Distributed Computing: A Glimmer of a Theory

Eli Gafni
Computer Science Department
U.C.L.A.

1 Introduction

1.1 What is the area about?

Distributed Computing Theory develops computability and complexity theories for models whose computation involves many processors interacting in certain limited unpredictable manner through some communication objects, where a **processor** is shorthand for a sequential piece of code that includes instructions, some of which involve access to the communication objects.

One of the defining characteristics of the models tackled by distributed computing is a **nondeterminism** due to run-time events whose possibility of occurrence or the order in which they might occur is unpredictable. Moreover, distributed computing takes as its primary domain models whose nondeterminism gives their processors differing views of the world. Thus, for example, distributed computing will examine a model in which the failure of a processor is noticeable to some of its processors but not to others, but it will not examine a model in which the outcome of a flip of a coin (i.e., a randomized outcome) is made available to all processors simultaneously. In the parlance of knowledge [HM90], distributed computing studies systems in which the current state of a processor is not common knowledge, even when the input is known to all of the processors.

Research in distributed computing has focused on a number of models distinguished by the different communication objects and different timing constraints. Among those investigated most extensively are the asynchronous message-passing model, the asynchronous shared-memory model, asynchronous models in which at most t processors may fail-stop, and synchronous models with Byzantine, omission, or fail-stop failures.

1.2 What is this chapter about?

Here I argue that in the chaos that has characterized the field of distributed computing theory, we have begun to see signs of an underlying order. In the 1980s, as Lynch and Lamport [LL90] have observed, "the field seemed to consist of a collection of largely unrelated results about individual models." They were right at the time.

Researchers concentrated on obtaining efficient solutions to problems, and although this was and is an important and productive line of research, most of the ideas leading to efficiency seem ad hoc and are closely tied to a specific model and, at times, to a specific problem instance. Few of the ideas leading to efficient solutions supply a methodology that can be applied in other contexts.

Furthermore, the arguments that have led to the most fundamental impossibility results of the field seem to share no common thread. One fundamental negative result, FLP impossibility [FLP85], states that no agreement can be reached in an asynchronous system in which even a single processor may fail-stop unannounced. The original arguments that led to this result were procedural and very much tied to the properties of the operations specific to the model. Concurrently, the impossibility of reaching Byzantine agreement in time no more than the number of faults was proven [FL82] through the use of a **chain** of compatible views that lead from a run in which one decision has to be taken to a run in which another decision has to be taken. Consequently, the argument went, there must be a link in the chain at which incompatible decisions are taken by different processors in the same run. At about the same time, the procedural inductive proof of the nonexistence of a solution to the so-called two-armies problem was replaced by a slick proof using a newly developed knowledge theory [HM90, Gra78]. Although the connection between the latter two impossibility results, one in the context of synchronous systems and the other in the context of asynchronous systems, was at the time vaguely understood, the connection between them and FLP impossibly seemed nonexistent.

In recent years tools have been developed, and research has been fruitful in, making connections between results and models in a way that parallels what has been done in Complexity Theory. Where complexity theory is not much help in supplying a receipt to determine whether a specific problem at hand is $\Theta(n^2)$ or $\Theta(n^3)$ other then referral to some generic problems that have been thoroughly investigated, distributed computing can similarly provide only hints in the form of pointers to similar problems, in determining time and message complexity of a specific problem. Where complexity theory has tool and techniques to categorize problems in a "broad-brush," e.g. P vs. NP, distributed computing theory has over recent years built a formidable machinery to classify problems by the models in which they are solvable.

Problems have been identified that are analogous to **complete** problems. They characterize a model. The problem is solvable in the characterized model, and any model in which the problem is solvable, has a set of solvable problems that is a superset of the problems solvable in the characterized model. Collections of models, whose set of solvable problems is identical, have been identified, where in some models the level of the unpredictability of the possible order of events is lesser than in others. A problem is then checked for solvability in the model that exhibit the least unpredictability, if one exists. It provides, on the one hand, short uncluttered impossibility argument, or on the other hand, it provides a succinct protocol. Such a protocol may be viewed as written in a high level programming language, with a guaranteed automatic compilation of the protocol to a protocol in the target model.

Some starts that stalled, and some partial successes in the direction indicated above, occurred in the 80's. Knowledge Theory [HM90] was developed, showed some promise for certain problems, but eventually led, in my opinion, to no major break-through, in the understanding of distributed computing. A way of coaching 1-resilient models within graph theory, led to a complete characterization of these model [BMZ88], providing an automatic impossibility proof on one hand or a protocol on the other, to the question of a problem solvability in the model. This raised the hope that protocol derivation, or the realization of its nonexistence, can in general be automated. And finally, a helpful instance of the "high-level-programming,"mentioned above, was discovered with the realization of the equivalence between message passing systems and shared memory systems on the one hand, and between shared memory and atomic-snapshot shared-memory on the other. This instance of model equivalence allowed the investigation of the power of message-passing systems, within the much less detailed model of atomic-snapshot shared-memory.

The break-through was enabled when researchers started to zoom-in on complete problems. Herlihy [Her91] showed that the number processor that can achieve consensus characterizes and differentiates certain models. While consensus takes n input values and allows a single output, Chaudhuri proposed to check for models that allow several but less than n outputs. In 1993 three teams were able to show that her problems are also complete for certain models. The techniques employed resulted in the revelation of a connection between distributed computing and topology, be it algebraic [HS93], combinatorial [AR96], or point-set topology [SZ93].

