The Consensus Power of Shared-Memory Distributed Systems

Eric Ruppert

University of Toronto
2000

A thesis submitted in conformity

with the requirements for the degree of Ph.D.

Graduate Department of Computer Science

University of Toronto

Abstract

In many asynchronous distributed systems, processes communicate by accessing objects in a shared memory. The ability of systems to solve problems in a fault-tolerant manner depends on the types of objects provided. Here, the wait-free model of fault-tolerance is used: non-faulty processes must run correctly even if other processes experience halting failures. The consensus problem, where processes begin with private inputs and must agree on one of them, has played a central role in analysing the power of distributed systems. This thesis studies the ability of different types of objects to solve consensus.

An object type has consensus number n if it can be used (with `read/write registers`) to solve consensus among n processes but not among $n + 1$ processes. Conditions are given that are necessary and sufficient for an object type to have consensus number n. This characterization applies to two large classes of objects: readable objects and read-modify-write (RMW) objects. An object is readable if processes can read its state without changing the state. For a RMW object, all operations update the state and then return the previous state of the object. When the type is of bounded size, the characterization may be used to decide the question "Does the type `T` have consensus number n?", which is undecidable for arbitrary types. The characterization is also used to show that different readable and RMW types with consensus number n cannot be used in combination to solve consensus for $n + 1$ processes.

Ordinarily, processes may access only one object in shared memory at a time. This thesis also studies how much the consensus number of a type increases in the multi-object and transactional models, where processes can perform operations on up to m of the objects in a single atomic action. These models are much more convenient for programmers to use, since they guarantee that certain blocks of operations will be executed without interruptions from other processes. This thesis establishes bounds on the consensus numbers of multi-objects and transactional objects as a function of m and the consensus numbers of the corresponding single-access types.

Acknowledgements

I am deeply indebted to my advisor, Faith Fich, who has been an ideal mentor. I greatly appreciate her enthusiasm, diligence and patience as a supervisor.

I received guidance and helpful comments from the other members of my advisory committee, Allan Borodin, Stephen Cook, Vassos Hadzilacos, Michael Molloy and Ken Sevcik; and from my external examiner, Michael Merritt.

I am extremely grateful for all of the support and encouragement I have received from my family.

During my time at the University of Toronto, I was very fortunate to take courses from many wonderful instructors. They include Michael Black, Allan Borodin, Stephen Cook, Derek Corneil, Faith Fich and Charles Rackoff of Computer Science; James Arthur, Michael Lorimer and Pierre Milman of Mathematics; Derek York of Physics; and Richard Toporoski of Classics at Saint Michael's College. I learned much from them, not only about the course material, but also about the art of effective teaching.

I thank Yehuda Afek, James Aspnes, Maurice Herlihy, Kal Lin, Wai-Kau Lo and Eric Schenk for helpful discussions about distributed computing. I also thank the anonymous conference reviewers who provided comments on the presentation of some of the material that appears in this thesis.

Many fellow students have helped make my years here so pleasant, including Demetrios Achlioptas, Geneviève Arboit, Alex Budanitsky, Rafal Kustra, Stephen Lam, Jin Lee, Steven Myers, François Pitt, Natasa Przulj, Jim Rootham, Michael Soltys and Iannis Tourlakis.

The department's administrative staff have always been cheerfully helpful. I thank the Fields Institute for Research in Mathematical Sciences for its hospitality during its Complexity Theory Programme.

This research was done with the financial support of postgraduate scholarships from the Natural Sciences and Engineering Research Council of Canada.

E.R.

Toronto

December 7, 1999

Contents

Chapter 1

Introduction

THE ability of a shared-memory distributed system to solve problems in a fault-tolerant way depends crucially on the kind of shared memory that is provided by the system. The goal of this thesis is a better understanding of the reasons that some kinds of shared memories are more powerful than others. The consensus problem, where a collection of processes must agree on a common value, has played an extremely important role in studying the power of different distributed systems. This thesis gives a systematic way of determining the ability of a system to solve consensus. These results are applicable to large, natural classes of distributed systems. In addition, this thesis studies how the ability to solve consensus changes when certain parameters of the shared-memory model are altered.

§1.1 Distributed Systems

When two do the same thing, it is not the same thing after all. —proverb

A distributed system can be modelled as a collection of processes, together with some means for processes to communicate with one another. Each process executes a sequential algorithm. The processes run concurrently to solve problems cooperatively. The processes may run on distinct processors connected by a network, or processes may share a processor by having the steps of their executions interleaved. In either situation, processes will often run at different speeds or experience unpredictable delays for a variety of reasons. For example, processes might be temporarily pre-empted by other processes performing tasks that are assigned a higher priority by the system, or memory accesses by different processes may require different amounts of time due to the architecture of the system. It is important, therefore, that distributed algorithms work correctly in the presence of uncertain execution speeds. This thesis considers asynchronous models of distributed computation. This means that algorithms must work correctly even if processes run at arbitrarily varying speeds.

In the presence of asynchrony, the processes of a distributed system must communicate with one another in order to coordinate their actions. This thesis is concerned with shared-memory models, where processes communicate by accessing shared data structures, called objects. Different systems are equipped with different types of shared objects. An object type will be modelled as an automaton that is specified by giving a set of possible states and by describing, for each possible operation, the state transition that occurs and the result returned to the process that performs the operation.

It is important to ensure that processes update shared data in an orderly way so that it does not become corrupt. For example, if two processes attempt to increment a counter, it is not sufficient that each process read the current value of the counter, add one to this value and then write the new value back into the counter. If two executions of this simple algorithm run concurrently, with both reads occurring before both writes, the value of the counter will increase only by one, even though the two processes both believe they have successfully incremented the counter.

Many distributed algorithms avoid concurrent updates to shared data altogether by using locks: a process must obtain an exclusive lock before being allowed to access the data [25]. However, the entire system can be affected when a process that holds a lock runs slowly or fails. Some algorithms that employ locks can also experience deadlock, where two processes wait for locks that are held by each other so that neither can ever make any progress, or livelock, where a process cannot obtain a lock because every time the process takes a step, some other process owns the lock. In 1977, Lamport [53] gave a distributed algorithm that implements multi-digit `registers` from single-digit `registers` without using locks. More recently, there has been much interest in wait-free algorithms [31], in which locks cannot be used. In a wait-free algorithm, each process must be able to complete its execution correctly in a finite amount of time, even if any number of other processes in the system experience halting failures. In addition to providing fault-tolerance, wait-free algorithms ensure that no process spends time waiting for another process to perform some action, since it cannot tell whether the other process has failed or is merely running slowly.

§1.2 Solvability and Consensus

In the standard random-access machine model of single-process computation [80], a computer's memory is modelled as an array of variables that can be read and written. This model of computation is used to characterize the set of problems that can be solved using sequential computation. This characterization is extremely robust: reasonable changes to the set of operations that can be performed on the memory does not change the class of problems that can be solved. Furthermore, there are several very different models that capture precisely the same notion of computability, including characterizations in terms of Turing machines [82] and recursive functions [49], for example. (See the survey paper [83].) The situation for wait-free distributed computation is much more complicated. In the distributed setting, there are many elementary tasks that cannot be solved in a wait-free manner if the shared memory consists only of `read/write registers`; more complex primitive memory operations are required to solve these problems. For example, the *compare&swap*(old, new) operation compares the current value of the object to the value *old*, and updates it to *new* only if the comparison returned true. The read, comparison and contingent update are performed as a single atomic operation, without interruption by other processes. A shared memory that permits *compare&swap* operations can solve a much larger class of problems than a system equipped only with `read/write registers` [31].

Distributed systems based on different types of shared-memory objects are available. For example, The MIPS RISC [48] and the DEC Alpha [81] architectures provide versions of `LL/SC` objects (which will be defined on page 63). Menke, Moir and Ramamurthy [60] describe how to build systems based on `compare&swap` objects using SCRAMNet network cards.

Designers of distributed systems must ensure that the primitive shared objects provided are sufficiently powerful to implement required data structures and solve the problems for which the system is being built. A shared-memory system will usually be equipped with a small number of object types as primitives. Any other types of shared objects that algorithms are to use must be implemented from the primitive objects. One can also model a problem as a shared object type: processes pass their inputs to the object, and the object returns the desired output. Thus, the question of whether the problem is solvable can be answered by determining whether this object type can be implemented. A central question in distributed computing, then, is whether a given set of objects types is capable of implementing another object type, for a given number of processes.

Herlihy [31] showed that the consensus problem, in which each process begins with an input and all non-faulty processes must agree on one of the input values, plays a central role in the study of the power of object types to implement other object types in a wait-free manner. He proved that object types that solve the consensus problem for n processes are universal for systems of n processes: they can be used to provide a wait-free implementation of *any* object type for n or fewer processes. (It is assumed throughout that `read/write registers` are available as basic shared objects, in addition to any other types mentioned explicitly.) Herlihy's result led to the idea of classifying object types into a consensus hierarchy according to their power to solve consensus [31, 42]. An object type is said to be at level n of the consensus hierarchy if it can solve consensus among n processes, but not among $n+1$ processes. If there is no such n, the object is at level infinity of the hierarchy.

Knowledge about the consensus hierarchy gives a great deal of information about the central question of whether one object type can be implemented from others, for a given number of processes. Herlihy's universality result provides information about what can be done using objects at level n of the hierarchy. On the other hand, the classification of objects into the hierarchy also gives information about what cannot be done: it follows from the definition of the hierarchy that an object type at level n cannot implement an object at any higher level in a system of more than n processes. Extensions of these results to the situation where several different primitive object types are provided will be discussed in Chapter 6. Consequently, the classification of types according to their power to solve consensus has provided much information about the important question of whether given types can implement other types of objects. In addition, the proof that a particular type `T` is at a certain level in this hierarchy usually includes an algorithm for consensus, and this can be combined with Herlihy's construction [31] to provide an upper bound on the complexity of implementing other objects from objects of type `T`.

This thesis provides theorems about the solvability of consensus that apply to large, natural classes of object types. There have not been many other general results about classes of objects that can solve a particular problem. However, there has been previous work on the related problem of characterizing the class of problems that can be solved by a particular object type, or showing that no such computable characterization exists. Herlihy's work, mentioned above, showed that types that can solve consensus can implement any other type. Chor and Moscovici [22] gave a decidable characterization of tasks that can be solved by randomized algorithms that use **`registers`** only. Several researchers have used relationships between distributed computation and the topological properties of simplicial complexes to characterize the problems that can be solved by particular types, or to show that no such computable characterization is possible. This approach has been applied to various primitive objects, including **`registers`** [30, 35] and **`set-consensus`** objects [34]. Herlihy and Rajsbaum also used the topological method to describe the types of objects that can solve the set-consensus problem [33].

§1.3 Main Results in this Thesis

Previously, objects have been classified into the consensus hierarchy mostly on a case-by-case basis. The proof that a particular type is at level n of the hierarchy usually involves two steps. One must first show that there is a consensus algorithm for n processes. Wait-free algorithms are often difficult to construct and proofs of correctness are typically quite complex. One must also show that no consensus algorithm exists for $n+1$ processes. Such proofs are generally done by adapting the proof, given by Fischer, Lynch and Paterson [28], that consensus is impossible in a message-passing system. They often involve lengthy, technical arguments. It is desirable to have a general framework for proving such results easily.

The goal of this thesis is to provide a systematic way of classifying objects into the consensus hierarchy. Unfortunately, the problem of determining whether a given object belongs to a given level of the hierarchy is undecidable in general [47]. Much of the work in this thesis therefore focuses on two classes of objects: read-modify-write objects and readable objects. These are two large, natural classes of objects that include many of the objects that have been considered as the basis of shared-memory systems. An object is called readable if it is equipped with operations that allow processes to read the data stored in the object without changing it. A read-modify-write object is one where every operation reads and returns the value stored in the object and also updates it by applying some function to the old value. This thesis gives combinatorial characterizations of the objects in these classes that are at level n of the consensus hierarchy.

Knowledge of the consensus hierarchy provides information about how well individual object types can be used to solve consensus. However, this knowledge does not necessarily extend to systems in which several different types are available; it may be that several weak objects which are incapable of solving consensus on their own can be used in combination to solve consensus. The hierarchy is said to be robust if there are no object types that can be combined to solve consensus better than any of the individual types alone. Robustness is a desirable property, since information about the power of a system containing several types of objects can then be obtained by studying each of the types separately. It has been shown that the hierarchy is not robust in a variety of settings [18, 57, 64, 77]. However, the non-robustness proofs make use of some unusual types, so they do not rule out the possibility that the hierarchy is robust for some natural classes of objects. It will be shown here that the hierarchy is indeed robust for readable and read-modify-write types.

In the asynchronous models of distributed systems considered here, algorithms must work correctly regardless of the way that the steps of processes are interleaved. This unpredictability in the scheduling of steps makes an algorithm designer's task difficult. One way to moderate this difficulty is for the system to ensure that certain small blocks of code are executed without interruptions from other processes. This thesis studies how the consensus power of a system increases when the scheduling of processes is constrained in two ways. In the multi-object setting [4], one can specify that a bounded number of independent instructions on distinct objects must be executed as a single atomic action. In the transactional setting, blocks of code which perform a bounded number of shared-memory operations may be labelled as transactions, which are to be performed atomically. One possible approach to designing a distributed algorithm is to implement the more convenient multi-object types or transactional types from the corresponding single-access types that are more commonly provided in distributed systems. It will be shown that this approach is impossible in general, since multi-objects and transactional objects are usually strictly more powerful than the corresponding basic objects. Let m be the bound on the number of objects that can be accessed in a single action. It will be shown that, for any object type at level three of the consensus hierarchy or higher, the number of processes that can solve consensus increases by a factor of $\Omega(\sqrt{m})$ (or $\Omega(m)$ if the object type is readable) in the multi-object model. These results also apply to transactional objects, since they are less restrictive than multi-objects, but some even stronger results will be described. It will be shown that the consensus level of a transactional type built from any non-trivial base type is $\Omega(2^{m/2})$. (An object type is trivial if processes can simulate it without using shared memory at all.) A stronger lower bound of $\Omega(2^m)$ will be established for read-modify-write base types. It will also be shown that a transactional object can solve consensus among any number of processes whenever the basic objects used in the transactional object are readable and non-trivial.

§1.4 Outline of this Thesis

Chapter 2 describes the formal model of distributed computation that will be used. Chapter 3 gives a formal definition of the consensus problem and the consensus hierarchy. It also includes a description of a proof technique called a bivalency argument [28] which will be used several times in the succeeding chapters. The team consensus problem is a restricted version of the standard consensus problem that will be used in many of the proofs. It is also introduced in Chapter 3.

Chapter 4 gives a combinatorial characterization of the read-modify-write object types that are capable of solving consensus among n processes, for any n. A similar characterization is given for readable types in Chapter 5. Chapter 6 discusses the robustness of the consensus hierarchy.

Chapter 7 defines multi-objects and transactional objects formally. Chapters 8 and 9 describe how the power of object types to solve consensus is increased when multi-objects and transactional objects are used. The concluding chapter contains some discussion of the results and future directions for research related to the topics of this thesis.

Preliminary versions of most of the results in this thesis have appeared in three papers [72, 73, 74].

Chapter 2

The Model

It's a perfect situation
I'm trying hard to understand.
—Electronic [27]

THIS chapter gives a formal definition of the model that will be used for shared-memory systems in this thesis. The model is adapted from the description that was given by Herlihy and Wing [36] and later expanded and formalized by several different researchers [13, 16, 31, 41]. Lynch's text [59] discusses this model in terms of I/O automata. The basic definitions of object types, protocols, schedulers and implementations are given first; a discussion of more general definitions of object types that have been studied appears in Section 2.5.

§2.1 Shared Objects

A shared memory consists of a collection of *objects*. Each object is an instance of an *object type*. A sequential specification of an object type describes how the object behaves when single operations are performed on it. At any time, an object has a *state*, chosen from a set Q of possible states. There is a set, OPS, of operations that can be applied to the object. An element of this set includes both the name of the operation and any parameters of the operation. The set RES contains the possible responses that can be returned by the object when an operation is performed. A transition function δ describes the state transition that occurs and the response returned by the object when a given operation is performed in a given state.

Definition 2.1 A sequential specification of an object type is a tuple $(Q, \mathrm{OPS}, \mathrm{RES}, \delta)$, where Q, OPS and RES are sets and δ is a function from $Q \times \mathrm{OPS}$ to $Q \times \mathrm{RES}$.

Suppose $\delta(q, op) = (q', res)$ for some $q, q' \in Q$, $op \in$ OPS and $res \in$ RES. This is interpreted to mean that if an object is in state q and a process accesses the object using the operation op, the object will move into state q' and return the response res to the process.

To ensure that objects do not have unreasonable power, all objects considered in this thesis have computable transition functions δ and recursive (but not necessarily finite) sets Q, OPS and RES. (All references to computability in this thesis refer to computability in the sequential, random-access machine model.) Many of the results in this thesis hold even without these computability assumptions. This will be discussed in Section 10.1.

Example 2.2 A `test&set` object stores a single bit, and is equipped with a single operation that reads the current value of the bit before setting it to 1. It has the following sequential specification.

$$\begin{aligned} Q &= \{0, 1\} \\ \text{OPS} &= \{test\&set\} \\ \text{RES} &= \{0, 1\} \\ \delta(q, test\&set) &= (1, q), \text{ for } q \in Q \end{aligned}$$

When a process makes a request to access an object using one of the operations, it is said to *invoke* the operation. In reality, there is some interval of time between the moment that the operation is invoked and the moment when the object returns a response to the process that invoked the operation. The sequential specification of an object type completely describes how the object behaves when accessed by one process at a time. However, in a distributed system, it is possible for the intervals of operations performed by different processes on an object to overlap. One must give additional constraints to describe the legal behaviour of objects when this occurs. A number of such constraints have been considered, including sequential-consistency [54], weak-consistency [39] and set-linearizability [65]. This thesis will focus on *linearizable* objects [36], in which all operations must appear to occur instantaneously at some time during the operation's interval.

A more formal description of linearizability requires some terminology. Consider an object of type `T` starting in state q. A *history* of the object is a sequence of operation invocations and matching responses. Each invocation has at most one matching response, which must occur after the invocation. A history is intended to describe all operations that are performed on the object during the execution of some algorithm. A *linearization* of the history is a linear order of those operations in the history that have a response, together with some subset of the operations that have no response, satisfying the following conditions:

(1) if the response to operation op_1 occurs before the invocation of operation op_2 in the history, then op_1 must occur before op_2 in the linearization, and

(2) the response to any operation in the history is the same as it would be if the operations were performed sequentially, in the order given by the linearization, on an object of type `T` starting in state q.

An object in a distributed system is called *linearizable* if its behaviour under any possible history of accesses has a linearization. Under the assumption that objects are linearizable, the sequential specification of a type completely determines the legal behaviour of the type. Thus, when defining linearizable object types, it is sufficient to give a sequential specification for the type.

An important type of object is the `read/write register`, which is defined formally here.

Definition 2.3 A `read/write register` that takes values in $\mathbb{N}$ has the following sequential specification.

$$\begin{aligned}
Q &= \mathbb{N}\\
\mathrm{OPS} &= \{write(v) : v \in \mathbb{N}\} \cup \{read\}\\
\mathrm{RES} &= \mathbb{N} \cup \{ack\}\\
\delta(v, write(v')) &= (v', ack),\ \text{for } v, v' \in \mathbb{N}\\
\delta(v, read) &= (v, v),\ \text{for } v \in \mathbb{N}
\end{aligned}$$

These objects will be referred to simply as `registers` hereafter.

As mentioned in the introduction, it will always be assumed that `registers` are available, in addition to any other object types mentioned explicitly. Although this assumption simplifies the descriptions of algorithms, it is not crucial for the results in this thesis: Bazzi, Neiger and Peterson showed that a deterministic object type `T` can be used with `registers` to solve consensus among n processes if and only if consensus among n processes can be solved using only objects of type `T`. For simplicity, it is assumed that a `register` can store an arbitrary integer value. The impossibility results given in this thesis do not assume any bound on the size of the `registers`. However, for any algorithm described, it is sufficient that the `register` be able to store a "reasonable" amount of data such as a process id or an input value for a problem. It is also possible to implement the `registers` required by the algorithms from a much more basic type of `register` that stores a single bit and can only be read by one process and updated by one other process (see [12], Chapter 10).

§2.2 Protocols

A protocol is a description of the algorithm to be followed by each of the processes in a distributed system. In the situation being studied in this thesis, each process runs on a single processor, so it is natural to model the algorithm executed by a process as a random-access machine programme.

Definition 2.4 Let $\mathcal{O}$ be a set of objects and $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of processes. A *protocol* for the processes $\mathcal{P}$ using objects $\mathcal{O}$ consists of

(1) an assignment of initial states to the objects of $\mathcal{O}$, and

(2) a programme Π_i for each process $P_i \in \mathcal{P}$.

Each programme Π_i is a sequential algorithm containing standard random-access machine operations and operations on objects from the set $\mathcal{O}$.

For an instance of a distributed problem, processes often begin with private inputs. The private input of a process is stored initially in the memory of the random-access machine used to model the process. The value stored in the first variable of the random-access machine when a process halts can be interpreted as the output of that process. The accesses to shared-memory objects are treated like black-box subroutine calls which return a response. In this thesis, protocols will be given in pseudocode or using more informal descriptions with enough detail so that they could easily be formalized as random-access machine programmes. All protocols considered here are *deterministic*, although researchers have also studied randomized protocols. (See Lynch's textbook [59] for a variety of randomized protocols.) The consensus hierarchy collapses when randomization is permitted, since it is possible to solve consensus (with probability one) using `registers` alone [21].

In Definition 2.4, it is assumed that the programmer may specify the initial state of all objects used by a protocol. One might also consider the case where objects of a given type must begin in a fixed initial state q_0. Borowsky, Gafni and Afek [16, Lemma 3.2] showed that the consensus problem can be solved using a certain type of objects whose initial states are chosen by the programmer if and only if consensus can be solved using objects of that type starting in a specific state q_0, provided all states are reachable from q_0 by some sequence of operations. Since this thesis is concerned with the ability of objects to solve consensus, the ability to specify initial states of objects in Definition 2.4 is not crucial.

To ensure that protocols do not have unreasonably powerful abilities by virtue of the assignment of initial states to the objects, it will be assumed that the set of objects $\mathcal{O}$ is enumerable, and that the assignment of an initial state to each object is a computable function.

§2.3 The Scheduler and Executions

Sors inmanis
et inanis,
rota tu volubilis,
status malus
—mediaeval lyric [37]

This section describes the way in which protocols are executed by a distributed system. The systems considered here are asynchronous. This means that the system can schedule steps of the processes in an arbitrary order. It is useful to think of the scheduler as an adversary: each process that is permitted to take sufficiently many steps must complete its algorithm correctly regardless of the way the scheduler interleaves the steps of other processes.

Individual steps by a process are treated as if they occur instantaneously. This is a reasonable assumption, since all objects are linearizable, so one can think of the scheduler choosing the linearization of the operations.

Formally, a *schedule* is a mapping from the natural numbers to the set of processes. An *execution* of a protocol is determined by the private inputs to the processes and the schedule σ: process $\sigma(0)$ performs one step of its programme, then process $\sigma(1)$ takes one step, followed by a step of process $\sigma(2)$, and so on. If, at some point, the process scheduled to take the next step has already completed its programme, the process does nothing during its turn to take a step. A process P is said to experience a *(halting) failure* in an execution if there is some point in the execution where P has not completed its algorithm and, after which, P is never scheduled to take a step. More formally, P fails if there is some $i \in \mathbb{N}$ such that P has not completed its programme before the ith step of the execution of the protocol and $\sigma(j) \neq P$ for all $j \geq i$. Such a process is called *faulty*, and processes that do not experience a halting failure are called *non-faulty*.

If a single process takes all steps during a schedule, it is called a *solo execution* by that process.

One says that process P cannot distinguish two executions of a protocol (or that the two executions are indistinguishable to P) if P begins with the same private input in both executions and, for each shared-memory access performed by P, P receives the same response in the two executions. Many of the arguments in succeeding chapters will use the fact that a process must produce the same output value in any two executions of a protocol which are indistinguishable to it.

§2.4 Wait-Free Implementations

In the model considered here, protocols must tolerate asynchrony and any number of failures. This fault-tolerance property is called wait-freedom [31].

Definition 2.5 A protocol is called *wait-free* if, in any execution, all non-faulty processes complete their programmes after a finite number of steps.

An important theme of this thesis is the ability of a set of objects to implement or simulate another type of object. A formal definition of a wait-free implementation is given here.

