UNIVERSITY OF CALIFORNIA

Los Angeles

Capturing the Power of Resiliency and Set Consensus in Distributed Systems

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Elizabeth Borowsky

1995

1995

The dissertation of Elizabeth Borowsky is approved.

Robert Edwards

Elias Koutsoupias

Sheila Greibach

Eli Gafni, Committee Chair

University of California, Los Angeles

1995

TABLE OF CONTENTS

1 Introduction **1**

2 Model **6**
2.1 Distributed Systems 6
2.2 Distributed Tasks 9
2.3 System Resiliency 10
2.4 Solvability 12
2.5 Communication Models 13

3 Immediate Atomic Snapshots **16**
3.1 One-Shot Immediate Atomic Snapshots 18
3.2 K-Shot Immediate Atomic Snapshots 21
3.3 Multi-Shot Immediate Atomic Snapshots 22

4 View Graph Structure of Read-Write Memory **29**
4.1 Runs and Views 29
4.2 Topological Basics 32
4.3 Immediate Atomic Snapshot View Graph Structure 36
4.4 Necessary Condition for Read-Write Wait-Free Solvability 43

5 Narrowing Power of a Model **45**
5.1 Narrowing Power of the Wait-Free Model with Set Consensus 45
5.2 Narrowing Power of Active Resiliency and Set Consensus 47
5.3 Narrowing Power of t-Resiliency with Set Consensus 51

6 Simplex Agreement **55**
6.1 The Generalized Simplex Agreement Task 56
6.2 Topological Basis for Simplex Agreement 60
6.3 ϵ-Convergence 63
6.4 Generalized Simplex Agreement Algorithm 66
6.5 t-Resilient Generalized Simplex Agreement Algorithm 68

7 Simulation **73**
7.1 The Simulation by Agreement 73
7.2 The Structure of Active Resiliency and Set Consensus 81
7.3 The Structure of t-Resiliency and Set Consensus 90

8 Characterization Theorems **95**
8.1 Active Resiliency with Set Consensus Objects 95
8.2 t-Resiliency with Set Consensus Objects 98

9 Applications to Set Consensus **100**
9.1 Wait-Free Read-Write $k+1$ Processor k-Set Consensus 100
9.2 Impossibility of k-Resilient k-Set Consensus 102
9.3 The Set Consensus Hierarchy 104

10 Conclusion **108**

References **111**

LIST OF FIGURES

3.1 Immediate Atomic Snapshot Automaton. 17
3.2 Participating Set Algorithm. 19
3.3 One-shot Immediate Atomic Snapshot Algorithm. 20
3.4 Multi-Shot Immediate Atomic Snapshot. 28

4.1 One-Shot Immediate Atomic Snapshot View Graph. 37
4.2 Atomic Snapshot Memory from Iterated Immediate Snapshots. . . 39

5.1 Iterated Strong Election Algorithm. 50

6.1 Example of Complex Partitioning and Map s. 63
6.2 ϵ-Convergence Specification. 64
6.3 Generalized Simplex Agreement Algorithm. 72

7.1 Agreement Algorithm. 76

ACKNOWLEDGMENTS

I would like to thank my advisor, Eli Gafni, who has taught me by example that interesting results take hard thinking. I would like to thank my brothers for the thesis shepherd and thinking cap. I would like to thank my parents, who have encouraged me to follow my dreams and helped me keep things in perspective when I begin to drown in my work. Lastly, I thank my husband, Neal Lindeman, for his tremendous love, support, and patience – without these things I would not have made it this far. Thank you.

VITA

January 3, 1969	Born, Bethesda, MD
1990	B.A., Mathematics Williams College Williamstown, MA
1993	M.S., Computer Science University of California, Los Angeles
1992–1994	Deans Fellowship School of Engineering University of California, Los Angeles
1992-1995	Research Assistant Department of Computer Science University of California, Los Angeles
1993	Teaching Assistant Consultant Office of Instructional Development University of California, Los Angeles

PUBLICATIONS AND PRESENTATIONS

E. Borowsky, E. Gafni, and Y. Afek, "Consensus Power Makes (Some) Sense!" *Proceedings of the 13th Symposium on Principals of Distributed Computing 1994.*

E. Borowsky and E. Gafni, "Immediate Atomic Snapshots and Fast Renaming," *Proceedings of the 12th Symposium on Principals of Distributed Computing 1993.*

E. Borowsky and E. Gafni, "Generalized FLP Impossibility Result for t-resilient Asynchronous Computations," *Proceedings of the 25th Symposium on Theory of Computing 1993.*

ABSTRACT OF THE DISSERTATION

Capturing the Power of Resiliency and Set Consensus

in Distributed Systems

by

Elizabeth Borowsky

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1995

Professor Eli Gafni, Chair

In an asynchronous distributed system, the solvability of a task depends on the communication model and on the resiliency of the system to failures. Up to now, determining whether a given task is solvable or unsolvable in a system has been approached in an ad hoc manner and often has proven quite difficult. To address this problem, I completely characterize task solvability in read-write shared memory in the presence of failures under two distinct fault models: the model of t-resiliency in which no more than t processors of the system will fail, and the active resiliency model in which a function of the number of active processors in the system may fail. I also generalize the characterization to encompass communication models that use set consensus objects in addition to shared memory. An (n, k)-set consensus object can be used by n processors to restrict the number of inputs to at most k values. In fact, the power of a model to narrow the number of input values (through the resiliency and the communication primitives) is the determining factor in the characterization. Thus, I prove a trade-off between the communication model and the failure model with respect to the power of the given

system to solve (i, k)-set consensus for all i and k. To achieve this characterization, I present a simplified model of read-write memory, a simplex agreement algorithm that may be used as a universal distributed algorithm, and a simulation technique that allows the results results derived in one model to be used in another. The tools and the characterization contribute to the development of a framework for understanding distributed computation.

CHAPTER 1

Introduction

As distributed systems evolve, system designers must carefully evaluate the impact of design decisions in order to develop useful and highly reliable systems. By addressing the question of which tasks a given asynchronous distributed system can solve, this thesis provides a framework that will help distributed system developers make informed design choices. Specifically, it will aid a system developer in evaluating the trade-off in power between different systems and in determining the assumptions necessary for a task to be solvable. In particular, I investigate the impact of the communication medium and the number of processor failures on the set of tasks solvable in a system. The primary tasks I focus on are consensus and set consensus.

A distributed system is a set of processors that coordinate through some communication medium to solve *tasks*. A *distributed task* is a relation between a set of inputs (one for each processor) and the feasible sets of outputs. In the consensus task, processors must choose as output some processor's input such that all processors choose the same value. The consensus task is fundamental for processor synchronization in a system and can be used as a building block for a number of distributed algorithms. In a system of message-passing or shared memory (equivalent by [5]), the consensus task is solvable if no processor may fail-stop. In 1995, Fischer, Lynch and Paterson [16] proved that this is not true if even one processor may fail. The obvious question, then, is which tasks are solvable in message-passing or shared memory if one processor may fail? In 1988, Biran, Moran, and Zaks [7] answered this question, completely characterizing the tasks solvable in the face of one failure. Generalizing naturally, researchers then asked what tasks are solvable in the face of t failures.

In 1990, Soma Chaudhuri [12] took the first major step toward such a generalization. She reasoned that since proving the impossibility of consensus in the face of one failure was a precursor to characterization of that model, a task that is solvable in the face of $t-1$, but not t, failures would provide insight into the t failure model. She proposed the task of k-set consensus, a generalization of the consensus task. In k-set consensus, a processor chooses the input of some processor such that the number of distinct output values chosen is at most k. Consensus is the special case of 1-set consensus. Chaudhuri proved that k-set consensus is solvable in the face of $k-1$ failures and she conjectured that it is not solvable in the face of k. Here, I prove Chaudhuri's conjecture correct and then give a complete characterization of the tasks solvable when t processors may fail. Indeed, I will give a complete characterization of the tasks solvable in general failure models with communication models of shared memory plus communication objects that solve k-set consensus.

I turn first to the *wait-free* model in which up to N out of a system of $N+1$ processors may fail. In this model every processor must work at its own speed without waiting for any other processor to take a step; thus, wait-free algorithms are simpler than algorithms for a general model where t processors may fail. I introduce a simplified model of wait-free read-write memory called the *immediate atomic snapshot* model. This model is useful as it gives rise to a highly structured protocol graph. I exploit this structure to prove that N-set consensus is impossible in a wait-free system of $N+1$ processors using immediate atomic snapshot memory. In addition, as Herlihy and Shavit [19, 20] first noted, task solvability relies on the existence of a map between the protocol graph and a graph derived from the task outputs. Thus, identifying the protocol graph structure of a model is the first step towards determining the set of tasks solvable in the model. I give such a characterization of the tasks solvable wait-free in the read-write shared memory model.

To complete the characterization of the wait-free read-write model and to extend the characterization of solvability in shared memory to the general model of t failures, I introduce two new techniques. The first is a simulation technique, used as a reduction between models. The structure of a protocol graph in the t fault model is deduced through the simulation. Thus, following [19], in order for a task to be solvable in the model, there must be a map between a graph with the given structure and the task output graph. To show the graph and map to outputs sufficient, I present a second technique, a general simplex agreement algorithm. The simplex agreement algorithm for a given model allows processors to converge over any graph with structure corresponding to the model. Processors may then exploit the necessary map from this graph to the output graph to obtain outputs for the task. For the special case of wait-free read-write computation such an algorithm was first given by [20].

The techniques of simulation and simplex agreement apply not only to t-resilient read-write models but also to models where the number of processor failures is a function of the number of processor active in the system (instead of a constant t) and also to models with communication primitives that solve the set consensus task. The protocol graph structure of both of these types of models relies entirely on the *narrowing* power of the model: namely, given i, the minimum number $k(i)$ for which i processors in the model can solve $k(i)$-set consensus. This *narrowing* function $k(i)$ induces a trade-off between the failure model and the communication model with respect to the task-solving power of a system.

In addition to completely characterizing task solvability in systems with general failure models and communication through shared memory and set consensus objects, I use the techniques presented to derive specific results about the solvability of n processor k-set consensus. After proving Chaudhuri's conjecture that n processor k-set consensus is impossible when k processors may fail in a system of shared memory, I will prove exactly when a system with m processor j-set consensus communication primitives can (and cannot) solve the n processor k-set consensus task. This leads to a hierarchy among set consensus objects.

Characterization of the model of set consensus communication in addition to read-write shared memory with either a constant number of failures, or the number of failures a function of the processors active in the system, provides a major advance in the theory of distributed computing. As will be proven, there is a precise trade-off between the communication power and the resiliency to failures needed for task solvability in a system. Thus, a distributed system designer now has a well-defined choice in building a system. The designer may opt to implement strong communication primitives in the hardware of the system and settle for less reliable processors, or the designer may use very reliable processors and keep the communication primitives simple. In addition, this thesis shows that there is a hierarchy among set consensus objects and contributes new tools to the field. Thus, although achieving a characterization of the most general distributed system may be a long way off, this work lays the foundations of a theoretical framework for determining the computational power of any distributed system.

CHAPTER 2

Model

In this chapter, I define the central concepts used throughout this thesis. Most fundamental are the concepts of distributed systems and distributed tasks.

2.1 Distributed Systems

Definition A *distributed system* is specified by a number of processors and a model.

Throughout this work I will assume a system of $N+1$ processors, labeled by distinct IDs $0, ..., N$.

Definition A *model* of distributed computation is specified by a communication model and a scheduler model.

Definition A *communication model* is specified by a possibly infinite state Mealy machine.

The Mealy machine $\langle I, O, S, \delta \rangle$ is specified by a set of inputs I, a set of outputs O, a set of states S, and a transition function δ from a state and an input (or set of inputs) to the corresponding next state and output (or set of outputs). The inputs correspond to the communication operations of processors (for example, a read of shared memory), and the outputs to the communication operation response (the value read).

Definition A communication model is *linearizable* [21] if δ is defined to map a state and a single input into a state and a single output.

Definition A communication model is *set-linearizable* [29] if δ is defined to map a state and a set of inputs into a state and a set of outputs.

Definition An *object* is a linearizable communication model.

Definition An object is *deterministic* if the corresponding Mealy machine is deterministic.

When two communication models are combined, the resulting model can be specified by the cross-product of the respective Mealy machines.

Definition The *state* of a system, or *global state* is the state of the communication model Mealy machine plus the local state of each processor in the system.

Definition A *scheduler* interfaces between a set of processors and a Mealy machine to dictate the order in which requests (or sets of requests) from the processors are submitted to the state machine.

For a linearizable communication model, the scheduler continually chooses the next operation (of those requested by the processors) to submit to the Mealy machine and subsequently return the response. For a set-linearizable communication model, the scheduler chooses sets of operations requested by the processors to submit to the Mealy machine. Upon receiving the set of outputs, the scheduler distributes the appropriate output to each processor of the set and then chooses a new set of requests to service. Operations submitted in the same set by the scheduler are said to be *concurrent*. I assume the behavior of each processor is *well formed* in that a processor alternates between requests and returns, starting with a *Request*. For a given communication model, a sequence of input requests submitted to the Mealy machine and the subsequent responses is said to be an *execution* or a *run*. In this work I consider only deterministic Mealy machines, so the execution is specified simply by the sequence of requests. Each execution prefix determines a global state of the system.

Definition Global states s_1 is an *immediate predecessor* of global state s_2 if there exist execution prefixes S_1 and S_2 that (respectively) determine s_1 and s_2 such that S_2 is sequence S_1 appended by a single request.

Definition Two global states are *consecutive* if one is an immediate predecessor of the other.

We assume the system is *asynchronous*, namely, that the only restriction placed on a scheduler is the number of processors it may ignore indefinitely. Thus, there is no bound on the number of times a given processor may be scheduled before another is scheduled. Different restrictions on the scheduler model different fault assumptions of a system.

2.2 Distributed Tasks

Each processor of a system is assumed to start with a private value called an *input*. A processor's input may be its ID, its initial state, or simply an arbitrary value. Processors coordinate in order to execute *tasks*.

Definition A *task* is specified by a set I of input sets, a set O of output sets, and a relation Δ between input sets $S \in I$ and sets of output sets $T \subseteq O$. The elements of an input set (respectively an output set) are [ID, input value] pairs (respectively [ID, output value] pairs) such that, for each distinct ID i, there is at most one pair [i, value] in a given input or output set. Moreover, $\forall S \in I,\ [i, \text{input value}] \in S,\ T \in \Delta(S)$, the output set T contains an element [i, output value].

In words, for each input set, a task determines the set of output sets compatible with that input. Each input set contains at most a single input per processor, and a compatible output set contains one output for each processor in the input set. Because some processors may take outputs before others expose their inputs, the task is specified such that for any input set $S \subset S'$, each set $T \in \Delta(S)$ is contained in some set $T' \in \Delta(S')$. Thus, an output set is *extendible* to some output compatible for an extension of the input. A processor's output is said to be its *decision value*. I will assume that once a processor decides on its output it writes the value and halts. Thus, once a processor has decided, it remains decided (with that output value) from that point on. Tasks are inherently *one-shot* since all processors have a single input and must obtain a single output.

The primary tasks studied in this thesis are the consensus task and its generalization, the k-set consensus task. In *consensus*, each processor decides on the input of some processor such that all processors decide on the same input. Formally, for each input set $S = \{[i, input_i], ..., [j, input_j]\}$, and, for all k such that $[k, input_k] \in S$, the set $\{[i, input_k], ..., [j, input_k]\} \in \Delta(S)$. For such an output set, processor k and $input_k$ will be referred to as the *winner* of the consensus.

A generalization of the consensus task, proposed by Chaudhuri [12], is the task of k-set consensus. In *k-set consensus*, each processor chooses the input of some processor such that at most k inputs are chosen. The consensus task is simply the special case of 1-set consensus.

2.3 System Resiliency

Definition A processor is *faulty* if it takes a finite number of steps in an infinite execution without halting.

For example, if the scheduler ignores a processor request indefinitely, that processor is considered faulty. If the scheduler is restricted so that no processor request may be ignored indefinitely, the system is said to be totally reliable. Unless there are no requests pending, the scheduler is required to schedule a request.

Definition A system is *t-resilient* if the scheduler may ignore up to t processors indefinitely.

In the t-resilient model, at most t processors of the system may fail. Thus, since $N+1-t$ processors must be scheduled eventually, each processor can repeatedly read the memory until it sees the write operations of $N+1-t$ processors before continuing. A processor that continually reads until the return value satisfies a given property is said to *wait* for the property to hold. Since processors in the t-resilient model may wait until the inputs of at least $N+1-t$ processors are written, a task input-output relation for this model is defined only for sets of size at least $N+1-t$.

Definition An $N+1$ processor system is said to be *wait-free* if it is N-resilient.

In a wait-free system, all processors but one may fail, so a processor may not wait to see an operation of any other processor.

Although most research has been aimed at the t-resilient model ([12, 7, 16] for example), this model is not suitable for some applications. In the t-resilient model a processor that takes no steps at all is considered faulty. However, if processors are coordinating to allocate shared resources, then it may be that only a subset of the processors need to participate in the protocol (those that require the shared resource). In light of this, a sleeping processor is not necessarily faulty and the number of processors that may execute a task is a variable. Thus, for a variable number of processors in a task, why should the number of possible faults be a constant? To reflect this view I define the *active resiliency* model, where for i processors active in an execution, the system is resilient to $g(i)$ failures, for some function $g(i)$. First, however, I will define when a processor is considered active.

Definition Given a global state of a system, a processor is said to be *sleeping* if it has taken no steps.

Upon taking a step a sleeping processor is said to *wake up*.

Definition Given a global state of a system, a processor is *active* if it is neither sleeping nor decided.

Definition A system is said to be *g-active resilient* for some function $g : N \to N$ if, out of i active processors, the scheduler may ignore at most $g(i)$ indefinitely.

To be on intuitively sound ground, we assume that the function g is nondecreasing and that it increases by at most one with each increase in i.

Notice that in the active resiliency model, the scheduler may ignore a sleeping processor indefinitely. In such a model, the total number of processors in the system need not be known unless the function g relies on this number.

I will study both the active resiliency and t-resiliency models, beginning with the wait-free model. The wait-free model bridges the gap between the active and t-resiliency models, as it can be viewed either as the N-resilient model in a system of $N+1$ processors or as the g-resilient model with $g(i) = i - 1$.

2.4 Solvability

Definition A task $\langle I, O, \Delta \rangle$ is *solvable* in a system if there exists a protocol such that each processor halts with an output decision after a finite number of steps, and, for any global state of any execution of the protocol where the set of awake processors and their inputs constitute input set S, the decided processors and their outputs constitute a set T such that $T \subseteq T'$ for some $T' \in \Delta(S)$.

Definition Two systems are *equivalent* if any task solvable in one is solvable in the other.

Definition System A is said to *implement* system B if there exists a map θ from global states of A to global states of B that satisfies the following conditions [1].

1. For any consecutive global states b, b' of system B there exist consecutive global states a, a' of system A such that $\theta(a) = b$ and $\theta(a') = b'$.

2. For any consecutive global states a, a' of system A, either $\theta(a) = \theta(a')$ or $\theta(a), \theta(a')$ are consecutive global states of system B.

2.5 Communication Models

The most common communication models in distributed systems are message-passing and read-write shared memory. These have been proven equivalent [5], thus without loss of generality I assume processors communicate through a base model of read-write shared memory.

Even within the shared memory model, however, there are varying levels of complexity. For example, are memory registers single-writer or multi-writer, single-reader or multi-reader? How much data can a memory register hold: a single bit, a word, possibly an infinite amount? Can a processor read the entire memory in a single step (an atomic snapshot) or only a single register? A long line of research focuses on determining the equivalence of various shared memory models ([23, 33, 11] among others). In 1986, Lamport [23] took the first step, showing that single-bit registers can implement multi-bit registers. Singh, Anderson, and Gouda [33] followed, providing an implementation of single-writer multi-reader registers from single-writer single-reader registers. Most recently, Afek et al. [2] showed that a model of single-writer multi-reader registers is equivalent to atomic snapshot shared memory, in which a read returns not just the value of one register, but the state of the entire memory at a single point in time.

Since the focus of this thesis is computability, not complexity, I will assume memory registers of infinite size. To avoid processors overwriting the operations of other processors, I assume single-writer registers. Thus, for an $N+1$ processor system, let the memory be composed of $N+1$ single-writer cells $C_0, ..., C_N$, one for each processor. Initially I assume atomic snapshot read, although in the next section I will simplify read-write shared memory one step further by introducing the *immediate atomic snapshot* model in which the write and atomic snapshot read operations are fused into a single step.

In addition to shared memory, the communication model may include *set consensus objects*. Processor i interacts with an object via input action $submit_i(value)$ and output action $receive\_response_i(value)$. The submit action enables the object's transition corresponding to the input and the processor. We assume that a processor's behavior in an execution is well formed in the sense that each submit is followed immediately by the corresponding response.

An (n,k)-*set consensus object* is a linearizable deterministic object that returns to each of the first n *submit* requests some input value submitted such that at most k distinct inputs are returned. Any request linearized after the first n receives the response *null*. Functionally, then, an (n,k)-set consensus object solves the k-set consensus task among a maximum of n processors. To abbreviate, we will call an $(n, 1)$-set consensus object simply an n-consensus object. Since an n-consensus objects are linearizable, and the single winning input is among those submitted, the first processor to be linearized necessarily wins the consensus. When a system has access to (n, k)-set consensus objects, we assume an unlimited number of such objects are available to the processors.