Extensive research effort followed. If by "theory" one means a body of tightly related results, building one upon the other, and readily providing explanations to many observable phenomena, then, as I will attempt to show in this chapter, a theory of distributed computing has emerged.

The survey in this chapter has a narrow focus. The advances outlined above are presented. The utility of the "modern tools" that result in are demonstrated in deriving past results. The next section, on models, discusses distributed systems in general and defines the various notions of emulation and tasks which are central to this chapter. We then embark upon a sequence of emulations. We start with the asynchronous shared-memory model for two and three processors, characterizing them for fail-stop and Byzantine failures, and, via transformation, we apply all these results to the message-passing model. Then we investigate the shared-memory model in the synchronous domain, and we show how an impossibility result in the asynchronous model translates into lower bound on complexity in the synchronous model. The next step is to link the two models through the notion of failure detectors, and the new notion of an iterated model, and to explicate the utility of characterizing a synchronous system as an asynchronous one with a (weak) failure detector [CHT96]. The concluding section raises some of the major questions that, in my opinion, are facing this emerging theory of distributed computing. Obviously, the chapter leaves un-addressed many of the very large number of techniques developed in distributed computing; in most cases, how they may be incorporated into the framework reported here is not yet clear. The reader is encouraged to look up this wealth of techniques in the excellent text-book by Lynch [Lyn96], and two the surveys by Lamport and Lynch [LL90], and Attiya [Att95].

2 Models

An asynchronous distributed computation model is the set of all sequential interleaving of communication actions performed by sequential processes or processors on shared communication objects. A communication action may be thought of as the invocation of a remote procedure call by a processor at an object. An object may be thought of as a processor that executes the remote procedure call, changes its state, and responds to the invoking processor by returning a value. The returned value causes the invoking processor to change its state, which in turn determines the next parameter for the next access to a communication object. In this chapter we assume that processors never halt, and therefore after each return of an invocation, they enter a state in which a new invocation is enabled. In the network model, processors access unidirectional point-to-point communication channels. A single communication object is associated with two processors, called sender and receiver, respectively. The sender can invoke an action **send**(m) on the object. The effect of the action is to place a message m in the buffer of the object. After placing the message in its buffer the object responds by returning an *ok* to the sender. The receiver invokes an action **receive** which moves a message from the buffer of the object to the receiver, if such a message exists, or the object responds by notifying exception otherwise. In the shared memory model, communication objects are read/write registers on which the action of read and write can be invoked.

In this chapter we assume that communication objects do not fail. Yet, in light of the view of a communication object as a "restricted" processor, it is not surprising that when communication failures are taken into account [AGMT95], they give rise to results reminiscent of processor failures.

Given a **protocol** - the instantiation of processors with codes - and the initial conditions of processors and objects, we define a space R of **runs** to be a subset of the infinite sequences of processors names. Since we assume that a processor has a single enabled invocation at a time, such a sequence when interpreted as the order in which enabled invocations were executed completely determines the evolution of the computation. Before the system starts all runs in which the processor has its current input are possible. As the system evolves, the local state of the processor excludes some runs. Thus with a local state of a processor we associate a **view** - the set of all runs in R which are not excluded by the local state. By making processors maintain their history in local memory we may assume that consecutive views of a processor are monotonically non-decreasing. Thus, with each run $r \in R$ with a protocol p

we can associate a limit view $lim(V_i(r,p))$ of processor P_i. A protocol f is **full-information** if for all i, r and p we have $lim(V_i(r,f)) \subseteq lim(V_i(r,p))$. Intuitively, a full-information protocol does not economize on the size of its local state, or the size of the parameter to its object invocation. Models which are **oblivious**, that is, the sequence of communication objects a processor will access is the same for all protocols, possess a full-information protocol. All the models in this chapter do. In the rest of this chapter, a protocol stands for the full-information one, and correspondingly a model is associated with a single protocol - its full-information protocol. One can define the notion of full-information protocol with respect to a specific protocol in a non-oblivious model, but we will not need this notion here. A sequence of runs $r_1, r_2, \ldots$ converges to a run r, if r_k and r share a longer and longer prefix as k increases.

It can be observed from the definition of a view, that two views of the same processor, are either disjoint, or related by containment. Given an intermediate view $V_i(r)$ of a processor P_i in run r, we say that a processor **outputs** its view in r if for all P_j which have infinitely many distinct views in r, $lim(V_j(r)) \subseteq V_i(r)$. Processor P_i is **faulty** in r if it outputs finitely many views. Processor P_i is **participating** in r if it outputs any nontrivial view. Otherwise it is **sleeping** in r. A model A with n processors with communication object O_A **wait-free emulates** a model B with with n processors and communication objects O_B if there is a map m from runs R_A in A to runs R_B in B such that:

1. The sets of sleeping processors and faulty processors in r and $m(r)$ are identical.
2. The map m is continuous with respect to prefixes. That is if $r_1, r_2, \ldots$ in A converges to r, then $m(r_1), m(r_2), \ldots$ in B converges to $m(r)$. This captures the idea that the map does not predict the future.
3. The map m does not utilize detailed information about the past of a run, if this detailed information is not available through processors' views. Formally, for all P_j non-faulty and for all r in A, $m(lim(V_j(r))) \subseteq lim(V_j(m(r)))$

We say that A **non-blocking** emulates B if we relax the first condition by allowing the mapping m to fail any non-faulty processor as long as an infinite sequence is mapped into an infinite sequence. Two models are wait-free (non-blocking) equivalent if they wait-free (non-blocking) emulate each other.