Definition 2.6 Let $\mathcal{O}$ be a set of objects and $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of processes. Let $\mathtt{T} = (Q, \mathrm{OPS}, \mathrm{RES}, \delta)$ be an object type. A *(wait-free) implementation* of an object of type T from $\mathcal{O}$ for the processes $\mathcal{P}$ consists of

(1) an assignment of initial states to the objects of $\mathcal{O}$ for each possible initial state in Q, and

(2) a programme Π_i^{op} for each process $P_i \in \mathcal{P}$ and each operation $op \in \mathrm{OPS}$.

Each programme is of the same form as described in Definition 2.4 and outputs a value in RES. The implementation must satisfy the following correctness condition. Suppose the objects of $\mathcal{O}$ are initialized for the state $q \in Q$, and each process P_i performs some sequence of programmes $\{\Pi_i^{op_{i,j}}\}$ according to some schedule. Then each programme $\Pi_i^{op_{i,j}}$ must terminate in a finite number of steps, if P_i does not fail during its execution. These executions of the programmes define a history of an object of type T, with an invocation of $op_{i,j}$ corresponding to the first instruction of the programme $\Pi_i^{op_{i,j}}$, and a response corresponding to the termination of the programme. There must be a linearization of this history matching the specification of type T, starting from state q.

Informally, this means that an object of type T can be replaced in any protocol by the implementation from $\mathcal{O}$: accesses to the object are simulated by running the appropriate programme. Implementations provide a way of building complicated protocols in a modular way by constructing complex shared data structures from simpler ones. This modular approach can often simplify proofs of correctness. Implementations can be composed: if an object X of type T can be implemented from a set $\mathcal{O}$ of objects and all of the objects in $\mathcal{O}$ can, in turn, be implemented from a set $\mathcal{O}'$, then X can be implemented from the set $\mathcal{O}'$ as follows. In the implementation of X from $\mathcal{O}$, each operation on an object of $\mathcal{O}$ can be replaced by the programme that implements it using the objects of $\mathcal{O}'$. The initial state of X determines the initial values of the objects of $\mathcal{O}$ and this determines the initial values to be used for the objects of $\mathcal{O}'$. (Lynch's textbook [59, Chapter 13] gives a more formal description of implementations and compositions of implementations in terms of I/O automata.)

For simplicity, Definition 2.6 does not permit the programme Π_i^{op} to depend on the initial state of the implemented object. There is no real loss of generality in such a restriction, provided `registers` can be used in the implementation. An implementation can include a read-only `register` that is initialized with the initial state of the simulated object. The information stored in this `register` can be used by the programme Π_i^{op} to determine how to simulate the operation *op*.

§2.5 Variants of the Model

A few variants of the model are described in this section. These are defined here so that previous work related to the topics in this thesis can be described precisely.

§2.5.1 Non-determinism

In Definition 2.1, object types are *deterministic*: the actions of the object in response to an operation are uniquely determined by the current state of the object and the operation applied. If the uniqueness condition is relaxed, a larger class of object types, called *non-deterministic* types is obtained. In this case, the specification of the object type will give a set of possible actions that the object can take in response to any operation invocation. Formally, such objects can be defined by replacing the transition function $\delta : Q \times \text{OPS} \rightarrow Q \times \text{RES}$ in Definition 2.1 by a function $\nu : Q \times \text{OPS} \rightarrow \mathcal{P}(Q \times \text{RES}) - \{\emptyset\}$, which gives the (non-empty) set of possible state transitions and responses that can occur when a given operation is applied to an object in a given state.

Protocols that use non-deterministic objects are required to work correctly regardless of the non-deterministic choices made by the objects. Although most of the objects considered in the field of distributed computing are deterministic, non-deterministic specifications are sometimes useful. For example, if an object that stores an array of records has an operation that sorts the records according to some key, it may be desirable to introduce some non-determinism into the specification so that records with the same key can be arbitrarily ordered after the *sort* operation. This provides some additional freedom in the implementation of the object, with the knowledge that protocols that use the object will work correctly regardless of the details of the implementation of the *sort* operation, which may or may not shuffle records with the same key.

The issue of determinism for objects is quite separate from the distinction between deterministic and randomized protocols. Allowing non-deterministic objects makes the protocol designer's task more difficult, since there is additional uncertainty about how objects will behave when the protocol is executed. Allowing randomization, on the other hand, gives the protocol designer extra tools to use.

§2.5.2 Non-oblivious Objects

Another variation of the basic definition of object types involves non-oblivious types. The object types defined in Definition 2.1 are called *oblivious* [50] because they behave in the same way towards all processes: the action an object takes in response to an operation cannot depend on the identity of the process that invoked the operation. Sometimes, one might want to restrict the set of processes that can perform an action, or possibly allow an object to respond differently to an operation, depending on the process that invoked it. *Non-oblivious objects* are used to model these situations. A non-oblivious object has a number of different *ports* to which processes must attach themselves before invoking operations. The set of available operations at each port is specified separately. Formally, such an object can be specified by giving a tuple $(Q, \mathrm{OPS}_p, \mathrm{RES}_p, \delta_p)$, as in Definition 2.1, for each port p of the object.

Example 2.7 A `single-writer, single-reader register` allows only one process to change the state and only one process to read the state. It has two ports: the writer's port called w and the reader's port called r. Suppose $Q = \mathbb{N}$. Then the behaviour of the writer and reader can be described as follows.

$$
\begin{array}{rclcrcl}
\mathrm{OPS}_w & = & \{\mathtt{write}(v) : v \in \mathbb{N}\} & \qquad & \mathrm{OPS}_r & = & \{\mathtt{read}\} \\
\mathrm{RES}_w & = & \{ack\} & & \mathrm{RES}_r & = & \mathbb{N} \\
\delta_w(v, \mathtt{write}(v')) & = & (v', ack),\ \text{for } v, v' \in \mathbb{N} & & \delta_r(v, \mathtt{read}) & = & (v, v) \text{ for } v \in \mathbb{N}.
\end{array}
$$

There are several different ways to model how processes attach themselves to the ports of a non-oblivious object to gain access to the object [16, 41]. The three most commonly used models are called soft-wired, hard-wired and restricted hard-wired objects.

In the *hard-wired* model, at most one process may access each port of each object, and the assignment of processes to ports is part of the description of the protocol: processes are bound to ports at compile-time. The *restricted hard-wired* model adds the constraint that each process may be bound to at most one port of each object. In the *soft-wired* binding model, different processes may access the same port of an object at different times during the execution of a programme, and the ports used by a process may be chosen at run-time. Before invoking an operation on port i of a soft-wired object, a process must execute a special operation $bind(i)$ on the object, and when it has finished using the port, the process executes another command $unbind(i)$ on the object. A protocol is legal only if no process ever attempts to bind itself to a port that is currently bound to another process. If a process fails while it is bound to a port of a soft-wired object, no other process may ever use that port. Any protocol that uses the restricted hard-wired model will also be legal in the ordinary hard-wired model, and any protocol that uses the hard-wired model will also work in the soft-wired model.

One could also consider a restricted version of the soft-wired model, where each process may be bound to at most one port of each object at a time. However, this version is equivalent to the ordinary soft-wired model. A protocol Π in the standard soft-wired model can easily be transformed into a protocol Π' in the restricted soft-wired model by removing all bind and unbind operations, and then adding operations before and after each shared-memory access: the process binds to a port during its step just before it needs to access an object and releases the port in its next step after accessing the object. The protocol Π' clearly satisfies the restriction that a process is bound to at most one object at a time.

Furthermore, no two processes will ever be bound to the same port at the same time in any execution of the new protocol Π', since a process is bound to a port in the new protocol only if it was already bound to that port in the corresponding execution of the original protocol. To see why Π gives the correct output, consider any execution E' of Π'. One can obtain a corresponding execution E of Π as follows. First, remove all bind and unbind steps from E', but leave the steps that access shared objects intact. Next, insert the bind and unbind operations of each process P in between the access steps of P as specified by P's programme in Π. (The relative order of binds and unbinds by different processes can be chosen arbitrarily.) In the resulting execution E, each process is performing its operations as specified by Π in the correct order. Furthermore, the sequence of accesses to each shared object is identical in E and E', so that accesses return the same responses in the two executions. Thus, E is a legal execution of Π.

Since Π is correct, and both executions produce the same results, Π' is also correct.

It is also possible to alter the definition of non-oblivious objects to allow for non-deterministic, non-oblivious objects by defining each port's transition function to give a non-empty set of state transitions and responses, as in Section 2.5.1.

Chapter 3

Consensus

HAMLET. *Do you see yonder cloud that's almost in shape of a camel?*
POLONIUS. *By the mass, and 'tis like a camel, indeed.*
HAMLET. *Methinks it is like a weasel.*
POLONIUS. *It is backed like a weasel.*
HAMLET. *Or like a whale?*
POLONIUS. *Very like a whale.* —Shakespeare [78]

THIS chapter describes the consensus problem and gives a general summary of previous work on the problem. Section 3.2 discusses the consensus hierarchy, and Section 3.3 describes an important proof technique called a bivalency argument. Section 3.4 discusses a special case of consensus called team consensus.

§3.1 The Consensus Problem

In the consensus problem, a set of two or more processes must agree on a common value from a set of input values. Each process is given a value as input. Assume the input values are natural numbers. Initially, no process has any information about the input value of any other process. The goal is to have processes decide on output values so that the following three properties are satisfied in every possible execution.

Termination: Every non-faulty process outputs a value after a finite number of steps.

Agreement: All output values of processes are the same.

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

A faulty process may or may not output a value; if it does, the value must satisfy the agreement and validity properties. The validity condition ensures that the problem cannot be solved trivially by having every process always output zero. The consensus problem for a single process is sometimes considered to be a degenerate case which can be solved trivially without using shared memory at all by having the single process return its own value. The consensus problem for n processes is sometimes called the n-consensus problem.

Several variants of the consensus problem have been studied. A version of the consensus problem was first studied by Pease, Shostak and Lamport [55, 66]. De Prisco, Malkhi and Reiter [24] describe five versions of the problem where the validity constraint is weakened in different ways. Leader election and binary consensus are special cases of the ordinary consensus problem. For *leader election*, the input to each process is its own (unique) process id. For *binary consensus*, all inputs to processes are either 0 or 1. Any binary consensus protocol can be used repeatedly to solve the leader election problem, if shared `registers` are available. Each process first writes its id to a `register`, and then executes a protocol to agree on the output id one bit at a time. Process P begins by proposing the bits of its own id. If the agreed value for a bit ever differs from the value P proposed, the process reads the `registers` to find a process id whose bits are the same as the bits that P has already agreed on, and then begins proposing the remaining bits of that id. A leader election protocol can be used to solve consensus, if `registers` are available, by having each process write its input to a `register` before electing a leader; each process then outputs the value written by the leader. Thus, the consensus, binary consensus and leader election problems can all be reduced to one another (if `registers` are available).

Chaudhuri introduced the problem of *k-set consensus* [20, 33] where, instead of having all processes return the same value, the set of values returned by processes must have at most k elements. Thus, the ordinary consensus problem is 1-set consensus. For *approximate agreement* [26, 75], the values returned need not be identical; instead, they must all be within an interval of bounded size. In the *simplex agreement* problem [16] and *loop agreement* problems [34], processes begin with a vertex in a simplicial complex and must decide on a vertex of the complex so that all vertices chosen lie on a single simplex (subject to certain validity conditions).

§3.1.1 Universal Constructions

Section 1.2 pointed out the importance of the consensus problem in studying the power of different kinds of distributed systems. This is largely due to Herlihy's universality result.

Theorem 3.1 [31] If an object type can be used (with `registers`) to solve consensus among n processes, then it can be used, along with `registers`, to implement any object type in a system of n processes.

The essential idea of Herlihy's universal construction is to have the system build a linked list of nodes, each one representing one operation on the simulated object. The simulated operations are linearized in the order that they appear on the list. Processes solve consensus to agree on the order in which nodes should be added to the end of the list. For each node in the list, a process can compute the response and new state that results from the simulated operation by applying the transition function to the state resulting from the previous node's simulated operation. To ensure that the implementation is wait-free, processes spend part of their time helping one another to add nodes to the list. This helping mechanism was designed to ensure that every simulated operation is eventually added to the list.

Plotkin gave a similar kind of universal construction [68]. Jayanti and Toueg [47] gave a more practical version of Herlihy's construction. Researchers have created more efficient universal constructions from various primitive object types [2, 19, 32]. Recently, Jayanti [43, 45] has studied some fundamental limitations on the efficiency of universal constructions.

§3.1.2 `Consensus` Objects

It is sometimes useful to think of the consensus problem in terms of an object type. A `consensus` object is accessed by a *propose* operation with a non-negative integer value as its argument. If the object starts in the initial state $\perp$, it returns, as the response to all operations, the first value that is proposed to it. Formally, the object type can be specified as follows.

$$\begin{aligned} Q &= \{\perp\} \cup \mathbb{N} \\ \text{OPS} &= \{\mathit{propose}(v) : v \in \mathbb{N}\} \end{aligned}$$

$$\begin{aligned} \text{RES} &= \mathbb{N} \\ \delta(\perp, \textit{propose}(v)) &= (v, v), \text{ for } v \in \mathbb{N} \\ \delta(v', \textit{propose}(v)) &= (v', v'), \text{ for } v, v' \in \mathbb{N} \end{aligned}$$

Let n be a positive integer. An n-`consensus` object has similar external behaviour to a `consensus` object for the first n accesses, if it is initialized to the state $\perp$. However, after n accesses, the object never again reveals the first value proposed to it. The formal description below captures this by adding to the state a counter which keeps track of the number of *propose* operations that have been performed since the object was in the state $\perp$.

$$\begin{aligned} Q &= \{\perp\} \cup (\mathbb{N} \times \{1, 2, \ldots, n\}) \\ \text{OPS} &= \{\textit{propose}(v) : v \in \mathbb{N}\} \\ \text{RES} &= \mathbb{N} \cup \{\textit{ack}\} \\ \delta(\perp, \textit{propose}(v)) &= ((v, 1), v), \text{ for } v \in \mathbb{N} \\ \delta((v', i), \textit{propose}(v)) &= ((v', i+1), v'), \text{ for } v, v' \in \mathbb{N}, 1 \leq i \leq n-1 \\ \delta((v', n), \textit{propose}(v)) &= ((v', n), \textit{ack}), \text{ for } v, v' \in \mathbb{N} \end{aligned}$$

The problem of solving consensus among n processes is equivalent to the problem of implementing an n-`consensus` object.

§3.2 The Consensus Hierarchy

Because of the central role of the consensus problem in studying the relative power of different types of shared memory, researchers have classified types of shared memory according to their power to solve consensus.

The *consensus number* of an object type `T`, denoted cons(`T`), is the maximum number of processes that can solve consensus using objects of type `T` and `registers`, or infinity if no such maximum exists. This classification is called the consensus hierarchy [31, 42]. When the consensus number of a type is ∞, the type is called *universal*, since it can be used to implement any object in a system with any number of processes, by Theorem 3.1. If a type has consensus number n, it is said to be at level n of the consensus hierarchy. Thus, higher levels in the hierarchy contain more powerful objects.

Jayanti [42, 44] discussed several similar hierarchies that classify object types into the levels $\mathbb{Z}^+ \cup \{\infty\}$. He identified several desirable properties of such classifications. A hierarchy is called a *wait-free* hierarchy if any object type at level n is capable of implementing every object type in a system of n processes. By Theorem 3.1, the consensus hierarchy is a wait-free hierarchy. A wait-free hierarchy is called *tight* if every type is at as high a level as possible: *i.e.* any type that can implement every other type for n processes is at level n. The consensus hierarchy is the unique tight wait-free hierarchy. A hierarchy is called *robust* if no object at one level of the hierarchy can be implemented by a collection of object types from lower levels. Intuitively, this means that weak objects cannot be combined to implement stronger objects. Robustness of the consensus hierarchy will be discussed in Chapter 6.

The first type whose consensus number was known was the `register`. It is at level one of the hierarchy. Loui and Abu-Amara [58] proved this, building on the first impossibility result for the consensus problem in a message-passing system with halting failures, by Fischer, Lynch and Paterson [28]. Herlihy's initial paper on the consensus hierarchy [31] gave the consensus numbers of several fundamental types. For example, he showed that the type `queue` is at level two, that `compare&swap` is at level infinity, and that a collection of `registers` is at level $2m-2$, if processes may write to $m > 1$ of the `registers` in a single atomic action.

§3.2.1 Consensus Using Several Object Types

Distributed systems sometimes contain several different types of objects. Consensus numbers are defined for single types. There is no real loss of generality in defining them this way, however, since for any set of types, one can define a single type that has equivalent power.

Definition 3.2 Let $\mathtt{T}_1 = (Q_1, \mathrm{OPS}_1, \mathrm{RES}_1, \delta_1)$ and $\mathtt{T}_2 = (Q_2, \mathrm{OPS}_2, \mathrm{RES}_2, \delta_2)$ be two types. Assume that the state sets Q_1 and Q_2 are disjoint. (If they are not disjoint, the states of one of the types can be renamed so that they are.) Define the type

$$\mathtt{T}_1 \vee \mathtt{T}_2 = (Q_1 \cup Q_2, \mathrm{OPS}_1 \cup \mathrm{OPS}_2, \mathrm{RES}_1 \cup \mathrm{RES}_2 \cup \{ack\}, \delta),$$

where $\delta(q, op) = \delta_i(q, op)$ if $q \in Q_i$ and $op \in \mathrm{OPS}_i$, for some $i \in \{1, 2\}$ and $\delta(q, op) = (q, ack)$ otherwise.

The $\vee$ operator can be used to build complex types out of simpler types. Afek, Merritt and Taubenfeld [4] described a variety of similar operators, called *combinators*. An object of type $\mathtt{T}_1 \vee \mathtt{T}_2$ behaves either like an object of type $\mathtt{T}_1$ or like an object of type $\mathtt{T}_2$, depending on the state in which it is initialized. If it starts in a state of Q_1, it behaves like an object of type $\mathtt{T}_1$. If a process then tries to apply any operation that is not defined on $\mathtt{T}_1$, it has no effect on the object. This definition (and the following results) can be extended in the natural way from two types to any finite set of types.

Proposition 3.3 Consider one system equipped with objects of types $\mathtt{T}_1$ and $\mathtt{T}_2$ and another system equipped with objects of type $\mathtt{T}_1 \vee \mathtt{T}_2$. Any protocol that runs in one system can be adapted to run in the other.

Proof: Any protocol that uses objects of types $\mathtt{T}_1$ and $\mathtt{T}_2$ can be run in a system that has objects of type $\mathtt{T}_1 \vee \mathtt{T}_2$, simply by replacing each of the objects by an object of type $\mathtt{T}_1 \vee \mathtt{T}_2$.

Consider any protocol that uses objects of type $\mathtt{T}_1 \vee \mathtt{T}_2$. It can be run in a system with objects of types $\mathtt{T}_1$ and $\mathtt{T}_2$ by replacing each object O by an object of type $\mathtt{T}_1$, if the initial state of O is a state of $\mathtt{T}_1$, or by an object of type $\mathtt{T}_2$ otherwise. Whenever the protocol attempts to perform an operation on one type of object that is only defined for the other type of object, it can be simulated without using shared memory by simply returning the value *ack*. □

Corollary 3.4 A system equipped with objects of types $\mathtt{T}_1$, $\mathtt{T}_2$ and `registers` can solve consensus among n-processes if and only if $\text{cons}(\mathtt{T}_1 \vee \mathtt{T}_2) \geq n$.

§3.2.2 Deciding Consensus Numbers

It's a perfect situation
I can't hope to understand.
—Electronic [27]

The proofs required to classify individual object types into the consensus hierarchy are often fairly intricate (see, for example, [56]), so a general technique for determining the consensus number of a given type would be desirable. However, Jayanti and Toueg [47] showed that there is no algorithm that can determine the consensus number of an object type, given its sequential specification. The sequential specification can be treated as a finite input by using a Turing machine to describe the transition function. In fact, Jayanti and Toueg showed that it is impossible to determine whether the type is at level one of the consensus hierarchy. Their proof uses some unusual object types which first simulate a Turing machine and then solve consensus if and only if the simulation halts. Thus, they showed that the undecidable halting problem for Turing machines reduces to the problem of determining the consensus number of object types.

Jayanti and Toueg's undecidability result means that the goal of automatically classifying object types into the consensus hierarchy cannot be achieved fully. This thesis therefore concentrates on specific classes of objects and gives characterizations of the types within those classes of objects that are at level n of the consensus hierarchy. It will be shown that, when the object types in question have finite state sets, the conditions that characterize the ability to solve consensus among n processes are decidable, although the decision algorithms do not run in polynomial time. The classes considered are readable objects and read-modify-write objects, which are natural classes that contain many of the most commonly considered shared object types. This thesis also gives general results about the consensus numbers of complex types built from simpler types whose consensus numbers are known.

§3.3 Bivalency Arguments

Fischer, Lynch and Paterson [28] introduced an important proof technique, called a *bivalency argument*, which has been widely used to establish the impossibility of consensus for many different types of shared memory. Some of the proofs in this thesis use bivalency arguments. This section introduces some useful terminology for bivalency arguments.

A *configuration* of a protocol at any moment in its execution consists of the state of every shared object, together with the internal state of every process. A configuration captures all of the information about the current status of the execution. A configuration of the protocol before any process has taken a step is called an *initial configuration* of the protocol. There are different initial configurations, one for each possible input to the protocol. One can think of the configurations of a protocol as the nodes of a directed graph that has an edge from configuration A to configuration B whenever a single step by some process will take the system from A to B. The configuration B is called a *successor* of A. Each path starting from an initial configuration represents an execution of the protocol, so that such a graph encodes all information about possible executions. A configuration of a consensus protocol is *univalent* if all possible executions that continue from that configuration lead to the same output value. The configuration is called *multivalent* otherwise.

A bivalency argument to prove that consensus is impossible in some shared-memory model works by showing that the final decision about the output value can never take place. The argument, a proof by contradiction, proceeds as follows. One assumes that there does exist some protocol that solves consensus. One first shows that any consensus protocol has a multivalent initial configuration, using the validity condition. Next, one shows that every multivalent configuration has a multivalent successor. Thus, it is possible to construct an infinite execution in which every configuration is multivalent, violating the termination condition.

A *critical configuration* C is a multivalent configuration, all of whose successors are univalent. Essentially, this means that the system has not, collectively, made a decision about the final output value in configuration C, but one further step by any process will determine the output value. The first step that a process takes after a critical configuration is called the *critical step* of that process. Since C itself is multivalent, critical steps taken by the processes from C must lead to at least two different output values. The output value that is eventually decided if process P is scheduled first after the critical configuration is called the *critical value* of process P (for configuration C). The earlier assumption that object types are deterministic is essential for this notion of a critical value to be well-defined: if the object accessed in the critical step is non-deterministic, different critical steps by the same process could lead to different decision values. Showing that multivalent configurations always have multivalent successors is often done by showing that the existence of a critical configuration leads to a contradiction. Frequently, this involves a technical argument by cases about what steps could possibly be taken by each process when the system is in the critical configuration.

The following well-known lemmata are important elements of most bivalency arguments.

Proofs are included, because the proofs later in this thesis build on the techniques used here. The proofs in this section are based on the original bivalency impossibility proofs [28, 58] which were done in a model with a weaker adversary that could fail at most one process.

Lemma 3.5 [28] Any consensus protocol has a multivalent initial configuration.

Proof: Consider the initial configuration when one process, P, has input 0 and another process, Q, has input 1. By the validity condition and termination conditions, a solo execution by process P must terminate with the output value 0, since P cannot distinguish this execution from the situation where all processes have input 0. Similarly, a solo execution by process Q must have output value 1. Thus, the initial configuration is multivalent. □

Lemma 3.6 [28] For any consensus protocol, there is some execution that includes a critical configuration.

Proof: Assume there are no critical configurations in any execution of the algorithm, to derive a contradiction. Then every multivalent configuration has a multivalent successor. By Lemma 3.5, there is a multivalent initial configuration. So there is an infinite execution σ in which every configuration is multivalent. However, whenever a process outputs a value, the protocol must be in a univalent configuration, since, by the validity condition, the outputs of all processes must be the same as the output already produced. Therefore, no process can ever output a value in the execution σ. This violates the termination condition of consensus. □

Lemma 3.7 [58] For any critical configuration of a consensus protocol, there is one object that is accessed by the critical step of every process. For each process, there is some execution from the critical configuration where the critical step of the process changes the state of the object.