CHAPTER 3

Immediate Atomic Snapshots

In this chapter I will strengthen shared memory models one step beyond atomic snapshot memory. Instead of separate memory operations for read and write steps, I propose a model with only a single memory access operation, *WriteRead-Request(v)*. The *WriteReadRequest(v)* of processor i has the effect that the value v is written into memory cell C_i, and an atomic snapshot of the memory is returned. The immediate atomic snapshot model is set-linearizable, with the effect that the writes of concurrent *WriteRead* operations occur before the snapshot reads of the accesses. Thus, concurrent operations return identical snapshots. If two operations are not concurrent, the read of one occurs strictly before the write of the other. The model is called *immediate atomic snapshot* because the reads occur immediately after the writes.

Formally, following [2], immediate atomic snapshot memory for $N+1$ processors can be specified as an automaton A. Processor i can access A through a single input action $WriteReadRequest_i(v)$ and through a single output action $WriteReadReturn_i(C_0, ..., C_N)$. The $WriteReadRequest_i(v)$ is a request by processor i to write v into cell C_i memory, after which a snapshot is immediately returned to processor i. This snapshot is held as the data $C_0, \ldots, C_N$ returned by $WriteReadReturn_i(C_0, ..., C_N)$.

The automaton has a single type of internal action, $Update_S$ for all $S \subseteq \{0, ..., N\}$. Each state of the automaton contains two $(N+1)$-entry arrays: $Mem =$

$\mathbf{WriteReadRequest}_i(v)$	
Effect:	$H_i := \mathbf{WriteReadRequest}_i(v)$
$\mathbf{Update}_S$	
Precondition:	$\forall i \in S$
	$H_i = \mathbf{WriteReadRequest}_i(v_i)$
	$Mem = (C_0, \ldots, C_N)$
Effect:	$\forall i \in S \ C_i := v_i$
	$\forall i \in S \ H_i := \mathbf{WriteReadReturn}_i(\bar{C}_0, \ldots, \bar{C}_N)$
	where $\bar{C}_j = \begin{cases} v_j & \text{if } j \in S \\ C_j & \text{otherwise} \end{cases}$
$\mathbf{WriteReadReturn}_i(C_0, \ldots, C_N)$	
Precondition:	$H_i = \mathbf{WriteReadReturn}_i(C_0, \ldots, C_n)$
Effect:	$H_i := \perp$

Figure 3.1: Immediate Atomic Snapshot Automaton.

$[C_0, ..., C_N]$, representing the $N+1$ cells of the memory, and $Interface = [H_0, ..., H_N]$, representing the $N+1$ interface variables. Each interface variable may hold as its value any interface action or a special value $\perp$. As shown in Figure 3.1, each $Update_S$ action takes a set of processors S, all requesting a *WriteRead* operation, and updates the memory Mem to reflect the new value for each processor in S. The automaton then returns the updated memory to each processor in S. Thus each update operation consists of a set of processors writing concurrently and then, immediately after the writes are finished, reading concurrently. Each set S chosen for an *Update* action is called a *concurrency class* since the corresponding operations occur concurrently.

An automaton M (multi-shot) implements A if M has the interface, input, and output actions of A and every well-formed behavior of M is also a behavior of A. M one-shot implements A if M has the interface, input, and output actions of A and each processor makes a single *Request* action followed by a single *Return*; M K-shot implements A if the single *Request-Return* is replaced by K *Request-Returns*. We will show first that the immediate atomic snapshot automaton A

can be one-shot implemented by an $(N+1)$-process single-writer atomic snapshot shared memory.

3.1 One-Shot Immediate Atomic Snapshots

The crucial difference between immediate atomic snapshot memory and atomic snapshot memory is that the write and read operations are fused into a single step. Thus, in the one-shot immediate atomic snapshot, if a processor i reads the input of processor j, then the operation of j is either concurrent with the operation of i or it precedes the *WriteRead* operation of i. We state the one-shot immediate atomic snapshot as the following *participating set* problem.

The Participating Set Task: Each processor i writes its ID into shared memory and returns a set of IDs, S_i, subject to the following conditions:

1. *Self-Containment*: $i \in S_i$.
2. *Atomic Snapshot*: For all i, j, either $S_i \subseteq S_j$ or $S_j \subseteq S_i$.
3. *Immediacy*: For all i, j, if $i \in S_j$ then $S_i \subseteq S_j$.

The first two conditions are satisfied by processors writing and then reading in atomic snapshots, but the third condition is not. This can be seen when the first processor i to write is the last to read. In this scenario, processors may obtain snapshots that contain processor i but do not contain i's snapshot. Given the self-containment and atomic snapshot conditions, the immediacy condition implies that if $i \in S_j$ and $j \in S_i$ then $S_j = S_i$. This can be used as an alternate statement of the immediacy condition.

We now show the participating set task is solvable in a wait-free system of read-write shared memory. The algorithm follows the idea of Peterson's mutual

```
Shared Variables:
  array level[0, ..., N] of integer, initially ⊥;    (*flag array*)
Local Variables:
  array SS[0, ..., N] of integer, initially ⊥;       (*snapshot*)
  Set: set of integer, initially ∅;
ParticipatingSet(i): Returns Set;
  level[i] := N+2;
  Repeat                                              (*drop levels*)
      level[i] := level[i] − 1;
       SS := Snapshot{level};
       Set := ∅;
      For j = 0 to N Do
          If SS[j] ≤ level[i] Then;
              Set := Set ∪ {j};
          Fi;
      Od;
  Until |Set| ≥ level[i];
Return Set;
```

Figure 3.2: Participating Set Algorithm.

exclusion algorithm [30] in which processors drop flags down a series of levels. Each processor begins by posting a flag at level $N+1$. A processor at level i drops the flag to level $i-1$ if it sees less than i flags on levels 1 through i. Thus a processor stops on a level if it cannot drop its flag any further. When a processor stops at level i, it sees exactly i processors on level i and below (Lemma 3.1.1). These processors are taken to be the participating set.

The participating set algorithm is presented in Figure 3.2. It is used as the main subroutine to the one-shot immediate atomic snapshot algorithm, presented in Figure 3.3. To implement a one-shot immediate atomic snapshot given any implementation to the participating set problem, processors write their values, execute the participating set algorithm, and then take the write values associated with the processors in the participating set returned.

Lemma 3.1.1 *In the participating set algorithm of Figure 3.2, for all processors*

```
Shared Variables:
  array write-value[0, ..., N] of registers, initially ⊥;
Local Variables:
  array SS[0, ..., N] of registers, initially ⊥;
  Set: set of integer;
OneShotWriteReadRequest(i, v): Returns snapshot;
  write-value[i] := v;
  Set := ParticipatingSet(i);
  For j = 0 to N Do
      If j ∈ Set Then
          SS[j] := value[j];
      Fi;
  Od;
Return OneShotWriteReadReturn(SS);
```

Figure 3.3: One-shot Immediate Atomic Snapshot Algorithm.

i, $|Set| = |\ \{j|SS[j] \leq level[i]\}\ | \leq level[i]$.

Proof: Assume to the contrary that $|\{j|SS[j] \leq level[i]\}| = |Set| > level[i] = k$. Since processors drop their flags one level at a time, at least $k+1$ processors must have dropped from level $k+1$ to level k. Thus, let i be the k'th $+1$ processor on level $k+1$ to take a snapshot. This processor must obtain a snapshot $SS[0, ..., N]$ such that $|\{j|SS[j] \leq level[i]\}| \geq level[i] = k+1$. Therefore, processor i will not drop to level k, contradicting the assumption that $k+1$ processors occupy levels 1 to k, and so the lemma holds. ∎

Corollary 3.1.2 *A processor halting on level i obtains a participating set of exactly i processors.*

Proof: By Lemma 3.1.1 there are less than or equal to i processors on levels 1 to i. By the algorithm, however, a processor halts on level i only if there are greater than or equal to i processors on levels 1 to i. So, upon halting at level i, a processor returns a set of exactly i processors. ∎

Lemma 3.1.3 *The participating set algorithm is correct.*

Proof: We show that the three participating set conditions (self-containment, atomic snapshot, and immediacy) are satisfied. Self-containment is satisfied since a processor halting with a flag on level i will take as output all processors with flags on levels 1 through i. For the atomic snapshot condition consider the output sets of processors i and j halting on levels l_i and l_j. Without loss of generality, let $l_i \geq l_j$. By corollary 3.1.2 there are exactly l_i processors on levels 1 to l_i and exactly l_j processors on levels 1 to l_j. Since no processor ever moves its flag up, the snapshot of processor i contains that of processor j. For the immediacy condition, notice that if processor i is in the set of processor j, then processor i resides on a level less than or equal to that of processor j. Thus the set of processor i is contained in the set of processor j. ∎

Since the participating set algorithm solves the participating set task, it is easy to see that the one-shot immediate atomic snapshot algorithm (Figure 3.3), which simply adds write values to the participating set output, solves the one-shot immediate atomic snapshot task.

3.2 K-Shot Immediate Atomic Snapshots

Generalizing the one-shot immediate atomic snapshot algorithm to a K-shot implementation is simple. If each processor is to make K consecutive immediate atomic snapshots, we need only increase the number of levels in the one-shot immediate atomic snapshot algorithm to compensate for the additional snapshots.

Thus, each processor starts at level $K(N+1)$, dropping flags by the same rule as before. Once an immediate snapshot is completed, a processor simply uses a new flag for the next immediate atomic snapshot, starting again at level $K(N+1)$. It is easy to see that this algorithm satisfies the immediate atomic snapshot requirements. This algorithm is unsatisfactory, however, since the time it takes for a given immediate atomic snapshot is a function of the total number of snapshots to be taken. In the next section we will show that the immediate atomic snapshot model can be implemented in atomic snapshot read-write memory even when there is no bound on the number of immediate atomic snapshots to be taken.

3.3 Multi-Shot Immediate Atomic Snapshots

The difficulty in extending the idea of the one-shot algorithm to the multi-shot case is that processors must determine a finite level from which to begin dropping flags. Further, processors must reserve space for the processors that may begin dropping at lower levels. To achieve these two constraints I impose an external structure on the array of levels by associating levels with atomic snapshots. An atomic snapshot containing i write values is associated with level i. Since atomic snapshots can be ordered by containment, every level will be associated with at most one snapshot. Processors will determine a range of levels in which to negotiate for an immediate atomic snapshot by writing, taking atomic snapshots, and posting the snapshots. This range falls between the highest level with a snapshot containing the processor's previous write and the lowest level containing the processor's current write. Thus, the processor determines the immediate atomic snapshot by combining information from the flags and the atomic snapshots associated with the levels. We now present the algorithm in detail (figure 3.4).

Following the one-shot implementation, we have an array of levels, each of which may become associated with a snapshot. The first step in each immediate atomic snapshot access associates a snapshot with a level. A processor begins by writing its value, taking an atomic snapshot, and writing down the snapshot. A snapshot containing i operations is associated with level i. After reading the snapshots, a processor begins dropping from the level of the the smallest snapshot containing its current write value. Since the snapshot just written by a processor contains its current write, such a snapshot exists.

The atomic snapshots associated with the levels form a framework, dictating the range in which processors may negotiate for an immediate atomic snapshot. Intuitively, a processor negotiates for an immediate snapshot on the levels between the smallest snapshot containing its current write and the largest snapshot containing only its previous write. Due to the asynchrony inherent in the model, this range of levels may change over time; thus, when dropping through a level, a processor must record whether or not it sees a snapshot associated with that level. To do this, a processor executes a one-shot immediate atomic snapshot on each level, writing its current write value and whether or not it saw a snapshot associated with that level. A processor may drop to the next level if either no processor in its one-shot snapshot sees a snapshot associated with the level or the associated snapshot contains the processor's current write value. Thus, a processor halts on a level when some member of its one-shot immediate snapshot sees that the snapshot associated with the level contains only the processor's previous write. A processor may then take as output a snapshot corresponding to the union of the atomic snapshot associated with the level and the one-shot immediate snapshot taken on the level. Due to the fact that processor i leaves a record of its current write as it descends levels (through the one-shot snapshots), the snapshot of any processor halting on a level that processor i has dropped from will contain the current operation of i.

Since the range of levels a processor proceeds through in any memory access is at most N, at most N one-shot immediate atomic snapshots are taken. Thus, the time needed to implement a single memory access is independent of the total number of accesses. For ease of implementation, a snapshot is stored as an $N+1$ element vector SS, with $SS[i]$ holding the number of writes processor i has completed in the snapshot. The actual snapshot can be determined by reading the write value array, an infinite array holding all write values of the processors. The algorithm is presented in full in Figure 3.4.

Lemma 3.3.1 *In any execution of the multi-shot immediate atomic snapshot algorithm, the final snapshots of processors that terminate on the same level can be ordered by containment.*

Proof: For processors i and j that terminate on the same level, let the one-shot immediate atomic snapshots returned on that level be IAS_i and IAS_j, respectively. Assume without loss of generality that IAS_i contains IAS_j. The atomic snapshot associated with the level is unique. Thus, since the final immediate atomic snapshots of i and j are determined by combining all operations of the level snapshot with operations of IAS_i and IAS_j, respectively, the final snapshot of processor i will necessarily contain the final snapshot of processor j. ∎

Lemma 3.3.2 *In any run of the multi-shot immediate atomic snapshot algorithm, the final snapshots of processors terminating on different levels can be ordered by containment.*

Proof: Let processors i and j halt at levels l_i and l_j such that $l_i > l_j$. Since snapshots associated with levels can be ordered by containment, SS_{l_i} contains SS_{l_j}. Thus, in order for the snapshot returned by processor i to fail to contain the snapshot of processor j, there must be an element $p_k \in IAS_j$ such that $p_k \notin IAS_i$ and $p_k \notin SS_{l_i}$. However, if $p_k \notin SS_{l_i}$, then to reach level l_j p_k must have taken an immediate snapshot on level l_i. Since $p_k \notin IAS_i$, it holds that $IAS_i \subset IAS_p$. Since i halts on level l_i, an element of IAS_i sees the snapshot associated with l_i and, thus, so does an element of IAS_p. Since $p_k \notin SS_{l_i}$, processor p must have halted at this level and cannot appear in the snapshot of processor j, contradicting the assumption that the final snapshots are not ordered by containment. ∎

Lemma 3.3.3 *For each immediate atomic snapshot access, a processor accesses at most N levels.*

Proof: To determine the range of levels, a processor reads the atomic snapshots taken by all processors, taking the smallest snapshot containing its current write value to be its initial level. Since there are only $N + 1$ processors, there can be at most N unwritten snapshots that lie between this snapshot and the next smallest snapshot written — one that does not contain the processors current write value. Thus, the processor will traverse through at most N levels for each immediate atomic snapshot access. ∎

Lemma 3.3.4 (Self-Containment) *WriteReadRequest(i, $write_i^k$) returns a snapshot that contains $write_i^k$.*

Proof: When a processor executing its k'th memory access halts at a level, its final snapshot is the union of the atomic snapshot associated with the level and its one-shot immediate atomic snapshot. Since the k'th write of the processor is necessarily contained in its one-shot immediate atomic snapshot, this write will appear in its final snapshot. ∎

Lemma 3.3.5 (Atomic Snapshot) *Any two snapshots SS_i and SS_j that are returned by WriteReadRequest($write_i^k$) and WriteReadRequest($write_j^m$) can be ordered by containment.*

Proof: Lemmas 3.3.1 and 3.3.2, any two snapshots returned, whether the processors halt on the same or different levels, can be ordered by containment. ∎

Lemma 3.3.6 (Immediacy) *Given snapshots SS_i and SS_j returned by operations WriteReadRequest($write_i^k$) and WriteReadRequest($write_j^m$), if $write_i^k \in SS_j$ then $SS_i \subseteq SS_j$.*

Proof: Let l_i and l_j be the halting levels of processors i and j, respectively, and assume $SS_i \not\subseteq SS_j$. Then, by Lemma 3.3.2, if $l_i \neq l_j$ then $l_i > l_j$. Thus, assume $l_i > l_j$ and $write_i^k \in SS_j$. This is a contradiction since the snapshots of level $l_i > l_j$ must contain only $write_i^{k-1}$ (since i halted at this level) and also must contain the snapshot of level l_j. Thus, it must be that $l_i = l_j$. Since i halts on this level, only $write_i^{k-1}$ appears in the associated snapshot; thus, for $write_i^k$ to appear in SS_j, $IAS_i \subseteq IAS_j$. This contradicts the assumption that $SS_i \not\subseteq SS_j$, and the lemma holds. ∎

Corollary 3.3.7 *The algorithm of Figure 3.4 implements multi-shot immediate atomic snapshots.*

Proof: By Lemmas 3.3.4, 3.3.5, and 3.3.6, the self-containment, atomic snapshot and immediacy properties are satisfied. By Lemma 3.3.3 and the fact that the protocol is wait-free, each processor obtains a snapshot after a finite number of steps. Thus, the algorithm correctly implements multi-shot immediate atomic snapshots. ■

It is easy to see that the executions of the immediate atomic snapshot model form a subset of the executions of the single-writer atomic snapshot-reader shared memory model. Thus, the immediate atomic snapshot model is at least as powerful as read-write shared memory. Conversely, the multi-shot immediate atomic snapshot model can be implemented by the read-write shared memory model, proving read-write shared memory is at least as powerful as immediate atomic snapshot memory. Thus, the two models are equivalent. To investigate solvability in shared memory it suffices to focus on the immediate atomic snapshot model. In the next chapter, I will show that a highly structured *view graph* can be derived from a variant of the wait-free immediate atomic snapshot model. This structure will pave the way for a characterization of task solvability in the wait-free read-write model, which in turn will lead to characterization of more general models.

```
Shared Variables: array write-value[0..N, 1..∞] of registers;
  array write-number[0..N] of integers;
  array LevelSS[1..∞, 0..N] of integers;
  array SnapSeen[1..∞, 0..N] of boolean;
Local Variables: done: boolean;
  level: integer;
  array currentSnap, OneShotSnap[0..N] of integers;
  array finalSnap[0..N] of registers;

WriteReadRequest(i, write_i^k): Returns(snapshot);
  write-value[i, k] := write_i^k;
  write-number[i] := k;
  currentSnap := AtomicSnapshotRead{write-number};
  LevelSS[∑_{j=0}^{N}(currentSnap[j])] := currentSnap;
  level := min{j| LevelSS[j, i] = k};
  Repeat                                  (*drop levels*)
      level := level −1;
      SnapSeen[level,i] := Not (LevelSS[level] =⊥);
      OneShotSnap := OneShotWriteReadRequest(i, level, k)
      For j = 0 to N Do
          If (j ∈ OneShotSnap) ∧ (SnapSeen[level,j] = true) ∧ (LevelSS[level,i] < k)
          Then done := true;
          Fi;
       Od;
  Until done;
  For  j = 0 to N Do
       If j ∈ OneShotSnap Then
            finalSnap[j] := write-value[j,OneShotSnap[j]];
      Else
          finalSnap[j] := write-value[j,LevelSS[level,j]];
       Fi;
  Od;
Return WriteReadReturn(finalSnap);
```

Figure 3.4: Multi-Shot Immediate Atomic Snapshot.

CHAPTER 4

View Graph Structure of Read-Write Memory

To study computation in the immediate atomic snapshot model, I focus on *full-information protocols* [17], in which processors write their entire local state in every write step. Initially, the local state of a processor is simply its input. After a *WriteRead* step, a processor's local state is determined by adding information returned by the read to the processor's previous state. Following [17], we will refer to the state of a processor upon halting as its *view* in the protocol. Since we concentrate on full-information protocols, the protocol need only determine which local states are halting states and to what output each halting state corresponds. Thus, a protocol determines a map from halting states to output values. The existence of such a map is crucial for determining the solvability of a task. Whether such a map exists depends on the structure formed by the halting states and the structure formed by the task outputs. In this chapter, I determine the structure induced by processor halting states in the wait-free immediate atomic snapshot memory model.

4.1 Runs and Views

Given a wait-free protocol, the execution determines the precise number of steps a processor will take before halting. Let n_i be the greatest number of steps processor i takes in any execution of the protocol. Since a processor may continue writing and reading without affecting its decision value (it simply ignores what it reads), we assume without loss of generality that in every execution processor i takes n_i steps and then halts.

In a full-information immediate atomic snapshot protocol, a particular execution is totally specified by the processor inputs and the sequence of concurrency classes $C_1, ..., C_m$. Thus, for a single input protocol over a set of processors I, a run may be denoted as a sequence of sets $C_1, .., C_m$ such that for all i, $C_i \subseteq I$, and, for all $j \in I$, the number of sets containing j equals n_j. The set of all such runs is denoted $R_I\{(i, n_i)\}$ for $i \in I$.

Throughout this section, I concentrate on the *view* of a processor in an execution. For processor $j \in I$ and a run $r \in R_I\{(i, n_i)\}$, the view of processor j in r will be denoted $view(j, r)$. The local state $view(j, r)$ is represented by the processor ID followed by a sequence of sets, defined inductively as follows.

Definition The *view* of processor j in $r = C_1, ..., C_m$ is:

1. $j : \lambda$ if $\forall i \in \{1, ..., m\}$, $j \notin C_i$, where λ is the empty sequence.
2. The set C_1 followed by the view of j in $r' = C_2, ..., C_m$, if either $j \in C_1$ or $\exists i \in C_1$ such that some set in the sequence of $view(j, r')$ contains i.
3. The view of j in $r' = C_1 \cup C_2, ..., C_m$ otherwise.