A specification of a problem Π on n processors, is a relation from runs to sets of "output-sequences." Each output in the sequence is associate with a unique processor. A model with n processors wait-free solves Π if there exists a map from views to outputs, such the map of the projection of a run on views that are output in the run, is an output-sequence that relates to the run. A problem Π on n processors is non-blocking solvable in a model, if the relaxed problem $\bar{\Pi}$ is wait-free solvable, where $\bar{\Pi}$ takes each element in Π and closes the output-sequence set with respect to removal of infinite suffixes of processors' outputs (as long as the sequence remains infinite).

A **task** is a relaxation of a problem Π such that only bounded prefixes of output-sequences matter. That is to say that past some number of outputs any output is acceptable. Since the notion of participating set is invariant over models, the runs that are distinguished by different output requirements in a task, are those that differ in their participating set. Thus in this chapter we employ the notion of task in this restricted sense. In the **consensus** task a processor first outputs its private value, which is either 0 or 1, and then outputs a consensus value. Consensus values agree, and match the input of at least one of the participating processors. In the **election** task a processor outputs its ID and then outputs an election value which is an ID of a participating processor. All election values agree. A run with a single participating processor is a **solo-execution** of that processor.

A model is **t-resilient** if we require that it solves a problem only over runs in which at most t processors are faulty.

A **synchronous** model is one which progresses in rounds. In each round all the communication actions enabled by the beginning of the round are executed by the end of the round.

3 Asynchronous Models

3.1 Two-processor shared-memory model

Consider a two-processor single-writer/multi-reader (SWMR) shared-memory system. In such a system, there are two processors P_1 and P_0, and two shared-memory cells C_1 and C_0. Processor P_i writes exclusively to C_i, but it can read the other cell. Both shared-memory cells are initialized to $\perp$. W.l.o.g. computation proceeds with each processor alternately writing to its cell and reading the cell of the other processor.

Can this two-processor system 1-resiliently (wait-free in this case, since for $n = 2$, $n - 1 = 1$) elect one of the processors as the leader? No one-step full-information protocol, and consequently no one-step protocol at all, for solving this problem exists. Consider the state of processor P_1 after writing and reading. It could have read what processor P_0 wrote (denoted by $P_1 : w_0$), or it could have missed what processor P_0 wrote (denoted by $P_1 : \perp$). Thus, we have four possible views, two for each processor, after one step.

In the graph whose nodes are these views, two views are connected by an undirected edge if there is an execution that gives rise to the two views. The resulting graph appears in figure 1. Since a processor has a single view in an execution, edges connect nodes labeled by distinct processor IDs. The two nodes of distinct IDs which do not share an edge are $P_1 : \perp$ and $P_0 : \perp$. This follows from the fact that in shared memory in which processors first write and then read, the processor that writes second must read the value of the processor that writes first.

The edge $\{P_1 : \perp, P_0 : w_1\}$ corresponds to the execution: P_1 writes, P_1 reads, P_0 writes, P_0 reads. If we could map the view of a processor after one step into an output, then processor P_1 in this edge is bound to elect P_1, since the possibility of a solo execution by the processor has not yet been eliminated. Similarly, in the edge

Figure 1: One-step view graph.

$\{P_0 : \perp, P_1 : w_0\}$, processor P_0 is bound to elect P_0. Thus, no matter what processor is elected by P_1 and P_0 in the views $P_1 : w_0$ and $P_0 : w_1$, respectively, we are bound to create an edge where at one end P_1 is elected and at the other end P_0 is elected. Because there is an execution in which both processors are elected, we must conclude that there is no one-step 1-resilient protocol for election with two processors.

To confirm that no k-step full-information protocol exists, we could draw the graph of the views after k steps, and observe that the graph contains a path connecting the views $\{P_1 : \perp\}$ and $\{P_0 : \perp\}$.

It is not easy to see that indeed the observation above holds. Given that our goal is an argument that will generalize to more than two processors, we have to be able to get a handle on the general explicit structure of the shared-memory model for any number of processors. This has been an elusive challenge. Instead, we turn to iterated shared memory, a model in which the structure of the graph of a k-step two-processor full-information protocol is easily verified to be a path. We then argue that for two processors, the shared-memory model and the iterated shared-memory model are nonblocking equivalent. We then show that this line of argumentation generalizes to any number of processors.

3.2 Two-processor iterated shared-memory model

For any model M in which the notion of one-shot use of the model exists, one can define the iterated counterpart $\bar{M}$ of M. In $\bar{M}$, the processors go through a sequence of stages of one-shot use of M in which the output of the $(k-1)$th stage is in turn the input to the kth stage.

To iterate the two-processor SWMR shared-memory model, we take two sequences of cells: $C_{1,1}, C_{1,2}, C_{1,3}, \ldots$, and $C_{0,1}, C_{0,2}, C_{0,3}, \ldots$. Processor P_1 writes its input to $C_{1,1}$ and then reads $C_{0,1}$. Inductively, P_1 then takes its view after reading $C_{0,(k-1)}$, writes this view into $C_{1,k}$, reads $C_{0,k}$, and so on.