Proof: Assume the statement is false to derive a contradiction. Then there must be either two processes, P and Q, that have different critical values but access different objects during their critical steps or one process, Q whose critical step never changes the state of the object. Consider an execution from the critical configuration in which P takes the first step, Q takes the second step, and then P runs until it terminates. Process P cannot possibly distinguish this execution from the execution starting from the critical configuration where Q takes one step and then P runs until it terminates. Thus, P must output the same value in both of these executions, contradicting the fact that P and Q have different critical values. □

This proof illustrates an important idea in bivalency arguments. The critical steps of processes P and Q on different objects are said to *commute*, since processes cannot tell the order in which the two steps are executed. That is, no process can distinguish an execution continuing from the critical configuration with P and Q taking a single step each from a similar execution where the critical steps of P and Q are reversed. If two processes have different critical values, their critical steps cannot commute with each other.

Lemma 3.8 [58] For any critical configuration of a consensus protocol, the object accessed by all processes in their critical steps is not a `register`.

Proof: Consider two processes P and Q with different critical values. By Lemma 3.7, they must both access the same object during their critical steps. Assume that object is a `register` to derive a contradiction. Neither process can perform a *read* during its critical step, since the critical step must be capable of changing the state of the object. So each process must perform a *write* operation in its critical step. Consider an execution from the critical configuration in which Q takes one step and then P runs until it terminates. Process P cannot distinguish this execution from the execution starting from the critical configuration in which P runs alone until it terminates. This contradicts the fact that P and Q have different critical values. □

This proof illustrates another idea frequently used in bivalency arguments. The *write* step of process P is said to *overwrite* process Q's step, since no process can tell whether Q executed its critical step before P's critical step or not. The critical step of one process cannot overwrite the critical step of another process if the two processes have different critical values.

These tools are sufficient to prove the first impossibility result for shared-memory consensus.

Theorem 3.9 [58] $\text{cons}(\texttt{register}) = 1$.

Proof: Assume there is a consensus protocol for two processes that uses only `registers`. By Lemmata 3.6 and 3.7, the protocol has a critical configuration, and all processes access the same object in their critical steps. Since `registers` are the only shared objects available, Lemma 3.8 cannot be satisfied, so the assumption must be false. □

In addition, one can prove that no level of the consensus hierarchy is empty using the n-`consensus` object defined in Section 3.1.2.

Theorem 3.10 cons(n-`consensus`) $= n$.

Proof: There is an obvious consensus protocol for n processes that uses one n-`consensus` object initialized to the state $\perp$. Each process proposes its input value to the object and outputs the response it receives.

Assume there is a consensus protocol for the processes $P_1, \ldots, P_{n+1}$ that uses n-`consensus` objects and `registers`. By Lemma 3.6 there is a critical configuration C of the protocol. All processes access the same object X in their critical steps after this configuration, by Lemma 3.7. The object X cannot be a `register`, by Lemma 3.8, so it must be an n-`consensus` object.

Processes must have at least two different critical values, since C is multivalent. Without loss of generality, suppose processes P_1 and P_2 have different critical values. Let S_1 be the configuration reached from C by having processes $P_1, P_2, \ldots, P_n$ each take a single step, in this order. Define S_2 similarly, except with the steps of P_1 and P_2 reversed. Regardless of the state of X in C, its state in S_1 and S_2 will have second component n. Thus, any further operations on X will return only the response *ack*. Since all other objects are in the same state in S_1 and S_2, a solo execution by P_{n+1} from either of these configurations must lead to the same decision value. This contradicts the fact that P_1 and P_2 have different critical values. □

The following result, originally proved by Jayanti and Toueg [47] using different objects, follows easily.

Corollary 3.11 There are object types at every level of the consensus hierarchy.

§3.4 Team Consensus

But ere sterne conflict mixt both strengths, faire Paris stept before
The Trojan host. Athwart his backe a Panther's hide he wore,
A crooked bow and sword and shooke two brazen-headed darts,
with which well-arm'd, his tongue provok't the best of Grecian hearts
To stand with him in single fight.
—Homer [38]

Let n_1 and n_2 be positive integers. The team(n_1, n_2) problem is defined to be a restricted version of the general consensus problem among $n_1 + n_2$ processes where the processes are divided (in advance) into two non-empty teams of sizes n_1 and n_2 and all processes on a team receive the same input value. Proposition 3.13 shows that this restricted version of the consensus problem can be used to solve the general version. This fact will be used in succeeding chapters to simplify proofs that consensus is solvable using various types of objects.

The following lemma follows directly from the definition of the team consensus problem.

Lemma 3.12 If a set $\mathcal{O}$ of objects can be used to solve the team(n_1, n_2) problem, then $\mathcal{O}$ can be used to solve the team(n_1', n_2') problem for all n_1' and n_2' such that $1 \leq n_1' \leq n_1$ and $1 \leq n_2' \leq n_2$.

Proof: Any protocol for the team(n_1, n_2) problem may be viewed as a protocol for the team(n_1', n_2') problem if one thinks of $n_1 - n_1'$ processes on the first team and $n_2 - n_2'$ processes on the second team failing before executing any of their steps. $\square$

To solve the general consensus problem using team consensus, each team recursively chooses, from among the team's inputs, a value to represent the team, and then a team consensus protocol is used to determine which of the two representative values is the final output. The algorithm behaves like a tournament algorithm. Initially, the input value of every process is eligible to be the output. This value is proposed to successive executions of team consensus protocols with increasing numbers of processes until one of those team consensus protocols chooses a different output. The value chosen by the team consensus protocol that is run among all n processes is the final output of the consensus protocol.

Proposition 3.13 For all i such that $2 \le i \le n$, let a_i and b_i be a pair of positive integers that sum to i and suppose O_i is a shared object that can be used with k_i `registers` to solve the team(a_i, b_i) problem for all i. Then n-consensus can be solved using objects $O_2, \ldots, O_n$, together with $\sum_{i=2}^{n} k_i$ `registers`.

Proof: The proposition will be proved by induction on n. For $n = 2$, the team consensus problem is identical to the general problem of 2-consensus. So, consensus among two processes can be solved using O_2 and k_2 `registers`.

Suppose the claim holds when the number of processes is less than n (where $n > 2$). This means that, for all $m < n$, $O_2, \ldots, O_m$ can be used together with $\sum_{i=2}^{m} k_i$ `registers` to solve m-consensus. This hypothesis will be used to provide a protocol for n-consensus. Divide the n processes into two non-empty teams A and B with a_n and b_n processes, respectively. The processes of team A will first execute a consensus protocol to agree on one of their input values. If $a_n = 1$, no shared objects are used. Otherwise, team A can agree on an input value using $\sum_{i=2}^{a_n} k_i$ `registers` and the shared objects $O_2, \ldots, O_{a_n}$, by the induction hypothesis. Similarly, the processes of team B agree on one of their input values. If $b_n = 1$, no shared objects are used. Otherwise, it can be done using $\sum_{i=a_n+1}^{n-1} k_i$ `registers` and the shared objects $O_{a_n+1}, \ldots, O_{n-1}$. This is possible, since O_{i+a_n-1} can be used with k_{i+a_n-1} `registers` to solve the team consensus problem among i processes for $2 \le i \le b_n$, by Lemma 3.12.

Next, the processes agree on which team's value should be used as the common decision value. The processes execute the protocol for the team(a_n, b_n) problem, with each process using the decision value of its team as its input. This can be done using O_n and an additional k_n `registers`. The total number of `registers` used by the protocol is $\sum_{i=2}^{n} k_i$.

By the inductive hypothesis, the consensus protocol within each team is wait-free. The team consensus protocol used to decide between the two teams' values is also wait-free. So, the entire consensus protocol is wait-free. The protocol satisfies the validity condition, since the value agreed upon by the winning team must be one of the input values of a process on that team, by the inductive hypothesis. The protocol satisfies the consistency condition, since the protocol for team(a_n, b_n) must satisfy the consistency condition. □

Proposition 3.14 Suppose that one object of type `T` and k **`registers`** can be used to solve team(n_1, n_2) for some n_1 and n_2 that sum to n. Then, n-consensus can be solved using an array of $n-1$ objects of type `T` and $k(n-1)$ **`registers`**.

Proof: By Lemma 3.12, one object of type `T` and k **`registers`** can be used to solve team consensus among i processes (for some division of the i processes into teams) for each i such that $2 \leq i \leq n$. Proposition 3.13 can then be applied with the objects $O_2, \ldots, O_n$ each being objects of type `T`. $\square$

Chapter 4

Read-Modify-Write Objects

CONSIDER an object type that has state set Q. For any function $f : Q \to Q$, the read-modify-write operation RMW_f returns the old value of the state and modifies the state by applying the function f. A *read-modify-write (RMW)* type is one where each operation is of the form RMW_f for some function f. The exact nature of the type depends on the set of functions that can be applied by RMW operations. This chapter studies the ability of different RMW types to solve consensus.

§4.1 RMW Types

The class of RMW types was defined by Kruskal, Rudolph and Snir [52]. A formal definition follows.

Definition 4.1 An object type with state set Q is called a *RMW type* if there is a set F of functions mapping Q to Q so that the sequential specification of the type has the following form.

$$\begin{aligned} \text{OPS} &= \{RMW_f : f \in F\} \\ \text{RES} &= Q \\ \delta(q, RMW_f) &= (f(q), q), \text{ for } q \in Q \text{ and } f \in F \end{aligned}$$

The sets Q and F need not be finite.

Example 4.2 The `compare&swap` type was mentioned on page 3. It has state set $Q = \mathbb{N}$, and it is equipped with operations of the form *compare&swap*(old, new), where old and new are elements of the state set. This operation changes the state of the object to *new* if and only if the object is in the state *old*, and it returns the state of the object before the operation was applied. The operation can be viewed as a *RMW* operation that applies the function

$$f_{old,new}(x) = \left\{ \begin{array}{ll} new & \text{if } x = old, \\ x & \text{if } x \neq old, \end{array} \right\}$$

where x, old and new are elements of the state set. Thus, the `compare&swap` type is a RMW type with $F = \{f_{old,new} : old, new \in \mathbb{N}\}$. This is an example of a type that is at level ∞ of the consensus hierarchy [31].

In addition to *compare&swap* operations, many common operations can be viewed as *RMW* operations. A *read* operation that returns the state of an object without altering the state is a *RMW* operation that applies the identity function.

If an object type has state set $\mathbb{N}$, the *fetch&add*(v) operation applies the function f_v defined by $f_v(x) = x + v$, where x and v are elements of $\mathbb{N}$.

The `test&set` object described in Example 2.2 is a RMW object where Q has two states, 0 and 1, and F contains only the constant function $f(x) = 1$.

It is a crucial point of the definition of RMW types that every operation returns the previous value of the object. Thus, a `register` is not a RMW type, since a *write* command returns only the response *ack*.

Herlihy [31] showed that a RMW type has consensus number greater than one if and only if it is equipped with some operation other than the trivial one that reads the state without changing it. Herlihy also gave a necessary (but not sufficient) condition on the set F to describe when RMW objects can be used with `registers` to solve consensus among three processes. This chapter gives a characterization of the RMW types that are capable of solving consensus among n processes, for all positive integers n. The characterization is trivial for $n = 1$, since any type can solve consensus for one process, so it is assumed throughout this chapter that $n \geq 2$.

§4.2 n-discerning RMW Types

Consider a RMW object type `T` with state set Q whose *RMW* operations can apply functions from the set F. Let q_0 be a state in Q. Partition a set of n processes $\{P_1, \ldots, P_n\}$ into two non-empty teams, A and B. Associate a function $f_i \in F$ with each process P_i. The functions f_i need not be distinct. The type `T` is called *n-discerning* if there exist choices for $q_0, A, B, f_1, \ldots, f_n$ so that, in any schedule in which each process P_i applies the single operation RMW_{f_i} to an object X of type `T` which initially has state q_0, every process can determine (from the value returned by its operation) whether a process from team A or a process from team B was the first process to take a step in the schedule. This is illustrated in Figure 4.1 for the case where $n = 3$. Team A contains the processes P and Q, and team B contains only the process R. The figure shows an execution tree that contains all possible executions where three processes each take a single step. Each process can deduce, from the response it receives, which team went first during the execution. Such a type is called n-discerning because processes can easily use an object of the type to discern the difference between the schedules that start with a process on team A from those that begin with a process on team B. This definition is formalized below. The notation $f \circ g$ is used to denote functional composition: $(f \circ g)(x) = f(g(x))$.

Definition 4.3 Let $n \geq 2$. The RMW type `T` defined by the state set Q and the set of functions F is *n-discerning* if there exist

- $q_0 \in Q$,
- a partition of the set of processes $\{P_1, \ldots P_n\}$ into two non-empty teams A and B, and
- a function $f_i \in F$ for each process P_i

such that

I. for all $j \in \{1, \ldots, n\}, Q_{A,j} \cap Q_{B,j} = \emptyset$,

II. for all $P_j \in B, q_0 \notin Q_{A,j}$, and

III. for all $P_j \in A, q_0 \notin Q_{B,j}$, where

$Q_{A,j} = \{(f_{i_\alpha} \circ \cdots \circ f_{i_1})(q_0) : P_{i_1} \in A, \alpha \geq 1$ and $i_1, \ldots, i_\alpha$ are distinct process indices, not including $j\}$, and

$Q_{B,j} = \{(f_{i_\alpha} \circ \cdots \circ f_{i_1})(q_0) : P_{i_1} \in B, \alpha \geq 1$ and $i_1, \ldots, i_\alpha$ are distinct process indices, not including $j\}$.

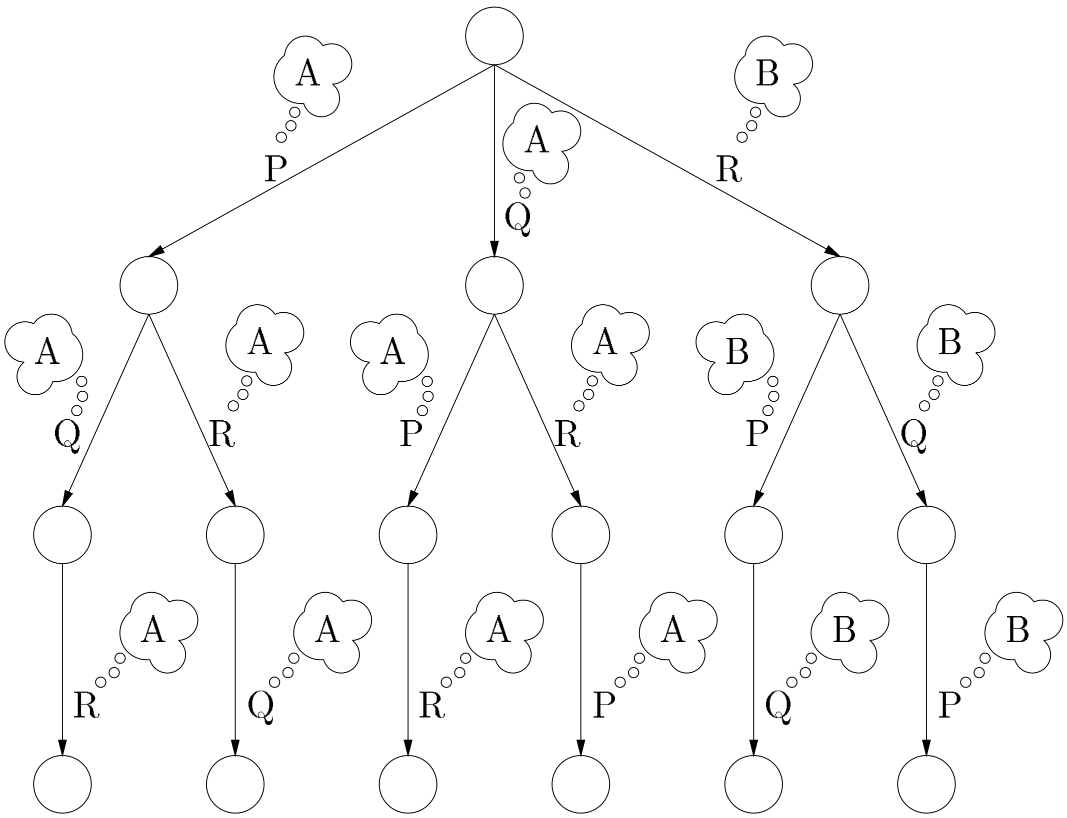

Figure 4.1: Conditions for a 3-discerning RMW type

In this definition, each process P_i is assigned the operation RMW_{f_i}. Suppose all processes apply their assigned operations in some order to an object X of type T initially in state q_0. Consider a process P_j on team A. If P_j is the first process to take a step, it receives the response q_0. The set $Q_{A,j}$ consists of the responses P_j can receive if some other process on team A performs the first step. (The set $Q_{A,j}$ will be empty if P_j is the only process on team A, since any sequence of process indices that starts with the index of a process on team A must include j. It may be the case that q_0 is in $Q_{A,j}$.) The set $Q_{B,j}$ is the set of responses P_j can receive if a process from team B takes the first step. Thus, condition III ensures that P_j can distinguish, using the response it receives, executions in which P_j itself takes the first step from those executions in which a process from team B takes the first step. Condition I ensures that P_j can distinguish executions starting with another process on team A from those in which a process on team B takes the first step. (Similarly, if the process P_j is on team B, conditions I and II guarantee that it can tell which team took the first step.)

It will be shown, in Theorems 4.6 and 4.9, that RMW objects of type `T` can be used with `registers` to solve n-consensus if and only if `T` is n-discerning.

Example 4.4 Consider the `test&set` type defined in Example 2.2. This RMW type is 2-discerning, but not 3-discerning.

To see that the `test&set` type is 2-discerning, let $q_0 = 0$ and divide the processes into the teams $A = \{P_1\}$ and $B = \{P_2\}$. For this type, F contains only the constant function $f(x) = 1$, so this function must be used as f_1 and f_2. Then, $Q_{A,1} = \emptyset, Q_{B,1} = \{1\}, Q_{A,2} = \{1\}$ and $Q_{B,2} = \emptyset$. These sets satisfy the conditions in the definition of a 2-discerning RMW type.

To see that the `test&set` type is not 3-discerning, consider any possible choice of q_0, A, B, f_1, f_2 and f_3. Without loss of generality, suppose the teams are $A = \{P_1, P_2\}$ and $B = \{P_3\}$. Each of f_1, f_2 and f_3 must be the constant function $f(x) = 1$. For any choice of q_0, $Q_{A,1} = Q_{B,1} = \{1\}$, violating condition I of the definition. Thus, the `test&set` type is not 3-discerning.

§4.3 Necessity of the Condition

In this section, it will be shown that any RMW type that can be used to solve consensus among n processes is n-discerning. The proof uses a bivalency argument to show that the behaviour of a consensus protocol just after a critical configuration satisfies the criteria of Definition 4.3.

Lemma 4.5 Consider any critical configuration S_0 of an n-process consensus protocol. If all processes access the same RMW object X of type `T` during their critical steps, then `T` is n-discerning.

Proof: Suppose the process P_i applies the operation RMW_{f_i} to X during its critical step, for $1 \leq i \leq n$. Let q_0 be the state of X in the configuration S_0. Let a be the critical value of one of the processes with respect to S_0. Let A be the set of processes whose critical value with respect to S_0 is a, and let B contain the rest of the processes. Team A is non-empty by construction. Since S_0 is multivalent, team B must also be non-empty.

Condition I of Definition 4.3 must hold for these values of $q_0, A, B, f_1, \ldots, f_n$. Otherwise, let j be a process index such that $Q_{A,j} \cap Q_{B,j}$ is non-empty. Then, X has the same state in two configurations that can be reached from S_0 by sequences of steps in which each process takes at most one step, process P_j takes no steps, and the first steps in the two sequences are taken by processes on opposite teams. These two configurations are indistinguishable to P_j. A solo execution by P_j from either of these configurations would therefore lead to the same decision value, contradicting the definitions of the teams A and B.

Condition II must also hold. Otherwise, let j be the index of a process on team B such that $q_0 \in Q_{A,j}$. Then, some sequence of processes, not including some process $P_j \in B$ and starting with a process on team A, could each take a step to arrive at a configuration that P_j cannot distinguish from S_0. A solo execution of P_j from these two configurations would then lead to the same decision, contradicting the fact that P_j is on team B.

The argument that condition III holds is symmetric. □

This lemma can be combined with a bivalency argument to prove that the conditions for being n-discerning are necessary for solving n-process consensus using RMW objects.

Theorem 4.6 If the n-consensus problem can be solved using objects of a RMW type `T` and `registers`, then `T` is n-discerning.

Proof: Suppose there is some protocol for n-process consensus using objects of RMW type `T` and `registers`. By Lemma 3.6, there is a critical configuration, S_0, of the protocol. By Lemma 3.7, the critical step of each process after S_0 must access the same object, X. This object X must be of type `T`, by Lemma 3.8. It follows from Lemma 4.5 that the type `T` is n-discerning. □

This theorem provides a condition on a RMW type `T` that is necessary for that type to be at level n of the consensus hierarchy or higher. In fact, one can also use this theorem to obtain a necessary condition that applies to any object type at all. Let `T` be an object type with the sequential specification $(Q, \mathrm{OPS}, \mathrm{RES}, \delta)$. One can define a related RMW type `T'` given by the sequential specification $(Q, \mathrm{OPS}, \mathrm{RES}, \delta')$ where $\delta'(q, op) = (q', q)$ whenever $\delta(q, op) = (q', res)$. An object of type `T'` can trivially implement an object of type `T`: to simulate an operation op on the object of type `T`, a process simply applies the same operation to the object of type `T'`.

The object will undergo the same state transition as the simulated object, say from state q to state q'. To determine the result of the simulated operation, the process can simply compute $\delta(q, op)$, since q is returned as the result of the operation on the object of type $\mathtt{T}'$. Thus, $\text{cons}(\mathtt{T}') \geq \text{cons}(\mathtt{T})$. So, for any type $\mathtt{T}$, $cons(\mathtt{T}) \geq n$ only if the related RMW type $\mathtt{T}'$ is n-discerning. A similar (and strictly stronger) general necessary condition will be given when n-discerning readable objects are discussed in Section 5.3.

The condition of the previous paragraph can also be extended to non-deterministic object types. Given a non-deterministic type $\mathtt{N}$ specified by $(Q, \text{OPS}, \text{RES}, \nu)$, one can obtain a strictly stronger deterministic type $\mathtt{T}$ by fixing one of the possible choices that the object can make in response to each action. That is, for each state q and operation op, define $\delta(q, op)$ to be one element of the set $\nu(q, op)$, chosen arbitrarily. Let $\mathtt{T}$ be the type specified by $(Q, \text{OPS}, \text{RES}, \delta)$. The deterministic type $\mathtt{T}$ can trivially implement the non-deterministic type $\mathtt{N}$. Let $\mathtt{T}'$ be the type constructed from $\mathtt{T}$ as in the previous paragraph. Then, if $\mathtt{N}$ can solve consensus among n processes, $\mathtt{T}'$ must be n-discerning.

§4.4 Sufficiency of the Condition

In this section, it is shown that any n-discerning RMW type is at level n of the consensus hierarchy or higher. The properties of n-discerning types are first used to construct a protocol for team consensus among n processes, which can be used to solve the general consensus problem as described in Section 3.4.

Lemma 4.7 Let $\mathtt{T}$ be an n-discerning RMW type. An object of type $\mathtt{T}$ and two `registers` can be used to solve team consensus among n processes.

Proof: Divide the processes into two non-empty teams A and B, assign a function f_i to each process P_i (for $1 \leq i \leq n$), and choose q_0 to satisfy the conditions of Definition 4.3. An algorithm will be constructed for the team$(|A|, |B|)$ problem. The algorithm uses an object X of type $\mathtt{T}$ that initially has state q_0 and two `registers` called R_A and R_B.

Each process P_j first writes its team's common input value into the `register` R_A, if it belongs to team A, or into the `register` R_B, if it belongs to team B. Each process P_j then performs its assigned operation RMW_{f_j} on the object X and uses the result of this operation to determine whether a process from team A or from team B was the first to access X.

Without loss of generality, suppose that process P_j belongs to team A. If P_j's RMW operation returns the response q_0, then a process from team A was the first to access X. To see why this is true, suppose some processes, starting with a process from team B, did access X before P_j. Let $i_1, \ldots, i_\alpha$ be the sequence of indices of these processes, in the order that they accessed X. Then $(f_{i_\alpha} \circ \cdots \circ f_{i_1})(q_0) = q_0$ and $P_{i_1} \in B$, violating condition III. If P_j's RMW operation returns a response different from q_0, the process must be able to deduce which team accessed X first by checking whether the response belongs to $Q_{A,j}$ or $Q_{B,j}$. These two finite sets are disjoint (by condition I), and contain all possible states of the object X that can be observed by process P_j in this protocol. Once P_j decides whether team A or team B accessed X first, it returns the value in R_A or R_B, respectively.

Each process performs only O(1) steps, so wait-free termination is guaranteed. The protocol satisfies the validity condition, since the winning team's value is written into the team's `register` before any process from that team can access the object X. The protocol satisfies the consistency condition, since all processes agree on the winning team and return the value of that team's `register` (which never changes after it is first written, since all processes on the same team have the same input value). □

The main results of this section now follow easily.