Given a view $v = view(i, r)$, the *viewing processor* of v is i. Further, the run r is said to *give rise to* the view v. The view contains only operations that occur concurrently with, or before, the last operation of processor j in the execution. Concurrency classes C_i, C_{i+1} of an execution are merged in a view when the viewing processor cannot determine the order of the classes. This occurs when no member of C_i takes another step until after the final operation of processor j is completed. Thus, no processor of C_i reveals to j that the operations occur in the execution strictly before the operations of C_{i+1}. The view represents all information a processor knows about the linearization of the run.

Two views are said to be *compatible* if they are views of a common run. Lemma 4.1.3 proves that if a set of views are pairwise compatible, then they are mutually compatible — namely there exists a single run that gives rise to all the views. We first state Lemma 4.1.2 in order to streamline the proof of Lemma 4.1.3.

Lemma 4.1.1 *Two distinct views of processor i are not compatible.*

Proof: Since the view of processor i in a run r is a deterministic function of the run, there can be no run that gives rise to two distinct views of processor i. Hence, two distinct views of processor i can not be compatible. ∎

Lemma 4.1.2 *Given two compatible views $i : s_1, ..., s_m$ and $j : s'_1, ..., s'_{m'}$, the sets s_1 and s'_1 are related by containment.*

Proof: Let $r = C_1, ..., C_n$ be a run that gives rise to both views. Assume $i \in r$ and $j \in r$, otherwise the resulting view would be empty and the condition holds trivially. By the definition of view, the first sets of $view(i, r)$ and $view(j, r)$ must be of the form $\bigcup_{i=1}^{k} C_i$ for some k. Thus, the first sets can be ordered by containment. ∎

Lemma 4.1.3 *A set of pairwise compatible views are mutually compatible.*

Proof: We prove the lemma by inductively building a run R, initially λ, that gives rise to every view of the set. Let the first set s_1 of view v_j be the smallest nonempty first set of any view. By Lemma 4.1.2, s_1 is contained in the first set of every nonempty view. Since every view v_i is compatible with v_j, there must exist a run $r_i = s_1, si_2, ..., si_{m_i}$ that gives rise to v_i. Then, either the first set of v_i is s_1, and v_i is simply $view(i, (si_2, ..., si_{m_i}))$ prefaced by s_1, or the first set of v_i properly contains s_1 and, $\forall j \in s_1$, processor j does not appear in any set of $view(i, (si_2, ..., si_{m_i}))$. Thus, any run that gives rise to this view, prefaced by s_1, gives rise to v_i. Moreover, the prefix s_1 is compatible with all views. Inductively then, the views $v_i' = view(i, (si_2, ..., si_{m_i}))$ are compatible, and so, letting R be R followed by s_1, we inductively continue the procedure on the views v_i'. The induction terminates when all views $v_i' = i : \lambda$, and R is a sequence of concurrency classes that gives rise to every view v_i. ■

Lemma 4.1.4 *A set of $N+1$ pairwise compatible views determines a unique run.*

Proof: By Lemma 4.1.3, the views are mutually compatible, si there exists a run that gives rise to all views. By Lemma 4.1.1, each of the views must be a view of a distinct processor. Since we have a run of each processor, each set of the run is determined uniquely by the views of processors in that set (due to condition 2 of the definition of view). Thus, the views together determine the unique sequence of sets that constitutes the single run giving rise to all views. ■

4.2 Topological Basics

In this section, I present the topological preliminaries needed for defining the *view complex* of a wait-free full-information immediate atomic snapshot protocol. The view complex is comprised of nodes and edges, with one node labeled $view(i, r)$ for all processors i and all possible executions r. Notice that a single node may represent the view of processor i in different runs of the protocol. Two nodes of the view complex are adjacent if the corresponding views are compatible. By Corollary 4.1.1, no view of processor i is adjacent to any other view of processor i. Thus, using processor IDs as colors, the view complex of an $N+1$ processor protocol can be properly $(N+1)$-colored. We refer to a node by the view that labels it. Thus, for any node $v = view(i, r)$, we define $ID(v) = i$, the viewing processor of the view labeling the node.

By Lemma 4.1.4, $(N+1)$-cliques in the view complex of a protocol correspond exactly to runs. In the terminology of algebraic topology, such an $(N+1)$-clique is called a *simplex*. The dimension of a simplex is the number of nodes in it minus 1, denoted by the superscript on the simplex. Thus, a simplex s^q corresponds to a $q+1$-clique and will be referred to as a q-simplex. For a simplex of views s^q, we define $IDs(s^q) = \{ID(v) | v \in s^q\}$.

Given a simplex S, a *face* of S is the simplex containing any nonempty subset of nodes $S' \subseteq S$. The face relation will be denoted $S' \leq S$. The view *complex* is then a set of simplexes. A fundamental notion of algebraic topology, a *complex* is formally defined below.

Definition A *complex* is a set of simplexes C such that for any simplex $S \in C$ and any simplex $S' \leq S$, $S' \in C$.

Given a set of simplexes C, the complex *induced* by C is the set of simplexes $C \cup \{S' | S' \leq S \in C\}$. Abusing notation, I will refer to the complex induced by a set C simply as complex C.

Definition A *primary* simplex of a complex is one that is not a proper face of any other simplex.

Definition A complex is *pure* N-dimensional if every primary simplex is of dimension N.

Definition A complex C^N is *colored* if each vertex of the complex is labeled by a color.

Definition A colored complex C^N is said to be $(N+1)$-*chromatic* (or simply *chromatic*) if the underlying graph is properly $(N+1)$-colored (i.e., no two adjacent nodes have the same color).

Thus, with a coloring induced by processor IDs, the view complex for an $N+1$ processor protocol is a pure N-dimensional $(N+1)$-chromatic complex. For an arbitrary chromatic complex, the set of colors corresponding to a simplex s is denoted as $IDs(s)$. For a set of colors $p \subseteq IDs(s)$, the face induced by nodes with colors in p as $face_p(s)$. Alternately, $face_{\bar{p}}(s)$ is the largest face of s containing no vertices with IDs in p. Throughout this section, I assume that all simplexes and complexes of dimension N are $(N+1)$-chromatic.

Every complex has an *embedding* in Euclidean space of high enough dimension so that the vertices of any simplex are affinely independent, and any two simplices intersect in a face or not at all. Without loss of generality a complex may be assumed to be embedded. When we an abstract complex C^N and its embedding must be distinguished, the embedding will be denoted as $|C^N|$. Each simplex in an embedded complex is considered to be all points contained in the convex hull of its vertices.

Definition Complex $|B^n|$ is a *subdivision* of $|A^n|$ if for any simplex $b^q \in B^n$ there is a simplex of $a^q \in A^n$ such that $b^q \subseteq a^q$ and, if $\forall a^q \in A^n,\ a^q$ is the union of finitely many simplices of B^n.

Definition Given a subdivision B^n of A^n, the *carrier* of a simplex $b \in B^n$ with respect to A^n, denoted $carrier(b, A^n)$, is the smallest simplex $a \in A^n$ such that a contains b.

Definition The chromatic subdivision B^n of chromatic simplex S^n is *Sperner chromatic* if for all $s^q \in B^n,\ IDs(s^q) \subseteq IDs(carrier(s^q, S^N))$.

Given the subdivision A^n of an embedded simplex S^n, the notion of a face of S^n can be extended to the subdivision. Formally, $face_p(A^n)$ is the complex induced by the set of simplexes $\{a^q | (a^q \in A^n) \wedge (carrier(a^q, S^n) \leq face_p(S^n))\}$. In words, the face of a subdivided simplex is the induced subdivision of the corresponding face of the simplex. An abstract complex C will be called a subdivided simplex if there is an embedding of C as a subdivided simplex.

Definition The *open star* of $s^q \in C^n$, denoted $\overset{\circ}{st}(s^q, C^n)$, is defined to be the set of simplexes $\{s^p | s^q \leq s^p \in C^n\}$.

Definition The *star* of simplex s^q in complex C^n, denoted $st(s^q, C^n)$, is the complex induced by $\overset{\circ}{st}\,(s^q, C^n)$.

Definition The *linking sphere* of a simplex s^q in complex C^n, denoted $ls(s^q, C^n)$, is the complex $st(s^q, C^n) - \overset{\circ}{st}\,(s^q, C^n)$.

Definition A map $\mu : A \to B$ from complex A to complex B is *simplicial* if for any simplex $s \in A$, $\mu(s)$ is a simplex in B.

A *simplicial map*, is a mapping from vertices of one complex to vertices of another which *preserves adjacency* in that simplexes map to simplexes.

Definition A simplicial map $\mu : A \to B$ is *chromatic* if A and B are chromatic and $IDs(s) = IDs(\mu(s))$ for all $s \in A$.

4.3 Immediate Atomic Snapshot View Graph Structure

In this section, I prove that the view complex of a wait-free immediate atomic snapshot protocol is the simplicial image of a subdivided simplex. I assume a single-input protocol in which each processor i takes exactly n_i steps. For a set of processors I, the complex is denoted $G_I\{(i, n_i) | i \in I\}$.

In 1994, Herlihy and Shavit [20] proved that a variant of the immediate atomic snapshot model gives rise to a view complex that is a subdivided simplex. They propose the *iterated* immediate atomic snapshot model. In this model, each processor i takes n one-shot immediate atomic snapshots and then halts. Thus, an execution in this model is a sequence of n individual executions of the one-shot immediate atomic snapshot algorithm. The view complex of a one-shot immediate atomic snapshot for $N+1$ processors is a simple subdivision of an N-simplex (Figure 4.1). Herlihy and Shavit propose the iterated immediate atomic snapshot model since the view complex for a k iteration protocol is a recursive subdivision of the view complex of a $k-1$ iteration protocol. In other words, such a complex is a subdivided simplex. The face opposite the vertex labeled i in the subdivided simplex ($face_{\bar{i}}(C^N)$) contains all views (of processors other than i) for which none of the n one-shot immediate atomic snapshots contain processor i.

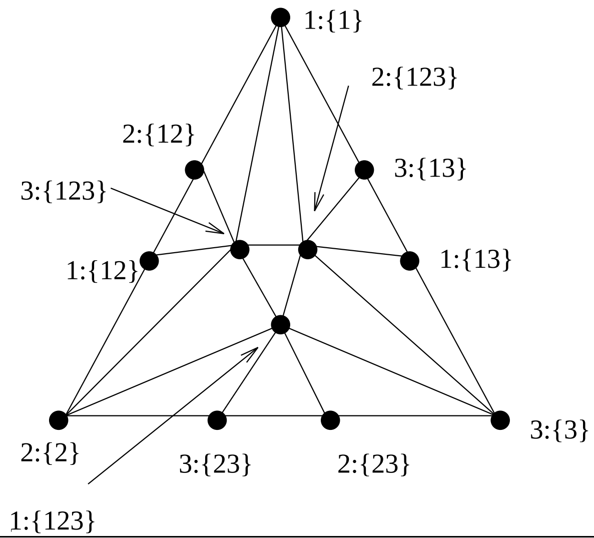

Figure 4.1: One-Shot Immediate Atomic Snapshot View Graph.

In this section, I prove that the iterated immediate atomic snapshot model is equivalent to the read-write shared memory model. I prove that the immediate atomic snapshot model can be implemented in the iterated immediate atomic snapshot model, which means that there exists a simplicial map from the view complex of the iterated model to the view complex of the immediate atomic snapshot model. Thus, the view complex of the immediate atomic snapshot model is the simplicial image of a subdivided simplex. Moreover, I will prove the boundary of the structure is preserved through the simplicial map. Since I have already shown that immediate atomic snapshots may be implemented in atomic snapshot memory, and since here I am concerned only with wait-free protocols in which the number of accesses is bounded, it is sufficient to implement k-shot atomic snapshot memory from iterated immediate atomic snapshot memory. The algorithm follows (Figure 4.2).

A processor initiates a write operation by submitting its input plus its previous snapshot to the i'th one-shot immediate atomic snapshot. Initially $i = 0$ and the snapshot is empty. If the size of the one-shot snapshot returned equals i, the write operation is complete. Otherwise, the processor submits to the i'th $+1$ one-shot immediate atomic snapshot the union of operations returned by the i'th one-shot snapshot. When an operation succeeds at the i'th one-shot snapshot, a processor initiates its next operation at the i'th $+1$ one-shot snapshot.

For a read operation, a processor initiates a write, submitting the special value *null* plus its previous snapshot. The value *null* is used as a placeholder for calculating the size of the snapshot, but it contributes no actual information to the snapshot. When the write of *null* is complete, the processor takes the snapshot returned to be its read value. The protocol is presented in Figure 4.2.

Lemma 4.3.1 *Every set submitted to the i'th one-shot immediate snapshot contains at least i values.*

Proof: For $i = 1$ the lemma holds. Inductively then, assume the lemma holds for $i-1$. Since every processor submits a set of at least $i-1$ values to the i'th -1 one-shot snapshot, a processor must obtain as output a set of $\geq i-1$ values from this snapshot. If the set contains exactly $i-1$ values, the operation is complete, and the processor's next operation will submit these $i-1$ values plus either *null* or

```
Local Variables:
  S: set of values, initially ∅; (*Snapshot*)
  level: integer, initially =1;
Procedure MemoryAccess(i, value, level): Returns Set;
  Repeat
      S := OneShotWriteReadRequest_level{i,value,level};
      value := {op|op ∈ S };
     level := level+1;
  Until |value| = level−1;
Return S;

Protocol(i,input);
  level := 1;
  S := ∅;
  Repeat
      If NextOp = write Then
          S := MemoryAccess(i, S ∪ input, level);
      Else (* NextOp = read *)
          S := MemoryAccess(i, S ∪null, level);
  Until view ∈ HaltingViews;
```

Figure 4.2: Atomic Snapshot Memory from Iterated Immediate Snapshots.

the new write value to the one-shot snapshot at level i. If the set from level $i-1$ contains greater than $i-1$ values, this entire set will be submitted to the one-shot snapshot at level i. Thus, it holds that any set submitted to the i'th one-shot snapshot contains at least i values. ■

Lemma 4.3.2 *The snapshot returned by a memory access that terminates at the i'th one-shot snapshot is unique.*

Proof: Since the i'th one-shot snapshot returns sets that can be ordered by containment, there is a single snapshot that contains exactly i values. ■

Lemma 4.3.3 *The snapshots returned by any two memory accesses may be ordered by containment.*

Proof: Let the operations terminate at one-shot snapshots i and j, and without loss of generality assume $i \leq j$. If $i = j$ then we are done, by Lemma 4.3.2. Thus, assume $i < j$. The set of operations seen in any snapshot received at iteration i contains all operations of a memory access terminating at this iteration. Inductively, all such operations returned will be submitted to the one-shot snapshot of iteration $i+1$. Thus, all sets submitted to one-shot snapshot j contain the snapshot returned by an access terminating at iteration i, and so the lemma holds. ■

Lemma 4.3.4 *The write and read operations are linearizable.*

Proof: A read operation is assumed to take place atomically just as it terminates. A write operation is assumed to take place just before the first read operation that contains it or upon termination, whichever is earlier. Since Lemma 4.3.3 proves that the snapshots returned by all operations may be ordered by containment, this is a valid linearization of the run. ■

Lemma 4.3.5 *The algorithm implements k-shot atomic snapshot memory.*

Proof: At least i operations must terminate within the first $N + i$ one-shot snapshots, so, since the number of total memory accesses is finite, the algorithm terminates. By Lemma 4.3.3 the reads are ordered by containment, and by Lemma 4.3.4 all operations are linearizable, thus atomic snapshot memory is implemented. ■

I now prove that the view complex of immediate atomic snapshot memory is the image of a subdivided simplex. However, I would also like to prove that the boundary is preserved under the simplicial map. For this I proceed first with some definitions.

Definition Simplex $s^{N-1} \in C^N$ is a *boundary* simplex of the pure N-dimensional complex C^N if it is contained in a single N-simplex of the complex.

Definition Simplex $s^{N-1} \in C^N$ is an *interior* simplex of pure complex C^N if it is not a boundary simplex.

Definition A *solo execution* of a set of processors S is an execution that contains only operations of processors in S.

Let the view complex of an iterated immediate atomic snapshot protocol that implements an immediate atomic snapshot protocol be IVC, and the view complex of the immediate atomic snapshot protocol be VC.

Lemma 4.3.6 *There exists a chromatic simplicial map from a subdivided simplex to the view complex VC of an immediate atomic snapshot protocol.*

Proof: By [20] the view complex IVC of an iterated immediate atomic snapshot protocol is a subdivided simplex. By Lemma 4.3.5, there exists such an iterated immediate atomic snapshot protocol that implements any wait-free immediate atomic snapshot protocol. Thus, since compatible views of the iterated protocol implement compatible views of the immediate atomic snapshot protocol, we can define a chromatic simplicial map that takes a final view of processor i in the iterated protocol to the final view of processor i implemented. Thus, the lemma holds. ∎

Lemma 4.3.7 *A simplex s^{N-1} of the view complex VC of an immediate atomic snapshot protocol is a boundary simplex iff it corresponds to a solo execution by processors in $IDs(s^{N-1})$.*

Proof: Without loss of generality, let $IDs(s^{N-1}) = \{0, ..., N-1\}$. If some view of the s^{N-1} contains an operation of processor N, then there are two $N+1$ processor runs compatible with the simplex, depending on whether the last operation of processor N contained in a view is concurrent with or precedes the concurrency class it is contained in with respect to s^{N-1}. Thus, such a simplex must be an interior simplex.

Alternately, if no view of s^{N-1} contains an operation of processor N, then, inductively, by Lemma 4.1.4, s^{N-1} determines a solo execution by processors $0, ..., N-1$. The only view of processor N that is compatible with s^{N-1} arises from the solo execution defined by s^{N-1} followed by n_N operations of processor N. Thus, the lemma holds. ■

Lemma 4.3.8 *A boundary $(N-1)$-simplex of the iterated immediate atomic snapshot view complex IVC maps to a boundary $(N-1)$-simplex of the immediate atomic snapshot view complex VC.*

Proof: Since boundary simplices of IVC see no operation by some processor, they must correspond to views of VC that see no operation of the processor. Inductively, since a face of IVC corresponds to the view complex of a smaller set of processors, and since the map from IVC to VC is simplicial, the face of VC consisting of all views that see no operation of a processor i must be the lower dimensional view complex of a protocol over all processor except i. ■

Theorem 4.3.9 *$G_I\{(i, n_i)\}$ is the image of a subdivided simplex, where, for all $j \in I$, the face opposite j is $G_{I-\{j\}}\{(i, n_i)\}$.*

Proof: By Lemma 4.3.6, there exists a chromatic simplicial map from views of IVC to views of the immediate atomic snapshot view complex $G_I\{(i, n_i)\}$. By Lemma 4.3.8, the boundary of IVC maps to the boundary of $G_I\{(i, n_i)\}$, with $face_{\bar{j}}$(IVC) mapping to $face_{\bar{j}}(G_I\{(i, n_i)\}) = G_{I-\{j\}}\{(i, n_i)\}$. By [20], IVC is a subdivided simplex; thus, $G_I\{(i, n_i)\}$ is the image of a subdivided simplex, whose boundary comprised of the lower dimensional view complexes $G_{I-\{j\}}\{(i, n_i)\}$, which are themselves recursively images of subdivided simplexes. ■

4.4 Necessary Condition for Read-Write Wait-Free Solvability

Based on the structure of iterated immediate atomic snapshot view complexes, I present a necessary condition for task solvability in the wait-free read-write model. Formally, in the terminology of algebraic topology, a task can be represented as a mapping from simplexes labeled by inputs to sets of simplexes labeled by outputs (the legal outputs for those inputs). Thus, following [19], a task is a 3-tuple $\langle I^N, O^N, \Delta\rangle$, where I^N is the chromatic input complex (colored by processor IDs and labeled by inputs), O^N the chromatic output complex (colored by processor IDs and labeled by outputs), and Δ the mapping from each simplex of I^N to the legal output simplexes of O^N determined by the task. If a task is solvable wait-free in the immediate atomic snapshot model, then there must be a chromatic simplicial map from a subdivided simplex to the output complex that satisfies the task specification. The subdivided simplex can be thought of as a subdivision of the input simplex. The map to the output must be simplicial, since each view maps to an output and since compatible views map to compatible outputs. Herlihy and Shavit [19] were the first to propose this necessary condition for solvability, proving it through homology theory [28].

Lemma 4.4.1 ([19, 20]) *If the task $T = \langle I^N, O^N, \Delta\rangle$ is solvable wait-free in read-write memory, then there exists a Sperner chromatic complex C^N that subdivides I^N, and a chromatic simplicial map $\mu : C^N \rightarrow O^N$ such that for $s^q \in C^N$, $\mu(s^q) \subseteq \Delta(carrier(s^q, I^N))$.*

Proof: Consider the case of a single-input task; that is, I^N is a single chromatic simplex. Then, if T is solvable, there must exist a full-information iterated immediate atomic snapshot protocol that solves T, by Lemma 4.3.5. By [20], the view complex of this protocol is a subdivided chromatic N-simplex and so can be viewed as a subdivision C^N of the input simplex I^N. By assumption, the protocol implements T and thus defines a simplicial map μ from C^N to O^N. By [20], any simplex $s^i \in face_P(C^n)$ (i.e., $carrier(s^i, I^N) \leq face_P(I^N)$) contains views that see only the inputs of processors in the set P. Thus, for such a simplex, the outputs $\mu(s^i)$ are contained in $\Delta(face_P(I^N))$, and thus, for I^N a single input, the theorem holds, since for any simplex s^q in C^N, $\mu(s^q) \subseteq \Delta(carrier(s^q, I^N))$.