The iterated model is related to the notion of a **communication-closed layer** [EF82]. This accounts for why algorithms in the iterated model are easy both to understand and to prove correct: one may imagine that there is a barrier synchronization after each stage such that no processor proceeded to the current stage until all processors had executed (asynchronously) the previous stage.

In an execution, if the view of P_1 after reading $C_{0,(k-1)}$ is X and the corresponding view for P_0 is Y, then the graph of the views after the kth stage, given the view after the $k-1$'st stage, appears in figure 2. This graph is the same as the graph for the one-shot SWMR shared-memory model when X is the input to P_1 (and stands therefore for w_1) and Y is the input to P_0 (and stands therefore for w_0). To get the graph of all the possible views after the kth stage, we inductively take the graph after the $(k-1)$th stage, replace each edge with a path of three edges, and label the nodes appropriately. Thus, after k stages, we get a path of 3^k edges. At one end of this path is a view that is bound to output P_1, at the other a view bound to be output P_0, which leads to the conclusion that there is no k-stage protocol in the model for any k.

Figure 2: One-step view graph after the kth stage with input X,Y.

It is easy to see that the shared-memory model non-blocking implements its iterated version by dividing cell C_i into a sequence of on-demand virtual cells $C_{i,1}, C_{i,2}, C_{i,3}, \ldots$. Processors then "pretend" to read only the appropriate cell. To the see that a non-blocking emulation in the reverse direction is also possible we consider processors to $WriteRead$ sequence numbers. Processor P_1 keeps an estimate v_{p_0} of the last sequence number P_0 wrote. To $WriteRead_1(v)$, processor P_1 writes the pair v_{p_0}, v into the next cell in the sequence. It then read the other. If it contains $\perp$, or it contains the same pair it has written, then the operation terminates, and P_1 returns the pair it wrote to its cell. Otherwise, it updates v_{p_0} to the maximum between the value it held and the value it read, and continues [BG97].

3.3 Characterization of solvability for two processors

What tasks can two processors in shared memory solve 1-resiliently? They can solve, for instance, the following task. Processor P_1 in a solo execution outputs 1, and processor P_0 in a solo execution outputs 10. In every case, the two processors must output values between 1 and 10 whose absolute difference is exactly 1. This task can be solved easily, since in the iterated shared-memory model, after three stages the path contains 10 nodes and these nodes can be associated one-to-one with the integers 1 through 10. The task may be represented using domino pieces. There is an infinite number of pieces, each labeled P_1, x on one side and P_0, y on the other side, where all x and y are real numbers and $|x-y| = 1,\ 1 \le x \le 10,\ 1 \le y \le 10$. The task is solvable iff one can create a domino path with the pieces such that one side of the path is labeled $P_1, 1$ and the other side $P_0, 10$. It is easy to see that if processor P_0 had to output 11 (rather than 10) in a solo execution, the problem would not be solvable.

A generalized version is solvable if processors in solo executions output their integer inputs (which come, for the moment, from a bounded domain of integers) and the tuples of inputs are such that one input value is odd and the other is even. To see that the input (1,8) is solvable, take the output from the third stage of the iterated shared-memory model and fold three consecutive view edges on a single output edge. In algebraic and combinatorial topology, an edge is called a **1-simplex**, a node is called a **0-simplex**, an edge that is subdivided into a path is called a **1-dimensional subdivided simplex**, and a graph is called a **complex**. Thus, for a problem to be solvable 1-resiliently by two processors, the output complex must contain a subdivided simplex in which the labels on the two boundary nodes are the processors with their corresponding solo-execution outputs and the ID labels on the path alternate (colored subdivided simplex).

What if we want to solve the infinite version of the task where the possible integer inputs are not bounded? In this case, the difficulty is that we cannot place an a-priori upper bound on the number of iterated steps we need to take in the iterated model. One solution is to map the infinite line to a semi-circle, then do the appropriate convergence on the semi-circle, and map back. Another solution, denoted by Π, proceeds as follows. Processor P_1 with input k_1 takes k_1 steps if it reads $\perp$ continuously. Otherwise, after P_1 reads a value for P_0, it stops once it reads $\perp$ or reads that P_0 has read a value from P_1. Clearly, if the input values are k_1 and k_2, then the view complex is a path of length $k_1 + k_2$ that can be folded into the interval $[k_1, k_2]$ with enough views to cover all the integers (see [ALS94]).

In the case of shared memory (not iterated), if a processor halts once it takes some number of steps or once it observes the other processor take one step, we say that the processor halts within a unit of time. Consider the full-information version of Π (a protocol in the iterated model). An execution of the full-information protocol can be interpreted as an execution of a nonblocking emulation of the atomic-snapshot shared-memory model. If we take this view, then Π translates into a unit-time algorithm.

This conclusion, in fact, holds true for any task that is solvable 1-resiliently by two processors. It is easy to see that by defining the unit as any desirable $\epsilon > 0$, we can get two processors to output real numbers that are within an ϵ-ball (ϵ-agreement). To solve any problem, we fix an embedding of a path that may account for the solvability of the task. Consequently, there exists an ϵ such that for any interval I of length ϵ, all the simplexes that overlap the interval have a common intersection. Processors then conduct ϵ-agreement on the path, and each processor adopts as an output the node of its label which is closest to its ϵ-agreement output.