Theorem 4.8 The RMW objects $X_2, \ldots, X_n$ can be used, together with $2(n-1)$ `registers`, to solve n-consensus if the type of X_i is i-discerning.

Proof: Immediate from Proposition 3.13 and Lemma 4.7. □

Theorem 4.9 An array of $n-1$ RMW objects of type `T` can be used, together with $2(n-1)$ `registers`, to solve n-consensus if `T` is n-discerning.

Proof: Immediate from Proposition 3.14 and Lemma 4.7. □

Theorems 4.6 and 4.9 can be combined to get the following characterization.

Theorem 4.10 Suppose `T` is a RMW type. Then, cons(`T`) $\geq n$ if and only if `T` is n-discerning.

§4.5 Consequences of the Characterization

The constructive proof of Theorem 4.9 includes an upper bound on the complexity of solving n-consensus using n-discerning RMW objects. Thus, it establishes a kind of "gap": for any RMW type, the consensus problem can either be solved in linear time per process and linear space, or it cannot be solved at all. If, when choosing the teams to satisfy the criteria of Definition 4.3, the two teams can be chosen to be of roughly equal size, then the number of steps per process can be reduced to $O(\log n)$.

Combining the characterization in Theorem 4.10 with Herlihy's universality result (Theorem 3.1) yields a useful corollary.

Corollary 4.11 Objects of a RMW type `T` can be used, with `registers`, to implement every type of object in a system of n processes if and only if `T` is n-discerning.

When finite RMW object types are considered, the characterization gives a decision algorithm for a problem that is undecidable in general (see Section 3.2.2).

Corollary 4.12 If the set of possible states for the RMW objects is finite, the following question is decidable: "Given an integer n and the sequential specification of a RMW type `T`, can n-consensus be solved using objects of type `T` and `registers`?".

Proof: If the state set of the type is finite, then the number of different functions that can be applied by *RMW* operations is also finite. One can check whether the conditions of Definition 4.3 are satisfied for each of the finite number of possible choices of q_0, A, B and $f_1, \ldots, f_n$ in a finite amount of time. □

For every value of n, there is a RMW object type with consensus number exactly n. This can be shown by considering a RMW type that behaves like a `sticky bit` [68] that gets reset after n accesses.

Proposition 4.13 Let $n \geq 2$ be an integer and $Q = \{\bot\} \cup (\{A, B\} \times \{1, \ldots, n-1\})$. Let F be the set $\{f_A, f_B\}$, where, for team $\in \{A, B\}$,

$$
\begin{aligned}
f_{\text{team}}(\bot) &= (\text{team}, 1), \\
f_{\text{team}}(y, z) &= (y, z+1), \text{ for } y \in \{A, B\} \text{ and } 1 \leq z < n-1, \text{ and} \\
f_{\text{team}}(y, n-1) &= \bot, \text{ for } y \in \{A, B\}.
\end{aligned}
$$

Then the RMW type $\mathtt{R}_n$ defined by Q and F is n-discerning, but not $(n+1)$-discerning.

Proof: First, it will be shown that the type $\mathtt{R}_n$ is n-discerning. Let $q_0 = \bot$, $A = \{P_1\}$, $B = \{P_2, \ldots, P_n\}$, $f_1 = f_A$, and $f_2 = \ldots = f_n = f_B$. Then, each element of $Q_{\text{team},j}$ is an ordered pair whose first component is team. The conditions of Definition 4.3 are therefore clearly satisfied.

Next, it will be shown that $\mathtt{R}_n$ is not $(n+1)$-discerning. Consider any choice of q_0, A, B, $f_1, \ldots, f_{n+1}$.

If q_0 is an ordered pair, let z be the second component of q_0. Let P_j be any process. Both $Q_{A,j}$ and $Q_{B,j}$ contain the element $\bot$, since any sequence of $n - z$ functions applied to q_0 will lead to the state $\bot$. This violates condition I in the definition of an $(n+1)$-discerning RMW type.

If $q_0 = \bot$, let P_j be a process in team B. The set $Q_{A,j}$ contains q_0, since any sequence of n functions, when applied to $\bot$ leads to the state $\bot$. This violates condition II of the definition of an $(n+1)$-discerning RMW type. □

The RMW type `compare&swap`, defined in Example 4.2 is at level ∞ of the consensus hierarchy [31], and the trivial RMW type with a single state is at level one of the consensus hierarchy. These facts together with the preceding proposition show that there are RMW types at every level of the consensus hierarchy.

Chapter 5

Readable Objects

THE class of RMW object types, studied in the preceding chapter, contains a wide variety of different types. However, many types are equipped with at least some operations that are not *RMW* operations. For example, `registers` are not RMW objects, since the *write* operation does not return the old state of the `register`. This chapter considers the class of readable objects, which allow operations of a much less restricted form. Instead of requiring that every operation returns complete information about the previous state of the object, the definition of a readable object requires that processes are able to determine the state of the object by *read* operations, which are available in addition to any other operations. The state of an object is often stored explicitly in the shared memory of a system, so that a *read* operation that simply returns this state would be one of the operations with which many object types would naturally be equipped. Thus, readable object types form a large and natural class. This chapter gives a combinatorial characterization of the readable types that are capable of solving n-process consensus, for any n. Since any type can solve consensus for one process, it is assumed throughout this chapter that $n \geq 2$. The characterization and proofs given in this chapter are similar to, though somewhat more complicated than, those given for RMW types in the previous chapter.

§5.1 Readable Object Types

An object type is called readable if processes can read the object's state, without altering it. The *read* operation need not read the entire state in a single atomic action; instead, processes might only be able to read the state piece-by-piece. This is formalized in the following definition.

Definition 5.1 Let T be an object type whose state set is the Cartesian product $Q = \bigtimes_{k \in \Gamma} Q_k$, where Γ is an index set and Q_k is a set for each $k \in \Gamma$. If, for each $k \in \Gamma$, the type T is equipped with an operation *read*(k), which does not change the state of the object and returns the current value of component k of the state, then T is called a *readable object.* Neither Γ nor Q_k need be finite. In addition to the *read* operations, the type T may be equipped with any other collection of operations, called *update* operations.

The `register` type, defined in Definition 2.3 is an example of a readable type. In this case, Γ is a singleton set, since a process can read the entire state of the object using a single operation. This is an example of a type that is readable, but is not a RMW type.

Example 5.2 This example presents an idealized version of the random access memory. Consider a shared object made up of a collection of memory cells, indexed by the natural numbers. Suppose each cell contains a natural number. Then the state set is $Q = \bigtimes_{k \in \mathbb{N}} \mathbb{N}$. The operation *read*($k$) returns the contents of the kth cell. The operation *write*(k, v) stores the natural number v into the kth cell of memory. The operation *indirect-write*(k, v) writes the value v into the cell whose index is stored in cell k. The operation *swap*(k_1, k_2) exchanges the values stored in cells k_1 and k_2. This is a readable type. It is necessary to treat the entire collection of cells as a single object, since a single *swap* operation can change the value of any pair of cells. For this object type, the exact cells that will be accessed by an operation might not be predictable at compile-time.

There are some types that belong to the class of RMW types and to the class of readable types. In fact, any RMW type that is augmented with a *read* operation belongs to both classes, since the *read* operation can be viewed as a RMW operation that applies the identity function.

The `test&set` object type, defined in Example 2.2, is an example of a RMW type that is not readable, since any operation that is performed on the object could change the state of the object. A `stack` is an example of an object type that is neither a RMW type nor a readable type.

§5.2 n-discerning Readable Objects

As for RMW types, a readable type will be called *n-discerning* if n processes divided into two teams can each perform a single update operation on an object and then deduce which team accessed the object first in some simple way without performing further update operations. More formally, a readable type $\mathtt{T}$ is defined to be n-discerning if a set of n processes can be partitioned into two non-empty teams and a single operation can be assigned to each process so that if any group of processes each perform their own operation on an appropriately initialized object X of type $\mathtt{T}$, then each process in the group could determine which team accessed X first, provided that it could see the final state of X. The operation assigned to each process cannot be a *read* operation: it must be possible for the operation to update the state of the readable object. This can be stated precisely as follows.

Definition 5.3 Let $n \geq 2$. The readable type $\mathtt{T}$ is called n-discerning if there exist

- a state $q_0 \in Q$,
- a partition of the set of processes $\{P_1, \ldots, P_n\}$ into two non-empty teams A and B, and
- an update operation op_i, for $1 \leq i \leq n$,

such that

$R_{A,j} \cap R_{B,j} = \emptyset$, for all $j \in \{1, \ldots, n\}$

where $R_{A,j}$ is the set of pairs (r, q) for which there exist distinct process indices $i_1, \ldots, i_\alpha$, including j and with $P_{i_1} \in A$, such that if $P_{i_1}, \ldots, P_{i_\alpha}$ each perform their operations $op_{i_1}, \ldots, op_{i_\alpha}$ (in that order) on an object of type $\mathtt{T}$ that is initially in state q_0, P_j gets the result r, and the object ends in state q. The set $R_{B,j}$ is defined similarly as the set of pairs (r, q) for which there exist distinct process indices $i_1, \ldots, i_\alpha$ including j with $P_{i_1} \in B$ such that if $P_{i_1}, \ldots, P_{i_\alpha}$ each perform their operations (in that order) on an object of type $\mathtt{T}$ that is initially in state q_0, P_j gets the result r, and the object ends in state q.

The conditions of this definition are illustrated in Figure 5.1 for the case where $n = 3$. Here, processes P and Q are on team A, and the process R is on team B. The figure shows three of the possible executions. In each of the executions shown, each process performs a single update operation on an object, and then is able to see the state of the object at some later time (as

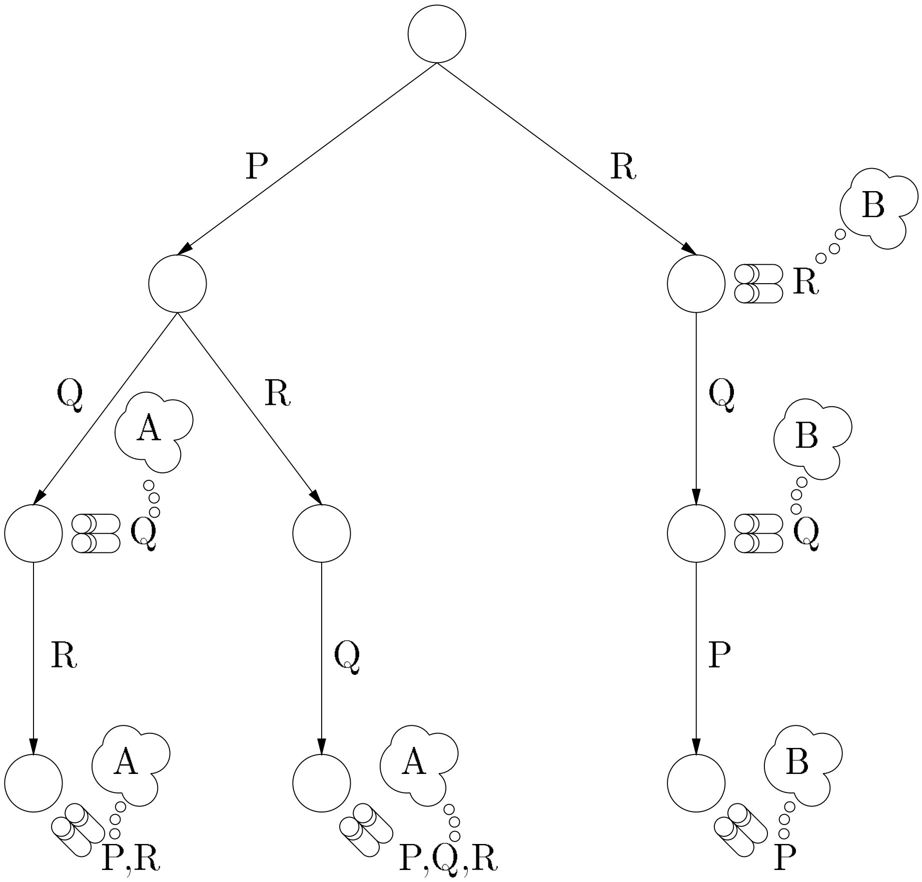

Figure 5.1: Three possible executions for a 3-discerning readable type

indicated by the binoculars). When the process sees the state, it can deduce which team took the first step. For example, consider the leftmost execution shown. Processes P and Q perform their operations on an object. Then process Q observes the state of the object. Using this information and the response to its operation, it has enough information to determine that a process on team A took the first step. Then process R performs its operation. Finally, processes P and R each observe the state of the object and can then conclude that a process from team A took the first step in the execution.

If the state of the readable object has more than one component, the process will not be able to see the entire state of the object by applying a single operation. The condition only states that if the process somehow happened to see the entire state at some point after it has performed the operation, then it would be able to deduce which team went first in the execution.

It will be shown, in Theorems 5.5 and 5.8, that readable objects of type `T` can be used, together with `registers`, to solve the consensus problem for n processes if and only if `T` is n-discerning.

§5.3 Necessity of the Condition

It will be shown in this section that any readable type with consensus number at least n is n-discerning. The following lemma is analogous to Lemma 4.5.

Lemma 5.4 Consider any critical configuration S_0 of an n-process consensus protocol. If all processes access the same readable object X of type `T` during their critical steps, then `T` is n-discerning.

Proof: Let q_0 be the state of X in S_0 and op_i be the operation performed by P_i in its critical step after S_0. As shown in Lemma 3.7, all of these operations must be update operations. Let a be one of the critical values. Let team A be the set of all processes whose critical value is a, and let B be the set of all processes whose critical value is different from a. The two teams are non-empty, since the configuration S_0 is multivalent.

To derive a contradiction, suppose these choices for $q_0, A, B, op_1, \ldots, op_n$ do not satisfy Definition 5.3. Then, there is a pair $(r, q) \in R_{A,j} \cap R_{B,j}$ for some j. There is some sequence $i_1, \ldots, i_\alpha$ of distinct process indices, including j, such that $P_{i_1} \in A$ and if processes $P_{i_1}, \ldots, P_{i_\alpha}$ each perform their next operation, in that order, starting from S_0, process P_j will receive the result r, and the system will end in a configuration S_A where X is in state q. There is some other sequence $k_1, \ldots, k_\beta$ of distinct process indices, including j, such that $P_{k_1} \in B$ and if processes $P_{k_1}, \ldots, P_{k_\beta}$ each perform their next operation, in that order, starting from S_0, process P_j will again receive the result r, and the system will end in a configuration S_B where X is in state q. The configurations S_A and S_B are indistinguishable to P_j, so a solo execution by P_j from either of these two configurations would lead to the same decision value. This contradicts the fact that S_A and S_B are univalent configurations that lead to different decision values. □

Combining this lemma with a bivalency argument yields the following theorem.

Theorem 5.5 If n-process consensus can be solved using any combination of `registers` and objects of a readable type `T`, then `T` is n-discerning.

Proof: Consider an n-process consensus protocol that uses `registers` and objects of type `T`. There is a critical configuration S_0 of the protocol, by Lemma 3.6. Lemmata 3.7 and 3.8 ensure that the critical step of every process after S_0 applies an operation to a single object X, and that X is not a `register`. Hence, by Lemma 5.4, `T` is n-discerning. □

Theorem 5.5 may be used to provide a necessary condition for a type `T` to be able to solve consensus that applies to any `T`, whether it is readable or not. This could be useful for establishing upper bounds on the consensus number of a type. If the type `T` is at level n of the consensus hierarchy or higher, then the set of update operations with which it is equipped must satisfy the definition for n-discerning readable types. This is because the addition of a *read* operation to the specification of type `T` would create a readable type `T′` that is at least as powerful as type `T`, so `T′` must be n-discerning. But the update operations of `T′` are identical to the update operations of `T`, so the update operations of `T` must also satisfy the n-discerning conditions for readable types.

§5.4 Sufficiency of the Condition

The great sin, as I have said, is to assume that something that has been read once has been read forever. —Robertson Davies [23]

The team consensus problem will now be used to show that n-consensus can be solved using n-discerning readable objects and `registers`. This provides a converse to Theorem 5.5.

Lemma 5.6 Let `T` be an n-discerning readable object type. An object of type `T` and two `registers` can be used to solve team consensus among n processes.

Proof: Choose $q_0, A, B, op_1, \ldots, op_n$ to satisfy Definition 5.3.

A protocol will be developed for team consensus among n processes. The protocol will use one `register` for each team and one shared object X of type `T`. Each process P_j writes its team's common input value into its team's `register`. It then applies the operation op_j to X and attempts to read enough of the state of X to determine which team accessed X first.

The state set of $\mathtt{T}$ has the form $Q = \bigtimes_{k\in\Gamma} Q_k$. Since Γ may be an infinite set, it will first be shown that process P_j can determine the winning team from the values of a finite number of the components. Let $R_{A,j}$ and $R_{B,j}$ be the disjoint sets defined in Definition 5.3. These sets are finite, since the number of ways to choose $\alpha, i_1, \ldots, i_\alpha$ in the definitions of $R_{A,j}$ and $R_{B,j}$ is bounded by $n \cdot n!$. For $(r, q) \in R_{A,j}$ and $(r', q') \in R_{B,j}$, let $k(q, q')$ be an element of Γ that indexes some state component where q and q' differ, if such a component exists. Let Δ_j be the set of such indices $k(q, q')$ for all possible choices of (r, q) and (r', q'). The number of such choices is finite, so Δ_j is a finite set. Let π_j be the projection function from Q to the set $\bigtimes_{k\in\Delta_j} Q_k$. This projection function discards all components of the state, except for the finite number of components indexed by the elements of Δ_j.

Suppose the sets $\{(r, \pi_j(q)) : (r, q) \in R_{A,j}\}$ and $\{(r', \pi_j(q')) : (r', q') \in R_{B,j}\}$ have an element in common. Then there are two distinct pairs $(r, q) \in R_{A,j}$ and $(r, q') \in R_{B,j}$ such that $q \neq q'$ and $\pi_j(q) = \pi_j(q')$. This is impossible, since $k(q, q') \in \Delta_j$. Thus, process P_j can discern executions in which team A performed the first update from executions in which team B performed the first update, using only the response to its own update operation and the projection $\pi_j(q)$ of the state q of X at any time after P_j's update has been performed.

After performing its update operation, the process P_j reads, one by one, the components of the state that are indexed by Δ_j. The state of X may be updated by other processes while process P_j is performing this scan of the components. Each scan produces a view of the image of the state of X under the projection π_j. Such a view is called *accurate* if it correctly reflects the state of X at some moment during the execution of the scan. If an update occurs during a scan, the resulting view may not be accurate, but any scan that is not interrupted by an update will produce an accurate view.

To ensure that P_j can correctly determine which team accessed X first, the scan of the components of X is repeated $2n - 1$ times. An update of X can occur during at most $n - 1$ of these scans, so at least n of the scans will return an accurate view of the state of X. By Definition 5.3, P_j can correctly determine which team accessed X first using the information from any accurate scan and the result of its operation op_j. Since a majority of the scans are accurate, P_j can correctly determine which team accessed X first. Process P_j then decides on the value stored in the `register` belonging to the team that accessed X first.

The validity condition for the consensus problem is satisfied, since every process must agree on the team that accessed X first. The consistency condition is also satisfied, since a process from the winning team must have written its value to its team's `register` before accessing X. The protocol is wait-free, since each of the $2n-1$ scans reads only a finite number of components of X. □

The idea of performing repeated scans to obtain a useful view has been used before in the context of shared `registers` [1, 7, 53]. However the criteria for deciding which views are accurate are quite different from the criteria used in the above proof. Version numbers are usually attached to the data so that two consecutive scans that have the same version number are known to be accurate.

The main results of this section follow easily from the preceding lemma and the results on team consensus from Chapter 3.

Theorem 5.7 The readable objects $X_2, \ldots, X_n$ can be used, together with $2(n-1)$ `registers`, to solve n-consensus if the type of X_i is i-discerning.

Proof: Immediate from Proposition 3.13 and Lemma 5.6. □

Theorem 5.8 An array of $n-1$ readable objects of type `T` can be used, together with $2(n-1)$ `registers`, to solve n-consensus if `T` is n-discerning.

Proof: Immediate from Proposition 3.14 and Lemma 5.6. □

Taken together, Theorems 5.5 and 5.8 provide the following characterization.

Theorem 5.9 Suppose `T` is a readable type. Then, cons(`T`) $\geq n$ if and only if `T` is n-discerning.

§5.5 Consequences of the Characterization

The characterization of readable types that can solve n-process consensus provides an upper bound on the space required for consensus protocols. If a readable type `T` has consensus number at least n, then it is possible to solve n-consensus using a tournament algorithm of the special form given in the proof of Proposition 3.13, which uses $n-1$ objects of type T and $2(n-1)$ registers. Unlike the situation for RMW types (as described on page 43), the proofs here do not provide a good general upper bound on the time complexity of solving consensus, since the number of steps required depends on the size of the set Δ_j, which was defined in the proof of Lemma 5.6 and can have exponentially many elements.

A combination of the characterization in Theorem 5.9 and Herlihy's universality result (Theorem 3.1) provides the following corollary.

Corollary 5.10 Readable objects of type T can be used, with registers, to implement any type of object in a system with n processes if and only if T is n-discerning.

The characterization gives a decision algorithm for determining whether a given finite readable type is capable of solving consensus among a given number of processes. This is analogous to Corollary 4.12 for RMW types.

Corollary 5.11 If the state set of object type T and the set of possible operations on object type T are both finite, then the following question is decidable: "Given a positive integer n and a readable type T, can n-consensus be solved using only objects of type T and registers?"

Proof: The conditions of Definition 5.3 can be checked for each of the finite number of choices of $q_0, A, B, op_1, \ldots, op_n$ in a finite amount of time. □

The conditions for being an n-discerning readable type can also be used to generalize a result about atomic snapshot objects. An atomic snapshot object behaves like a finite array of registers with an additional *scan* operation that reads the entire array at once. It is known that the addition of the *scan* operation does not increase the power of an array of registers to solve consensus, since the atomic snapshot object can be implemented from ordinary registers [1, 7, 10]. Here, it will be shown that the addition of an atomic *scan* operation does not increase the power of *any* readable object type to solve consensus.

Corollary 5.12 Let T be a readable type. Let T′ be a type that is the same as T, except that it allows an additional *scan* operation that reads the entire state of T. Then T and T′ have the same consensus number.

Proof: Let n be the consensus number of $\mathtt{T}'$. Clearly, the consensus number of $\mathtt{T}$ is at most n. By Theorem 5.5, $\mathtt{T}'$ is an n-discerning readable type. Let $q_0, op_1, \ldots, op_n, A$ and B be chosen to satisfy Definition 5.3 for $\mathtt{T}'$. None of the operations can be a *scan*, since *scan* operations never update the state of an object. Therefore, the choice of $q_0, op_1, \ldots, op_n, A$ and B will also satisfy the Definition 5.3 for type $\mathtt{T}$. By Theorem 5.8, it is possible to solve n-consensus using objects of type $\mathtt{T}$ and `registers`. □

It can be shown that, for each integer $n > 1$, there is a readable object type that has consensus number n. The type is similar to the RMW type defined in Proposition 4.13. It is equipped with two update operations, *propose*(A) and *propose*(B). If the object begins in the state $\perp$, it "remembers" which of the two update operations was first performed. However if more than n update operations are performed, the object gets reset to the state $\perp$. A more formal definition follows.

Definition 5.13 Let $n > 1$. Let $\mathtt{T}_n$ be an object type with state set $Q = \{\perp\} \cup (\{A, B\} \times \{1, \ldots, n\})$. The object has a *read* operation defined on it, which returns the entire state, as well as two update operations defined as follows.

$$\begin{aligned}
\delta(\perp, \mathit{propose}(\text{team})) &= ((\text{team}, 1), \mathit{ack}) \text{ for team} \in \{A, B\} \\
\delta((\text{team}', z), \mathit{propose}(\text{team})) &= ((\text{team}', z+1), \mathit{ack}), \text{ for team, team}' \in \{A, B\} \text{ and } 1 \leq z < n \\
\delta((\text{team}', n), \mathit{propose}(\text{team})) &= (\perp, \mathit{ack}), \text{ for team, team}' \in \{A, B\}
\end{aligned}$$

Proposition 5.14 For any $n > 1$, the readable type $\mathtt{T}_n$ is n-discerning, but not $(n+1)$-discerning.

Proof: To see that the type $\mathtt{T}_n$ is n-discerning, choose $q_0 = \perp$, $A = \{P_1, \ldots, P_{n-1}\}$, $B = \{P_n\}$, $op_1 = \cdots = op_{n-1} = \mathit{propose}(A)$, and $op_n = \mathit{propose}(B)$. If any sequence of processes take apply their assigned operation to an object that is initialized to the state q_0, the resulting state will be an ordered pair whose first component will be the team of the first process to take a step.