In the multi-input case, the proof extends verbatim. By [20], the subdivision of any simplex of inputs is the view complex corresponding to that set of inputs. Thus, the protocol induces a subdivision of the entire input complex. Again, the protocol determines the simplicial map μ to output values. Due to fact that the subdivision of any face of an input simplex is a lower dimensional view complex, the carrier property of μ is satisfied, and the theorem holds. ■

CHAPTER 5

Narrowing Power of a Model

As I will prove in Chapters 6, 7, and 8 the power of a model to solve tasks can be captured by the power of the model to solve i processor k-set consensus for all i and k. Here, I present the $(i, g(i))$-narrowing task, which specifies, for i active processors, the number of distinct outputs ($g(i)$) that they decide on. We derive the narrowing function $g(i)$ for models of active resiliency and t-resiliency and with access to a set of set consensus types $\{O_1 = (m_1, k_1), ..., O_l = (m_l, k_l)\}$. We proceed first with the wait-free model of shared memory and set consensus objects.

5.1 Narrowing Power of the Wait-Free Model with Set Consensus

In this section we present and solve the $(i, g(i))$-narrowing task, a dynamic set consensus.

Definition In the $(i, g(i))$-*narrowing task*, for each input set $\{[j, input_j] |\ j \in P\}$, the output of each processor is the input of some processor such that at most $g(|P|)$ distinct outputs are chosen.

We begin by determining the narrowing power of a wait-free model with set consensus objects. Subsequently, we will combine the narrowing power of the set consensus with the narrowing power of resiliency. To simplify the statement and proof of the following lemmas, we assume that any model with access to (m, j)-set consensus has access to (k, j)-set consensus for all $k < m$.

Lemma 5.1.1 *A wait-free system of i processors using (m, j)-set consensus objects can implement $(i, (i \text{ div } m)j + \min\{j, i \bmod m\})$-set consensus.*

Proof: Arrange a sequence of $\lceil i/m \rceil$ (m, j)-set consensus objects in a row. Let processors $1, ..., m$ access object 1; let processors $m + 1, ..., 2m$ access object 2; and so on for processors 1, ..., i. The first (i div m) objects will produce at most $j(i \text{ div } m)$ outputs, and the last object will produce $\min(j, i \bmod m)$ outputs. ∎

Lemma 5.1.2 *In a wait-free system of i processors with access to a set of set consensus types $S = \{O_1 = (m_1, j_1)\text{-set consensus}, ..., O_l = (m_l, j_l)\text{-set consensus}\}$, if $m_k \leq i$ and $\forall h \leq i$, it holds that $m_k/j_k \geq m_h/j_h$. Then define $n_i = (i \text{ div } m_k)j_k + n_{i_1}$, where $i_1 = i \bmod m_k$. Then (i, n_i)-set consensus is solvable in the system.*

Proof: Following the proof of Lemma 5.1.1, build a sequence of objects that can achieve (i, n_i)-set consensus. The sequence consists of (i div m_k) objects of type O_k, returning at most $j_k(i \text{ div } m_k)$ outputs, plus (recursively) the sequence that achieves the $(i \bmod m_k, n_{i_1})$-set consensus for the ($i \bmod m_k$) processors remaining. Since for all $h \leq i$ it holds that $m_k/j_k \geq m_h/j_h$, and since we assume if an (m_i, j_i)-set consensus object is available then so is an $(m_i - 1, j_i)$-set consensus object, O_k will narrow the largest set of the i processors to the smallest number possible. The remainder are narrowed recursively. ∎

Lemma 5.1.3 *A wait-free model over $N+1$ processors with access to a set of set consensus types $S = \{O_1 = (m_1, j_1)$-set consensus, ..., $O_l = (m_l, j_l)$-set consensus$\}$ can solve the (i, n_i)-narrowing task.*

Proof: For all i between 1 and $N+1$, let there be a designated set of communication objects that solve (i, n_i)-set consensus (via Lemma 5.1.2). Upon entering the narrowing protocol, each processor takes a one-shot immediate atomic snapshot. Obtaining a snapshot of i processors, a processor submits its input to the (i, n_i)-set consensus. Incrementing i, the processor then proceeds, submitting to the (i, n_i)-set consensus its output from the $(i-1, n_{i-1})$-set consensus. When $i = N+1$, the processor halts, deciding on its output from the $(N+1, n_{N+1})$-set consensus.

Any processor that completes level $N+1$ and halts while only i processors are active must have executed the (i, n_i)-set consensus. Thus, the only inputs submitted to the $(i+1, ..., N+1)$-set consensus objects are the outputs from the (i, n_i)-set consensus and at most n_i outputs are chosen. This solves (i, n_i)-narrowing. ∎

5.2 Narrowing Power of Active Resiliency and Set Consensus

The narrowing problem has a dynamic nature since the number of distinct outputs relies on the number of processors executing the task. In the active resiliency model, the number of processors that may fail is a function of the number of active processors. Thus, to combine the dynamic nature of the narrowing problem with a resiliency function, I propose the strong $f(i)$ election problem. As I have generalized the k-set consensus problem to a variable k in the $(i, g(i))$-narrowing problem, the strong $f(i)$ election problem (defined below) is a generalization of the l-exclusion problem [3] to an exclusion problem with a variable l. The number of processors $f(i)$ that may enter the critical section increases as the number of active processors i increases. A solution to the strong $f(i)$ election problem will prove the $(i, f(i))$-narrowing task is solvable in the model.

The strong $f(i)$ election problem will be used in the simplex agreement algorithm of the next chapter. The simplex agreement algorithm proceeds in a series of rounds. Thus, to maintain consistency between one round of the simplex agreement algorithm and the next, I also propose an iterated version of the strong $f(i)$ election problem. In iterated strong $f(i)$ election, the number of active processors may decrease as well as increase. This captures the case that some processors halt before reaching a given iteration.

The Strong $f(i)$ Election Problem: Processors solve the $(i, f(i))$-narrowing problem with the additional requirement that a processor may reveal its input only in the last step before it halts.

The strong $f(i)$ election problem is distinct from the $(i, f(i))$-narrowing problem because a processor can not choose the input of any other processor until that processor has halted.

The Iterated Strong $f(i)$ Election Problem: After solving the strong $f(i)$ election problem, and committing a decision by writing it to shared memory, a processor determines (via its local state) whether it proceeds to the next iteration or halts. If j processors proceed to the next iteration of strong election without halting, they narrow to at most $f(j)$ outputs from among their j inputs, revealing their inputs only in the last step before completing the iteration.

The strong election problem is simply the subcase of iterated strong election in which every processor halts after the first iteration. Thus, I present only a solution to the iterated election. The solution views previous election phases simply as computation in which a processor either halts or proceeds to the election. The solution uses as a subroutine the participating set algorithm of Chapter 3. We solve the iterated strong $f(i)$ election problem in a model of g-resiliency with access to set consensus objects for $f(i) = n_{g(i)+1}$ where n_i is defined in Lemma 5.1.2. The iterated strong election algorithm is presented in Figure 5.1.

Solution to Iterated Strong $n_{g(i)+1}$ Election: I solve the iterated strong election problem in an $N+1$ processor $g(i)$-resilient system with set consensus objects $\{O_1 = (m_1, k_1)\text{-set consensus}, ..., O_l = (m_l, k_l)\text{-set consensus}\}$. Processors entering the election first write to signal that they are awake. Inductively, a processor proceeds through iterations $1, ..., t-1$, either posting that it is halting or proceeding to iteration t of the strong election. To execute election, processor j implements the participating set algorithm and posts the final-set[j] returned in the memory.

Let the processors awake but not halted in a prior iteration be W. Any processor $i \in W$ that has not posted its final-set[i] in this iteration is considered at level 0. This includes processors active in prior iterations as well as those that have not posted their participating sets in this iteration. The level of a processor $i \in W$ that has posted final-set[i] equals the number of elements in final-set[i]. Processor j now orders all processors of W lexicographically by (level, ID), updating this order as processors post their final-set of this iteration or post that they are halting in an earlier iteration. For $|W| = i$, processor j waits until either its location

```
Shared Variables: round, final-value: array [0,...,N] of register;
  final-set, round-output: array [0,...,N][1, ..., ∞] of set;
  halted, awake: array [0,...,N] of boolean; initially = false
Local Variables: part-set, active, pair-set: set;
  SS: snapshot;
  a,b := integer;
Iterated Narrowing(i, round, input)
  awake[i] := true;
  round[i] := round;
  part-set := ParticipatingSet(i);
  final-set[i, round] := part-set;
  Repeat
      SS := atomic-snapshot;
      active := {j| (SS.awake[j] ≠⊥)∧ ((SS.halted[j] =⊥)∨ (SS.round[j] ≥ round))};
      pair-set := ∅;
      For j ∈ active Do
          If SS.final-set[j,round] ≠⊥ Then
              pair-set := pair-set ∪(|SS.final-set[j, round]|, j);
          Else pair-set := pair-set ∪(0, j);Fi;
      Od;
      Sort(pair-set);
      For j = 1 to g(|active|) Do
          (a,b) := pair-set[j];
          If round-output[b,round] ≠⊥ Then
              round-output[i,round] := round-output[b,round];
          Else If b = i Then
              round-output[i, round] := CriticalSection(i, round, input);Fi;
      Od;
  Until round-output[i, round] ≠⊥;
Return round-output[i,round];
```

Figure 5.1: Iterated Strong Election Algorithm.

in the order is $\leq g(i) + 1$ or until some processor has written an input, declaring that it has halted with that value. In the latter case, processor j adopts the input value written as output, posts the output, and halts. In the former case, we say processor j has entered the critical section. A processor in the critical section may wait-free access the set consensus objects, submitting its own input and receiving an output. The processor adopts the set consensus value as its output, writes the value, and halts.

Lemma 5.2.1 *The algorithm of Figure 5.1 solves $(i, n_{g(i)+1})$-narrowing in a model of $g(i)$-resiliency with access to set consensus objects such that (i, n_i)-set consensus is wait-free solvable.*

Proof: In each iteration, out of the i processors that have not halted before entering the election, at most $g(i)$ will fail. Thus, eventually either there will be $g(i) + 1$ processors in the critical section or some processor will post an input. In either case (since at most $g(i)$ may fail), at least one processor is guaranteed to post an input, enabling noncritical section processors to complete the election protocol.

The function g decreases by at most 1 with each decrease in i, and the value i decreases (by at most 1) only when a processor ordered at level 0 declares it has halted (before entering the election). Thus, once a processor enters the critical section, its location in the order will remain less than $g(i)+1$, and so it will henceforth belong in the critical section. Since at any point there are at most $g(i) + 1$ processors in the critical section, at most $n_{g(i)+1}$ final values will be decided (due to the consensus objects used), and iterated strong election is solved. ∎

Lemma 5.2.2 *A $g(i)$-resilient system of i processors with access to a set of set consensus types $\{O_1 = (m_1, k_1)$-set consensus, ..., $O_l = (m_l, k_l)$-set consensus$\}$, can solve $(i, n_{g(i)+1})$-narrowing.*

5.3 Narrowing Power of t-Resiliency with Set Consensus

The foremost distinction between the t-resilient model and the model of active resiliency is that in the t-resilient model sleeping processors are considered faulty. To accommodate this discrepancy, I view the t-resilient model as a model of active resiliency in which the number of faults is a function of both the number of active processors and the number of halted processors. Below I define this resiliency function $T(i,j)$. Notice that once the number of halted processors is at least $N+1-t$, the active processors work wait-free.

Definition In an $N+1$ processor t-resilient system of shared memory, define the function $T(i,j) = \min(i-1, i+j-(N+1-t))$ to be the number of possible failures when i processors are active, j halted, and $i+j \geq N+1-t$.

In this section, I determine the narrowing power of the t-resilient model with access to set consensus objects. In an $N+1$ processor t-resilient system, at least $N+1-t$ processors are guaranteed to wake up and reveal their inputs, so the task input-output relation is not specified for simplexes of dimension less than $N-t$. Thus, the narrowing problem is only specified for input sets of size at least $N+1-t$. To solve the narrowing problem I adapt the solution to the iterated strong election problem to the t-resilient model. Since the number of failures in the t-resilient model is dependent on the number of processors that have woken up, not the number currently active, the narrowing function of each iteration of the strong agreement is dependent on both the number of processors currently active and the number of processors that have halted.

Lemma 5.3.1 *A t-resilient system of $N+1$ processors with access to a set of set consensus types $\{O_1 = (m_1,k_1)$-set consensus, ..., $O_l = (m_l,k_l)$-set consensus$\}$ can solve, for i active and j halted processors such that $j+k \geq i = N+1-t$, iterated strong $(i, n_{T(i,j)+1})$ election.*

Proof: I prove the lemma by solving the iterated strong $n_{T(i,j)+1}$ election problem in a t-resilient system in which i processors are active, j are halted, and $i+j \geq N+1-t$. Following the active resiliency solution, each processor p enters the election, posting that it is awake, and then waits until at least $i \geq N+1-t$ processors have woken. Processor p then executes the participating set algorithm for the current election phase, writing the participating set in the memory. As in the active resiliency algorithm, each processor orders all processors lexicographically by (level, ID), with any active processor that has not posted its participating set in that iteration considered at level 0.

Now, let i be the number of processors active and j the number that have halted. Processor p waits either for its location in the order to be $\leq T(i,j)+1 = \min(i, i+j-(N+1-t)+1)$ or for a processor to write an input, declaring that it has halted with that value. In the latter case, processor p adopts the input value written as output and halts, posting that it is halting with that value. In the former case, the processor enters the critical section to wait-free access the set consensus objects, submitting its own input. Once a processor in the critical section obtains a set consensus value, it adopts this input as output, writes the value, and halts.

When i processors are active in the election and j have halted, at most $T(i,j) = \min(i-1, i+j-(N+1-t))$ processors may fail. Thus, in a given phase, there will eventually be either $T(i,j)+1$ processors in the critical section or an input posted. In either case, at least one processor will post an input, enabling processors that are not in the critical section to complete the election protocol. At most $T(i,j)+1$ processors will enter the critical section, thus yielding at most $n_{T(i,j)+1}$ inputs chosen. ■

Corollary 5.3.2 *A t-resilient system of $N+1$ processors with access to a set of set consensus types $\{O_1 = (m_1, k_1)$-set consensus, $..., O_l = (m_l, k_l)$-set consensus$\}$ can solve, for i active and j halted processors such that $j + k \geq i = N + 1 - t$, $(i, n_{T(i,j)+1})$-narrowing.*

As we will see in the following chapters, this narrowing function captures the task-solving power of a model.

CHAPTER 6

Simplex Agreement

In Chapter 4, I gave a necessary condition for task solvability in the wait-free read-write model. The condition is based on the structure of the view graph and the existence of a simplicial map from the view graph to the output complex. To complete the characterization, it must be shown that the existence of the subdivided simplex and the simplicial map is sufficient as well as necessary. To this end, Herlihy and Shavit [20] proposed the *simplex agreement* problem. In the simplex agreement problem, each processor i has as input chromatic complex C^N and takes a node $v \in C^N$ as output such that $ID(v) = i$ and the set of all output nodes is a simplex of C^N.

Herlihy and Shavit [20] have solved the simplex agreement problem in the wait-free read-write model for input complex C^N a subdivided simplex. They employed topological arguments to prove that the view complex of an iterated immediate atomic snapshot protocol can be perturbed to approximate an arbitrary chromatic subdivided simplex. Since their simplex agreement algorithm relies on the precise subdivision of iterated immediate atomic snapshot view complexes, generalizing their algorithm to different models would entail determining the precise structure of each model. Thus, I take a different approach, solving the simplex agreement problem algorithmically by using the simplex agreement input complex directly. I first present the algorithm to solve simplex agreement in models of active resiliency with communication through shared memory and set consensus objects on the general class of g-solid complexes defined below. Subsequently I adapt the algorithm to solve simplex agreement in the t-resilient model with access to shared memory and set consensus on the class of $g(i, j)$-solid complexes defined in Section 6.5. When restricted to the wait-free read-write model with input simplex a subdivided simplex, the solution presented here is entirely different from that of [20].

6.1 The Generalized Simplex Agreement Task

In the simplex agreement problem on a subdivided simplex, a trivial solution would be to predefine an output simplex, letting each processor take the node of the simplex with its ID as output. I avoid this solution by requiring that processors choose nodes on the face of the input complex determined by the active processors. A wait-free read-write solution to the simplex agreement problem on a subdivided simplex completes the characterization of wait-free read-write shared memory since processors obtaining adjacent nodes of the subdivided simplex may then use the simplicial map (necessary by Lemma 4.4.1) to obtain compatible outputs. The formal task specification follows.

The Simplex Agreement Problem for a Subdivided Simplex [20]: Each processor i of an $N+1$ processor system has as input the chromatic subdivided simplex S^N and takes as output a vertex $v_i \in S^N$ such that the following conditions hold.

1. *(color preserving)* $ID(v_i) = i$.
2. *(adjacency)* $v_0, ..., v_N$ define a simplex of S^N.
3. *(nontriviality)* For any input set $\{[i, S^N] |\ i \in P\}$, the corresponding output

nodes v_i are contained in $face_P(S^N)$.

The simplex agreement problem for a subdivided simplex is solvable wait-free in read-write shared memory. For models of different resiliency with communication through shared memory and set consensus objects, the simplex agreement problem is solvable on more general complexes. Thus, the first step in generalizing to these models is to determine the class of input complexes for which simplex agreement is solvable.

Definition The *boundary* of a subdivided simplex $Subd(S^n)$, also called an *n-sphere*, is the subcomplex $Bd(Subd(S^n)) = \{s^q \in Subd(S^n) | carrier(s^q, S^n) < S^n\}$.

The boundary of a subdivided simplex is the complex induced by all boundary simplices of the subdivision.

Definition The simplicial image $\mu(S)$ of sphere S in complex C *has a span* in C if there exists a subdivided simplex S' with boundary S and a simplicial map $\bar{\mu} : S' \rightarrow C$ such that $\bar{\mu}$ agrees with μ on S.

Definition A set of vertices S in a complex C^N *can be spanned* if, inductively, the span of a vertex $v \in S$ is v, and the sphere image determined by the spans of the $i-1$ vertex subsets of S_i has a span.

Clearly if a set of nodes S can be spanned, then any subset of S can be spanned. Thus, the span for such a set of points is determined by taking the span of each pair of points, then taking the span of the 2-spheres they create, and so on. In the simplex agreement algorithm, I assume that, for each set of vertices S, a span is (inductively) predefined to be a span of the sphere image created by the predefined spans of the subsets of V. Based on the notion of span I define a k-solid complex for a constant k.

Definition A complex C^N is *k-solid of dimension N* if:

1. Any sphere image in C^N of dimension $\leq k$ has a span in C^N.
2. The linking sphere $ls(s^q, C^N)$ of any q dimensional simplex $s^q \in C^N$ is inductively k-solid of dimension $N - q$.

The nontriviality condition of the simplex agreement problem for a subdivided simplex requires that a solution lies in the face determined by the set of active processors. Thus, in order to generalize the simplex agreement problem to other input complexes, I generalize the notion of face to a complex. To do this, I define a correspondence between the faces of a simplex and simplices of the complex. In a subdivided simplex, the face of the simplex implicitly corresponds to the set of simplexes that subdivide that face. Let S^N be a chromatic simplex and C^N be a chromatic complex over the same set of IDs. Let δ be a point-to-set chromatic map from faces $s^i \leq S^N$ to sets of simplexes $\{c^i | c^i \in C^N\}$ such that $\delta(s^0) = \{c^0\}$, $\delta(S^N) = C^N$ and, for $s^i \leq s^j$, $\delta(s^i) \subseteq \delta(s^j)$. Then C^{N-1} has *underlying face structure* δ. For a set of IDs P and a complex C^{N-1} with underlying face structure δ, define $face_P(C^{N-1}) = \delta(face_P(S^{N-1}))$. I now extend the definition of k-solidity to complexes with underlying face structure. This class of complexes, g-solids with underlying face structure δ, will be used as input to the simplex agreement problem.

Definition Given a chromatic n-dimensional complex C^n and a function $g : N \rightarrow$

N, the complex C^n is a *g-solid of dimension n with underlying face structure δ* if:

1. Any sphere image in C^n of dimension $\leq g(n+1)$ has a span in C^n.

2. The linking sphere $ls(s^q, C^n)$ of any q-dimensional simplex $s^q \in C^n$ is inductively a g-solid of dimension $n-q$.

3. The complex $face_S(C^n)$ for any set of processors $S \subseteq IDs(C^n)$ is inductively a g-solid of dimension $|S|-1$ with underlying face structure δ.

In the next chapter, I will show that, in a model of active resiliency with communication through set consensus, the view complex is a g-solid (with the function g determined by the model) with underlying face structure δ. Theorem 6.1.1 shows that this definition generalizes a subdivided simplex. The generalized simplex agreement problem follows.