This view of convergence does not generalize easily to more than two processors. Therefore, we propose another interpretation for the two-processor convergence process [BG97]. After an $\epsilon/2$-agreement as above, P_i observes the largest common intersection s_i of the simplexes overlapping the ϵ-length interval that is centered around P_i's ϵ-agreement value. It must be that $s_1 \cup s_2$ is a simplex and that $s_1 \cap s_2 \neq \emptyset$. P_i posts s_i in shared memory. It then takes the intersection of the s_j's it observes posted. If a node labeled by P_i's ID is in the intersection, P_i outputs that node. Otherwise, P_i sees only one node, v, in the intersection, and v has a label different from P_i's ID. In this case, P_i outputs one of the nodes labeled by its own ID which appear in a simplex along with v (these nodes are said to be "in the **link** of v").

Thus, since solvability amounts to ϵ-agreement, and ϵ-agreement can be achieved 1-resiliently within a unit of time, we conclude that any task 1-resiliently solvable by two processors can be solved within a unit of time.

3.4 Three-processor 2-resilient model

We consider now the three-processor 2-resilient SWMR shared-memory model. W.l.o.g. processors alternate between writing and reading the other two cells one by one. Obviously, we cannot elect a leader (since two processors cannot), but perhaps we can solve the $(3, 2)$ set-consensus problem in which processors elect at most two leaders (i.e., each processor outputs an ID of a participating processor, and the union of the outputs is of cardinality at most 2).

The structure of the full-information protocol of the one-shot shared-memory model for three processors is not as easy to identify as that for two processors. Two-processor executions have many hidden properties that are lost when we have three processors. For example, in a two-processor execution, when a processor reads the value of the other processor's cell, then in conjunction with the value it has last written to its own cell, the processor has an instantaneous view of how the memory looks, as if it read both cell in a single atomic operation. Such an instantaneous copy is called an **atomic snapshot** (or, for short, a snapshot) [AAD+90]. In a three-processor system, this property is lost; we cannot interpret a read operation as returning a snapshot.

To get the effect of processor P_1 reading cells C_2 and C_0 instantaneously, we have P_1 read C_2 and C_0 repeatedly until the values the processor reads do not change over two consecutive repetitions. If all values written are distinct, then the values the processor reads reside simultaneously in the memory in an instant that is after the first of the two consecutive repetitions and before the second of the repetitions. Thus, P_1 may safely return these values.

Clearly, three processors can 2-resiliently nonblocking implement one-shot snapshot memory–a memory in which a processor writes its cell and then obtains a vector of values for all cells, and this vector is a snapshot. Yet, a one-shot snapshot may give processors the following views: $P_1 : w_1, w_2, w_0$, $P_2 : w_1, w_2, w_0$, $P_0 : \perp, w_2, w_0$. This is the result of the execution: P_2 writes, P_0 writes, P_0 takes a snapshot, P_1 writes, P_1 and P_2 take a snapshot. Can we require that the set of processors S_i that return at most i values return snapshots only of values written by processors from S_i? In the example above, we have $S_2 = \{P_0\}$, but P_0 returns a value from P_2 which is not in S_2. A snapshot whose values are restricted in this way is called an **immediate snapshot** [SZ93, BG93]. A recursive distributed procedure will return immediate snapshots (program for P_i):

1. Procedure-Immediate-Snapshot $ISN(P_i, k)$.

2. Write input to cell $C_{k,i}$.

3. For $j = 1$ to n, read cell $C_{k,j}$.

4. If the number of values ($\neq \perp$) is k, then return everything read; else, call immediate snapshot $ISN(P_i, k-1)$.

We assume that all cells are initialized to $\perp$ and that in a system with n processors, P_i starts by calling immediate snapshot $ISN(P_i, n)$.

It is not difficult to prove that the view complex of a one-shot immediate snapshot is a subdivided simplex. Simple extension of the algorithm outlined for two processors shows that, in general, the iterated immediate-snapshot model nonblocking implements the shared-memory atomic-snapshot model [BG97]. The view complex for three processors is shown in figure 3.

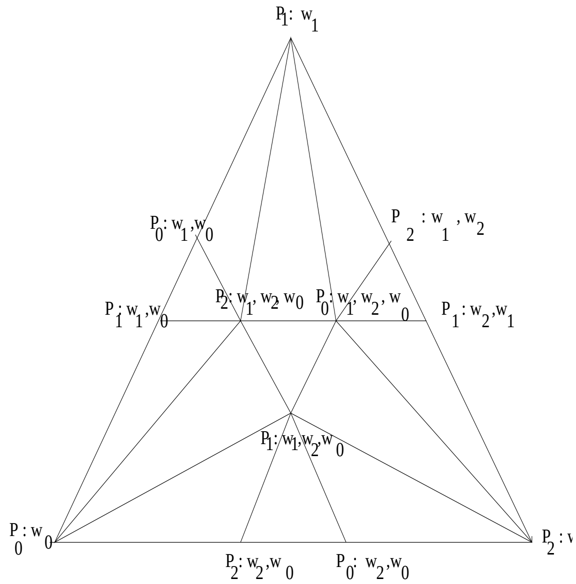

Figure 3: Three-processor one-shot immediate-snapshot view complex.