To show that $\mathtt{T}_n$ is not $(n+1)$-discerning, consider any choice of $q_0, A, B, op_1, \ldots, op_{n+1}$. Let X be an object of type $\mathtt{T}_n$ that is initially in state q_0. There are two cases to consider.

First, suppose q_0 is an ordered pair with second component n. Consider three processes: P_i and P_j on one team and process P_k on the opposite team. Without loss of generality, suppose $op_j = \mathit{propose}(A)$. If processes P_i and P_j perform their operations on X, it ends in state $(A, 1)$. If processes P_k and P_j perform their operations on X, it will also end in state $(A, 1)$. Thus, $(\mathit{ack}, (A, 1)) \in R_{A,j} \cap R_{B,j}$, violating the conditions of Definition 5.3

Now suppose q_0 is $\perp$ or an ordered pair whose second component is not n. Let

$$k = \left\{ \begin{array}{ll} n+1 & \text{if } q_0 = \perp, \\ n+1-z & \text{if } q_0 = (y, z). \end{array} \right\}$$

Let $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ be two sequences of k distinct process indices both containing j such that $P_{i_1} \in A$ and $P_{j_1} \in B$. If either sequence of processes each perform their operation on X, in order, then X will end in state $\perp$. Thus $(\mathit{ack}, \perp) \in R_{A,j} \cap R_{B,j}$, violating the conditions of Definition 5.3.

Thus, $\mathtt{T}_n$ is not $(n+1)$-discerning. □

`Registers` are readable objects at level one of the consensus hierarchy. If the `compare&swap` type defined in Example 4.2 is augmented with a *read* operation, it becomes a readable type at level ∞ of the hierarchy. The preceding proposition shows that there are readable objects at every other level of the hierarchy as well.

Chapter 6

Robustness of the Consensus Hierarchy

THE consensus number of a type provides information about the power of a system that has objects of that type, together with `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. Object types which are weak when used by themselves might be more powerful when used together. This issue was first addressed by Jayanti [42, 44] who questioned whether the consensus hierarchy is robust. He defined a wait-free hierarchy to be *robust* if it is impossible to implement any type at some level n from any finite set of types that are each at levels strictly less than n, in a system of n processes. Robustness is a desirable property since it allows one to study the power of a system with several types by reasoning about each of the types individually. The following theorem makes this explicit, using the arguments given by Jayanti [41]. The proof holds even for non-deterministic object types. Recall that a system with objects of type $\mathtt{T}_1 \vee \mathtt{T}_2$ is equivalent in power to a system with objects of types $\mathtt{T}_1$ and $\mathtt{T}_2$, by Proposition 3.3.

Theorem 6.1 The following are equivalent.

(1) The consensus hierarchy is robust.

(2) For any types $\mathtt{T}_1$ and $\mathtt{T}_2$, $\mathrm{cons}(\mathtt{T}_1 \vee \mathtt{T}_2) = \max(\mathrm{cons}(\mathtt{T}_1), \mathrm{cons}(\mathtt{T}_2))$.

(3) For any finite set of types $\{\mathtt{S}_1, \ldots, \mathtt{S}_k\}$, $\mathrm{cons}(\mathtt{S}_1 \vee \cdots \vee \mathtt{S}_k) = \max\limits_{1 \le i \le k} \mathrm{cons}(\mathtt{S}_i)$.

Proof: (1)$\Rightarrow$(2): By Proposition 3.3, $\mathtt{T}_1$ and $\mathtt{T}_2$ can be implemented by $\mathtt{T}_1 \vee \mathtt{T}_2$, so that $\max(\mathrm{cons}(\mathtt{T}_1), \mathrm{cons}(\mathtt{T}_2)) \le \mathrm{cons}(\mathtt{T}_1 \vee \mathtt{T}_2)$, and furthermore the type $\mathtt{T}_1 \vee \mathtt{T}_2$ can be implemented from objects of types $\mathtt{T}_1$ and $\mathtt{T}_2$. If the hierarchy is robust, it follows that $\text{cons}(\mathtt{T}_1 \vee \mathtt{T}_2) \leq \max(\text{cons}(\mathtt{T}_1), \text{cons}(\mathtt{T}_2))$.

(2)⇒(3): Follows easily by mathematical induction.

(3)⇒(1): To prove the contrapositive, assume the hierarchy is not robust. Then there is a type $\mathtt{T}$ at level n of the hierarchy that can be implemented from some types $\mathtt{S}_1, \cdots, \mathtt{S}_k$ such that $\text{cons}(\mathtt{S}_i) < n$ for all i. There is a protocol for consensus among n processes that uses objects of type $\mathtt{T}$ and `registers`. The objects of type $\mathtt{T}$ can be replaced by implementations from the type $\mathtt{S}_1 \vee \cdots \vee \mathtt{S}_k$ (by Proposition 3.3 and mathematical induction), so it follows that $\text{cons}(\mathtt{S}_1 \vee \cdots \vee \mathtt{S}_k) \geq n > \max\limits_{1 \leq i \leq k} \text{cons}(\mathtt{S}_i)$. □

§6.1 Previous Results on Robustness

Since Jayanti first defined robustness, many researchers have studied the robustness of the consensus hierarchy. Whether the hierarchy is robust depends on the class of objects that are considered and the exact nature of the model used. It turns out that, although the hierarchy is not robust in general, there are some classes of objects for which the hierarchy is robust.

There have been a number of non-robustness results proved during the past few years. Chandra *et al.* [17, 18] showed that the consensus hierarchy is not robust in general. Their proof uses non-deterministic, non-oblivious objects with the hard-wired binding model. (See page 17 for a description of binding models.) Moran and Rappoport [64] showed that the hierarchy is not robust even when restricted to deterministic (but non-oblivious) objects, in the restricted hard-wired binding model. Peterson [67] also describes a non-robustness result for this model.

Schenk [77] proved that the consensus hierarchy is not robust for oblivious objects. However, his proof uses objects that can sometimes choose non-deterministically from an infinite number of possible state transitions when an operation is invoked. In addition, the proof uses a slightly different definition of wait-freedom, where the number of steps taken by a process must be bounded, although the bound may depend on the input to the protocol. This definition of wait-freedom can be shown to be equivalent to the standard definition of wait-freedom for objects with bounded non-determinism using König's Lemma [51]. Lo and Hadzilacos [57] improved

Schenk's result by showing that the hierarchy is not robust even when non-determinism is bounded and objects are oblivious.

Although the hierarchy is known not to be robust in general, all of the non-robustness proofs make use of rather unusual types of objects. In particular, all of the proofs use types that are either non-deterministic or non-oblivious. They leave open the possibility that the hierarchy is robust for some large and natural classes of objects.

Borowsky, Gafni and Afek [16] claimed that the consensus hierarchy is robust for all deterministic objects using the soft-wired binding model for non-oblivious objects. Their paper is quite complex, and a full version has not yet appeared.

Jayanti's survey paper [41, Section 5] contains a more detailed discussion of the history of the robustness question.

§6.2 Robustness for RMW and Readable Types

Here, the characterizations of RMW and readable objects that can solve n-process consensus will be used to provide a concise proof of the robustness of the hierarchy when restricted to deterministic RMW and readable objects.

Theorem 6.2 Let $\mathtt{T}$ be a readable or RMW object type. Let $\mathcal{S}$ be a finite set of readable and RMW object types such that $\text{cons}(\mathtt{T}') < \text{cons}(\mathtt{T})$ for each $\mathtt{T}' \in \mathcal{S}$. Then an object of type $\mathtt{T}$ cannot be implemented using objects whose types are from the set $\mathcal{S}$.

Proof: Let $n = \max\{\text{cons}(\mathtt{T}') \mid \mathtt{T}' \in \mathcal{S}\} + 1$. This quantity is finite, since $\text{cons}(\mathtt{T}')$ is less than $\text{cons}(\mathtt{T})$, and therefore finite, for each $\mathtt{T}' \in \mathcal{S}$, and $\mathcal{S}$ is a finite set. To derive a contradiction, assume that $\mathtt{T}$ can be implemented from objects whose types are in $\mathcal{S}$.

Since $\text{cons}(\mathtt{T}) \geq n$, there is a protocol using objects of type $\mathtt{T}$ and `registers` that solves consensus among n processes. The objects of type $\mathtt{T}$ can be simulated by objects whose types are in $\mathcal{S}$ to obtain a protocol for n-process consensus that uses types from the set $\mathcal{S} \cup \{\mathtt{register}\}$. By Lemma 3.6, there is a critical configuration, S_0, of this protocol. All processes access the same object, X, during their critical step after S_0, by Lemma 3.7. Let $\mathtt{T}_X$ be the type of object X. The object X cannot be a `register`, by Lemma 3.8, so $\mathtt{T}_X$ is either a RMW type or a readable type.

First, suppose that $\mathtt{T}_X$ is a RMW type. The type $\mathtt{T}_X$ is an n-discerning RMW type, by Lemma 4.5. It follows from Theorem 4.9 that $\mathrm{cons}(\mathtt{T}_X) \geq n$, contradicting the definition of n.

Now suppose that $\mathtt{T}_X$ is a readable type. The type $\mathtt{T}_X$ is an n-discerning readable type, by Lemma 5.4. By Theorem 5.8, $\mathrm{cons}(\mathtt{T}_X) \geq n$, which again contradicts the definition of n. $\square$

It follows from this theorem that the ability of a system built from a finite set $\mathcal{S}$ of different RMW and readable types to solve consensus is no greater than the ability of the strongest type in $\mathcal{S}$.

Theorems 6.1 and 6.2 allow the decidability results of Corollaries 4.12 and 5.11 to be extended to finite sets of object types. If $\mathcal{S}$ is any finite set of finitely-specified RMW and readable object types, then one can decide whether objects whose types are in $\mathcal{S} \cup \{\mathtt{register}\}$ can be used to solve n-process consensus, by checking whether any of the types in $\mathcal{S}$ are n-discerning.

Chapter 7

Multiple-Access Object Types

THE design of algorithms for asynchronous systems is often simplified when processes can perform more complex operations in a single atomic action. This chapter describes multi-object types and transactional object types which are built from a basic type by allowing processes to perform several of the basic type's operations in a single action. The goal of Chapters 8 and 9 is to determine how much the power of a basic type is increased in these two settings. The concept of a direct implementation, which will be used to prove lower bounds on the consensus numbers of multiple-access types is described in Section 7.3.

§7.1 Definitions

Let $\mathtt{T}$ be a shared object type, and let m be a positive integer. Afek, Merritt and Taubenfeld [4] defined a *multi-object* of type $\mathtt{multi}(\mathtt{T}, m)$ to be a collection of objects of type $\mathtt{T}$, indexed by the natural numbers, on which up to m independent operations can be performed in a single atomic action. (The notation $\mathtt{T}^m$ for $\mathtt{multi}(\mathtt{T}, m)$ has been used previously in the literature.) The type $\mathtt{T}$ is called the *base type* of the multi-object type $\mathtt{multi}(\mathtt{T}, m)$. The objects of type $\mathtt{T}$ that make up the multi-object are called the *base objects*. An operation on the multi-object consists of at most m pairs of the form (op, i), where op is an operation on the base type $\mathtt{T}$, and i is a natural number that indexes one of the base objects. The indices used in an operation on the multi-object must be distinct from one another. The operation on the multi-object applies the operations on all of the specified base objects as a single atomic action. The operation applied to one base object cannot depend on the result of an operation on another base object that is part of the same multi-object operation, since all of the operations on the base objects are specified in advance. The response returned by an operation on the multi-object is simply a tuple of the responses returned to all of the operations on the base objects.

A transactional object type is a more powerful kind of multiple-access type. A transactional object of type $\texttt{trans}(\texttt{T}, m)$ consists of a collection of base objects of type T, indexed by the natural numbers. An operation on the transactional object (called a *transaction*) can be thought of as a block of code where any execution of the code performs at most m operations on the base objects and performs no other shared-memory operations. The response a transaction returns is computed from the responses to all of its shared-memory accesses. More formally, a transaction is a collection of $m+1$ (computable) functions $f_1, \ldots, f_m, R$. Each function f_i maps tuples of $i-1$ responses from objects of type T either to "nil" or to an operation on an object of type T and a natural number. The function R maps tuples of up to m responses from objects of type T to responses from the transactional object. Thus, $f_i(r_1, \ldots, r_{i-1})$ gives the operation to be performed and the index of the base object to be accessed during the ith shared-memory access of the transaction when the responses received from the first $i-1$ shared-memory accesses in the transaction are $r_1, \ldots, r_{i-1}$. If, during some execution of the transaction, the function f_i evaluates to "nil", this indicates that the transaction should terminate after the first $i-1$ shared-memory accesses. The response returned by the transaction is then $R(r_1, \ldots, r_{i-1})$. Although this definition of a transaction permits processes to do arbitrarily complicated internal computations within a transaction, all of the algorithms in this thesis that use transactional types perform very simple computations.

§7.2 Related Work

Herlihy [31] showed that an array of `registers`, where processes are allowed to read individual `registers` or write to m `registers` in a single atomic action, has consensus number $2m-2$, for $m > 1$. Merritt and Taubenfeld [61] showed that n-process consensus can be solved using the multi-object type $\texttt{multi}(\texttt{register}, m)$ in the presence of at most t process failures if and only if $t \leq \max(2m-3, 0)$.

The `multi` operator is an example of an *object combinator*, which is an operator that can be used to construct complex object types from simpler ones. Afek, Merritt and Taubenfeld [4] introduced the notion of object combinators and defined several, including the `multi` operator. They suggested that the study of such combinators would provide insight into the nature of shared-memory systems. Their original paper studied the consensus numbers of multi-objects that use, as their base objects, `queues`, `consensus` objects and `swap` objects. (A `swap` object is equipped with a single operation that writes a new value into the state of the object and returns the old value of the state). They showed that $\text{cons}(\texttt{multi}(\texttt{queue}, 2)) = \infty$, $\text{cons}(\texttt{multi}(n\texttt{-consensus}, m)) = \Theta(n\sqrt{m})$ and $\text{cons}(\texttt{multi}(\texttt{swap}, m)) = \Theta(\sqrt{m})$. They also observed that any multi-object built from a commutative base type has consensus number at most two. (An object type is *commutative* if, for any pair of operations and any possible state of the object, the state that results from applying the two operations does not depend on the order in which they are applied.) One example of a commutative type is the `test&set` type defined in Example 2.2.

Jayanti and Khanna [46] defined a different kind of multi-object, which can be denoted by $\texttt{multi}(\texttt{T}, \infty)$, where *any* finite number of base objects of type `T` can be accessed in a single atomic action. They showed that, for any type `T`, $\text{cons}(\texttt{multi}(\texttt{T}, \infty)) \in \{1, 2, \infty\}$. Their proof used the result that $\text{cons}(\texttt{multi}(3\texttt{-consensus}, m)) = \Omega(\sqrt{m})$ [4] to show that $\text{cons}(\texttt{multi}(\texttt{T}, m)) = \Omega(\sqrt{m})$ for any base type `T` with consensus number greater than two. It follows that $\text{cons}(\texttt{multi}(\texttt{T}, \infty)) = \infty$ whenever $\text{cons}(\texttt{T}) > 2$, since $\text{cons}(\texttt{multi}(\texttt{T}, \infty))$ is equal to $\lim_{m \to \infty} \text{cons}(\texttt{multi}(\texttt{T}, m))$. Anderson and Moir [9] described an implementation of multi-objects from object types, such as `compare&swap`, which have consensus number ∞.

A number of researchers have studied transactional objects for particular base types. Most of the research has focused on implementing versions of transactional objects from universal object types such as `compare&swap` and `LL/SC`. The `LL/SC` type is equipped with two operations: the *load-linked* operation, *LL*, reads the value stored in the object, and the *store-conditional* operation, $SC(v)$, operation updates it to a new value v. However, a *SC* operation performed by a process succeeds if and only if the object has not been updated since the last *LL* applied to the object by that process. Israeli and Rappoport [40] described a variety of implementations of versions of `LL/SC` and `compare&swap` objects where the *SC* and *compare&swap* operations are generalized to access several objects in a single atomic action. Shavit and Touitou [79]

Base Type `T`	`T` Readable?	cons(`T`)	cons(multi(`T`, m)) for $m > 1$		cons(trans(`T`, m)) for $m > 1$	
`trivial`	Yes	1	1	[46]	1	
`register`	Yes	1	$2m - 2$	[31]	∞	Theorem 9.1
`toggle`	Yes	1	∞	Proposition 8.7	∞	
any commutative `T` with cons(`T`) $= 2$	Maybe	2	2	[4]	$\Omega(2^{m/2})$	Theorem 9.7
`test&set`	No	2	2	[4]	∞ $(m \geq 3)$	Proposition 9.5
`swap`	No	2	$\Theta(\sqrt{m})$	[4]	∞	Proposition 9.6
`queue`	No	2	∞	[4]	∞	
any non-trivial `T`	Maybe	n	$\Omega(n\sqrt{m})$ $(n \geq 3)$	Theorem 8.1	$\Omega(2^{m/2})$	Theorem 9.7
n-**`consensus`**	No	n	$\Theta(n\sqrt{m})$ $(n \geq 3)$	[4]	$\Omega(n^{m-1})$	Proposition 9.2
any readable `T`	Yes	n	$\Omega(nm)$ $(n \geq 3)$	Theorem 8.3	∞ (`T` non-trivial)	Theorem 9.1
$\mathtt{T}_n$	Yes	n	$\Theta(nm)$ $(n \geq 3)$	Proposition 8.5	∞ $(n \geq 2)$	Theorem 9.1
`toggle` $\vee$ $\mathtt{T}_n$	Yes	n	∞	Proposition 8.7	∞	
any RMW `T`	Maybe	n	$\Omega(n\sqrt{m})$ $(n \geq 3)$	Theorem 8.1	$\Omega(2^m)$ $(n \geq 2)$	Corollary 9.4

Table 7.1: Summary of results about multiple-access objects

described how to use LL/SC objects for a non-blocking implementation of atomic transactions on registers. (A non-blocking implementation [32] has a weaker notion of fault-tolerance than wait-freedom: there can never be an infinite execution in which no operation is completed.) Moir [63] gave a similar wait-free implementation. Attiya and Dagan [11] discussed a non-blocking implementation of operations that can access two base objects in a single action, using LL/SC objects as their primitive. They also showed that *LL* and *SC* operations can be used to solve a problem more efficiently if processes may perform operations on more than one base object in an atomic action.

Table 7.1 summarizes known results about the consensus numbers of multiple-access types, including the results in this thesis.

§7.3 Direct Implementations

Afek, Merritt and Taubenfeld [4] introduced the notion of direct implementations as a tool for studying consensus numbers of multi-objects. Type $\mathtt{T}_1$ *directly implements* type $\mathtt{T}_2$ if an implementation of $\mathtt{T}_2$ can be constructed from objects of type $\mathtt{T}_1$ and registers so that each simulated operation of $\mathtt{T}_2$ can be linearized at the moment the simulation first accesses an object of type $\mathtt{T}_1$. (It is implicit in this definition that every simulated operation contains at least one access to an object of type $\mathtt{T}_1$.) The following lemma is an easy consequence of the definition of direct implementations.

Lemma 7.1 [4] If type $\mathtt{T}_1$ directly implements type $\mathtt{T}_2$, then $\mathtt{multi}(\mathtt{T}_1, m)$ directly implements $\mathtt{multi}(\mathtt{T}_2, m)$ for any positive integer m.

Proof: Assume that the direct implementation of $\mathtt{T}_2$ from $\mathtt{T}_1$ uses objects of type $\mathtt{T}_1$ called $X_0, X_1, \ldots$ and registers called $Y_0, Y_1, \ldots$. Suppose registers $R_0, R_1, \ldots$, and an object O of type $\mathtt{multi}(\mathtt{T}_1, m)$ are available. A direct implementation of an object of type $\mathtt{multi}(\mathtt{T}_2, m)$ will be described.

Let τ be any computable bijection from $\mathbb{N}^2$ to $\mathbb{N}$. (For example, one could use the function $\tau(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$.) Suppose a process P must simulate an operation on a $\mathtt{multi}(\mathtt{T}_2, m)$ object that consists of the operations $op_1, \ldots, op_r$ on the base objects with indices $i_1, \ldots, i_r$, respectively, where $1 \leq r \leq m$. Process P simulates op_j on the i_jth base object using the direct implementation of $\mathtt{T}_2$ from $\mathtt{T}_1$, replacing each access to an object X_k by an access to the base object of O indexed by $\tau(i_j, k)$ and replacing each access to `register` Y_k by an access to the `register` $R_{\tau(i_j,k)}$. To ensure that all r of the simulated operations on the base objects can be linearized at the same moment, the steps of these simulated operations are executed by P in the following order. First, P executes consecutively all of the simulations up to, but not including, the first access to an object of type $\mathtt{T}_1$. It then performs the next step of every simulated operation in a single atomic operation on O. The simulated operation on the object of type $\mathtt{multi}(\mathtt{T}_2, m)$ can be linearized at the moment this (first) operation on O is performed. Finally, P completes the executions of every simulation one by one. □

In some of the proofs, it will be useful to talk about team consensus *objects*, which are related to the team consensus problem that was defined in Section 3.4. The object type $\mathtt{team}(n_1, n_2)$ is equipped with operations of the form *propose*(i, value). The value argument is a natural number. All *propose* operations performed on an object must have distinct arguments i in the range $1 \leq i \leq n_1 + n_2$. Furthermore, all *propose* operations with $i \leq n_1$ must have the same value, and all *propose* operations with $i > n_1$ must have the same value. If these conditions are met, the object returns, as the result of each operation, the first value proposed to it. Processes can solve the team(n_1, n_2) problem with a $\mathtt{team}(n_1, n_2)$ object: each process is assigned a distinct index i that corresponds to its team, and then executes a single *propose* operation to solve the problem.

The following lemma will be useful for proving lower bounds on the consensus numbers of multi-objects. The proof uses a bivalency argument to convert a consensus algorithm into a direct implementation of a team consensus object.

Lemma 7.2 Let $n > 1$. Suppose objects of type $\mathtt{T}$ can be used, with `registers`, to solve consensus among n processes. Then, objects of type $\mathtt{T}$ can directly implement a $\mathtt{team}(n_1, n_2)$ object for some positive integers n_1 and n_2 with $n_1 + n_2 = n$.

Proof: Consider a consensus protocol Π for n processes that uses objects of type $\mathtt{T}$ and `registers`. By Lemmata 3.6 and 3.7, the protocol has a critical configuration, after which the critical step of each process is applied to the same object, say X. The object X must be of type `T`, by Lemma 3.8. Let a be the critical value of some process. Let n_1 be the number of processes that have a as their critical values and let $n_2 = n - n_1$.

Divide the processes of protocol Π into two teams. The n_1 processes with critical value a form one team and the remaining n_2 processes form the other team. The direct implementation of the `team`(n_1, n_2) object will use one `register` for each team, in addition to the objects used in the protocol Π, which are initialized to the states they have in the critical configuration of Π. When a processor P simulates a *propose*(i, val) operation on the `team`(n_1, n_2) object, it first writes its input value into the `register` of the team containing the ith process of Π. It then simulates the ith process of protocol Π, starting from the critical configuration. If the simulated process decides a when it completes protocol Π, P returns the value in the first team's `register`. Otherwise, P returns the value in the second team's `register`. This ensures that every simulated operation on the `team`(n_1, n_2) object returns the value proposed by the first process to access X, which is the first object of type `T` accessed by each process. Hence, this is a direct implementation of the `team`(n_1, n_2) object. $\square$

Chapter 8

Multi-objects

THIS chapter studies the consensus number of the type $\texttt{multi}(\texttt{T}, m)$ as a function of m and the consensus number of the base type T. The first result gives a lower bound on the consensus number of a multi-object built from any base type at level three of the consensus hierarchy or higher. A stronger bound is given for the class of readable base types in Section 8.2. Finally, Section 8.3 explains why one cannot provide similar upper bounds on the consensus numbers of multi-objects.

§8.1 A General Lower Bound

In this section, it will be shown that, for any base type T whose consensus number n is at least three, the consensus number of $\texttt{multi}(\texttt{T}, m)$ is $\Omega(n\sqrt{m})$. Thus, the ability to perform m independent operations in a single atomic action boosts the consensus number of a type by a factor of $\Omega(\sqrt{m})$. This general lower bound cannot be strengthened since the consensus number of $\texttt{multi}(n - \texttt{consensus}, m)$ is $\Theta(n\sqrt{m})$ [4]. However, a stronger lower bound will be proved for the class of readable base types T in the next section. The general lower bound is proved only for base types at level three of the consensus hierarchy or higher. It cannot be extended to base types at levels one and two, since $\text{cons}(\texttt{multi}(\texttt{trivial}, m)) = 1$ [47] and $\text{cons}(\texttt{multi}(\texttt{test\&set}, m)) = 2$ [4] for all positive integers m.