Theorem 6.1.1 *An n-dimensional subdivided simplex C^n is a g-solid of dimension n where $g(i) = i-1$.*

Proof: For conditions (1) and (2) above, see [31, 28]. To prove condition (3): since the face of a subdivided simplex is a subdivided simplex of lower dimension, an i-dimensional face is inductively a g-solid of dimension i. ∎

The Simplex Agreement Problem for a g-Solid Complex: For a system of $N+1$ processors, each processor i has chromatic complex C^N with underlying face structure δ as input, and takes as output a vertex $v_i \in S^N$ such that the following conditions hold.

1. *(color preserving)* $ID(v_i) = i$.

2. *(adjacency)* $v_0, ..., v_N$ define a simplex of C^N.

3. *(nontriviality)* For input set $\{[i, C^N]|\ i \in P\}$, output $\{[i, v_i]|\ i \in P\}$ is such that $v_i \in face_P(C^N)$.

The determining factor in whether the generalized simplex agreement problem is solvable in a given model is the solidity of the given complex C^{N-1} and the narrowing power of the model. Here I solve generalized simplex agreement over a g-solid complex in any model of active resiliency and set consensus in which $(i, g(i)+1)$-narrowing is solvable.

6.2 Topological Basis for Simplex Agreement

In the simplex agreement problem, processors are given a g-solid complex C^N as input and must obtain nodes of a common simplex as output. To achieve this, processors proceed in a series of rounds, each of which contains a narrowing phase and a convergence phase. The convergence takes place across spans in the complex, with processors obtaining points of the span within distance ϵ from each other. For small enough ϵ, there exists a function s mapping points of an ϵ-neighborhood to simplexes such that the set of simplices $\{s(p)|\ p \in \epsilon\text{-Nbd}\}$ has nonempty intersection s^q (Lemma 6.2.3). Any processor i such that $i \in IDs(s^q)$ will take the node of its ID as output and halt, guaranteeing termination after at most $N+1$ rounds. Processors without ID in the intersection will proceed to the next round, converging across a span of the linking sphere of the union of vertices (possibly) chosen in all previous rounds. By guaranteeing that the function from points to simplexes maps points of an ϵ-neighborhood to simplices with nonempty intersection, processors in the next round will be able to determine an appropriate span for convergence. In this section, I present the topological preliminaries for proving such an ϵ and function exist. All complexes are assumed to be finite and thus compact.

Lemma 6.2.1 (Lebesgue number lemma [27]) *Let $\mathcal{A}$ be an open cover of the metric space (X, d). If X is compact, there is a $\delta > 0$ such that for each subset of X having diameter less than δ, there exists an element of $\mathcal{A}$ containing it.*

The number δ is called the *Lebesgue number* for the covering $\mathcal{A}$ [27].

Lemma 6.2.2 *Let $\mathcal{A}$ be an open cover of a compact space. Let the Lebesgue number of the cover be δ. If $x \in A \in \mathcal{A}$, $y \notin A$, and, $\forall B \neq A,\ x \notin B$, then the minimum distance between $1/4\delta$ balls around x and y is at least $1/2\delta$.*

Proof: By the definition of the Lebesgue number, if x and y were within δ of one another then there must be an open set $C \in \mathcal{A}$ that contains both x and y. This contradicts the assumptions that $x \in A, y \notin A$, and, $\forall B \neq A,\ x \notin B$. ■

Lemma 6.2.3 *Let $|S^N|$ be an embedding of the finite complex C^N in R^{2N+1}. Then there exists a function $s : |C^N| \to C^N$ that maps $x \in |C^N|$ to a simplex $s(x) = s^m \in carrier(x, |C^N|)$ and an $\epsilon > 0$ such that, for any neighborhood $N(\epsilon) \subset |C^N|$ with diameter $\leq \epsilon$, $\bigcap_{y \in N(\epsilon)} s(y) \neq \emptyset$ and $\bigcup_{y \in N(\epsilon)} s(y) = s^l$ for some $s^l \in C^n$.*

Proof: The function $s : |C^N| \to C^N$ is defined in stages. In stage i, the function is defined over all points p with image $s(p)$ of dimension $\leq i + 1$. Thus, by stage N, the function is fully defined over $|C^N|$. Let $S_i = \{x | \ (x \in |C^N|) \wedge (dim(s(x)) = i)\}$. Then, to define the set of points that map to zero-dimensional simplexes of C^N (S_0), take the open cover of $|C^N|$ induced by the open stars of the zero-dimensional simplexes of C^N. Let the Lebesgue number of this open cover be $\delta_0 > 0$. By definition of the Lebesgue number, for each neighborhood $N_{\delta_0} \subset |C^N|$

of diameter $\leq \delta_0$, there exists an open star A such that $N_{\delta_0} \subseteq A$. Then define $S_0 = \{x \in |C^N| \mid \min_{s^0 \in C^N}(dist(x, s^0)) \leq \delta_0/4\}$. Let $interior(S_0) = S_0 - \{x \in |C^N| \mid \min_{s^0 \in C^N}(dist(x, s^0)) = \delta_0/4\}$; that is, $interior(S_0)$ is all points of S_0 less than $\delta_0/4$ distance from a zero-simplex. By definition of the Lebesgue number, it is easy to see that each $x \in S_0$ is included in S_0 via a unique vertex s^0_x. Hence, for $x \in S_0$, define $s(x) = s^0_x$.

Inductively, assume $S_0, ..., S_{i-1}$ are defined. To define S_i, take the open cover of $|C^n|$ consisting of the open stars of the i-dimensional simplexes and the sets $interior(S_k)$, $k = 0, ..., i-1$. Let the Lebesgue number of this open cover be δ_i. Then define $S_i = \{x | (\min_{s^i \in C^N}(dist(x, s^i)) \leq 1/4\delta_i) \wedge (x \notin \cup_{j=0}^{i-1} S_j)\}$. Again, for each $x \in S_i$, there is a unique s^i_x that accounts for x in the set. Thus, for $x \in S_i$ define $s(x) = s^i_x$.

Now, let $\epsilon = 1/4 \min_{i=0}^{n}(\delta_i)$. The union and intersection properties will be implied if I prove that, for any x and y, if $dist(x, y) \leq \epsilon$, then $s(x)$ and $s(y)$ are related by containment. This follows from the construction of the function $s(x)$ and Lemma 6.2.2. Namely, assume x and y are such that $dist(x, y) < \epsilon$, $s(x) \not\subseteq s(y)$, and $s(y) \not\subseteq s(x)$. Then without loss of generality let the dimension of $s(x) = k$ be less than or equal to dimension of $s(y)$. In the open cover considered in step k, all points in $s(x) - \cup_{i=0}^{k-1} S_i$ belong solely to the open star of $s(x)$, but points in $s(y)$ are not in the open star of $s(x)$, contradicting the fact that $\epsilon < 1/2\delta_k$. ■

Figure 6.1 shows the partitioning of a two-dimensional complex into regions determined by the function s of Lemma 6.2.3. The figure, which is not drawn to scale, is presented to help visualize the idea of the lemma.

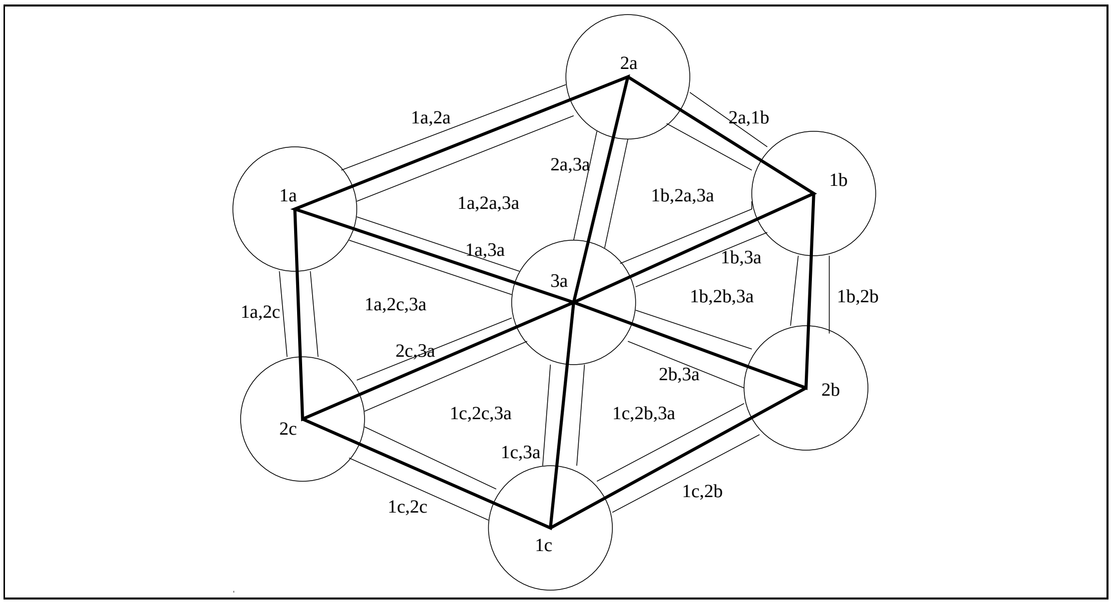

Figure 6.1: Example of Complex Partitioning and Map s.

6.3 ϵ-Convergence

The generalized simplex agreement algorithm proceeds in a series of rounds, with each round consisting of a narrowing phase and a convergence phase. For the narrowing phase, I use the iterated strong election of Chapter 5. Processors submit vertices to the narrowing phase, the outcome of which determines the corner points of a span for the convergence phase. In the convergence phase, processors converge across the span predefined as a function of the corner vertices. By definition, there exists a subdivided n-simplex that simplicially maps to the span. Thus, all convergence can take place on the subdivided simplex, yielding simplices via the function of Lemma 6.2.3 which simplicially map to the span. Without loss of generality, let the subdivided simplex be a subdivision of the unit simplex with vertices $(1, 0, ..., 0), (0, 1, 0, ..., 0), ..., (0, ..., 0, 1)$ in R^{n+1} and let ϵ be the minimum of the maximal ϵ' value that satisfies Lemma 6.2.3 for each span in the complex. Since the complex is finite, such an ϵ exists.

The ϵ-convergence problem is simply the vector generalization of the classic ϵ-approximate agreement problem [6, 14, 15]. In ϵ-approximate agreement, processors start with real numbers and are required to end with real numbers within ϵ of one another. For vector ϵ-convergence, the distance between two vectors (each representing a point in space) must be smaller than ϵ. To achieve this, I present an inefficient but conceptually simple algorithm. Faster algorithms appear in [6, 14, 15].

Let S be a unit simplex with corner points $\alpha_0, ..., \alpha_N$ in R^{N+1}; then, for a subdivision of S and an $\epsilon > 0$, and any N, there exists an M such that for any distribution of positive integer weights $\lambda_0, .., \lambda_N$ with $\sum_{i=0}^{N} \lambda_i = M$, the point $(\lambda_0\alpha_0/M, ..., \lambda_N\alpha_N/M)$ is within ϵ of any point $(\lambda'_0\alpha_0/m, ..., \lambda'_N\alpha_N/m)$ where for all $i, \lambda_i \leq \lambda'_i$ and $\sum_{i=0}^{N} \lambda'_i = m \leq M + N$. Thus, let each processor be associated with a corner point of the simplex (more than one processor may be associated with a given corner point). Then, let each processor repeatedly add a unit of weight to its vertex until it reads that the total weight of the system is at least M. At this point, a processor takes the weighted average of the vertex points to be its final point. In other words, if upon reading, a processor sees vertex weights $\lambda_0, ..., \lambda_N$ such that $\sum_{i=0}^{N} \lambda_i = K \geq M$, then the processor takes the point $(\lambda_0\alpha_0/K, ..., \lambda_N\alpha_N/K)$ to be output. Necessarily, final points lie within ϵ of each other.

Precondition: The vertex$_i$ is contained in $ls(\text{core}_i, \text{face}_{Set_i}(\text{complex}))$, and, for all i, the intersection and union of core$_i$ are nonempty simplices in $\text{face}_{Set_i}(\text{complex})$.

Postcondition: For all i, the intersection and union of simplex$_i$ are nonempty simplices, and simplex$_i$ is contained in $ls((\cap_j core_j), \text{face}_{\cup_j Set_j}(complex))$.

ϵ-**Converge**(i, vertex$_i$, Set$_i$, core$_i$, complex)
Return simplex$_i$;

Figure 6.2: ϵ-Convergence Specification.

The specification of the ϵ-convergence phase of the simplex agreement algorithm is presented in Figure 6.2. Each processor in the ϵ-convergence has a vertex, a simplex core, a set of awake processors, and a complex as input. Each processor i returns a simplex s_i that lies in the linking sphere of the intersection of the core simplexes such that the intersection and union of the s_i simplexes is a nonempty simplex.

I assume the span for any set of vertices, core simplexes, and awake sets is a predefined function of the (vertex, core, set) triples. The span lies in the linking sphere of the intersection of the core simplexes and in the face of the complex determined by the union of processor sets. Since the complex is finite, there are a finite number of predefined spans. By Lemma 6.2.3, each span determines a δ and function s for which points within δ of each other map to simplexes with nonempty union and intersection. Scale each subdivided simplex that maps to a span so that it is a subdivision of the unit simplex. From each such subdivision, I obtain a δ' such that points within δ' of each other in the subdivided unit simplex map to simplexes of the span with nonempty union and intersection. Letting ϵ be the minimum over all the δ', processors can solve ϵ-convergence on a subdivision of the unit simplex and map their output points to a simplex of the predefined span of the (vertex, core, set) inputs. Thus, the output simplexes to the ϵ-convergence phase have nonempty simplices as union and intersection, and lie in the linking sphere of the intersection of the core simplexes. The complete simplex agreement algorithm is specified below.

6.4 Generalized Simplex Agreement Algorithm

In this section, the generalized simplex agreement algorithm (Figure 6.3) is stated in full. I assume that the input complex for a model of active resiliency with access to set consensus objects such that $(i, g(i)+1)$-narrowing is solvable is a g-solid complex C^N with underlying face structure δ.

The simplex agreement algorithm proceeds in a series of rounds. After announcing that it is participating in the simplex agreement algorithm, a processor enters round 1. Initially, processor i's span-corner is the vertex $face_i(C^N)$ and its core simplex is empty. A processor in round d submits its span-corner to the d'th iteration of iterated strong $g(i)+1$ election, obtaining a vertex (reassigned to span-corner) as output. The processor then takes a snapshot to determine the awake processors and submits its span-corner, core simplex, and awake processor set to the ϵ-convergence.

There are at most $g(i)+1$ span-corners obtained from the iterated narrowing when i processors are active in the iteration. Thus, by the g-solidity of the complex, there exists a predefined span over the span-corners that lies in the face determined by the union of the awake sets on the intersection of the core simplexes. Thus, processors execute the ϵ-convergence and obtain simplexes with nonempty intersection and union which lie in the predefined span.

Upon completing the ϵ-convergence, each processor writes its output simplex ϵ-output-sx from the convergence and takes a snapshot, calculating the union and intersection of the convergence phase output simplexes posted. For processor i, if $i \in IDs(\bigcap_j \epsilon\text{-output-sx}_j)$, then the processor chooses as output the node of its ID, posts this node, and halts. Since the intersection is guaranteed to be nonempty, at least one processor halts in each round. Otherwise, if a processor's ID is not in the intersection, the processor updates its core simplex and span-corner proposal and begins round $d+1$. The new value of the core simplex for processor i is the intersection of the cores posted in round $d-1$ plus simplex $S = (\bigcup \epsilon\text{-output-sx})$ of round d, minus any node of ID i. Thus, the core contains all nodes that may have been taken as output in rounds up to this point. A processor chooses any node of its ID that lies on the linking sphere of the core simplex and in the face determined by the active processors as its new span-corner proposal.

Since at least one processor is guaranteed to take a value and halt in each round, the algorithm terminates in at most $N+1$ rounds. The algorithm is formally presented in Figure 6.3.

The next theorem unifies the results of this chapter. As will be evident in Chapter 8, this theorem provides the proof of sufficiency of the characterization for models of active resiliency and set consensus.

Theorem 6.4.1 *$N+1$ processors working in a model that can achieve $(i, g(i)+1)$-narrowing through the resiliency and set consensus objects available can solve simplex agreement on a g-solid complex C^N with underlying face structure δ.*

Proof: By assumption, the model solves $(i, g(i)+1)$-narrowing. It follows that, the narrowing phase of the algorithm (iterated strong election) will narrow the proposals of i processors active in the agreement to at most $g(i)+1$ vertices. Because the complex is g-solid, a span can be predefined for each set of $j \leq g(i)+1$ triples of (vertex v_i, core simplex cs_i, active processor set S_i) if the core simplexes intersect in a simplex s^q, $v_i \in ls(cs_i, face_{S_i}(C^N))$, and, for k such that $g(k)+1 \leq j$, $\bigcup_i S_i - IDs(s^q)$ contains at least k processors. Moreover, the predefined span of the vertices lies in the linking sphere of s^q and has faces recursively determined by the appropriate subsets of triples.

By the iterated narrowing, there are at most $g(i)+1$ triples corresponding to at least i active processors of the phase. By Lemma 6.2.3 and the fact that vertices chosen in any phase reside on the linking sphere of vertices chosen in previous phases, the core simplexes intersect in a simplex. Thus, in each round of simplex agreement, a span on which to carry out convergence exists.

Processors execute the ϵ-convergence phase, returning simplexes with intersection and union nonempty simplexes by Lemma 6.2.3. Thus, at least one processor per round will see its ID in the intersection and choose a final value. After at most $N+1$ rounds, the simplex agreement algorithm will terminate. The core simplex of any round contains all vertices chosen up to this round, and so the final set of vertices is a simplex since the nodes chosen in any round lie on the linking sphere of the intersection of the cores and thus are adjacent to all nodes chosen in previous rounds. ∎

The sufficiency of the necessary condition for wait-free read-write solvability presented in Lemma 4.4.1 is now evident.

6.5 t-Resilient Generalized Simplex Agreement Algorithm

In a t-resilient $N+1$ processor system, at least $N+1-t$ processors are guaranteed to wake up and reveal their inputs, thus for simplex agreement the convergence phase always takes place within a face of dimension at least $N-t$. To extend the simplex agreement algorithm to the t-resilient model, I first define the properties of an input complex in the t-resilient model with access to set consensus. Since no statements are made about the view complex until $N+1-t$ processors are active, I first generalize the notion of underlying face structure in order to limit the dimensions in which the face structure exists.

A complex C^N is said to have *underlying i face structure* δ of dimension N if $\delta : S^N \rightarrow C^N$ is a chromatic point-to-set map such that $\delta(S^N) = C^N$ and, for $q \geq i$ and $s^q \leq s^p \leq S^N$, $\delta(s^q) \subset \delta(s^p)$. Given a complex C^N with underlying i face structure δ of dimension N, the linking sphere $ls(c^q, C^N)$ of a simplex c^q in C^N is said to have *induced i face structure* δ if, $\delta(s^p) = \{c \in ls(c^q, C^N) | c \in \delta(s^p) \text{ w.r.t } C^N\}$.

Definition For a constant k and function $g : N \times N \rightarrow N$, complex C^N is defined to be a $g(i, j)$-solid of dimension N with underlying k face structure δ if:

1. Any sphere image $S \in C^N$ of dimension $\leq g(N + 1, 0)$ has a span in C^N.
2. The linking sphere of any simplex $s^q \in C^N$ is inductively a $g(i-(q+1), j+(q+1))$-solid of dimension $N - q$ with with induced underlying k face structure δ.
3. The complex $\delta(s^q)$ for $q \geq N - t$ is inductively g-solid of dimension q with induced underlying k face structure δ.

For a t-resilient $N + 1$ processor model of shared memory, I will show that the view complex is $T(i, j)$-solid with underlying $N - t$ face structure. Recall that $T(i, j) = \min(i - 1, i + j - (N + 1 - t))$ is the number of possible failures when i processors are active, j are halted, and $i + j \geq N + 1 - t$. For a model of t-resiliency and set consensus objects, I will show that the view complex is $(n_{T(i,j)+1} - 1)$-solid with underlying $N - t$ face structure, where n_i is defined in Lemma 5.1.2 for the set consensus objects available.

In the generalized simplex agreement problem for the t-resilient model, the input complex has underlying $N - t$ face structure. This poses no problem since processors may wait until at least $N + 1 - t$ processors become active, defining the appropriate face. Since the simplex agreement algorithm is wait-free except for the narrowing phase, I need only substitute the t-resilient iterated strong election of chapter 5 for the narrowing phase to adapt the simplex agreement algorithm to the t-resilient model. The simplex agreement is solvable since processors in a model that solves iterated strong $n_{T(i,j)+1}$ election can narrow to few enough vertices so that the appropriate dimension spans exist for the ϵ-convergence phase in a $(n_{T(i,j)+1} - 1)$-solid complex.

Theorem 6.5.1 *A t-resilient $N+1$ processor system of shared memory and set consensus objects in a model that can solve $(i, n_{T(i,j)+1})$-narrowing for $i + j \geq N+1-t$ can solve simplex agreement on a complex C^N that is $(n_{T(i,j)+1} - 1)$-solid with underlying $N-t$ face structure δ.*

Proof: The simplex agreement algorithm is the algorithm of Figure 6.3 with the t-resilient strong iterated narrowing used for the narrowing phase. In the t-resilient narrowing, processors wait until $N+1-t$ processors wake up before proceeding further, so the appropriate face in which to choose vertices is defined. By Lemma 5.3.1, when $i \geq N+1-t$ processors have woken and none have halted, processors can solve $(i, n_{T(i,0)+1})$-narrowing. By the $(n_{T(i,j)+1} - 1)$-solidity requirements, an $(n_{T(i,0)+1} - 1)$-dimensional span for the convergence phase exits in the i dimensional face corresponding to the processors that are awake. Thus, processors narrow to a set of vertices that can be spanned, and thus proceed with the ϵ-convergence phase.