We now argue by way of example that after the first stage in the iterated immediate-snapshot model, processors cannot elect two leaders. By definition, the view $P_i : w_i$ has to be mapped to P_i. The views $P_i : w_i, w_j$ are mapped to P_i or P_j. The rest are mapped to any processor ID. Such a mapping of views to processor IDs constitutes a Sperner coloring of the subdivided simplex [Spa66]. The Sperner lemma then says that there must be a triangle colored by the three colors. Since a triangle is at least one execution, we have proven that no election of two leaders is possible. The argument we made about the coloring of a path by two processors' IDs is just the 1-dimensional instance of the Sperner lemma. By the recursive properties of iterated models, we conclude that the structure of a k-step three-processor 2-resilient iterated immediate snapshot is the structure of a subdivided triangle.

What non-trivial tasks can we solve? For one, the task of producing a one-shot immediate snapshot is far from trivial. In general, we can solve anything that the view complex of a sufficiently large number of iterations of one-shot immediate snapshots can map to, color and boundary preserving, simplicially. This includes any subdivided triangle A [HS94].

We show algorithmically how any three processors converge to a triangle on a colored subdivided triangle A. Embedding a reasonably large-enough complex of the iterated immediate snapshots embedded over A yields 2-dimensional ϵ-agreement over A, since triangles of the view complex can be inscribed in smaller and smaller circles as a function of the number of iterations we take. Thus, as we did for two processors, we may argue now that an $\epsilon > 0$ exists such that for any ϵ-ball in A, all simplexes that overlap the ball have a common intersection. P_i then conducts $\epsilon/2$-agreement and posts the largest such intersection s_i of the simplexes overlapping the $\epsilon/2$-radius ball whose center is P_i's $\epsilon/2$-agreement value. P_i takes the intersection of the s_is posted and, if P_i's color is present, adopts the value of that A node. Otherwise, P_i removes a node of P_i's color, if one exists, from the union of the simplexes P_i observed to get a simplex x_i and then starts a new ϵ-agreement from a node of P_i's color in the link of x_i.

As we argued for two processors, when there are three processors, at least one processor will terminate after the first ϵ-agreement. If two processors show up for the second agreement, they have identified the processor that will not proceed; the link is at worst a closed path. The convergence of the remaining two processors is interpreted to take place on one side of the closed path rather than on the other, and the decision about on which side convergence takes place is made according to a predetermined rule that is a function of the starting nodes. The convergence of two processors on a path was outlined in the previous section.

To see that ϵ-agreement for three processors is solvable 2-resiliently on a triangle within a unit of time, we notice that if we again take an iterated algorithm with the stopping rule that processors halt once they reach some bound or once they learn that they read from each other, as in the case of two processors, it can easily be argued inductively that we get a subdivided simplex "growing from the center." As we increase the bound on the solo and pairs executions, we "add a layer" around the previous subdivided simplex. Thus, we have a mesh that becomes as fine as we need and results in ϵ-agreement. Our stopping rule, when converted via nonblocking emulation of shared memory by iterated shared memory, results in a unit-time algorithm. Since we have seen that solving a task amounts to ϵ-agreement on the convex hull, we conclude that in the snapshot model, an algorithm exists for any wait-free solvable task such that at least one processor can obtain an output within a unit of time (see [ALS94]).

Now, if we view three-processor tasks as a collection of triangular domino pieces that can be reshaped into triangles of any size we want, we see that again the question of solvability amounts to a tiling problem. Where in the two-processor case we had two distinct domino ends that we had to tile in between, we now have three distinct domino corners, corresponding to solo executions, and three distinct domino sides, corresponding to executions in which only two processors participated. To solve a three-processor task, we have to pack the domino pieces together to form a subdivided triangle that complies with the boundary conditions on the sides. Unfortunately, this 2-dimensional tiling problem is undecidable in general [GK95].

3.5 Three-processor 1-resilient model

What kind of three-processor tasks are 1-resiliently solvable [BMZ88]? It stands to reason that such systems are more powerful than two-processor 1-resiliency. Perhaps the two non-faulty processors can gang up on the faulty one and decide.

In light of our past reasoning, the natural way to answer this question is to ask what the analogue to the one-shot immediate snapshot is in this situation. A little thought shows that the analogue comprises two stages of the wait-free one-shot immediate snapshot where the nodes that correspond to solo executions have been removed. (We need to two stages because one stage with solo executions removed is a task that is not solvable 1-resiliently by three processors.) This structure can be solved as a task by the three-processor 1-resilient model, and the iteration of this structure is a model that can nonblocking implement a shared-memory model of the three-processor 1-resilient model.

An inductive argument shows that this structure is connected, and consequently consensus is impossible. Furthermore, the link of any node is connected. If we have a task satisfying these conditions, the convergence argument of the previous section shows how to solve the task. We start at possibly only two nodes because one processor may wait on one of the other two. By simple connectivity, we can converge so that at least one processor terminates. The other two can converge on the link of the terminating processor since this link is connected.

Thus a task is solvable 1-resiliently in a system with 3 processors if it contains paths with with connected links, connecting solo executions of 2 processors. Checking whether this holds amounts to a reachability problem and therefore decidable.

3.6 Models with Byzantine failure

What if a processor not only may fail to take further steps but also may write anything to its cell in the asynchronous SWMR memory? Such a processor is said to fail in a Byzantine fashion. Byzantine failures have been dealt with in the asynchronous model within the message passing system [Bra87]. Here we define it for the shared-memory environment and show that the essential difficulty is actually in transforming a message-passing system to a shared-memory one with write-once cells.