Theorem 8.1 For any type T with consensus number $n > 2$, $\text{cons}(\texttt{multi}(\texttt{T}, m)) = \Omega(n\sqrt{m})$.

Proof: By Lemma 7.2, T can directly implement a $\texttt{team}(n_1, n_2)$ object for some positive integers n_1 and n_2 that sum to n. Without loss of generality, assume that $n_1 \geq n_2$. By

Lemma 7.1, $\texttt{multi}(\texttt{T}, m)$ can directly implement a multi-object of type $\texttt{multi}(\texttt{team}(n_1, n_2), m)$. Thus, the consensus number of $\texttt{multi}(\texttt{T}, m)$ is at least as large as the consensus number of $\texttt{multi}(\texttt{team}(n_1, n_2), m)$.

Let $g = \lfloor\sqrt{m}\rfloor$ and $N = g \cdot \lfloor n_1/2\rfloor + n_2$. A multi-object of type $\texttt{multi}(\texttt{team}(n_1, n_2), m)$ will be used to solve the team$(g \cdot \lfloor n_1/2\rfloor, n_2)$ problem. Both teams will be non-empty, since $n_1 \geq \lceil n/2\rceil \geq \lceil 3/2\rceil$ and $n_2 \geq 1$. It will then follow from Lemma 3.12 that the type $\texttt{multi}(\texttt{team}(n_1, n_2), m)$ (and therefore $\texttt{multi}(\texttt{T}, m)$) has consensus number at least $N = \Omega(n\sqrt{m})$. The algorithm will be similar to the one used by Afek, Merritt and Taubenfeld [4] to prove that $\text{cons}(\texttt{multi}(n\text{-}\texttt{consensus}, m)) = \Theta(n\sqrt{m})$.

The algorithm will use a multi-object of type $\texttt{multi}(\texttt{team}(n_1, n_2), m)$ with g^2 base objects, arranged into a two-dimensional $g \times g$ array. Since $g^2 \leq m$, processes may access any number of the base objects in the array in a single atomic action. The algorithm is given in pseudocode in Figure 8.1.

Divide the N processes into two teams, A and B, containing $g \cdot \lfloor n_1/2\rfloor$ and n_2 processes, respectively. Divide the processes of team A into g groups, $A_1, \ldots, A_g$, each of size $\lfloor n_1/2\rfloor$. Suppose the inputs of processes on teams A and B are a and b, respectively. The algorithm will ensure that every process decides on the value of the first process to access the multi-object. Throughout the algorithm, each process will only propose its team's input value to base objects in the array. It will be shown below that the algorithm ensures that processes access each base object correctly: the *propose* operations performed by processes on team A use distinct numbers between 1 and n_1 as their first arguments, and *propose* operations performed by processes on team B use distinct numbers greater than n_1 as their first arguments.

The array of base objects is shown in Figure 8.2 for the case where $m = 16$. Each of the sixteen rectangles represents a $\texttt{team}(n_1, n_2)$ object, which is directly implemented from an object of type $\texttt{T}$.

The ith process of team B performs the operation $\textit{propose}(n_1 + i, b)$ to all base objects in a single operation. The base objects accessed by each process on team B are indicated by light shading in Figure 8.2. Only the right portion of each base object is shaded lightly to indicate that the *propose* operations executed by processes on team B use first arguments in the range

```
shared variable
    object of type multi(team(n_1, n_2), m) whose base objects are
        arranged into an array C[1..g, 1..g]

% Algorithm for ith process in group A_j
local variables
    r[1..g] : array of integers % responses from accessing multi-object
    out : integer % output value
for 1 ≤ y ≤ g do atomically
    r[y] ← result of performing propose(i, a) on C[j, y]
end for
out ← a
for y ← 1..g
    if r[y] ≠ a then out ← r[y]
end for
if out ≠ a then
    for 1 ≤ k ≤ g do atomically
        r[k] ← result of performing propose(⌊n_1/2⌋ + i, a) on C[k, j]
    end for
    for k ← 1..g
        if r[k] = a then out ← a
    end for
end if
halt with output out


% Algorithm for ith process of team B
local variables
    r[1..g, 1..g] : array of integers % responses from accessing multi-object
    out : integer % output value
for 1 ≤ x ≤ g, 1 ≤ y ≤ g do atomically
    r[x, y] ← result of performing propose(n_1 + i, b) on C[x, y]
end for
out ← b
for x ← 1..g
    for y ← 1..g
        if r[x, y] ≠ b then out ← r[x, y]
    end for
end for
halt with output out
```

Figure 8.1: Pseudocode for Theorem 8.1

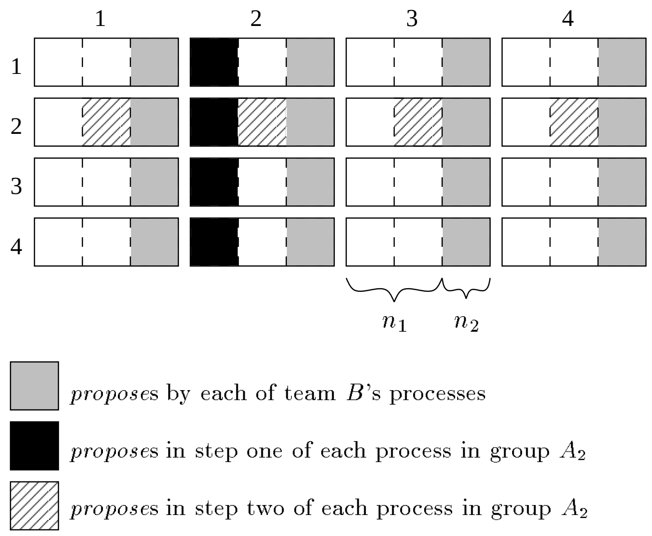

Figure 8.2: Accesses to the multi-object

n_1+1 to n_1+n_2. If every object returns b, this means that either $a=b$ or a process from team B accessed the multi-object first, so the process outputs b. Otherwise, some base object returns $a \neq b$. In this case, the process outputs a, since some process from team A must have accessed the multi-object first.

The ith process, P, of group A_j first performs the operation *propose*(i,a) to all objects in column j of the array in a single atomic step. If some base object returns a value b that is different from a, process P then performs the operation *propose*$(\lfloor n_1/2 \rfloor + i, a)$ on all of the base objects in row j as a second atomic action. The operations in this second step will not affect the output of any of the base objects, since P performs this second step only if all base objects in the array have been previously accessed by a process on team B. The base objects accessed in the first step of each process in group A_2 of team A are indicated by dark shading in Figure 8.2. The left portion of each of these base objects is shaded to indicate that these *propose* operations use first arguments in the range 1 to $\lfloor n_1/2 \rfloor$. The base objects accessed during the second step of each process in group A_2, if it is necessary, are indicated by stripes in Figure 8.2. The middle portion of each of these base objects is striped to indicate that these *propose* operations use first arguments in the range $\lfloor n_1/2 \rfloor + 1$ to n_1.

If every object accessed by the first step of P returns a, then either $b=a$ or a process from group A_j accessed the multi-object before any process from team B, so process P outputs a. Otherwise, it must be the case that a process from team B accessed the multi-object before any process in group A_j and $a \neq b$. Process P then uses the results of its second step to determine whether some process from another group of team A accessed the multi-object first. If every one of the base objects in row j returns b, then a process from team B must have accessed the multi-object first, so process P outputs b. Otherwise, the base object in some column, k, of row j returns a. This means that some process in group A_k accessed the multi-object before any process on team B, so P outputs a.

It will now be verified that this algorithm accesses each base object correctly: no two processes ever perform *propose* operations on any base object using the same first argument. Consider the base object in column k and row j. The $\lfloor n_1/2 \rfloor$ processes from group A_k access it using distinct first arguments in the range 1 to $\lfloor n_1/2 \rfloor$. The $\lfloor n_1/2 \rfloor$ processes from group A_j access it using distinct arguments in the range $\lfloor n_1/2 \rfloor + 1$ to $2 \lfloor n_1/2 \rfloor$. The n_2 processes from team B access it using distinct arguments in the range $n_1 + 1$ to $n_1 + n_2$. No other processes access the base object. Thus, the base objects must always provide the correct solution to the team(n_1, n_2) problem. It follows that the algorithm using the multi-object correctly solves the team$(g \cdot \lfloor n_1/2 \rfloor, n_2)$ problem. $\square$

It will now be shown that the lower bound for the consensus number of $\texttt{multi}(\texttt{T}, m)$ given in the previous proof exceeds the consensus number of the base type $\texttt{T}$ when m is at least sixteen.

Proposition 8.2 Let $\texttt{T}$ be an object type with $2 < \text{cons}(\texttt{T}) < \infty$. Objects of type $\texttt{T}$ and $\texttt{registers}$ cannot be used to implement the type $\texttt{multi}(\texttt{T}, 16)$ (in a system of more than $\text{cons}(\texttt{T})$ processes).

Proof: It was shown, in the proof of Theorem 8.1, that objects of type $\texttt{multi}(\texttt{T}, m)$, together with $\texttt{registers}$, can solve consensus among $\lfloor \sqrt{m} \rfloor \cdot \lfloor n_1/2 \rfloor + n_2$ processes, for some positive integers n_1 and n_2 with $n_1 \geq n_2$ and $n_1 + n_2 = \text{cons}(\texttt{T})$. If $n_1 = 2$, then $\text{cons}(\texttt{multi}(\texttt{T}, 16)) \geq 4 + n_2 > \text{cons}(\texttt{T})$. If $n_1 > 2$, then $\text{cons}(\texttt{multi}(\texttt{T}, 16)) \geq 4 \cdot \lfloor n_1/2 \rfloor + n_2 \geq 2(n_1 - 1) + n_2 = \text{cons}(\texttt{T}) + n_1 - 2 > \text{cons}(\texttt{T})$. $\square$

§8.2 Readable Multi-objects

In this section, a better lower bound on the consensus numbers of multi-objects is obtained for the large class of readable object types.

The proof is similar to the proof of Theorem 8.1. First, the main difference between the two proofs is described. The proof of Theorem 8.1 used a multi-object that consisted of a $\lfloor\sqrt{m}\rfloor \times \lfloor\sqrt{m}\rfloor$ array of base objects. All of the elements in column k of the array were used to determine whether a process from group A_k or a process from team B accessed the object first. The $\lfloor\sqrt{m}\rfloor$ base objects in the column were required to ensure that all processes could access this information. When the objects are readable, however, it will be shown that all processes can access the information from a single base object. This allows the square array of base objects to be replaced by a one-dimensional array consisting of a single row. The width of the array can then be increased from $\lfloor\sqrt{m}\rfloor$ to m, while still allowing processes on team B to access every base object in a single operation. This leads to the following improved bound for readable multi-objects.

Theorem 8.3 For any readable type $\mathtt{T}$ with consensus number $n > 2$, cons($\mathtt{multi}(\mathtt{T}, m)$) is $\Omega(nm)$.

Proof: By Theorem 5.5, type $\mathtt{T}$ is n-discerning, as described in Definition 5.3. Choose $A, B, op_1, \ldots, op_n$ and q_0 to satisfy the definition of an n-discerning readable type. Without loss of generality, assume that the processes are labelled so that $A = \{P_1, \ldots, P_{|A|}\}$ and $B = \{P_{|A|+1}, \ldots, P_n\}$, where team A has at least as many processes as team B. Hence, $|A| \geq \lceil n/2 \rceil \geq 2$.

Let $N = m(|A|-1)+|B|$. Since $|A| \geq n/2$, N is $\Omega(nm)$. It will be shown that $\mathtt{multi}(\mathtt{T}, m)$, which is also a readable type, is N-discerning. It will then follow from Theorem 5.8 that cons($\mathtt{multi}(\mathtt{T}, m)$) $\geq N$. Primes will be used to distinguish the variables in the definition of N-discerning for type $\mathtt{multi}(\mathtt{T}, m)$ from the variables used in the definition of n-discerning for type $\mathtt{T}$.

Values for $A', B', op'_1, \ldots, op'_N$ and q'_0 must be chosen so that each process P'_j can tell, from the result of applying op'_j, together with the state of the multi-object at any later time, which team accessed the multi-object first. Let the team A' consist of $m(|A|-1)$ processes, and let B' consist of the remaining $|B|$ processes. Both teams are non-empty, since $|A'| = m(|A|-1) \geq m(\lceil n/2 \rceil - 1) \geq m$ and $|B'| = |B| > 0$. Divide the processes on team A' into m groups, $A'_1, \ldots, A'_m$, each of size $|A|-1$.

Only m base objects, $O_1, \ldots, O_m$, will be used. In the initial state q'_0 of the multi-object, each of the base objects is in state q_0. Each process of team B' accesses each of the m base objects in the same way that the corresponding process on team B accesses an object of type T when determining whether team A or B accessed it first. Each process in group A'_i of team A' accesses only the base object O_i. It does so in the same way that the corresponding process on team A accesses an object of type T when determining whether team A or team B accessed it first. The base object O_i is used to determine whether a process from group A'_i accessed the multi-object before any process from team B'. It will be shown below that this information can be accessed by all processes (and not only those in $A'_i \cup B'$, which are actually performing updates to O_i) because only $n-1$ processes update the object O_i. Processes can determine which team accessed the multi-object first because they can determine whether or not a process from group A'_i accessed it before a process from team B', for each i.

The assignment of operations to processes will now be specified more precisely. The jth process of group A'_i performs the operation op_{j+1} on the base object O_i. The jth process of team B' performs the operation $op_{j+|A|}$ on all m of the base objects. The possible sequences of updates that can be performed on the base object O_i are exactly those that can be performed on an object of type T when some subset of processes from $\{P_2, \ldots, P_n\}$ perform their update operations as described in the definition of n-discerning.

The fact that T is n-discerning will now be used to show that these choices of teams, initial state and operations satisfy the definition of N-discerning for $\texttt{multi}(\texttt{T}, m)$. To derive a contradiction, suppose the type $\texttt{multi}(\texttt{T}, m)$ is not N-discerning. Then, for some j, there is a pair $(r', q') \in R'_{A',j} \cap R'_{B',j}$. This implies that there are two sequences of distinct processes, one beginning with a process in some group A'_i of team A' and the other beginning with a process of team B', such that the final state of the multi-object is q' if either sequence of processes perform their update operations on the multi-object. Let $op_{k_1}, \ldots, op_{k_\alpha}$ and $op_{l_1}, \ldots, op_{l_\beta}$ be the sequences of operations that are performed on O_i by these two sequences of updates to the multi-object. Both of these sequences leave object O_i in the same state. Both $k_1, \ldots, k_\alpha$ and $l_1, \ldots, l_\beta$ are sequences of distinct indices chosen from the set $\{2, \ldots, n\}$ with $P_{k_1} \in A$ and $P_{l_1} \in B$. Now consider the two sequences of update operations obtained by adding a final operation, performed by P_1. The two sequences $op_{k_1}, \ldots, op_{k_\alpha}, op_1$ and $op_{l_1}, \ldots, op_{l_\beta}, op_1$, when performed on an object of type $\mathtt{T}$ initialized to the state q_0, will both leave the object in the same state and return the same response to the operation op_1. This implies that $R_{A,1} \cap R_{B,1}$ is non-empty, contradicting the fact that $\mathtt{T}$ is n-discerning. Thus, type $\mathtt{multi}(\mathtt{T}, m)$ must be N-discerning. $\square$

The constant in Proposition 8.2 can be improved for the class of readable objects, as described in the following proposition.

Proposition 8.4 Let $\mathtt{T}$ be a readable object type with $2 < \text{cons}(\mathtt{T}) < \infty$. Objects of type $\mathtt{T}$ and **registers** cannot be used to implement $\mathtt{multi}(\mathtt{T}, 3)$ (in a system of more than $\text{cons}(\mathtt{T})$ processes).

Proof: Let n be the consensus number of type $\mathtt{T}$. It was shown in the proof of Theorem 8.3 that objects of type $\mathtt{multi}(\mathtt{T}, m)$, together with **registers**, can solve consensus among $N = m(|A|-1) + |B| = (m-1)(|A|-1) + n - 1 \geq (m-1)(\lceil n/2 \rceil - 1) + n - 1$ processes. Thus, $\text{cons}(\mathtt{multi}(\mathtt{T}, 3)) \geq n + 2 \cdot \lceil n/2 \rceil - 3 > n$, since $\lceil n/2 \rceil > 3/2$ when $n > 2$. $\square$

The following proposition shows that the lower bound on $\text{cons}(\mathtt{multi}(\mathtt{T}, m))$ for readable types $\mathtt{T}$ given in Theorem 8.3 cannot be improved. Recall the readable object type $\mathtt{T}_n$ defined in Definition 5.13. It behaves like a readable version of a binary consensus object that gets reset to the state $\perp$ once every $n+1$ operations. It was shown in Proposition 5.14 that the consensus number of $\mathtt{T}_n$ is n.

Proposition 8.5 Let $n > 1$. For the readable object type $\mathtt{T}_n$, $\text{cons}(\text{multi}(\mathtt{T}_n, m)) = \Theta(nm)$.

Proof: The lower bound $\text{cons}(\text{multi}(\mathtt{T}_n, m)) = \Omega(nm)$ follows from Theorem 8.3. Here, it is shown that $\text{cons}(\text{multi}(\mathtt{T}_n, m)) = \text{O}(nm)$.

Suppose the type multi($\mathtt{T}_n, m$) can be used to solve consensus among N processes. It will be shown that N is at most $2m(n+1)$. By Theorem 5.5, multi($\mathtt{T}_n, m$) is an N-discerning readable type. Choose $A, B, op_1, \ldots, op_N$ and q_0 to satisfy the definition of N-discerning for multi($\mathtt{T}_n, m$).

Let P_i and P_j be any two processes on teams A and B, respectively. Let $\mathcal{O}_{ij}$ be the set of base objects of the multi-object that are updated by both op_i and op_j. First, it will be shown that there is some base object in $\mathcal{O}_{ij}$ that is updated by at most $n+2$ of the operations $op_1, \ldots, op_N$.

To derive a contradiction, suppose each base object in $\mathcal{O}_{ij}$ is updated by at least $n+3$ of the operations. Consider the following two executions of the operations $op_1, \ldots, op_N$ on a multi-object initially in state q_0. In the first execution, op_i is performed first, op_j is performed second, and then the remaining $N-2$ operations are performed one at a time in some fixed order. In the second execution, op_j is performed first, op_i is performed second, and then the remaining $N-2$ operations are performed in the same order as in the first execution.

Let O be one of the base objects of the multi-object. Let s_1 be the state of O just before the final operation of the first execution is performed. Let s_2 be the state of O just before the final operation of the second execution is performed. If $O \notin \mathcal{O}_{ij}$ then $s_1 = s_2$: the relative order of op_i and op_j is unimportant, since at most one of them updates O. Now, consider the case where $O \in \mathcal{O}_{ij}$. Both executions perform the same number of operations on O. Therefore $s_1 = \perp$ if and only if $s_2 = \perp$. Furthermore, if s_1 and s_2 are ordered pairs, then their second components must be equal. In the latter case s_1 is also equal to s_2, since the first component of O depends only on the last n update operations performed on O, and it follows from the assumption that at least n operations are performed after op_i and op_j but before the last operation in both executions.

Thus, the last process to take a step will receive the same response from the multi-object in both executions and the multi-object will be left in the same state at the end of the two executions. This contradicts the fact that $A, B, op_1, \ldots, op_N$ and q_0 were chosen to satisfy the definition of N-discerning. So, it must be the case that some base object in $\mathcal{O}_{ij}$ is updated by at most $n+2$ of the operations $op_1, \ldots, op_N$.

Let P_i be any process on team A. Consider the set $\mathcal{S}$ of base objects that are updated by op_i and by at most $n+1$ of the other operations $op_1, \ldots, op_{i-1}, op_{i+1}, \ldots, op_N$. Since op_i updates at most m base objects, $|\mathcal{S}| \leq m$. It was shown that $\mathcal{O}_{ij} \cap \mathcal{S} \neq \emptyset$ for any $P_j \in B$, so every process on team B updates one of the base objects in $\mathcal{S}$. Thus, $|B| \leq |\mathcal{S}|(n+1) \leq m(n+1)$. Similarly, $|A| \leq m(n+1)$. Therefore, the total number of processes is $N = |A|+|B| \leq 2m(n+1) = \mathrm{O}(mn)$.

□

§8.3 The Toggle Object

In this section, it is shown that a general *upper* bound for the consensus numbers of multi-objects cannot be found, even for readable object types. This will be done using a readable type called `toggle`.

Definition 8.6 The `toggle` type is given by the following specification.

$$\begin{aligned} Q &= \{\perp, 0, 1\} \\ \mathrm{OPS} &= \{\mathit{toggle}(0), \mathit{toggle}(1), \mathit{read}\} \\ \mathrm{RES} &= Q \cup \{\mathit{ack}\} \\ \delta(q, \mathit{read}) &= (q, q) \text{ for } q \in Q \\ \delta(\perp, \mathit{toggle}(j)) &= (j, \mathit{ack}) \text{ for } i \in \{0,1\} \\ \delta(i, \mathit{toggle}(j)) &= (1-i, \mathit{ack}) \text{ for } i, j \in \{0,1\} \end{aligned}$$

The state transitions for the *toggle* operation are shown in Figure 8.3.

Proposition 8.7 $\mathrm{cons}(\texttt{toggle}) = 1$ but $\mathrm{cons}(\texttt{multi}(\texttt{toggle}, 2)) = \infty$.

Proof: First it will be shown that `toggle` is not a 2-discerning readable type. To derive a contradiction, suppose `toggle` is 2-discerning. Without loss of generality, suppose $A = \{P_1\}$ and $B = \{P_2\}$. If the two processes are assigned different *toggle* operations, say $op_1 = \mathit{toggle}(0)$ and $op_2 = \mathit{toggle}(1)$, then $(\mathit{ack}, 0) \in R_{A,1} \cap R_{B,1}$, which contradicts Definition 5.3. If both processes are assigned the same *toggle* operation, say $\mathit{toggle}(0)$, then $R_{A,1} \cap R_{B,1}$ contains $(\mathit{ack}, 0)$ if q_0 is 0, and contains $(\mathit{ack}, 1)$ if q_0 is either 1 or $\perp$, and this is again a contradiction.

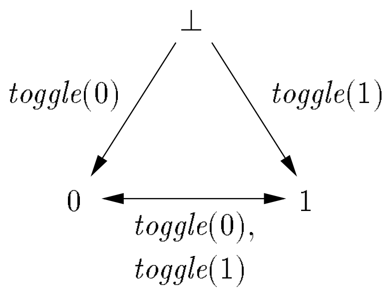

Figure 8.3: State transitions for the *toggle* operation.

It will now be shown that the readable type $\texttt{multi}(\texttt{toggle}, 2)$ is n-discerning, for any n, and therefore has infinite consensus number, by Theorem 5.8. Only three base objects O_1, O_2 and O_3 will be used. Each of the three base objects is initialized to the state $\perp$ in q_0. The n processes are partitioned into two non-empty teams A and B arbitrarily. Processes on team A perform a *toggle*(0) operation on O_1 and O_2. Processes on team B perform a *toggle*(1) operation on O_2 and O_3. The state transitions of the three base objects O_1, O_2 and O_3 in response to operations performed by processes on team A and B are shown in Figure 8.4. At any time after the first *toggle* has been performed, one can determine which team went first from the state of the multi-object as follows. If the base object O_1 is in state $\perp$, then team B went first. If the base object O_3 is in state $\perp$, then team A went first. Otherwise, the state of each of the three base objects will be in $\{0, 1\}$. The first team to take a step can be determined from the parity of these three states: team A went first if and only if zero or two of the base objects are in state 1. Thus, $\texttt{multi}(\texttt{toggle}, 2)$ is an n-discerning readable type. □

In the proof above, it was shown that the multi-object $\texttt{multi}(\texttt{toggle}, 2)$ can solve consensus among any number of processes even if all base objects must be initialized to the same starting state and all of the operations performed on base objects during one multi-object operation must be the same.

The following corollary shows that, for any n, one cannot obtain an upper bound on $\text{cons}(\texttt{multi}(\texttt{T}, m))$ (as a function of m) that applies to the class of object types $\texttt{T}$ with consensus number n, even if only readable objects are considered.