In subsequent narrowing phases, processors choose vertices on the linking sphere of some simplex s^q (reflecting that $q+1$ processors have halted). These i active processors can narrow their inputs to $n_{T(i,q+1)+1}$ vertices that again can be spanned on the linking sphere due to the solidity requirements. Thus, in each phase, a span for ϵ-convergence exists, so processors solve simplex agreement on a $(n_{T(i,j)+1} - 1)$-solid complex with underlying $N - t$ face structure δ. ■

```
Shared Variables: array core, ε-output-sx[0..N, 0..N] of simplex = ∅;
  array participating, halted[0..N] of boolean = ⊥;
  array final-vertex[0..N] of simplex = ∅;
Local Variables: done: boolean = false;
  intersection, union, span-corner, newcore: simplex = ∅;
  round: integer = 1;
  SS, simplexSS: snapshot;

Generalized Simplex Agreement(i, C^N): Returns(vertex);
  participating[i] := true;
  span-corner := face_i(C^N);
  Repeat
      span-corner := IteratedNarrowing(i,round,span-corner);
      SS := snapshot{participating};
      ε-output-sx[round,i] := ε-Converge(i, span-corner, SS, core[round,i], C^N);
      simplexSS := snapshot{core[round,0..N]};
      intersection, union := ∅
      For j = 0 to N Do
          If simplexSS[j] ≠⊥ Then
              intersection := intersection ∩ simplexSS[j];
              union := union ∪ simplexSS[j];
          Fi;
      Od;
      If i ∈ IDs(intersection) Then
          final-vertex[i] := {v|v ∈ intersection ∧ ID(v) = i};
          halted[i] := true;
      Else (*i ∉ IDs(intersection) *)
          round := round+1;
          newcore := ∅;
          For j = 0 to N Do
              If core[round−1] ≠⊥ Then
                  newcore := newcore ∩ core[round −1];
              Fi;
          Od;
          newcore := newcore ∪ union;
          If i ∈IDs(newcore) Then
              newcore := newcore −{v|v ∈ newcore ∧ ID(v) = i};
          Fi;
          round := round+1;
          core[round,i] := newcore;
          span-corner := choose{v|v ∈ ls(newcore, face_SS(C^N)) ∧ ID(v) = i};
      Fi;
  Until halted[i];
Return final-vertex[i];
```

Figure 6.3: Generalized Simplex Agreement Algorithm.

CHAPTER 7

Simulation

The simplex agreement algorithm solves simplex agreement on a g-solid complex in any model of active resiliency and set consensus that can achieve $(i, g(i)+1)$-narrowing and on a $g(i,j)$-solid complex in a model of t-resiliency and set consensus that can solve $(i, g(i,j)+1)$-narrowing. Thus, extending the characterization beyond the wait-free read-write model, I will show that the existence of a g-solid complex and simplicial map to the output are necessary for solvability in models that achieve $(i, g(i)+1)$-narrowing. In addition, I will show a $g(i,j)$-solid complex and simplicial map to output are necessary for solvability in models that achieve $(i, g(i,j)+1)$-narrowing. Instead of deriving the view complex structure for each new model directly, I derive properties of its structure by simulating the protocol wait-free in read-write memory. I then show that the simulation implies a simplicial map from the view complex of the simulation to the view complex of the system being simulated.

7.1 The Simulation by Agreement

Processors work wait-free in read-write memory to simulate protocols written for resilient systems of set consensus plus read-write memory. I start with a protocol for M processors written for a resilient model with set consensus objects. I assume without loss of generality that the protocol is full-information [17], namely, each read is a snapshot, each write and submit is the entire history, and no writes are ever overwritten. In addition, I assume without loss of generality that each access to a set consensus object is immediately followed by a write.

The codes of a resilient protocol for read-write memory include commands of the type *write, read,* and *wait.* Each wait statement is simply a loop of read operations and thus will be considered simply as a sequence of read statements. I will simulate executions in which the first read of the wait statement satisfies the wait condition, so only one read per wait statement will be necessary. When the codes correspond to a protocol in which processors have access to set consensus objects, the commands may also include *submit(object-id, value)*, and *receive-reply(object-id, value).* These two commands in a code always appear as a pair with matching *object-id*, the *receive-reply* immediately following the *submit.* The last two commands of any code are *write(decision value)* and *halt*, in that order.

The simulation system is composed of $N+1 \leq M$ simulators working wait-free in read-write shared memory. The M protocol codes (with inputs) are placed in the shared memory of the simulation system. The memory of the simulation system also contains *agreement sections*, with each individual read, write and receive-reply command corresponding to a unique agreement section. In addition, for each object-id, there is a single agreement section corresponding to the set of all submit commands for that object-id. An agreement section is a place in the memory where simulators conduct an *agreement protocol.* In the simulation, simulators read and write in order to simulate a step on behalf of one of the codes. As more than one simulator may attempt to simulate a particular statement of a code, an agreement protocol is needed so that all simulators agree on the state of each code. For example, to simulate a read statement, each simulator reads the memory and proposes this read value as the value of the read statement. Because the simulator system is asynchronous, simulators read at different times, proposing different states of the memory as the value of a single read statement. Each state proposed is inductively valid for the simulation. However, in order to simulate a single execution, simulators must determine the unique local state of a code before simulating any further statements of the code. To this end, processors use an agreement protocol to decide which proposal will be associated with the operation. Once all processors agree on the value of a given operation, that operation is considered simulated, and inductively, the local state of the code is determined. Simulators continually propose values and execute agreement protocols for the first unsimulated operation of any code, thus simulating an execution of the protocol.

The agreement protocol is the foundation of the simulation. Since the simulation is wait-free, and consensus is impossible if even one processor may fail [16], a failed simulator can block an agreement from completing. To ensure that the simulation progresses, I propose an agreement protocol that is wait-free except for a single wait statement as the last statement of the protocol. Moreover, only a processor in the wait-free section of the agreement protocol can block the protocol, so once all simulators active in the agreement protocol reach the wait statement, the simulators can determine the agreement value. The simulators time-share between agreement protocols for different codes of the simulation, only starting a new agreement protocol after reaching the wait statement in the current agreement protocol. In this way, each simulator can block only a single agreement protocol, so a nonfaulty simulator faces at most N blocked agreement protocols out of the $M > N$ codes to be simulated.

Although there are many appropriate agreement protocols that are wait-free except for a single wait statement as the last statement of the code, the one presented

```
Shared Variables: array proposal[0..N] of register = ⊥;
  array part-set[0..N] of set = ⊥;
Local Variables: done: boolean; initially false
  proposal-set, not-ready, small-set: set = ∅;

Agreement(i, OpType, proposal): Returns(output);
  proposal[i] := proposal;
  small-set := ParticipatingSet(i);
  part-set[i] := small-set;
  Repeat
      For j = 0 to N Do
          If part-set[j] ≠⊥ ∧|part-set[j]| ≤ |small-set| Then
              small-set := part-set[j] Fi;
      Od;
      proposal-set := ∅;
      not-ready := ∅;
      For j = 0 to N Do
          If j ∈ small-set Then
              proposal-set := proposal-set ∪ proposal[j];
              If part-set[j] =⊥ Then
                  not-ready := not-ready ∪j Fi;
          Fi;
      Od;
      If (OpType = submit(m,j)) ∧ (|not-ready| < j ∨ |proposal-set| ≤ j) Then
          done := true;
      Else If (OpType ≠ submit) ∧ (|not-ready| = 0 ∨ |proposal-set| = 1) Then
          done := true;
      Fi;
  Until done;
Return proposal[Min{small-set}];
```

Figure 7.1: Agreement Algorithm.

here is based on the participating set protocol of Chapter 3. Recall that output sets to the participating set problem satisfy the properties of self-containment, atomic snapshot, and immediacy.

Agreement Protocol (figure 7.1):

Each simulator i writes its proposal v_i to the agreement in the memory and then executes the participating set algorithm. Once processor i obtains a participating set S_i, it posts (i, S_i) in the memory and waits for one of the following conditions.

1. For all agreement sections except for a submit section, processor i waits until it reads a pair (j, S_j) such that either, $\forall p \in S_j, (p, S_p)$ is posted or, $\forall p \in S_j, v_p = v$ for some proposed value v. The processor then chooses as output the agreement proposal of the lowest ID processor in the smallest set posted.

2. For a submit section corresponding to an (m, k)-set consensus object, processor i waits until it reads a pair (j, S_j) such that either $|\{v_p \mid p \in S_j\}| \leq k$ or, for at least $|S_j| - (k-1)$ processors $p \in S_j$, the pair (p, S_p) is posted. The processor chooses as output the agreement proposal of the lowest ID processor in the smallest set posted.

The following lemmas prove that the protocol executes consensus for agreement sections corresponding to read, write and receive-reply operations, and k-set consensus for agreements corresponding to submit operations.

Lemma 7.1.1 *The number of distinct agreement values that result from a write, read, or receive-reply agreement is exactly one.*

Proof: Assume the agreement terminates because there exists a pair (j, S_j) such that for all processors $p \in S_j$ the pair (p, S_p) is written. Let S_i be the smallest set of any processor $p \in S_j$ (several processors may share this set). By the immediacy property of the participating set task, S_i is the unique smallest set that will be posted. Thus, the processor with the smallest ID in S_i is determined uniquely, and all processors will obtain the same agreement value.

Now, assume some processor terminates because there exists a pair (j, S_j) such that all processors $p \in S_j$ propose the same agreement value. Then, again by the participating set immediacy property, only processors in S_j will post sets smaller than S_j. Thus, the value of every processor in S_j will also be the value of the smallest ID processor in the smallest set, and again all processors obtain the same agreement value. ∎

Lemma 7.1.2 *If no processor in a write, read, or receive-reply agreement fails, then the agreement protocol solves consensus.*

Proof: If no processor is faulty, then eventually all processors i will post their participating set (i, S_i) in the memory. In particular, all members of the smallest set will have posted their sets. Thus, the agreement terminates with the value proposed by the processor with the smallest ID in the smallest set chosen as output by all processors. ∎

Lemma 7.1.3 *At most k distinct agreement values result from a submit agreement corresponding to AN (m, k)-set consensus object.*

Proof: Assume the agreement terminates when there is a pair (j, S_j) for which at most $k-1$ processors of S_j have not posted their sets. Thus, letting S_i be the smallest set written, there are at most $k-1$ processors in S_i that have not written their sets, and so at most $k-1$ sets that may be smaller than S_i. The smallest processor in a set is unique; thus, since there are at most $k-1$ sets smaller than S_i, there are at most $k-1$ processors that may be chosen from these sets. Adding in the input of the smallest processor from set S_i, there are at most k values chosen total.

Now, assume a processor terminates due to a pair (j, S_j) for which processors of the set propose at most k distinct values. Then, any set smaller than S_j contains at most these k values, the only values eligible to be chosen (via a set in which at most $k-1$ processors have not written sets). Thus, at most these k values will be chosen, and the lemma holds. ∎

Lemma 7.1.4 *If less than k simulators fail in an (n,k)-set consensus submit agreement, the agreement solves k-set consensus.*

Proof: If less than k simulators fail, then all but k simulators will have posted their participating sets (i, S_i) in the memory. In particular, the smallest set posted will contain at most $k-1$ processors p for which (p, S_p) is not posted. Thus, the wait statement is satisfied, and a simulator will choose the value proposed by the smallest processor of the smallest set posted. At most k values are chosen. ∎

To simulate an execution of a protocol, simulators propose values for each operation and execute the associated agreement protocol. Once an agreement is reached, the step is considered to be executed. Since simulators may fail, some agreement protocols may not terminate wait-free. Thus, a simulator time-shares between agreement protocols for different operations. Since a simulator can only block an agreement in the wait-free section of the agreement (before it posts its participating set), a simulator only starts a new agreement protocol once it reaches the wait statement of the present agreement. In this manner, the failure of a single simulator may cause the blocking of at most a single agreement protocol, the one for which it has not posted its participating set.

A simulator, starting with the first statement of each code, executes a code statement only after all previous statements of that code have been completed (i.e., the agreement values determined). Since each simulator reads the entire memory in a snapshot, it can determine the local state of each of the simulated codes by observing the history of operations completed for the code. Thus, a simulator may attempt to execute the first operation of any code for which the corresponding agreement value is not determined. A simulator proposes input agreement as follows.

write A simulator submits the local state of the processor's code, which can be deduced uniquely from the sequence of preceding operations for that code. (This step is introduced for the sake of modularity. All simulators necessarily bring the same input value to a write agreement, and consequently the agreement protocol is superfluous).

read A simulator submits the state of the shared memory that it interprets from its own atomic snapshot read. Simulators reading at different times may bring in different input values to a common read agreement. Since the read values submitted are deduced between the time a code's previous operation agreement completes and the time the read statement agreement completes, inductively, each of these input values is a legal extension to the history of the code.

submit Just as for a write, a simulator can infer uniquely the value that the code submits to the object. This value is proposed as input to the submit agreement. Since there is a single submit agreement for each (m, k)-set consensus, different simulators (corresponding to m different codes) may propose different values to the agreement. Each of the j possible outcomes of the agreement, distributed back to codes, is thus a value the code may have received in an actual execution.

receive-reply A simulator submits the value it received from the preceding submit agreement. At most j distinct values will be submitted to the receive-reply agreement, and one will be chosen.

Since each input to an agreement is a legal value for that operation of the code, when the agreement values are determined, a single execution of the protocol is simulated. However, there are some steps of the simulation that have not yet been specified. In particular, how does a simulator choose which operation to execute next (i.e., which agreement to enter), and when does a simulator halt? We define a *simulation rule* to be a deterministic function that maps the local state of a simulator into either an agreement to participate in or to a halting state. Thus, a simulation determined by a simulation rule is simply a deterministic protocol over the set of simulators.

To ensure that the simulators make progress in simulating a protocol, any simulation rule will allow the read operation of a wait statement to be simulated only once the wait condition has been met. Since the simulation rule maps a snapshot of the memory to the agreement to implement next, we simply require that no wait (read) agreement is executed until the read value to be submitted meets the wait condition. The simulation technique as applied to wait-free read-write simulation of resilient read-write systems has recently been formalized by Lynch and Rasjbaum [25].

7.2 The Structure of Active Resiliency and Set Consensus

The simulation technique enables simulators working wait-free in read-write memory to simulate runs of a protocol written for a larger set of processors working resiliently and using set consensus objects. In a run of the simulation, a code is said to be *simulated to completion* if all statements of the code have been simulated. The local state of any code that has been simulated to completion will be referred to as a *final code view*. I will define a simplicial map between views of the simulation view complex and views of the protocol view complex that has been simulated, mapping a view of the simulation to the final code view of some code simulated to completion in the simulation. Through these simplicial maps, I will identify properties of the view complex structure of active resilient full-information protocols using set consensus objects. I refer to the simulation view complex as SVC and the simulated code view complex as CVC. I will show that CVC for any model in which $(i, g(i)+1)$-narrowing is solvable is g-solid.

Lemma 7.2.1 *For a given simulation rule, and execution of the simulation, the set of final code views of a simulation constitutes a simplex in the view complex of the simulated protocol (CVC).*

Proof: Since simulators simulate operations of the codes one at a time, the local states of the simulated codes are inductively compatible with the prefix of an execution in which the operations occur in the order they are simulated. Thus, given a set of codes simulated to completion, the corresponding local states define views that are compatible with a common execution. Thus, the views are adjacent in CVC. ■

Lemma 7.2.2 *For any simulation rule, if each view of a simulator contains at least one final code view, then there exists a simplicial map $\bar{\mu}$ from simulator views (SVC) to code views (CVC) such that $\bar{\mu}(SVC)$ is the image of a subdivided simplex.*

Proof: For a given simulation rule, the simulation is a deterministic wait-free read-write protocol, thus SVC, the view complex of the simulation, is the image of a subdivided simplex (Lemma 4.3.9). Since each view of SVC sees at least one final code view, the simplicial map $\bar{\mu}$(SVC) can be defined to map each view v of SVC to a view of CVC seen in v. By Lemma 7.2.1, this map is simplicial, and so $\bar{\mu}$(SVC) also is the simplicial image of a subdivided simplex. ∎

The first step in determining the g-solidity of a view complex CVC is to define the underlying face structure δ. For a chromatic N-simplex S^N labeled by the processor IDs of CVC, define $\delta(s^q)$ to be the set of all simplexes of CVC induced by the set of all views that contain only operations of processors in $IDs(s^q)$. Then, for any set of IDs S, $face_S$(CVC) is the complex $\delta(s^q)$, where s_q is labeled by the IDs of S. Clearly, $\delta(S^N)$ =CVC, $|\delta(s^0)| = 1$, and, for $s^q \leq s^p \leq S^N$, $\delta(s^q) \subseteq \delta(s^p)$.

Given this underlying face structure δ, I will prove that within every i-face of the complex each $g(i)$-sphere image has a span. To achieve this, I first show through a sequence of lemmas that a system that can achieve $(i, g(i)+1)$-narrowing (through resiliency and set consensus) can be simulated by $g(i)+1$ processors guaranteeing each simulator sees at least one final code view. Then I show that for every $g(i)$-sphere image of CVC and set of $g(i)+1$ simulators, there exists a particular simulation rule and simplicial map μ for which μ(SVC) is a span of the $g(i)$-sphere image. I begin with the relation between the number of simulators and the model simulated.

Lemma 7.2.3 *A wait-free protocol for i processors with access to a set of set consensus types $S = \{O_1 = (m_1, j_1)$-set consensus, ..., $O_l = (m_l, j_l)$-set consensus$\}$ can be simulated by n_i processors guaranteeing that each simulator sees one final code view, where n_i is defined in Lemma 5.1.2.*

Proof: For $m_k \leq i$ such that for all $h \leq i$, $m_k/j_k \geq m_h/j_h$, n_i is defined to be $(i \text{ div } m_k)j_k + n_{i_1}$ where $i_1 = i \bmod m_k$. Since for all $h \leq i$, $m_k/j_k \geq m_h/j_h$, a submit agreement for an (m_k, j_k)-set consensus object will block the greatest number of codes with the fewest number of simulator failures. Thus, $j_k(i \text{ div } m_k)$ failed simulators can block $i - (i \bmod m_k)$ codes. Inductively, n_{i_1} is the minimal number of simulators needed to block the remaining $i_1 = i \bmod m_k$ codes. Since the simulation is wait-free, if one simulator is nonfaulty, there must be at least one unblocked code available for simulation to completion. Thus, the lemma holds. ∎

Lemma 7.2.4 *A $g(i)$-resilient protocol for a system of i processors with access to set consensus types $\{O_1 = (m_1, k_1)$-set consensus, ..., $O_l = (m_l, k_l)$-set consensus$\}$ can be simulated by $n_{g(i)+1}$ processors guaranteeing that each simulator will see at least one final code view.*

Proof: By the previous lemma, $n_{g(i)+1}$ is the minimal number of simulator failures needed to block $g(i)+1$ codes through submit, write, read, and receive-reply agreements. Thus, for $n_{g(i)+1}$ wait-free simulators, a single non-faulty simulator will face at most $g(i)$ blocked codes out of the i codes. Since the protocol is $g(i)$-resilient, the remaining $i - g(i)$ codes can be simulated to completion, satisfying the lemma. ∎

Lemma 7.2.5 *Any M processor protocol written for a model of resiliency and set consensus that can solve $(i, g(i)+1)$-narrowing can be simulated by $g(M)+1$ simulators such that each simulator is guaranteed to see a code final view.*

Proof: By Lemmas 5.2.2 and 7.2.4, if a model can achieve $(i, g(i)+1)$-narrowing, an M code protocol in this model can be simulated by $g(M)+1$ processors such that every simulator can simulate to completion at least one code. Thus, with a simulation rule that is fair to each code in which a simulator halts only when no unblocked agreement remains to be executed, each simulator will see at least one final code view. ∎

Since a protocol in a model that can solve $(i, g(i)+1)$-narrowing by virtue of resiliency and set consensus can be simulated by $g(i)+1$ processors so that every simulator sees at least one final code view, then by Lemma 7.2.2 the image of a subdivided $g(i)$-simplex can be found in every i-face of CVC. Lemma 7.2.9 will prove that for a given image of a subdivided $g(i)$-simplex of CVC, simulators can negotiate to determine a simulation rule that gives rise to a simulation view complex SVC that maps directly into the image. I use this lemma to prove that every image of a $g(i)$-sphere in CVC has a span in CVC. First, I define some terms to simplify the proofs of the following lemmas.

I assume that a protocol for shared memory and set consensus objects is full-information, and thus every write and submit is the processor's complete local state. This local state is the *value* of the write or submit. In addition, the *value* of a read or receive-reply is the state of the memory read or the value of the submit returned, respectively. In a given execution, a write operation of processor i is *revealed* to processor j when j reads the write in the memory. Since a write of processor i is the processor's local state, every operation of processor i up to the write is *revealed* to processor j when the write itself is revealed to processor j. A submit operation of processor i to set consensus object k is *revealed* to processor j if the submit is returned in the receive-reply of processor j from object k. Accordingly, since the submit is the local state of processor i, all operations of processor i up to the submit are revealed to processor j when the submit is revealed to processor j. Read and receive-reply operations are only revealed once a subsequent write or submit operation is revealed. When an operation of processor i is revealed to processor j, the value of the operation is revealed as well. Every operation of processor i is revealed to processor i.