Why a Byzantine failure is not more harmful than a regular fail-stop failure when in a write-once cells shared-memory environment? If we assume that cell C_i is further subdivided into sub-cells $C_{i1}, C_{i2}, \ldots$ and that these sub-cells are write-once only, we can nonblocking emulate iterated snapshot shared memory in which a processor writes and then reads all cells in a snapshot. All processors now have to comply with writing in the form of snapshots–namely, they must write a set of values from the previous stage, otherwise what they write will be trivially discarded. Yet, faulty processors may post snapshots that are inconsistent with the snapshots posted by other, non-faulty processors. We observe that we can resolve inconsistent snapshots by letting their owners to revise these snapshots to snapshots that are the union of the inconsistent ones. This allows processors to affirm some snapshots and not affirm others. A processor that observes another snapshot that is inconsistent with its own snapshot will suggests the union snapshot if its snapshot has not been affirmed yet. This processor waits with a snapshot consistent with the others until one of its snapshots has been affirmed. It then writes one of its affirmed snapshots as a final one. Thus, all other processors may check that a processor's final snapshot have been affirmed.

A processor affirms a snapshot by observing it to be consistent with the rest and writing an affirmation for it. If we wait for at least $n/2+1$ affirmations (discard a processor which affirms two inconsistent snapshots as faulty), then it can be seen that no inconsistent snapshots will be affirmed. In other words, we have transformed Byzantine failure to fail-stop at the cost of nonblocking emulating the original algorithm if it was not written in the iterated style.

The next problem to be addressed is how to nonblocking emulate subdivision to a write-once sub-cell of a cell C_i when a processor may overwrite the values previously written to the cell. The following procedure nonblocking emulates a read of $C_{i,k}$:

1. If read $C_{i,k} \neq \perp$ or if read $f+1$ processors claiming a value $v \neq \perp$ for $C_{i,k}$, claim v for $C_{i,k}$.

2. If read $2f+1$ processors claiming v for $C_{i,k}$ or if read $f+1$ processors wrote Confirm(v) for $C_{i,k}$, write Confirm(v) for $C_{i,k}$.

3. If read $2f+1$ processors claiming Confirm(v) for $C_{i,k}$, accept v for $C_{i,k}$.

Clearly, once a processor accepts a value, all processors will accept that value eventually and so a value may not change.

Since the availability on write-once cells allows Byzantine agreement we conclude that like Byzantine agreement the implementation of write-once cells requires that less of a third of the processors fail. In hindsight, we recognize that Bracha discovered in 1987 that a shared-memory system with Byzantine failure and write-once cells can be nonblocking implemented on a message-passing system with the same failures as the shared-memory system provided that $3f < n$. It took another three years for researchers to independently realize the simpler transformation from message passing to shared memory for fail-stop faults [ABND95].

3.7 Message-passing model

Obviously, shared memory can nonblocking emulate message passing. The conceptual breakthrough made by Attiya, Bar-Noy, and Dolev (ABD) [ABND95] was to realize that for $2f < n$, message passing can f-resiliently wait-free emulate f-resilient shared memory, as follows. To emulate a write, a processor takes the latest value it observed, sends the value to all of the processors, and waits for acknowledgment from a majority. To read, a processor asks for the latest value for the cell from a majority of the processors. A processor keeps an estimate of the latest value for a cell; this estimate is the value with the highest sequence number that the processor has ever observed for the cell. Since two majorities intersect, the emulation works.

All results of shared memory now apply verbatim, by either the fail-stop or the Byzantine transformation, to message passing. To derive all the above directly in message passing is complex, because, in one way or the other, hidden ABD or Bracha transformations sit there and obscure the real issues. The ABD and Bracha transformations have clarified why shared memory is a much cleaner model than message passing to think and argue about.

3.8 From safe-bits to atomic snapshots

We now show that in retrospect the result of the research conducted at the second part of the 80's on the power of shared-memory made out of safe-bits, is not surprising.

We started this section with a SWMR shared-memory model. Can such a model be nonblocking implemented from the most basic primitives? Obviously, this is a question about the power of models, specifically about relaxing the atomicity of read-write shared-memory registers. This problem was raised by Lamport [Lam86] in 1986, and quite a few researchers have addressed it since then. Here we show that the nonblocking version of the problem can be answered trivially.

The primitive Lamport considers is a single-writer/single-reader safe-bit. A **safe-bit** is an object to which the writer writes a bit and from which the reader can read provided that the interval of operation execution of reading does not overlap with writing.

We now show how a stage of the iterated immediate-snapshot model can be nonblocking implemented by safe-bits. We assume an unlimited number of safe-bits, all initialized to 0, per pair of processors. To write a value, a processor writes it in unary, starting at a location known to the reader. To read, a processor counts the number of 1s it encounters until it meets a 0.

We observe that the recursive program for one-shot immediate snapshot itself consists of stages. At most k processors arrive at the immediate-snapshot stage called k (notice that these stages run from n down to 1). All we need is for at least one out of the k processors to remain trapped at stage k. The principle used to achieve this is the flag principle, namely, if k processors raise a flag and then count the number of flags raised, at least one processor will see k flags. For this principle to hold, we do not need atomic variables. Processors first write to all the other processors that they are at stage k and only then start the process of reading, so the last processor to finish writing will always read k flags.

Given that a processor writes its input to all other processors before it starts the immediate-snapshot stage, when one processor encounters another in the immediate snapshot and needs to know the other processor's input, the input is already written.