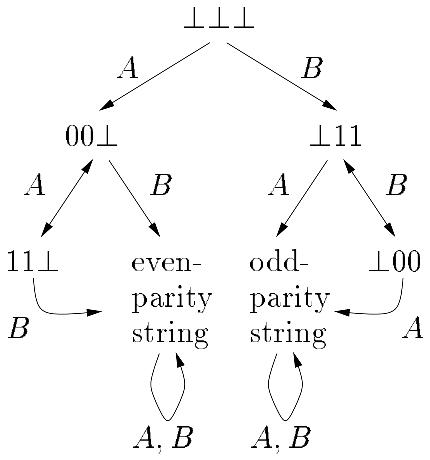

Figure 8.4: State transitions of the multi-object in the protocol of Proposition 8.7

Corollary 8.8 Let n be a positive integer. For the readable type $\mathtt{T}_n$ defined in Definition 5.13, $\mathrm{cons}(\mathtt{toggle} \vee \mathtt{T}_n) = n$, but $\mathrm{cons}(\mathtt{multi}(\mathtt{toggle} \vee \mathtt{T}_n, m)) = \infty$ for all $m > 1$.

Proof: By Theorem 6.2, the consensus hierarchy is robust for readable objects. So, by Theorem 6.1, $\mathrm{cons}(\mathtt{toggle} \vee \mathtt{T}_n) = \max(\mathrm{cons}(\mathtt{toggle}), \mathrm{cons}(\mathtt{T}_n)) = \max(1, n) = n$. However, since an object of type $\mathtt{toggle} \vee \mathtt{T}_n$ can directly implement a $\mathtt{toggle}$ object trivially, it follows that $\mathrm{cons}(\mathtt{multi}(\mathtt{toggle} \vee \mathtt{T}_n, m)) = \infty$ for all $m > 1$. □

Chapter 9

Transactional Objects

THE preceding chapter described how the ability of a type to solve consensus increases as a function of the number of base objects that can be accessed (independently) in a single atomic action. In this chapter, it will be shown that the consensus number of a type increases considerably more in the transactional setting, defined on page 62, where the accesses that are part of an atomic action need not be independent.

Section 9.1 shows that any transactional type built from a non-trivial readable base type is universal. The transactional type built from the n-`consensus` base type is studied in Section 9.2, and the consensus algorithm developed there is adapted in Section 9.3 to obtain a lower bound on the consensus number of any transactional type with a non-trivial RMW base type. Proofs of the universality of $\texttt{trans}(\texttt{test\&set}, 3)$ and $\texttt{trans}(\texttt{swap}, 2)$ are also given. The final section of this chapter provides an exponential lower bound on the consensus number of a transactional type built from *any* non-trivial base type.

§9.1 Readable Base Types

The universality of transactional types based on non-trivial readable types will now be proved. Recall that an object type is called trivial if it can be simulated without using shared memory at all.

Theorem 9.1 For any non-trivial readable type `T`, $\text{cons}(\texttt{trans}(\texttt{T}, 2)) = \infty$.

Proof: Since `T` is non-trivial, there must be some update operation op that changes the state of the object from some state q to some other state r.

The transactional type $\texttt{trans}(\texttt{T}, 2)$ is, itself, a readable type: it is possible to read each of the base objects (possibly in a piecewise manner), so it is possible to read any component of the transactional object's state without altering the state.

By Theorem 5.8, it suffices to show that $\texttt{trans}(\texttt{T}, 2)$ is n-discerning (as specified in Definition 5.3) for every natural number n. The initial state q_0 of the transactional object has every base object in state q. Partition a collection of n processes into two teams $A = \{P_1\}$ and $B = \{P_2, \ldots, P_n\}$. Let the base objects of the transactional object be denoted by $O_1, O_2, \ldots$. The operation op_1 assigned to process P_1 is simply an application of the update operation op to O_1. For $i > 1$, assign to process P_i a transaction as its op_i: it first performs a *read* of O_1 and, if the state returned is q, it then applies op to O_i.

Consider any execution where some non-empty set of processes each perform their assigned operations. If the process P_1 on team A takes the first step, all of the objects $O_2, \ldots, O_n$ will always remain in state q. However, if a process P_i on team B takes the first step, it will change the state of O_i to r, and O_i will remain in state r for the rest of the execution. Thus, one can determine whether a process on team A took the first step from the state of the transactional object by checking whether all of the objects $O_2, \ldots, O_n$ are in state q. The object $\texttt{trans}(\texttt{T}, 2)$ is therefore n-discerning for any n. □

It follows from this theorem that a readable type T can be used to implement $\texttt{trans}(\texttt{T}, 2)$ if and only if T is either trivial or universal. Furthermore, transactional objects can be much more powerful than the corresponding multi-objects. For example, if m is any integer greater than one, the multi-object $\texttt{multi}(\texttt{register}, m)$, which has consensus number $2m - 2$ [31], is incapable of implementing even $\texttt{trans}(\texttt{register}, 2)$.

§9.2 Consensus Base Types

Meanwhile, in the forest,
In a parliament of trees...
—Rheostatics [71]

This section considers transactional memory that uses base objects of the n-**consensus** type, defined in Section 3.1.2. The algorithm presented here shows that the ability to use transactions causes an exponential increase in the consensus number of a shared-memory system. This contrasts with the multi-object setting, where Afek, Merritt and Taubenfeld [4] showed that if the base objects of type $\mathtt{T}$ may be accessed by at most n processes, $\text{cons}(\mathtt{multi}(\mathtt{T}, m))$ is $O(n\sqrt{m})$.

The algorithm uses a collection of tree data structures. Processes access these trees by starting at a leaf and working towards the root. Each node contains an n-**consensus** object. Each child of a parent node acts as a filter to control access to the parent: only one of the processes that access the child will be allowed to access the parent. The first value proposed to the first tree's root becomes the output of all processes. However, since each base object in the transactional object can be accessed at most n times, information about the output value must be carefully distributed to all processes using a series of other trees that use similar filtering mechanisms to control access to the nodes. The algorithm given here will be adapted in Sections 9.3 and 9.4 to prove more general results.

Proposition 9.2 The transactional object $\mathtt{trans}(n\text{-}\mathtt{consensus}, m)$ can be used to solve consensus among n^{m-1} processes.

Proof: When $n = 1$, the result is trivial, so assume that $n \geq 2$. A consensus protocol for $N = n^{m-1}$ processes, $P_1, \ldots, P_N$, will be constructed using a transactional object of type $\mathtt{trans}(n\text{-}\mathtt{consensus}, m)$. All of the base objects are initialized to the state $\perp$. Arrange the base objects into trees as follows. The first tree, T, is a complete n-ary tree of height $m - 2$. Each of the remaining trees, T_k^j for $1 \leq k \leq n$ and $1 \leq j < n^{m-2}$, consists of a root with one child which is, itself, the root of a complete n-ary tree of height $m - 3$. The tree T will be used to determine the output of the consensus protocol. The remaining trees will be used to distribute this information to all of the processes. A part of the data structure is shown in Figure 9.2. The triangles represent complete n-ary trees of height $m - 3$.

Divide the processes into n^{m-2} groups of size n. Associate each group with a different leaf of the tree T. Let G_k be the set of n^{m-2} processes that are associated with leaf descendants of the kth child of T's root. The trees T_k^j, for $1 \leq j < n^{m-2}$, will be used to distribute information to the processes in G_k. Associate each of the n-process groups in G_k with one of the n^{m-3} leaves of T_k^1. The group is also associated with the corresponding leaf in each of the trees T_k^j,

```
shared variable
    An object of type trans(n-consensus, m) whose base objects are arranged as follows
        T : complete n-ary tree of height m − 2, each node containing one base object
        T_k^j (1 ≤ k ≤ n, 1 ≤ j < n^{m−2}): a tree whose root has one child.
            That child is the root of a complete n-ary tree of height m − 3.
            Each node contains one base object.
        T_k^{n^{m−2}} (1 ≤ k ≤ n) : a single node with one base object (used only to simplify code).

private global variables
    done : boolean % becomes true when process has determined its output
    v : integer % output value

subroutine traverse(p, next : pointer to node)
    local variable
        winner : {1, ..., N}
    begin transaction
        winner ← i
        loop while winner = i and p has a parent
            % traverse the path of length m − 2 towards the root until a loss
            winner ← result of applying propose(i) to p
            p ← parent of p
        end loop
        if winner = i then % access root and copy result to next root
            v ← result of applying propose(v) to p
            apply propose(v) to next
            done ← true
        end if
    end transaction
end traverse

% main routine
local variable
    j : {1, ..., n^{m−2} − 1}
done ← false
v ← input value of process P_i
traverse(lth leaf of kth subtree of T, root of T_k^1)
j ← 1
loop until done
    traverse(lth leaf of T_k^j, root of T_k^{j+1})
    j ← j + 1
end loop
halt with output v
```

Figure 9.1: The code for a process P_i, associated with the lth leaf of the kth subtree of tree T.

for $2 \leq j < n^{m-2}$.

Figure 9.1 shows the algorithm for a process P_i that is associated with the lth leaf of the kth subtree of tree T. The assignment of processes to leaves can be done by taking $k = \lceil \frac{i}{n^{m-2}} \rceil$ and $l = \lceil \frac{1+(i \bmod n^{m-2})}{n} \rceil$, for $1 \leq i \leq n^{m-1}$.

Process P_i in group G_k begins with a transaction in which it accesses base objects of the tree T. The process accesses nodes in the tree, each time proposing its own process identifier, i, and receiving a response. The process is said to "win" at a node if the response it receives is the value it proposed, and it is said to "lose" at the node if it receives some other response. First, it accesses the leaf of T with which its group of processes has been associated. If it wins at the leaf, process P_i goes on to access the parent of the leaf, again proposing the value i. It continues in this way, traversing the path from the leaf to the root until it loses. If the process loses at some node below the root, it ends its transaction. If it does reach the root, P_i proposes its own input value (instead of i) to the object located there. Let v be the value that the process P_i receives as a response from the root. The process then proposes the value v to the root of T_k^1, terminates its transaction, outputs v and halts. The access to the root of T_k^1 has the effect of copying v into that object: all future access to that object will return the value v. (It will be shown, below, that no base object ever gets reset to the state $\perp$.)

After performing its first transaction, process P_i knows the outcome only if it accessed the root of T. However, this outcome has also been stored in the root of the tree T_k^1. In an attempt to retrieve this information, each process in group G_k that has not yet halted performs its second transaction on T_k^1. It traverses a path from its leaf towards the root in the same way as it did in T during the first transaction. If the process does reach the root of T_k^1, it receives the output value v, which it then copies into the root of T_k^2 by applying the operation *propose*(v). If the process does not reach the root of T_k^1, it continues in this way, performing transactions on T_k^2, T_k^3, and so on, until it successfully reaches the root of a tree. Whenever a process accesses the root of a tree T_k^j, it immediately copies the value v stored there to the root of the next tree, T_k^{j+1}, as part of the same transaction, outputs v and halts. All outputs of this protocol will be the input value of the first process to perform a transaction.

Figure 9.2 shows the accesses to the transactional object by some process of group G_1 in

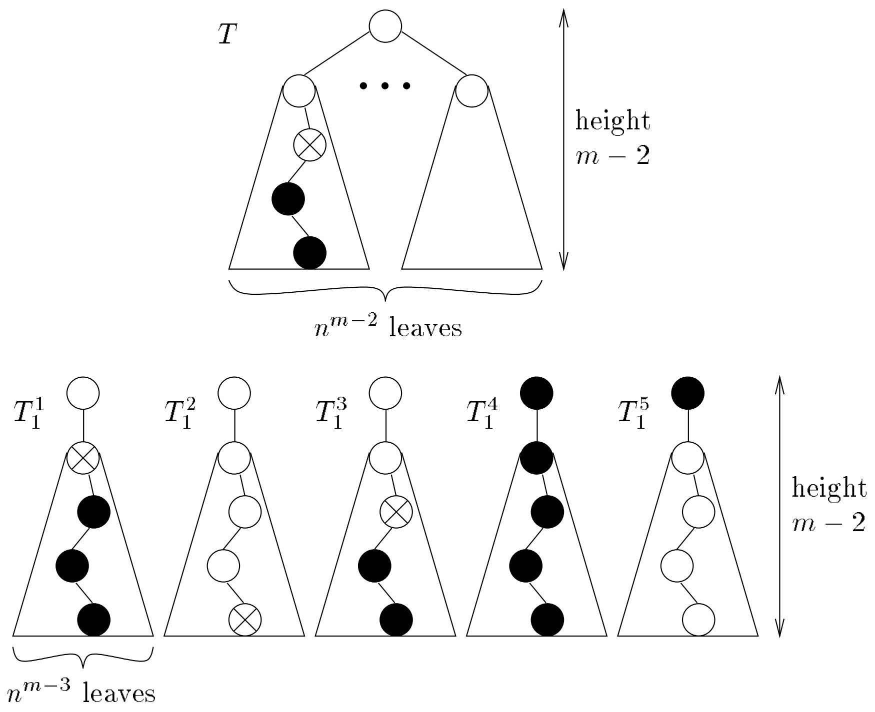

Figure 9.2: Accesses to the transactional object by one process

one execution. The solid circles represent nodes where the process wins. The circles marked with an "X" represent the nodes where it loses. In this example, the process performs five transactions. In the first, it attempts to gain access to the root of T by working its way up from a leaf, but fails. The process then tries unsuccessfully to gain access to the roots of T_1^1, T_1^2 and T_1^3 in its next three transactions. In the final transaction, it successfully reaches the root of T_1^4 and then accesses the root of T_1^5 directly.

It must be shown that the algorithm terminates. Suppose the process P_i attempts to access one of the trees T_k^j, but loses before reaching the root. Then some other process must have performed its transaction on the tree before P_i. The first such process will successfully reach the root and terminate. Thus, each time process P_i fails to access the root of a tree, some other process in G_k terminates. There are $n^{m-2}-1$ processes in G_k that attempt to access the information from the trees in the set $\{T_k^j : 1 \leq j < n^{m-2}\}$, since one process in G_k gets the information directly from the root of T. Thus, process P_i must successfully access the root of one of the $n^{m-2}-1$ trees T_k^j.

Since any path from a leaf to a root of one of the trees has $m-1$ nodes, no transaction contains more than m operations on the shared memory. It must also be checked that no base object is accessed more than n times, since an n-**consensus** object gets reset to the state $\perp$ after n accesses. For $1 \leq j < n^{m-2}$, at most one process will access the root of T_k^j to read the value stored there (during the $(j+1)$th transaction of the process), since only one process will win at the child of the root. At most one process will access the root of T_k^1 to record the winning value (during the first transaction of the process), since only one of G_k's processes can reach the root of T. For $1 < j < n^{m-2}$, at most one process will access the root of T_k^j to record the winning value (during the jth transaction of the process), since only one process can reach the root of T_k^{j-1} in its jth transaction. Thus, at most two processes access the root of any tree T_k^j. The leaf of any tree is accessed only by the n processes in the group associated with that leaf. Any other internal node is accessed only by those processes that win at the children of the node, and there are at most n such processes. □

§9.3 RMW Base Types

Herlihy [31] showed that any non-trivial RMW object type has consensus number at least two. An algorithm similar to the one given in Proposition 9.2 for the case where $n = 2$ can be used to establish an exponential lower bound on the consensus number of transactional memory built from any non-trivial RMW object type. In fact, the lower bound applies to an even more general class of objects. It is applicable whenever two processes can each apply a single operation and immediately know which operation occurred first. (It will be shown below that any non-trivial RMW type has this property.)

Theorem 9.3 Let $\mathtt{T}$ be any type. Suppose there exist two (not necessarily different) operations op_0 and op_1, and a state q of type $\mathtt{T}$ so that op_0 and op_1 each return different responses depending on the order that the two operations are performed on an object initially in state q. Then, $\text{cons}(\mathtt{trans}(\mathtt{T}, m)) \geq 2^{m-1}$.

Proof: For $i = 0, 1$, let R_i be the response returned by op_i when an object of type $\mathtt{T}$ is in state q, and R_i' be the response returned by op_i when it is preceded by the other operation, op_{1-i}.

The hypothesis requires that $R_0 \neq R'_0$ and $R_1 \neq R'_1$.

Consider a system with 2^{m-1} processes. Partition the set of processes into two non-empty teams, A and B, each with 2^{m-2} processes. By Proposition 3.14, it suffices to show that $\mathtt{trans}(\mathtt{T}, m)$ can solve the team$(2^{m-2}, 2^{m-2})$ problem.

Arrange the base objects into trees and assign groups of processes to each leaf as in the proof of Proposition 9.2 for the case where $n = 2$. Processes on team A should be assigned to the left subtree of T, and processes on team B should be assigned to the right subtree of T. Initialize all base objects to the state q.

The team consensus protocol will mimic the operation of the algorithm given in Proposition 9.2 with $n = 2$. However, instead of agreeing on an input value, the algorithm will be used to agree on the team of the first process to access the transactional object. Once this can be accomplished, it is easy to solve the team$(2^{m-2}, 2^{m-2})$ problem: each process first writes its input value into a `register` belonging to its team, and when the identity of the winning team is known, the value stored in that team's `register` is returned as the output.

shared variables
  One `register` for each team
  An object of type $\mathtt{trans}(\mathtt{T}, m)$ whose base objects are arranged as follows
    T : complete binary tree of height $m-2$, each node containing one base object
    T_k^j $(1 \leq k \leq 2, 1 \leq j < 2^{m-2})$: a tree whose root has one child. That child
      is the root of a complete binary tree of height $m-3$. Each node contains
      one base object.
    $T_k^{2^{m-2}}$ $(1 \leq k \leq 2)$: a single node with one base object (used only to simplify code).

private global variables
  done : boolean % becomes true when process has determined its output
  team : $\{A, B\}$ % identity of winning team

function useOp(*previous*, p : pointer to node) : $\{0, 1\}$
  % Determines whether process should apply op_0 or op_1 when accessing a node.
  if *previous* = nil then return i mod 2 % a leaf is being accessed
  elsif *previous* is left child of p then return 0
  else return 1 % previous is right child of p
  end if
end useOp

Figure 9.3: Code for P_i to solve the team$(2^{m-2}, 2^{m-2})$ problem (Part I).

subroutine traverse(p, *next* : pointer to node)
    local variables
        win : boolean % did P_i win when accessing a node?
        previous : pointer to node % node previously accessed during traversal
    begin transaction
        $win \leftarrow$ true
        $previous \leftarrow$ nil
        loop while *win* and p has a parent
            if result of applying $op_{\text{useOp}(previous,p)}$ to p is $R'_{\text{useOp}(previous,p)}$ then $win \leftarrow$ false
            $previous \leftarrow p$
            $p \leftarrow$ parent of p
        end loop
        if *win* then
            if p is the root of T then
                if result of applying $op_{\text{useOp}(previous,p)}$ to p is $R_{\text{useOp}(previous,p)}$ then
                    $team \leftarrow$ team to which P_i belongs
                else $team \leftarrow$ team to which P_i does not belong
                end if
            else % p is root of some T_k^j.
                if result of applying op_1 to p is R_1 then $team \leftarrow A$
                else $team \leftarrow B$
                end if
            end if
            $done \leftarrow$ true
            if $team = B$ then apply op_0 to *next* % record winning team in *next*
        end if
    end transaction
end traverse

% main routine for a process P_i on team A
local variable
    $j : \{1, \ldots, 2^{m-2} - 1\}$
$done \leftarrow$ false
write input value into team A's `register`
traverse($\lceil i/2 \rceil$th leaf of T, root of T_1^1)
$j \leftarrow 1$
loop until *done*
    traverse($\lceil i/2 \rceil$th leaf of T_1^j, root of T_1^{j+1})
    $j \leftarrow j + 1$
end loop
halt, using the value in *team*'s `register` as the output

Figure 9.4: Code for P_i to solve the team$(2^{m-2}, 2^{m-2})$ problem (Part II).

```
% main routine for a process P_i on team B
local variable
    j : {1, ..., 2^{m-2} - 1}
done ← false
write input value into team B's register
traverse(⌈i/2⌉th leaf of T, root of T_2^1)
j ← 1
loop until done
    traverse(⌈i/2 - 2^{m-1}⌉th leaf of T_2^j, root of T_2^{j+1})
    j ← j + 1
end loop
halt, using the value in team's register as the output
```

Figure 9.5: Code for P_i to solve the team$(2^{m-2}, 2^{m-2})$ problem (Part III).

The algorithm for process P_i to solve team consensus is given in Figures 9.3, 9.4 and 9.5. It assumes that processes are associated to the leaves of T in order: P_i is associated to the $\lceil i/2 \rceil$th leaf of T. Then $A = \{P_1, \ldots, P_{2^{m-2}}\}$ and $B = \{P_{2^{m-2}+1}, \ldots P_{2^{m-1}}\}$.

To simulate the filtering action of a node below the root of a tree, one process that accesses the node performs op_0 and the other process performs op_1. If the process that performed op_i receives the response R_i, then it has won at that node. If it receives the response R'_i, then it has lost at that node. The two processes that access the root of T determine which team accessed the root first in exactly the same way.

To simulate the operation that stores a value into the root of T_k^j (during the jth transaction performed by a process), the process applies the operation op_0 to the object if and only if it wants to record the fact that a process from team B took the first step. If it wants to indicate that a process from team A took the first step, it does nothing.

To simulate the operation that reads the contents of the root of T_k^j (during the $(j+1)$th transaction performed by a process), the process applies the operation op_1 to the base object located at the root. It interprets the response R_1 as indicating that a process from team A went first, and the response R'_1 as indicating that a process from team B went first.

Clearly, each transaction contains at most m operations on base objects. The correctness of the algorithm can be shown in exactly the same way as in the proof of Proposition 9.2. □

Corollary 9.4 For any non-trivial RMW type $\mathtt{T}$, $\text{cons}(\mathtt{trans}(\mathtt{T}, m)) \geq 2^{m-1}$.

Proof: Since $\mathtt{T}$ is non-trivial, there exists some operation, *op*, that applies a function f with $f(q) \neq q$ for some state q. Otherwise, one could trivially simulate $\mathtt{T}$ without using shared memory by returning the initial state of the object as the response to every operation. The operation *op* can be used as both operations op_0 and op_1 of Theorem 9.3. If *op* is performed when the object is in state q, it returns the response q. However, if *op* is performed when another *op* has already been performed on an object in state q, the second *op* returns a different response, $f(q)$. The lower bound follows from Theorem 9.3. □

The `test&set` object defined in Example 2.2 is perhaps the simplest non-trivial RMW type. The following proposition demonstrates that this very simple type becomes even more powerful in the transactional setting than Corollary 9.4 suggests. The algorithm used in the proof has a different flavour from the algorithm in the previous proof. Once a base object has been accessed, further accesses never change its state, so it is not necessary to carefully control access to the base objects to avoid erasing information stored in them.

Proposition 9.5 The type $\mathtt{trans}(\mathtt{test\&set}, 3)$ is universal.

Proof: Let n be any integer greater than one. An algorithm will be given to solve the team$(1, n-1)$ problem using an object of type $\mathtt{trans}(\mathtt{test\&set}, 3)$. It follows from Proposition 3.14 that the consensus number of $\mathtt{trans}(\mathtt{test\&set}, 3)$ is infinity. Here, it is described how every process can determine which team first accesses the transactional object. Once this can be done, it is easy to solve team consensus using two additional `registers`, as in the preceding proof.

Divide the n processes into two teams $A = \{P_1\}$ and $B = \{P_2, \ldots, P_n\}$. The algorithm uses $2n+1$ base objects, labelled $A_1, \ldots, A_n, B_1, \ldots, B_n$ and C, all of which are initially in state 0.

The object C is used to determine which team wins: every process will eventually discover which team accessed C first. The other objects are used to distribute the information about the winning team to all of the processes. First, the method of distributing this information will be described informally. Processes access the objects $A_1, \ldots, A_n$ in order so that A_{i+1} is never set to 1 while A_i still has value 0. The same comment applies to the objects $B_1, \ldots, B_n$. Consider some moment in the computation. Let a be the largest index such that A_a has been set to 1. Let b be the largest index such that B_b has been set to 1. The information about the first team to access object C is stored in the other base objects by maintaining the following invariant after each complete transaction.

Invariant: If a process from team A accessed the transactional object first, then $a = b+1$. If a process from team B accessed the transactional object first, then $b = a+1$.

Each process retrieves information about which team first accessed the base object C by accessing pairs (A_i, B_i) for increasing values of i until it finds a pair where only one of the two base objects is set to 1.

The algorithm will now be described in detail. Process P_1 performs a single transaction. It first performs a *test&set* operation on object C. If it receives the response 1, P_1 knows that some other process has already accessed C, so it can conclude that a process on team B accessed the transactional object first, and it need not perform any further actions. On the other hand, if it receives the response 0, then it knows it is the first process to access the transactional object. It then performs a *test&set* operation on object A_1 to ensure that the invariant holds and performs no further actions.