Definition The *set of operations revealed to processor* i in an execution contains all operations of processor i and, for each $j \neq i$, operations 1 to n_j of processor j, where operation n_j is the last (write or submit) operation revealed to processor i in the execution.

Lemma 7.2.6 *The set of operations revealed to processor* i *in an execution is well defined.*

Proof: The execution determines processor i's view, or local state upon halting. This local state is determined by the receive-reply from every set consensus object accessed by i, the last state of the memory read by i, and every operation i itself takes. Thus, for each $j \neq i$, let operation n_j be the last operation of processor j revealed to processor i. By definition, operation n_j is either a submit or a write. Thus, the set of operations revealed to processor i in the view contains all operations of processor i as well as operations 1 to n_j of processor j for all j in the view, and no other operation. ■

Lemma 7.2.7 *Given adjacent views* v_i, v_j *of processors* i *and* j *in the view complex of a full-information protocol using read-write memory and set consensus objects, any operation revealed to both views has the same value in both views.*

Proof: Since the views are adjacent there exists a run that gives rise to both views. Thus, the value of any operation revealed to both views is the value of the operation in the run, a single value. ∎

Lemma 7.2.8 *Given a view of process of processor i in an execution, the shortest sequence of operations that gives rise to the view is unique, modulo commuting operations.*

Proof: All operations revealed to the view must appear in any sequence of operations that gives rise to the view. Accordingly, any operation that is not revealed to the view need not occur in the sequence of operations, hence the set of operations comprising the sequence is unique. The linearization of this set is then determined, modulo commuting operations, by the value of each operation in the set. ∎

Lemma 7.2.9 *Let $\hat{\mu}(S^q) \subseteq CVC$ be the simplicial image in CVC of a subdivided simplex S^q. There exists a wait-free read-write protocol for $q+1$ processors with view complex VC and a simplicial map $\bar{\mu} : VC \rightarrow CVC$ such that $\bar{\mu}(VC) = \hat{\mu}(S^q)$.*

Proof: The $q+1$ processor protocol consists of a negotiation phase and a simulation phase. In the negotiation phase, each processor determines the simulation rule to be used in the simulation phase. In the negotiation phase, processors execute the ϵ-convergence phase of the simplex agreement algorithm of Chapter 6 on the span $\hat{\mu}(S^q)$ to obtain simplexes of $\hat{\mu}(S^q)$ that are contained in a common simplex. Each processor i chooses an arbitrary node v_i of its ϵ-convergence output simplex. These nodes are adjacent views of CVC. A processor i determines the unique shortest sequence of operations (modulo commuting operations) that gives rise to view v_i. The simulator determines its simulation rule (and what to submit to each agreement) in accordance with this sequence and enters the simulation phase, halting when view v_i has been produced.

By Lemma 7.2.7, simulators will execute operations according to sequences that agree on the values of any operation common to both sequences. Thus, each simulator submits to a read, write, submit, or receive-reply agreement the value determined by the sequence. Since a set of processors executing a read, write, or receive-reply agreement will submit the same value, the agreement will not block. For a submit agreement corresponding to an (m, j)-set consensus object, at most j values will be submitted, so any one is a legal output value, and again the agreement will not block. Thus, a simulator can return from a submit agreement to a receive-reply proposing the value that won in the view sequence. Since no agreement is blocked, processors simulate exactly the view $\hat{\mu}(v)$ they intended, and the simplicial map $\bar{\mu}$ is defined to map the resulting simulation view to $\hat{\mu}(v)$. Thus, the simulation rule determined via ϵ-convergence on S^q gives rise to a simulation view complex SVC and simplicial map $\bar{\mu}$ such that $\bar{\mu}$(SVC)= $\hat{\mu}(S^q)$. ■

Now I am ready to prove the pivotal lemma for the g-solidity of CVC in a model that can achieve $(i, g(i)+1)$-narrowing. This will prove condition 2 of the g-solidity of the CVC view complex.

Lemma 7.2.10 *Given a protocol for a model that achieves $(i, g(i)+1)$-narrowing by using a combination of set consensus objects and resiliency, then for any set P of i processor inputs, any $g(i)$-sphere image in $face_P(CVC)$ has a span.*

Proof: Let $g(i)+1$ simulators simulate the i codes under the following rules. Each simulator starts at a unique corner of the subdivided $g(i)$-simplex that simplicially maps to the sphere image. The simulators converge on the face they define as long as they see $\leq g(i)$ processors simulating. Upon converging to any node v, a processor begins simulating the protocol using the simulation rule determined by the view of CVC that v maps to. A simulator that sees $g(i)+1$ active simulators or gets blocked in an agreement may resort to any fair simulation rule that includes halting when every code has either terminated or been blocked through an agreement. If less than $g(i)+1$ simulators arrive, all and only views on the sphere image will be produced. Thus, the faces of SVC can be simplicially mapped to the sphere by mapping a simulator view to the vertex that determined its simulation rule. Since the simulation view complex is a subdivided simplex, I define an extension to the simplicial map (from the boundary of SVC) to the sphere image of CVC that maps interior nodes of SVC to any view of CVC seen simulated. Thus, the simplicial image of SVC is a span of the sphere image in CVC. ■

Theorem 7.2.11 (g-solidity) *The view complex CVC of a protocol for $N+1$ processors in a model that achieves $(i, g(i)+1)$-narrowing by using a combination of set consensus objects and resiliency is a g-solid of dimension N with underlying face structure δ.*

Proof:

1. By Lemma 7.2.10, with $i \leq N+1$, every $g(N+1)$-sphere image has a span.
2. Notice that given any simplex of views $s^q \in CVC$, the simulation may be started anew from the point where the last to halt view of s^q terminated.

Thus, the remaining $N+1-(q+1)$ codes may be simulated, creating views necessarily adjacent to views of s^q. Let the underlying face structure δ' of the linking sphere be defined so that $face_T(ls(s^q,\text{CVC}))$ contains all views of $ls(s^q,\text{CVC})$ in which, beyond operations seen in views of s^q, only operations of processors in T are seen. Then, by the simulation, the g-solidity of the $N-q$ dimensional complex $ls(s^q,\text{CVC})$ follows by analogy to the above.

3. By definition of the underlying face structure of CVC, for any set of IDs S, $face_S(\text{CVC})$ is simply a complete protocol over the processors in S. Thus, the protocol can be simulated and, inductively (applying Lemma 7.2.10), is g-solid.

∎

7.3 The Structure of t-Resiliency and Set Consensus

In Chapter 6, I showed that processors in a t-resilient model with set consensus objects can achieve simplex agreement on a $(g(i,j)-1)$-solid complex with underlying $N-t$ face structure in a model that solves $(i, g(i,j))$-narrowing when i processors are active and j are halted. Now I prove that the view complex of any full-information t-resilient protocol using set consensus objects is $(g(i,j)-1)$-solid with underlying $N-t$ face structure. As in the active resiliency model, we define the underlying face structure of a protocol view complex on the basis of the set of processors seen in each view. Given a chromatic simplex S^N, and a view complex VC^N, define $\delta : S^N \to VC^N$ as follows. For all $q \geq N-t$, let $\delta(s^q)$ be the complex induced by all vertices v such that for all processors i seen in view v, $i \in IDs(s^q)$. Since all processors in such a protocol wait until at least $N+1-t$ processors take a step, δ is not defined for faces of dimension less than $N-t$.

Lemma 7.3.1 *A protocol for $N+1$ processors with t-resiliency and access to a set of set consensus types $\{O_1 = (m_1, k_1)\text{-set consensus}, ..., O_l = (m_l, k_l)\text{-set consensus}\}$ can be simulated by $n_{T(N+1-j,j)+1}$ processors from any point where j codes are halted so that each simulator will see at least one final code view from the $N+1-j$ unhalted codes.*

Proof: By Lemma 5.1.2, n_i is the minimal number of simulator failures needed to block i codes through submit, write, read, and receive-reply agreements for a protocol using the set consensus types stated. Thus, for $n_{T(N+1-j,j)+1}$ wait-free simulators, a single nonfaulty simulator will face at most $T(N+1-j,j)$ blocked codes out of the $N+1-j$ unhalted codes. Since the protocol is t-resilient, the codes must be able to proceed when at most $T(N+1-j,j)$ codes are blocked. Thus, at least one of the $N+1-j$ active codes may be simulated to completion, satisfying the lemma. ■

Corollary 7.3.2 *A protocol for i processors with t-resiliency and access to a set of set consensus types $\{O_1 = (m_1, k_1)\text{-set consensus}, ..., O_l = (m_l, k_l)\text{-set consensus}\}$ can be simulated by $n_{T(i-j,j)+1}$ processors from any point where j codes are halted so that each simulator will see at least one final code view from the $i-j$ unhalted codes.*

Lemma 7.3.3 *Given an $N+1$ processor protocol with view complex CVC, in a model that solves $(i, g(i,j))$-narrowing by using a combination of set consensus objects and t-resiliency, then for any set of processors P such that $|P| \geq N+1-t$, any $(g(|P|,0)-1)$-sphere image in $face_P(CVC)$ has a span.*

Proof: Let $g(|P|,0)$ simulators simulate the codes of the processors in P under the following rules. First, processors negotiate for a simulation rule by executing ϵ-convergence starting at a unique corner of the subdivided $(g(|P|,0)-1)$-simplex that maps to the sphere image. The simulators converge on the face they define as long as they see $< g(|P|,0)$ processors. Upon converging to any node, a processor simulates the protocol using the simulation rule determined by the corresponding view of CVC. Any simulator that sees $g(|P|,0)$ simulators active or gets blocked in an agreement may resort to any fair simulation rule that includes stopping when every code either has terminated or is blocked through an agreement. If less than $g(|P|,0)$ simulators arrive, they will produce all and only views on the sphere image, thus there exists a simplicial map μ from faces of SVC to the sphere image. Since the simulation view complex is a subdivided simplex, define a simplicial extension to μ that maps interior nodes of SVC to any final code view of CVC seen simulated. Thus, the simplicial image of SVC is a span of the sphere image in CVC. ■

Lemma 7.3.4 *Given a protocol for $N+1$ processors with view complex CVC written for a model in which $(i, n_{T(i,j)+1})$-narrowing is solvable through the set consensus and t-resiliency when i processors are active, j are halted, and $i+j \geq N+1-t$, then, given a simplex $s^{j-1} \in CVC$, any $(n_{T(N+1-j,j)+1}-1)$-sphere image in $ls(s^{j-1}, CVC)$ has a span.*

Proof: Let $n_{T(N+1-j,j)+1}$ simulators negotiate for a simulation rule as follows. Each simulator starts at a unique corner of the subdivided $(n_{T(N+1-j,j)+1}-1)$-simplex that maps to the sphere image. The simulators converge on the face they determine as long as they see less than $n_{T(N+1-j,j)+1}$ simulators active. Upon converging to any node, a processor first simulates the views of s^{j-1} and then simulates the view it has converged to, using the simulation rule determined by the image of the node in the sphere (a view of CVC). A simulator that sees $n_{T(N+1-j,j)+1}$ other simulators or gets blocked in an agreement may resort to any fair simulation rule that includes first simulating views of s^{j-1} and then stopping when every code has either terminated or been blocked through an agreement. Since it takes the failure of $n_{T(N+1-j,j)+1}$ simulators to block the simulation of $N+1-j$ codes, if one simulator remains nonfaulty, then the views of s^{j-1} and at least one view of the $N+1-j$ will be simulated to completion.

If less than $N_{T(N+1-j,j)+1}$ simulators are active, they will produce all and only views on the sphere image, thus the faces of SVC can be simplicially mapped by some map μ to the sphere image (mapping a node to the vertex that defined its simulation rule). Since the simulation view complex is a subdivided simplex with each view simulating the final views of s^{j-1} and an additional final code view, we can define an extension to μ that maps interior nodes of SVC to any view of $ls(s^{j-1}$,CVC) seen simulated. Thus, the simplicial image of SVC is a span of the sphere image in $ls(s^{j-1}$,CVC). ■

Corollary 7.3.5 *Given a protocol for $N+1$ processors with view complex CVC written for a model in which $(i, n_{T(i,j)+1})$-narrowing is solvable through set consensus and t-resiliency when i processors are active, j are halted, and $i+j \geq N+1-t$, then, given a simplex $s^{j-1} \in face_P(CVC)$, with $|P| \geq N+1-t$, any $(n_{T(|P|-j,j)+1}-1)$-sphere image in $ls(s^{j-1}, face_P(CVC))$ has a span.*

The last two lemmas prove that the appropriate dimension spans exist in the view complex of a t-resilient protocol using set consensus objects for $(n_{T(i,j)+1})-1)$-solidity. The corollaries show that the appropriate spans exist in the faces as well.

The following theorem unifies these results to show that the view complex of an $N+1$ processor t-resilient protocol for shared memory with set consensus objects is $(n_{T(i,j)+1}-1)$-solid with underlying $N-t$ face structure. In Chapter 8 I present the full characterization of tasks solvable t-resiliently with set consensus objects and shared memory.

Theorem 7.3.6 *The view complex CVC of an $N+1$ processor t-resilient protocol in a model that solves $(i, n_{T(i,j)+1})$)-narrowing for $i+j \geq N+1-t$, among i active and j halted processors, through the resiliency and set consensus objects is $(n_{T(i,j)+1}-1)$-solid of dimension N with underlying $N-t$ face structure δ.*

Proof: Let the underlying face structure δ be determined by the set of awake processors seen in each view. Since the protocol is t-resilient, processors may wait until at least $N+1-t$ inputs are revealed, so the face structure is defined for sets of size $\geq N+1-t$. By Lemma 7.3.3, every i-sphere image of dimension $\leq n_{T(N+1,0)+1}-1$ has a span, so condition 1 of $(n_{T(i,j)+1}-1)$-solidity holds. By Lemma 7.3.4, the linking sphere of any $(j-1)$-simplex in CVC contains a span for every $(n_{T(N+1-j,j)+1}-1)$-sphere image. Since the number of processors that may fail is now dependent on the number awake instead of the number active, the face structure of the CVC induces the face structure of the linking sphere. By Lemma 7.3.3 and Corollary 7.3.5, each $q \geq N-t$ dimensional face of CVC is $(n_{T(i,j)+1}-1)$-solid as well, so CVC is $(n_{T(i,j)+1}-1)$-solid of dimension N with underlying $N-t$ face structure δ. ∎

CHAPTER 8

Characterization Theorems

In this chapter, I present a complete characterization of solvability in models of set consensus with either active resiliency or t-resiliency.

8.1 Active Resiliency with Set Consensus Objects

For models of active resiliency and set consensus objects, the heart of the characterization has already been proven. It remains only to unify the results into a single theorem. I first state the characterization for single-input tasks in models of active resiliency and set consensus. Then I extend the theorem to the multi-input case.

Theorem 8.1.1 *Let $T = (I^N, O^N, \Delta)$ be an $(N+1)$-processor single-input task. T is solvable in a model of active resiliency and set consensus that solves $(i, g(i)+1)$-narrowing iff there exist a g-solid complex Φ^N with underlying face structure $\delta : I^N \to \Phi^N$ and a chromatic simplicial map $\mu : \Phi^n \to O^n$ such that $\forall s^q \subseteq I^N,\ \mu(\delta(s^q)) \subseteq \Delta(s^q)$.*

Proof: ($\Rightarrow$) Assume T is solvable in the model of active resiliency and set consensus. Then, by Theorem 7.2.11 the view complex CVC of a protocol solving the task is g-solid with underlying face structure δ such that any view v in $\delta(s^q)$ contains only operations of processors in $IDs(s^q)$. Let Φ^n =CVC. Let μ be the chromatic simplicial map from views to outputs induced by the protocol. Then, $\forall s^q \subseteq I^N,\ \mu(\delta(s^q)) \subseteq \Delta(s^q)$ since a view of $\delta(s^q)$ sees only inputs of s^q.

($\Leftarrow$) Assume there exists a g-solid complex Φ^N with underlying face structure δ and a chromatic simplicial map $\mu : \Phi^n \to O^n$ such that, $\forall s^q \subseteq I^N,\ \mu(\delta(s^q)) \subseteq \Delta(s^q)$. By Theorem 6.4.1 and the assumption that the model solves $(i, g(i)+1)$-narrowing, the $N+1$ processors can solve simplex agreement on Φ^n such that if only a set of processors S is active, then the output vertices lie in $face_S(\Phi^N)$. Thus, a processor with node $v \in \Phi^N$ as output to the simplex agreement may take $\mu(v)$ as output to the task. The outputs will be compatible since μ is a chromatic simplicial map and since processors obtain nodes of a common simplex in the simplex agreement. The outputs meet the task specification since $\mu(\delta(s^q)) \subseteq \Delta(s^q)$. Thus, processors obtain compatible task outputs, valid for the inputs they have seen. ■

A multi-input task can be viewed as a collection of single-input tasks. Thus, the complex Φ must have an underlying face structure that reflects the entire input complex I^N. I now generalize the definition of *underlying face structure* δ as follows. Complex Φ^N has underlying face structure $\delta(I^N)$ if there exists a complex I^N and a chromatic map δ from simplices of I^N to sets of simplices of Φ^N such that $\delta(I^N) = \Phi^N, |\delta(s^0)| = 1$ and for $s^q \leq s^p \in I^N,\ \delta(s^q) \subseteq \delta(s^p)$. With the generalized definition of underlying face structure, the proof follows the single-input case directly.

Theorem 8.1.2 *Let $T = (I^N, O^N, \Delta)$ be an $N+1$ processor task. Then T is solvable in a model of active resiliency and set consensus that solves $(i, g(i)+1)$-narrowing iff there exists a complex Φ^N with underlying face structure $\delta(I^N)$ such that, $\forall s^q \in I^N, \delta(s^q)$ is g-solid and a chromatic simplicial map $\mu : \Phi^n \to O^n$ such that, $\forall s^q \subseteq I^N,\ \mu(\delta(s^q)) \subseteq \Delta(s^q)$.*

Proof: ($\Rightarrow$) Assume T is solvable in the model of resiliency and set consensus. Then, by Theorem 7.2.11, for each input simplex S^N, the view complex CVC of any protocol solving the task is g-solid with underlying face structure $\delta(S^N)$ such that views of $\delta(s^q)$ see only inputs s^q. Thus, this face structure agrees on the intersection of any two input simplexes and induces a face structure δ for CVC such that, for $s^q \leq s^p$, $\delta(s^q) \subset \delta(s^p)$. Let Φ^n =CVC. Then, letting μ be the chromatic simplicial map from views to outputs induced by the protocol, it holds that, $\forall s^q \subseteq I^N$, $\mu(\delta(s^q)) \subseteq \Delta(s^q)$, as in the single-input case.

($\Leftarrow$) Assume there exists a g-solid complex Φ^N with underlying face structure $\delta(I^N)$ and a chromatic simplicial map $\mu : \Phi^n \rightarrow O^n$ such that, $\forall s^q \subseteq I^N$, $\mu(\delta(s^q)) \subseteq \Delta(s^q)$. Then, since the simplex agreement algorithm is limited to the face defined by the active processors, the inputs of the active processors determine the complex over which the processors converge in simplex agreement. Thus, by Theorem 6.4.1 and the assumption that the model solves $(i, g(i)+1)$-narrowing, the $N+1$ processors can solve simplex agreement on Φ^n such that if only processors of S^q are active, then their output vertices lie in $\delta(S^q)$. Thus, a processor with node $v \in \Phi^N$ as output to the simplex agreement may take $\mu(v)$ as legal output to the task. ∎

The characterization condition relies solely on the ability of a model to solve $(i, g(i)+1)$-narrowing. Thus, any two models with the same narrowing function are equivalent in task implementation power. Since the set consensus and resiliency function determine the narrowing function, a trade-off exists between the set consensus and resiliency necessary to implement a given task. For example, any task solvable in a model that solves $(i, g(i)+1)$-narrowing is solvable either in read-write memory with resiliency function g or in a wait-free model with set consensus objects that can solve $(i, g(i)+1)$-narrowing.

8.2 t-Resiliency with Set Consensus Objects

Based on the solvability of simplex agreement and the structure of a t-resilient protocol view complex, the full characterization of the t-resilient model with access to shared memory and set consensus objects may be stated. In order to present the theorem directly for the multi-input case, I will first define the underlying $N-t$ face structure of the multi-input complex. Again, the multi-input task is viewed as a collection of single-input tasks, and thus the underlying $N-t$ face structure reflects the entire input complex I^N. Complex Φ^N has *underlying $N-t$ face structure* $\delta(I^N)$ if there exists a complex I^N and a chromatic map δ from simplices of I^N to sets of simplices of Φ^N such that $\delta(I^N) = \Phi^N$ and, for $q \geq N-t,\ s^q \leq s^p \in I^N,\ \delta(s^q) \subseteq \delta(s^p)$.