The algorithm for a process on team B is more complicated. In the first transaction, the process performs a *test&set* operation on the base object C. If the result is 0, it knows that it is the first process to access the transactional object. It then performs the operation *test&set* on B_1 as part of the same transaction in order to satisfy the invariant. If the process receives the result 1 from C, it knows that some other process has already accessed the transactional object. The process must then use the other base objects to determine which team made the first access. The process performs a number of transactions, accessing A_1 and B_1 in the first transaction, then A_2 and B_2 in the second transaction, and so on, until it gets different responses from two objects A_i and B_i. Suppose that A_i returns 1 and B_i returns 0, indicating that a process from team A accessed the transactional object first. To ensure that the invariant remains true, the process performs a *test&set* operation on A_{i+1} as part of the same transaction. It can then halt, knowing that team A performed the first transaction. The case where A_i returns 0 and

B_i returns 1 is symmetric.

It must be checked that this algorithm does terminate. The first process to perform a transaction that accesses the pair of objects A_i and B_i will halt at the completion of that transaction. Thus, at most $n-i$ processes on team B will perform the transaction that accesses A_i and B_i. Thus, every process will perform at most n transactions. It is easy to see that the invariant is true after each complete transaction, and the correctness of the algorithm follows. □

The algorithm given in this proof can easily be adapted to work for any base type that is equipped with an operation whose first invocation returns a response different from any subsequent invocation's response.

One other specific RMW type will be considered to illustrate the difference between multi-objects and transactional objects with RMW base types. A `swap` object stores an integer and is equipped with the operation $swap(v)$ which changes the state to the integer v and returns the previous state of the object. This type is at level two of the consensus hierarchy, and it is known that $\text{cons}(\texttt{multi}(\texttt{swap}, m)) = \Theta(\sqrt{m})$ [4]. Here, two proofs of the universality of $\texttt{trans}(\texttt{swap}, 2)$ are given.

Proposition 9.6 The type $\texttt{trans}(\texttt{swap}, 2)$ is universal.

Proof 1: The type $\texttt{trans}(\texttt{swap}, 2)$ can easily implement the type `compare&swap`, defined in Example 4.2, using a single base object. To simulate the operation *compare&swap*(old, new), a process performs a $swap(new)$ operation on the base object. If the response r is different from old, the process performs the operation $swap(r)$ on the base object as part of the same transaction. Since `compare&swap` is universal [31], so is $\texttt{trans}(\texttt{swap}, 2)$. □

The first proof employs transactions that access the same base object twice. However, the following proof establishes the result even if a transaction is restricted to access each base object at most once.

Proof 2: A consensus protocol for any number of processes will be described. It uses three base objects O_0, O_1 and O_2, each initially in state 0. The protocol will ensure that all processes output the input value of the first process to take a step. Each process first *swaps* the value 1 into O_0. This base object is used like a `test&set` object: a process knows it is the first process to access the object if and only if it receives the response 0. If the process does receive the response 0, it records its input value, incremented by one, in O_1 as part of the same transaction. (The input value is incremented so that the state 0 can be used as a "nil" value to indicate that information about the outcome is not recorded in the base object.) The process then outputs its own input value. If a process receives the response 1 from O_0, it must determine the input value of the process that took the first step. It does this by repeatedly performing a $swap(0)$

```
shared variable
    An object of type trans(swap, 2) with three base objects O_0, O_1, O_2, each initially 0.

local variable
    v : natural number
    done : boolean % becomes true when process knows output value
    out : natural number % output value
    i : {1, 2}

begin transaction
    v ← result of applying swap(1) to O_0
    if v = 0 then % no other process has taken a step
        apply swap(1 + input value) to O_1
        done ← true
        out ← input value
    else
        done ← false
    end if
end transaction
i ← 1
loop until done
    begin transaction
        v ← result of applying swap(0) to O_i
        if v ≠ 0 then
            apply swap(v) to O_{3-i}
            done ← true
            out ← v - 1
        end if
    end transaction
    i ← 3 - i
end loop
halt with output out
```

Figure 9.6: Code for Proof 2 of Proposition 9.6

operation, starting a new transaction each time and alternating between the base objects O_1 and O_2. Whenever the process receives a non-zero response from O_1 or O_2, it *swaps* the response into O_2 or O_1, respectively, as part of the same transaction. The process then subtracts one from the response and outputs it. A description of the algorithm in pseudocode is given in Figure 9.6.

Consider any execution of the protocol. Let f be the input value of the first process to take a step. It is easy to check that the following invariant holds after any step of the execution.

Invariant: The state of O_0 is different from 0. One of the base objects O_1 or O_2 is in state 0, and the other is in state $1 + f$.

It follows from this invariant that the protocol is wait-free: if any collection of processes are executing the loop, the first process that *swaps* 0 into the base object O_1 or O_2 that contains $1 + f$ exits the loop and terminates. The number of processes running continues to decrease in this way until all non-faulty processes have terminated. The invariant can also be used to show that every process outputs f. If a process receives the response 0 from the base object O_0, it must be the first process to perform a step, so it can output its own input value. If a process ever receives a non-zero response from O_1 or O_2, that response must be $1 + f$. The process then outputs f. $\square$

§9.4 A General Lower Bound

Theorem 9.3 gives a lower bound on the consensus numbers of transactional objects that applies to a large class of base types. In this section, it will be shown that the non-triviality of the base type $\mathtt{T}$ is sufficient to show that $\mathtt{multi}(\mathtt{T}, 2)$ satisfies the hypothesis of Theorem 9.3, and this fact can be used to prove a weaker, though still exponential, lower bound on the consensus number of a transactional type built from any non-trivial base type.

Theorem 9.7 For any non-trivial base type $\mathtt{T}$, $\text{cons}(\mathtt{trans}(\mathtt{T}, m)) \geq 2^{\lfloor m/2 \rfloor - 1}$.

Proof: Since $\mathtt{T}$ is non-trivial, the response to some operation, t_0 cannot depend entirely on the state in which the object is initialized. That is, there is some operation t_1 that takes an object of type $\mathtt{T}$ from a state s where it would return one response R to t_0 into a state where it would return a different response R' to t_0.

Now consider the type $\mathtt{multi}(\mathtt{T}, 2)$. It will be shown that this multi-object type satisfies the hypothesis of Theorem 9.3. Let O_0 and O_1 be two of the base objects. Let q be the state of the multi-object where all base objects are in state s. Let op_0 be the multi-object operation that applies t_0 to O_0 and t_1 to O_1. Let op_1 be the multi-object operation that applies to t_1 to O_0 and t_0 to O_1. Using the notation of the proof of Theorem 9.3, for $i = 0, 1$, let R_i be the response returned by op_i when the multi-object is in state q, and R'_i be the response returned by op_i when it is preceded by the other operation, op_{1-i}. It must be shown that $R_0 \neq R'_0$ and $R_1 \neq R'_1$. Let S be the response when t_1 is applied in state s and let S' be the response when t_1 is applied after t_0. Recall that $R \neq R'$, so $R_0 = (R, S) \neq (R', S') = R'_0$ and $R_1 = (S, R) \neq (S', R') = R'_1$.

By Theorem 9.3, $\text{cons}(\mathtt{trans}(\mathtt{multi}(\mathtt{T}, 2), \lfloor m/2 \rfloor)) \geq 2^{\lfloor m/2 \rfloor - 1}$. But $\mathtt{trans}(\mathtt{T}, m)$ is at least as strong a type as $\mathtt{trans}(\mathtt{T}, 2\lfloor m/2 \rfloor)$, which is equivalent to the type $\mathtt{trans}(\mathtt{trans}(\mathtt{T}, 2), \lfloor m/2 \rfloor)$. Since $\mathtt{trans}(\mathtt{T}, 2)$ can trivially implement $\mathtt{multi}(\mathtt{T}, 2)$, it follows that $\text{cons}(\mathtt{trans}(\mathtt{T}, m)) \geq \mathtt{trans}(\mathtt{multi}(\mathtt{T}, 2), \lfloor m/2 \rfloor) \geq 2^{\lfloor m/2 \rfloor - 1}$. □

The preceding proof shows that the algorithm given in Figures 9.3 to 9.5 can be adapted for use with any non-trivial base type. However, two base objects must be used in each node of the forest data structure. This has the effect of roughly halving the heights of the trees, and produces a corresponding reduction in the number of processes.

The following corollary shows that a transactional type is often more powerful than the corresponding base type $\mathtt{T}$. Proposition 8.2, a similar result for multi-objects, also applies to transactional types, since multi-object types are a restricted form of transactional types. However, the following corollary does not require that $\text{cons}(\mathtt{T}) > 2$, and this proof gives a better bound, for some types, on the parameter m for which the transactional type is more powerful.

Corollary 9.8 If $\mathtt{T}$ is any type that is neither trivial nor universal, there is some m such that $\mathtt{T}$ cannot provide a wait-free implementation of $\mathtt{trans}(\mathtt{T}, m)$.

Proof: If $m = 2\log_2(\text{cons}(\mathtt{T}) + 1) + 3$, then $\text{cons}(\mathtt{trans}(\mathtt{T}, m)) \geq 2^{\lfloor (2\log_2(\text{cons}(\mathtt{T})+1)+3)/2 \rfloor - 1} \geq 2^{\log_2(\text{cons}(\mathtt{T})+1)} = \text{cons}(\mathtt{T}) + 1$, so $\mathtt{trans}(\mathtt{T}, m)$ is strictly more powerful than $\mathtt{T}$. □

Chapter 10

Conclusion

Pour l'auteur ils pourraient s'appeler 'Conclusions' et pour le lecteur 'Incitations'. —Marcel Proust [69]

THE field of fault-tolerant shared-memory systems is fairly new: most of the work in this area has been done in the past fifteen years. One thread of research has focused on the design of efficient protocols for particular systems (for example, [1, 3, 62]). A second has used bivalency arguments to show impossibility results, again for particular types of shared objects [28, 31, 58]. These two threads gave rise to a third, which has the goal of proving more general results about the solvability of problems in shared-memory systems [15, 31, 35, 42, 45]. These general results have begun to provide a good overall picture of the relative powers of different types. In particular, the consensus hierarchy provides a good classification of types that provides information about which types can implement others.

The problem of proving that an object type belongs to a particular level of the consensus hierarchy remained a difficult task that was usually handled on a case-by-case basis. The results of this thesis are steps towards a general framework for such proofs, and are intended to provide some insight into the properties of a shared-memory system that determine where it fits into the consensus hierarchy. In fact, Chapters 4 and 5 gave concrete criteria for membership of individual types in each level of the hierarchy for two large classes of objects. The robustness result of Chapter 6 extends these criteria to systems containing several types. Finally, the results of Chapters 8 and 9 measure the increase in the consensus power of a system as parameters of the model are changed.

Although the focus of this thesis has been the third thread of research mentioned above, there are important connections with the other two. Many of the results were proved by giving concrete protocols, and the design of protocols for specific types of shared memory was often crucial in establishing that general bounds were as tight as possible. The bivalency arguments used to prove general results were often motivated by similar arguments previously used for specific types.

§10.1 Remarks on Computability

In the model of shared-memory systems used in this thesis, the sets used to give a sequential type specification are recursive and the transition function is computable (see page 10). These assumptions were made for two reasons. They ensure that object types can be described by a finite specification, so that the question of deciding the consensus number of a type can be formulated. In addition, they allow Herlihy's universality result (Theorem 3.1) to be formulated for processes that have the computational power of random-access machines: Herlihy's construction requires that processes can compute the transition function.

It should be pointed out that most of the central results of this thesis would hold even without these assumptions. An alternative model would allow arbitrary sets and transition functions in the definitions of object types. Theorems 4.10, 5.9, 6.2 and the bounds on the consensus numbers of multi-objects and transactional objects hold in this model, since the computability assumptions are not used in the proofs. In order for Theorem 3.1 to be true in this more general model, processes must be given some additional power. For example, they might be permitted to access an oracle that computes the transition function of a type to be implemented.

§10.2 Future Directions

There are many open questions related to the work in this thesis.

Chapters 4 and 5 characterized the RMW and readable types that can solve consensus among n processes. Can similar techniques be used to obtain characterizations for other natural classes of objects, or for other problems, such as set-consensus, or for different fault models?

The characterizations use the fact that processes have well-defined critical values for any critical configuration of a consensus protocol, something which is not true if non-deterministic object types are considered. Can the results be generalized somehow to cover non-deterministic RMW or readable types?

The proofs of Theorems 4.9 and 5.8 use a tournament-style consensus protocol. Thus, for readable and RMW types, if a consensus protocol exists at all, then there is a protocol of this very special form. If the idea of the tournament is extended to allow more than two teams in each competition, can it be shown that, for each type, some such tournament protocol is the best possible consensus protocol (in terms of space or time complexity).

Jayanti and Toueg showed that the consensus number of an object type is undecidable in general [47]. Their proof relies on the fact that an object type can be constructed to simulate the execution of a Turing Machine. This means that the object must be allowed to have an infinite state set. Corollaries 4.12 and 5.11 showed that one can decide whether a given finite RMW or readable type can solve consensus among a given number of processes. It would be interesting to determine whether consensus numbers are decidable for arbitrary finite types.

Another kind of decidability problem is to pick a particular primitive type and try to decide which types can be implemented by it. A good overview of work in this area was given by Herlihy and Rajsbaum [34]. For example, some systems are still equipped only with `registers` as their shared-memory objects. Is it possible to decide whether a given type is implementable in such a system? Gafni and Koutsoupias [30] showed that this is undecidable in general. However, Anderson and Moir [8] state (without proof) a result that, together with the results of this thesis, would imply that one can decide whether a given (finite) readable type can be implemented from `registers` alone.

Chapter 6 gave a concise proof that the hierarchy is robust for RMW and readable types. Two important properties of RMW and readable types are used: such objects are deterministic, and their state information can be freely accessed by all processes. Most types that have been used to show that the hierarchy is not robust [18, 57, 64, 76] have neither of these properties. They are non-deterministic and severely restrict the ways that processes may access their state information. Can robustness results be proved for some natural classes of non-deterministic types, such as non-deterministic readable types? This would perhaps give insight into the reasons that the hierarchy is robust for some classes, but not for others.

The results on consensus numbers of multi-objects and transactional objects that were given in Chapters 8 and 9 apply to deterministic types. Can they be extended to non-deterministic base types?

It is not known whether the lower bounds on the consensus numbers of transactional types given in Proposition 9.2 and Theorems 9.3 and 9.7 are tight. Indeed, no upper bounds on the consensus numbers of transactional types are known.

In the multi-object model, memory accesses within an atomic operation must be performed on different base objects. In the transactional model, processes are free to access base objects repeatedly during an atomic action, but this freedom was not required for any of the results in this thesis. Is there a case where the ability to access the same base object repeatedly really does help? Other variants of the multiple-access types could also be considered. For example, processes might be allowed to perform transactions on a single base object only. The definition of transactional objects allows great freedom in describing the way that shared-memory accesses can depend on earlier accesses within the transaction. One could require the dependencies between shared-memory accesses within a transaction to be of a specified, simple form to obtain multiple-access types that are in between multi-objects and the fully general transactional types.

Most of the work in this thesis has been concerned with computability and implementability in shared-memory systems. However, it is of practical importance to know how efficiently one type can solve a problem or implement another type. Although the results of this thesis often guarantee that implementations of one type from another are possible, the design of efficient implementations remains an important question. Recently, researchers have given bounds on the complexity of implementations as a function of the contention in the system, ensuring that the performance is better when fewer of the system's processes are accessing the same part of a data structure concurrently [2, 5]. Other recent developments include Jayanti's [43, 45] lower bounds on the complexity of the sort of universal construction that was used to prove Theorem 3.1. There have been some cases where reductions can be used to obtain lower bounds on time or space complexity from impossibility results [6, 29]. It would be interesting to see if further lower bounds could be derived in this way from the impossibility results in this thesis.

There is no case of Coming to an End but has about it something of an effort and a jerk, as though Nature abhorred it. —Hilaire Belloc [14]

FINIS

Bibliography

Iter est quacumque dat prior vestigium —Publilius Syrus [70]

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4), pages 873–890, September 1993.

[2] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[3] Yehuda Afek and Michael Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. 18th ACM Symposium on Principles of Distributed Computing*, pages 105–112, 1999.

[4] Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. The power of multi-objects. *Information and Computation*, 153(1), pages 117–138, August 1999.

[5] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling multi-object operations. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, pages 111–120, 1997.

[6] Yehuda Afek and Gideon Stupp. Optimal time-space tradeoff for shared memory leader election. *Journal of Algorithms*, 25(1), pages 95–117, October 1997.

[7] James H. Anderson. Composite registers. *Distributed Computing*, 6(3), pages 141–154, April 1993.

[8] James H. Anderson and Mark Moir. Towards a necessary and sufficient condition for wait-free synchronization. In *Distributed Algorithms, 7th International Workshop*, volume 725 of *LNCS*, pages 39–53, 1993.

[9] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[10] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.

[11] Hagit Attiya and Eyal Dagan. Universal operations: Unary versus binary. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 223–232, 1996.

[12] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.

[13] Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3), pages 117–127, 1997.

[14] H. Belloc. On coming to an end. In *On Nothing and Kindred Subjects*, page 255. Methuen, London, 1908.

[15] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 91–100, 1993.

[16] Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 363–372, 1994.

[17] Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. The h_m^r hierarchy is not robust. Manuscript, 1994.

[18] Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 334–343, 1994.

[19] Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998.

[20] Soma Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1), pages 132–158, July 1993.

[21] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4), pages 701–712, August 1994.

[22] Benny Chor and L. Moscovici. Solvability in asynchronous environments. In *Proc. 30th IEEE Symposium on Foundations of Computer Science*, pages 422–433, 1989.

[23] Robertson Davies. *The Merry Heart: Selections 1980–1995*, page 228. McClelland & Stewart, 1996.

[24] Roberto De Prisco, Dahlia Malkhi, and Michael K. Reiter. On k-set consensus in asynchronous systems. In *Proc. 18th ACM Symposium on Principles of Distributed Computing*, pages 257–265, 1999.

[25] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), page 569, September 1965.

[26] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3), pages 499–516, July 1986.

[27] Electronic (Bernard Sumner and Johnny Marr). Try all you want. On *Electronic* [Compact Disc Recording]. Warner Brothers, 1991.

[28] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), pages 374–382, April 1985.

[29] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.

[30] Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM Journal on Computing*, 28(3), pages 970–983, January 1999.

[31] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1), pages 124–149, January 1991.

[32] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5), pages 745–770, November 1993.

[33] Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 324–333, 1994.

[34] Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 589–598, 1997.

[35] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. Technical Report CS-96-03, Computer Science Department, Brown University, 1996. Available from `http://www.cs.brown.edu/people/mph/home.html`.

[36] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), pages 463–492, July 1990.

[37] A. Hilka and O. Schumann, eds. Carmen 17: O Fortuna. In *Carmina Burana*, volume 1, part 1. C. Winter, Heidelberg, 1930. O fate, monstrous / and empty, / you whirling wheel, / you are malevolent.

[38] Homer (translated by George Chapman). *Iliad, Book III.* 1608. Reprinted in *Chapman's Homer*, volume 1, Pantheon Books, New York, 1956.

[39] Phillip W. Hutto and Mustaque 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.

[40] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[41] Prasad Jayanti. Wait-free computing. In *Distributed Algorithms, 9th International Workshop*, volume 972 of *LNCS*, pages 19–50, 1995.

[42] Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4), pages 592–614, July 1997.

[43] Prasad Jayanti. A lower bound on the local time complexity of universal constructions. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 183–192, 1998.

[44] Prasad Jayanti. Solvability of consensus: Composition breaks down for nondeterministic types. *SIAM Journal on Computing*, 28(3), pages 782–797, September 1998.

[45] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

[46] Prasad Jayanti and Sanjay Khanna. On the power of multi-objects. In *Distributed Algorithms, 11th International Workshop*, volume 1320 of *LNCS*, pages 320–332, 1997.

[47] Prasad Jayanti and Sam 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.

[48] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*, pages A–63, A–93. Prentice-Hall, 1992.

[49] S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112, pages 727–742, 1936.

[50] Jon Kleinberg and Sendhil Mullainathan. Resource bounds and combinations of consensus objects. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pages 133–143, 1993.

[51] Dénes König. Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae: Sectio Scientiarum*

Mathematicarum, 3, pages 121–130, 1927. The required result is also stated as Theorem 3 in chapter VI of DÉNES KÖNIG, *Theory of Finite and Infinite Graphs*, Birkhäuser, Boston, 1990.

[52] CLYDE P. KRUSKAL, LARRY RUDOLPH, AND MARC SNIR. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4), pages 579–601, October 1988.

[53] LESLIE LAMPORT. Concurrent reading and writing. *Communications of the ACM*, 20(11), pages 806–811, November 1977.

[54] LESLIE LAMPORT. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9), pages 690–691, September 1979.

[55] LESLIE LAMPORT, ROBERT SHOSTAK, AND MARSHALL PEASE. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), pages 382–401, July 1982.

[56] WAI KAU LO. *On the relative power of shared objects in fault-tolerant distributed systems.* Ph.D. thesis, Department of Computer Science, University of Toronto, 1997.

[57] WAI-KAU LO AND VASSOS HADZILACOS. All of us are smarter than any of us: wait-free hierarchies are not robust. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 579–588, 1997.

[58] MICHAEL C. LOUI AND HOSAME H. ABU-AMARA. Memory requirements for agreement among unreliable asynchronous processes. In FRANCO P. PREPARATA, ED., *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987.

[59] NANCY A. LYNCH. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[60] STEPHEN MENKE, MARK MOIR, AND SRIKANTH RAMAMURTHY. Synchronization mechanisms for SCRAMNet+ systems. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 71–80, 1998.

[61] MICHAEL MERRITT AND GADI TAUBENFELD. Atomic m-register operations. *Distributed Computing*, 7(4), pages 213–221, May 1994.

[62] MARK MOIR. Practical implementations of non-blocking synchronization primitives. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[63] MARK MOIR. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop*, volume 1320 of *LNCS*, pages 305–319, 1997.

[64] SHLOMO MORAN AND LIHU RAPPOPORT. On the robustness of h_m^r. In *Distributed Algorithms, 10th International Workshop*, volume 1151 of *LNCS*, pages 344–361, 1996.

[65] GIL NEIGER. Set-linearizability. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, page 396, 1994.

[66] M. PEASE, R. SHOSTAK, AND L. LAMPORT. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), pages 228–234, April 1980.

[67] GARY L. PETERSON. Properties of a family of booster types. In *Proc. 18th ACM Symposium on Principles of Distributed Computing*, page 281, 1999.

[68] SERGE PLOTKIN. Sticky bits and universality of consensus. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pages 159–175, 1989.

[69] MARCEL PROUST. *Sur la Lecture,* the preface to his translation of *Sesame and Lilies* by JOHN RUSKIN, page 32. Société du Mercure de France, Paris, 1906. English translation, *On Reading*, published by Penguin, 1994. For the author they can be called 'Conclusions' and for the reader 'Incitements'.

[70] PUBLILIUS SYRUS. Sententia 328. In *Minor Latin Poets*, Loeb classical library, W. Heinemann, London, 1934. The road runs wheresoever a predecessor leaves his footprint.

[71] RHEOSTATICS (DAVE BIDINI, DAVE CLARK, MARTIN TIELLI AND TIM VESELY) AND J. MORASSUTTI. Northern wish. On *Melville* [Compact Disc Recording]. DROG Canadian Recordings, Guelph, Ontario, Canada, 1991.

[72] Eric Ruppert. Determining consensus numbers. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, pages 93–99, 1997.

[73] Eric Ruppert. Consensus numbers of multi-objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 211–217, 1998.

[74] Eric Ruppert. Consensus numbers of transactional objects. In *Distributed Computing: 13th International Symposium*, volume 1693 of *LNCS*, pages 312–326, 1999.

[75] Eric Schenk. Faster approximate agreement with multi-writer registers. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1995.

[76] Eric Schenk. *Computability and Complexity Results for Agreement Problems in Shared Memory Distributed Systems.* Ph.D. thesis, Department of Computer Science, University of Toronto, September 1996.

[77] Eric Schenk. The consensus hierarchy is not robust. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, page 279, 1997.

[78] William Shakespeare. Hamlet, III, ii, 1600.

[79] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2), pages 99–116, February 1997.

[80] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2), pages 217–255, April 1963.

[81] Richard L. Sites, ed. *Alpha Architecture Reference Manual*, pages (I)4–8–(I)4–11, (I)5–6. Digital Press, 1992.

[82] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Second Series*, 42, pages 230–265, November 1936.

[83] Peter van Emde Boas. *Machine Models and Simulations*, volume A, chapter 1. Elsevier, 1990.

Index