Theorem 8.2.1 *Let $T = (I^N, O^N, \Delta)$ be an $N+1$ processor task. Then T is solvable in a model of t-resiliency and set consensus that solves $(i, g(i,j))$-narrowing for i processors active and j halted such that $i+j \geq N+1-t$ iff there exists a chromatic complex Φ^N with underlying $N-t$ face structure $\delta : I^N \to \Phi^N$ and a chromatic simplicial map $\mu : \Phi^N \to O^n$ where, $\forall s^q \in I^N,\ \mu(\delta(s^q)) \subseteq \Delta(s^q)$, such that, $\forall s^N \in I^N,\ \delta(s^N)$ is $(g-1)$-solid with induced underlying $N-t$ face structure δ.*

Proof: $(\Rightarrow)$ Assume T is solvable in the model. Then, by Theorem 7.3.6, the view complex CVC of any protocol solving the task for a single input is $(g(i,j)-1)$-solid with underlying $N-t$ face structure δ. By Lemma 7.3.6, δ is defined such that any view v in $\delta(s^q)$ sees only operations of processors in $IDs(s^q)$. Let $\Phi^N =$CVC.

Then, letting μ be the chromatic simplicial map (induced by the protocol) that maps each view of CVC to an output value, the correctness of the protocol implies $\mu(\delta(s^q)) \subseteq \Delta(s^q), \forall s^q \subseteq I^N$.

($\Leftarrow$) Assume there exists a complex Φ^N with underlying $N-t$ face structure $\delta(I^N)$ such that $\forall S^N \in I^N$, $\delta(S^N)$ is $(g-1)$-solid, and assume there exists a chromatic simplicial map $\mu : \Phi^N \to O^N$ such that, $\forall s^q \subseteq I^N$, $\mu(\delta(s^q)) \subseteq \Delta(s^q)$. By Theorem 6.5.1, the $N+1$ processors can solve simplex agreement on Φ^N such that if only processors with inputs S^q are active and $q \geq N-t$, then their vertices lie in $\delta(S^q)$. Thus, a processor deciding node $v \in \Phi^N$ as output to the simplex agreement may take $\mu(v)$ as output to the task. This output meets the task specification since $\mu(\delta(s^q)) \subseteq \Delta(s^q)$, so processors obtain outputs compatible with the (at least $N+1-t$) inputs they have seen. ■

CHAPTER 9

Applications to Set Consensus

I have presented a full characterization of task solvability in models of active resiliency and t-resiliency with communication through shared memory and set consensus objects. The characterization condition is existential; in fact, Gafni and Koutsoupias [18] have proven that determining whether a task satisfies the condition is an undecidable problem. Thus, although the characterization provides insight into general computability theory, determining whether a specific task is solvable in a particular model requires other methods.

In this chapter, I prove several specific results concerning the computability of set consensus in various models. First, I show that k-set consensus is impossible in a system of $k+1$ processors working wait-free in read-write memory [9, 19, 32]. Generalizing to the t-resiliency model, I then confirm Chaudhuri's [12] conjecture that k-set consensus is not k-resiliently solvable. Finally, I prove that a partial hierarchy exists between set consensus objects.

9.1 Wait-Free Read-Write $k+1$ Processor k-Set Consensus

By [20], the view graph of any wait-free full-information iterated immediate atomic snapshot protocol is a subdivided simplex with lower dimensional view graphs as faces. In order for $k+1$ processors to wait-free implement k-set consensus in read-write memory, there must be a chromatic simplicial map from nodes of the view graph to outputs that satisfy the set consensus task. Without loss of generality, assume each processor's input is its own ID. Then, the simplicial map to the output complex can be viewed as a coloring of the view graph by output IDs. For the protocol to correctly implement the k-set consensus task, the coloring must satisfy two properties. First, to ensure that each run produces at most k outputs, each simplex must be colored by no more than k colors. Moreover, each processor must choose an active processor as output. Thus, since the view graph is a subdivision of a chromatic simplex with view graphs of lower dimension as faces, the color (output) of a node must be among the colors of the carrier of that node in the subdivided simplex. A coloring of a subdivided simplex satisfying this last requirement is a *Sperner coloring*, with such a subdivided simplex referred to as *Sperner chromatic.* I will now present Sperner's Lemma [8], which will be used to show k-set consensus is impossible in a system of $k+1$ processors working wait-free in read-write memory.

Lemma 9.1.1 (Sperner's lemma [8]) *A Sperner chromatic subdivided simplex C^k that contains an odd number of boundary $(k-1)$-simplexes s^{k-1} with $IDs(s^{k-1}) = \{0, ..., k-1\}$ contains an odd number of k-simplexes s^k with $IDs(s^k) = \{0, ..., k\}$. In particular, there must be at least one such $(k+1)$-colored k-simplex.*

Theorem 9.1.2 *The k-set consensus task is impossible in a wait-free $k+1$ processor system of read-write shared memory.*

Proof: Assume $k+1$ processors can solve k-set consensus wait-free in read-write memory. Then, for some N, there exists a full-information iterated immediate atomic snapshot protocol that solves k-set consensus in which each processor takes exactly N steps. By [20], the view graph of this protocol is a subdivided simplex with the view graphs of each subset of $k-1$ processors comprising the boundary.

Since the protocol implements k-set consensus there must be a Sperner coloring of the graph by outputs such that each k-simplex is colored by at most k colors.

The view graph (face) corresponding to the single processor 0 is a single vertex — a view that sees only processor 0 active. Thus, this node must be colored by color 0. Inductively, assume the view graph corresponding to the set of processors $\{0, ..., i\}$ contains an odd number of simplexes s^i such that $IDs(s^i) = \{0, ..., i\}$. This graph is the only face of the view graph of processors $\{0, ..., i+1\}$ that may contain i-simplexes colored $0, ..., i$, so the view graph of processors $\{0, ..., i+1\}$ contains an odd number of boundary i-simplexes colored with $IDs = \{0, ..., i\}$. By Sperner's lemma, the view graph of processors $0, ..., i+1$ must contain an odd number of i-simplexes colored $0, .., i+1$.

With $i = k-1$, Sperner's lemma proves that any sperner coloring of a wait-free read-write view graph contains at least one k-simplex colored by all $k+1$ colors. Such a simplex corresponds to a run in which k-set consensus is not achieved. Thus, k-set consensus is impossible for a $k+1$ processor wait-free read-write shared memory system. ∎

9.2 Impossibility of k-Resilient k-Set Consensus

To extend the impossibility proof of wait-free (k-resilient) $k+1$ processor k-set consensus to the general case of k-resilient k-set consensus, I use the simulation technique of Chapter 7. We reduce $k+1$ processor wait-free k-set consensus to n processor k-resilient k-set consensus. Thus, by the results of the last section, k-resilient k-set consensus must be impossible.

Theorem 9.2.1 *The k-set consensus task is impossible in a k-resilient system of $N+1>k$ processors with read-write memory.*

Proof: Assume a system of $N+1>k$ processors can solve k-set consensus k-resiliently in read-write memory, and let P be a protocol that solves it. Let $k+1$ processors wait-free simulate protocol P in read-write memory. If such a protocol exists, then the $k+1$ simulators can wait-free execute k-set consensus, contradicting Theorem 9.1.2

Before each code is simulated, simulators execute an agreement protocol to associate a simulator with the code. Simulators submit themselves to be associated with each code. Just as for read and write agreements, one faulty processor can block the agreement, and only one agreement value (one simulator) may be associated with each code. Once a code is associated with a simulator, it may be simulated as in Chapter 7. Since the $k+1$ simulators work wait-free, at most k of the $N+1>k$ codes will be blocked. Since the protocol is k-resilient, the $N+1-k$ unblocked codes may be simulated to completion by any remaining simulator. Thus, each simulator can be guaranteed to see at least one terminated code and hence one k-set consensus choice. A simulator takes as output to k-set consensus the simulator associated with any code winning the simulated k-set consensus. Since the protocol implements k-set consensus, at most k codes will win, and thus at most k simulators (associated with the winning codes) will be chosen as output. By the requirements of k-set consensus, any winning code must have had some of its operations simulated, and thus there must be a simulator associated with the code. Therefore, if an $N+1$ processor k-resilient read-write protocol exists for k-set consensus, a $k+1$ processor wait-free read-write protocol must exist for k-set consensus. This contradicts Theorem 9.1.2, so k-set consensus is k-resiliently impossible in read-write memory. ■

9.3 The Set Consensus Hierarchy

I have shown that set consensus is not achievable wait-free in read-write memory; however, I have yet to answer the question of whether k-set consensus for n processors is achievable when given access to (m, j)-set consensus objects. Here, I address exactly this question, showing that the answer depends entirely on the values n, k, m, and j. In particular, if $k < j\lceil n/m \rceil$ and $k - j\lfloor n/m \rfloor < n - m\lfloor n/m \rfloor$ [13], or if $m/j < n/k$, or if $j > k$, then n processor k-set consensus cannot be implemented in a wait-free system of shared memory and (m, j)-set consensus objects. Limiting the values of k and j to 1, this rederives the result of [22], which states that m processor consensus cannot implement n processor consensus for $m < n$. Although consensus objects can be partitioned into a strict hierarchy on this basis, set consensus objects form only a partial hierarchy. For example, object types (n, k)-set consensus and (m, j)-set consensus are *incomparable* in that neither object can be used to implement the other. An example of this is $(2, 1)$-set consensus versus $(5, 2)$-set consensus. On the one hand, $(5, 2)$-set consensus object cannot be used to achieve (1-set) consensus for even two processors. On the other hand, when used by five processors, $(2, 1)$-set consensus can only narrow inputs to three choices, thus implementing $(5, 3)$-set consensus but not $(5, 2)$-set consensus.

Lemma 9.3.1 *(m, j)-Set consensus cannot implement n processor k-set consensus if $m/j < n/k$.*

Proof: Assume the converse. Then there exists an n processor protocol using (m, j)-set consensus objects which achieves k-set consensus. Thus, by using j

copies of the protocol, there exists an nj processor protocol P using (m, j)-set consensus that achieves jk-set consensus. Letting $jk+1$ processors simulate P wait-free in read-write memory, I will show that the simulators can implement jk-set consensus wait-free in read-write memory — a contradiction. Simulators implement an agreement protocol (one for each code) to associate each code with a simulator before any operations of that code are simulated. Once codes are associated with simulators, the simulators may use any fair simulation rule that includes terminating only after at least one final view is simulated.

Since the codes use (m, j)-set consensus, the failure of j simulators can block at most m codes. If jk of the $jk+1$ wait-free simulators fail, they will block at most mk codes. Since $m/j < n/k$ is assumed, it holds that less than nj codes may be blocked. Thus, each nonfaulty simulator will simulate to completion at least one code, taking the simulator associated with the winning code as output. Since at most jk codes will win the jn processor jk-set consensus protocol, the simulators themselves will take at most jk values, solving wait-free $jk+1$ processor jk-set consensus in read-write memory. This contradicts Theorem 9.1.2, so there cannot be a wait-free n processor k-set consensus protocol using (m, j)-set consensus objects if $m/j < n/k$. ∎

Lemma 9.3.2 *(m, j)-Set consensus cannot implement n processor k-set consensus if $k < j$.*

Proof: Assume, to the contrary, that there exists an n processor k-set consensus protocol using (m, j)-set consensus objects and that $k < j$. Then, let $k+1 \leq j$ simulators simulate the protocol, executing an agreement to associate each code with a simulator and halting when at least one code is simulated to completion.

Since j failed simulators are required to block m codes through a submit agreement, and since only $k+1 \leq j$ processors may simulate, if one simulator remains nonfaulty, no submit agreement can be blocked. Thus, each failed simulator blocks at most one code, so at most k of the $n > k$ codes will be blocked. Thus, any nonfaulty simulator can simulate to completion at least one code. Upon determining a single code that wins the k-set consensus, a simulator can take as output the processor associated with the winning code. Since at most k codes win the simulated protocol set consensus, at most k of the $k+1$ simulators will be chosen. Thus, $k+1$ simulators can wait-free achieve k-set consensus in read-write memory, contradicting Theorem 9.1.2. Thus, the initial assumption is false, and the lemma holds. ∎

Lemma 9.3.3 ([13]) *(m, j)-Set consensus cannot implement n processor k-set consensus if $k < j\lceil n/m \rceil$ and $k - j\lfloor n/m \rfloor < n - m\lfloor n/m \rfloor$.*

The following lemma shows the converse, that (m, j)-set consensus can implement (n, k)-set consensus if $n/k \leq m/j$ and $k \geq j$ and either $k \geq j\lceil n/m \rceil$ or $k - j\lfloor n/m \rfloor \geq n - m\lfloor n/m \rfloor$. This proves the complete relationship between any two set consensus objects.

Lemma 9.3.4 ([13]) *A system of n processors with (m, j)-set consensus objects can solve k-set consensus wait-free if $n/k \leq m/j$ and $k \geq j$ and either $k \geq j\lceil n/m \rceil$ or $k - j\lfloor n/m \rfloor \geq n - m\lfloor n/m \rfloor$.*

Proof: From Chapter 6, a system of n processors using (m, j)-set consensus can achieve $j\lfloor n/m \rfloor + \min(j, n \bmod m) = i$-set consensus. Thus, to prove the lemma, $i \leq k$ must be shown.

First, let $n \bmod m = 0$. Then, $j\lfloor n/m \rfloor + \min(j, n \bmod m) = j(n/m) \leq k$, and the lemma holds. Notice that the last step of this equation is due to the fact that $n/k \geq m/j$.

Thus, assume $0 < n \bmod m$. If $k \geq j\lceil n/m \rceil$, then $j\lfloor n/m \rfloor + \min(j, n \bmod m) \leq j\lfloor n/m \rfloor + j \leq j\lceil n/m \rceil \leq k$.

Otherwise, $k - j\lfloor n/m \rfloor \geq n - m\lfloor n/m \rfloor$ and $j\lfloor n/m \rfloor + \min(j, n \bmod m) \leq k - n + m\lfloor n/m \rfloor + (n \bmod m) \leq k - n + n = k$.

Thus, the lemma holds, and (n, k)-set consensus can implement (m, j)-set consensus. ∎

CHAPTER 10

Conclusion

In 1985, Fischer, Lynch, and Paterson [16] proved that the set of tasks solvable in an asynchronous distributed system of message passing when no processors may fail is distinct from the set solvable when even one processor may fail. In other words, the consensus task is impossible in the face of a single possible failure, but solvable in a totally reliable system. In 1987, Loui and Abu-Amara [24] extended this result to the shared memory model. These results inspired a line of research investigating the precise tasks solvable in the presence of failures.

In 1988, Biran, Moran, and Zaks [7] completely characterized the distributed tasks solvable in asynchronous shared memory systems when at most one failure may occur. Their solution was based on the connectivity of a graph derived from the task output. Taking the first step toward extending these results to the case of multiple failures, Chaudhuri [12] defined k-set consensus, a generalization of the consensus task. She proved that the k-set consensus task was solvable in the face of $k - 1$ failures and conjectured that it was insolvable in the face of k. She was absolutely right.

In 1993, three teams [9, 19, 32] proved Chaudhuri's conjecture correct for a model of $k + 1$ processors in which k processors may fail — the wait-free model. One team, Herlihy and Shavit [19] saw directly that the solvability or impossibility of a task relied on the topological structure of the computation in the model and of the task output. They gave a necessary condition for solvability in the wait-free read-write model. In 1994, Herlihy and Shavit [20] were able to complete their characterization of the wait-free read-write model.

This thesis generalizes the work of [7, 9, 19, 20]. Bridging the gap between the characterization of the read-write model in the face of one failure [7] and its characterization in the face of all but one failure (i.e., wait-free computation) [19, 20], I characterize tasks solvable in the read-write model in the face of t failures, for arbitrary t. In addition, I extend the characterization to encompass communication models that allow any collection of set consensus types to supplement the shared memory. Further, I introduce a new fault model — the active resiliency model — and characterize the tasks solvable in this model with communication through shared memory with set consensus objects. In the active resiliency model, the number of failures is a function of the number of processors active in the system. This contrasts with the t-resilient model, in which at most a constant t processors may fail out of the $N+1$ processors in the system. Unlike the t-resiliency model, for large systems, in which the number of processors may not be known and in which processors may work on different tasks, the active resiliency model accurately captures the conditions under which tasks must be solved: only a subset of the processors may need to solve a particular task, and the number of possible failures among these is a probabilistic function of the number actively solving the task.

Characterizing general models of distributed computation clarifies the critical difficulties present in distributed system design. In this thesis, I have introduced a simplified model of read-write shared memory, which has already been used to ease algorithm design [10, 4, 26]. I have proven an explicit trade-off between models of set consensus communication with many possible failures and shared memory models with fewer possible failures. Indeed, for every model of set consensus, we now can deduce the resiliency function necessary for a shared memory system to implement the same set of tasks.

Adding to the tools available to distributed system researchers, I have introduced a simulation technique whereby results in the wait-free read-write model can be used to imply results in models of set consensus and resiliency. I used the simulation not only for the characterization but to prove k-set consensus is k-resiliently impossible in shared memory and to provide the complete relationship between any pair of set consensus objects. In addition, I have presented a general simplex agreement algorithm that can serve as a universal algorithm for distributed computation in models of resiliency and set consensus. Given a complex and a simplicial map to output for any task, processors may solve simplex agreement on the complex and thus solve the task.

The work presented here serves as a foundation for understanding asynchronous distributed computation in faulty systems of shared memory and set consensus. Such models are equivalent in power to all commonly implemented communication primitives: message passing, test-and-set, and read-modify-write registers for example. However, much work remains to be done. The characterization does not produce an algorithm for a given task, nor does it determine whether new communication objects are equivalent to any of the models characterized. Moreover, once an algorithm is produced, to be useful it must be efficient with respect to time and space constraints, issues neglected in the characterization. Thus, researchers in the field of distributed computing have many problems yet to solve. Nonetheless, this thesis smoothes the road by providing a framework for determining which tasks may be tackled successfully and by contributing a number of new techniques to the problem-solving arsenal of distributed system developers.

Bibliography

[1] Martin Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 2(82):253–284, 1991.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Mike Merrit, and Nir Shavit. Atomic Snapshots of Shared Memory. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 1–13, 1990.

[3] Yehuda Afek, Danny Dolev, Eli Gafni, Mike Merritt, and Nir Shavit. First-In-First-Enabled l-Exclusion. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, 1990.

[4] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A Completeness Theorem for a Class of Synchronization Objects. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 159–170, 1993.

[5] Hagit Attiya, A. Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message Passing Systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 363–375, 1990.

[6] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are Wait-Free Algorithms Fast? In *Proceedings of the 31st Symposium on Foundations of Computer Science*, 1990.

[7] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A Combinatorial Characterization of the Distributed Tasks which Are Solvable in the Presence of One Faulty Processor. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 263–275, 1988.

[8] J. Bondy and U. Murty. *Graph Theory with Applications.* North Holland, New York, 1976.

[9] Elizabeth Borowsky and Eli Gafni. Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 91–100, 1993.

[10] Elizabeth Borowsky and Eli Gafni. Immediate Atomic Snapshots and Fast Renaming. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 41–51, 1993.

[11] Jim Burns and Gary Peterson. Constructing Multi-Reader Atomic Values from Non-Atomic Values. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[12] Soma Chaudhuri. More *Choices* Allow More *Faults*: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105:132–158, 1993.

[13] Soma Chaudhuri and Paul Reiners. Understanding the Set Consensus Hierarchy. Manuscript, 1995.

[14] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33(3):499–516, 1986.

[15] Alan Fekete. Asymptotically Optimal Algorithms for Approximate Agreement. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, 1986.

[16] Michael Fischer, Nancy Lynch, and Michael Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[17] G. Frederickson and N. Lynch. Electing a Leader in a Synchronous Ring. *Journal of the ACM*, 34(1):98–115, 1987.

[18] E. Gafni and E. Koutsoupias. 3-processor Tasks are Undecidable. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995.

[19] Maurice Herlihy and Nir Shavit. The Asynchronous Computability Theorem for t-Resilient Tasks. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 111–120, 1993.

[20] Maurice Herlihy and Nir Shavit. A Simple Constructive Computability Theorem for Wait-Free Computation. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, 1994.

[21] Maurice Herlihy and Jeannette Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[22] Maurice P. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[23] Leslie Lamport. On Interprocessor Communication, Parts I and II. *Distributed Computing*, 1:77–101, 1986.

[24] M. C. Loui and H. H. Abu-Amara. Memory Requirements for Agreement Among Unreliable Asynchronous Processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, CT., 1987.

[25] Nancy Lynch and Sergio Rasjbaum. A New Look at the Borowsky-Gafni Simulation Algorithm. Manuscript, 1995.

[26] Mark Moir and James Anderson. Wait-Free Algorithms for Fast, Long-Lived Renaming. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 145–155, 1994.

[27] James R. Munkres. *Topology; A First Course.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[28] James R. Munkres. *Elements of Algebraic Topology.* Addison-Wesley, Menlo Park, CA, 1984.

[29] Gil Neiger. Set-Linearizability. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, page 396, 1994.

[30] Gary Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[31] C.P. Rourke and B.J. Sanderson. *Introduction to Piecewise-Linear Topology.* Springer-Verlag, New York, 1982.

[32] Michael Saks and Fotios Zaharoglou. Wait-Free k-Set Agreement Is Impossible: The Topology of Public Knowledge. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 101–110, 1993.

[33] Ambuj Singh, James Anderson, and Mohamed Gouda. The Elusive Atomic Register. In *Proceedings of the 6th ACM Symposium on the Theory of Computing*, pages 206–221, 1986.