This shows that any task solvable by the most powerful read-write objects can be solved by single-writer/single-reader safe-bits. Can one nonblocking emulate by safe-bits any object that can be read-write emulated? The ingenious transformations cited in the introduction have made it clear that the answer to this question is "yes." Nonetheless, the field still awaits a theory that will allow us to deal with wait-free emulation in the same vein as we have dealt with nonblocking emulation here.

3.9 Geometric Protocols

What if draw a 1-dimensional immediate snapshot complex on a plan and repeat subdividing it forever? We can then argue that each point in our drawing corresponds to an infinite run with respect to views that are output-ed. Obviously such a construction can be done for any dimension. We obtain an embedding of the space of runs in the Euclidean Unit Simplex. Such an embedding was the quest that eluded the authors of [SZ93]. An embedding gives rise to "geometric-protocols."

Consider the problem of 2-processors election when we are given that the infinite symmetric run will not happen. One may work out an explicit protocol (which is not trivial). Geometrically, we take an embedding, and a processor waits until its view is completely on one side of the symmetric run. It then decides according to the solo execution that that side contains.

4 Synchronous Systems

4.1 Shared-memory model

In retrospect, given the iterated-snapshots model in which conceptually processors go in lock-step, and its equivalence to the asynchronous shared memory, it is hard to understand the dichotomy between synchrony and asynchrony. In one case, the synchronous, we consider processors that are not completely coordinated because of various type of failures. In the other case, the asynchronous, processors are not coordinated because of speed mismatch. Why is this distinction of such fundamental importance? In fact, we argue that it is not. To exemplify this we show how one can derive a result in one model from a result in the other.

We consider the SWMR shared-memory model where its computation evolves in rounds. In other words, all communication events in the different processors and communication objects proceed in lockstep. At the beginning of a round, processors write their cells and then read all cells in any order. Anything written in the round by a processor is read by all processors. This model may be viewed as a simple variant of the Parallel RAM (PRAM) model in which processors read all cells, rather than just one cell, in a single round.

With no failures, asynchronous systems can emulate synchronous ones. What is the essential difference between synchronous and asynchronous systems when failures are involved? A 1-resilient asynchronous system can "almost" emulate a round of the synchronous system. It falls short because one processor (say, if we are dealing with the synchronous shared-memory system below) misbehaves–but this processor does not really misbehave: what happens is that some of the other processors miss it because they read its value too early, and they cannot wait on it, since one processor is allowed to fail-stop. If we call this processor faulty, we have the situation where we have at most one fault in a round, but the fault will shift all over the place and any processor may look faulty sooner or later. In contrast, in a synchronous system, a faulty behavior is attributed to some processor, and we always assume that the number of faults is less than n, the number of processors in the system.

We now introduce the possibility of faults into the synchronous system, and we will examine three types of faults. The first type is the analogue of fail-stop: a processor dies in the middle of writing, and the last value it wrote may have been read by some processors and not read by the others.

The algorithm to achieve consensus is quite easy. Each processor writes an agreement proposal at each round and then, at the next round, proposes the plurality of values it has read (with a tie-breaking rule common to all processors). At the first round, processors propose their initial value. After $t+1$ rounds, where t is an upper bound on the number of faults, a processor decides on the value it would have proposed at round $t+2$. The algorithm works because, by the pigeon principle, there is a **clean round**, namely, a round at which no processor dies. At the clean round, all processors take the plurality of common information, which results in a unanimous proposal at the next round. This proposal will be sustained to the end. Various techniques exist that can lead to early termination in executions with fewer faults than expected.

The next type of fault is omission [NT90]: a processor may be resurrected from fail-stop and continue to behave correctly, only to later die again to be perhaps resurrected again, and so on.

We reduce omission failure to fail-stop. A processor P_i that fails to read a value of P_j at round k writes at round $k+1$ that P_j is faulty. A processor that reads about another processor being faulty stops reading the faulty processor's values. We see that if a correct processor P_i fails to read a value from P_j, all processors will stop reading cell C_j at the next round, which is exactly the effective behavior of a fail-stop processor.

The last type of fault is Byzantine. We assume $n = 3f+1$. Here, in addition to omitting a value, a processor might not obey the protocol and instead write anything (but not different values ($\neq \perp$) to different processors). We want to achieve the effect of true shared memory, in which if a correct processor reads a value, then all other processors can read the same value after the correct processor has done so.

We encountered the same difficulty in the asynchronous case, with the difference that in the asynchronous case things happen "eventually." If we adopt the same algorithm, whatever must happen eventually (asynchronous case) translates into happening in some finite number of rounds (synchronous case). Moreover, if something that is supposed to happen for a correct processor in the asynchronous case does not happen for a processor in the synchronous case within a prescribed number of rounds, the other processors infer that the processor involved is faulty and announce it as faulty. Thus, the asynchronous algorithm for reading a value which we gave in the previous section translates to (code for processor P_i):

1. Round 1: $v := read(cell)$.
2. Round 2: Write v, read values for v from all non-faulty processors.
3. Round 3: If $2f+1$ for value $v \neq \perp$ in Round 2, write $confirm(v)$; else, write $v := faulty$, read cells.
4. Round 4: If $2f+1$ for v in Round 3 or if $f+1$ for $confirm(v)$, write $confirm(v)$; else, $v := faulty$. If $2f+1$ $confirm(v)$ up to now, then $accept(v)$. If $v = faulty$ and $accept(v)$, consider the processor that wrote v faulty.

A little thought shows that the "eventual" of the asynchronous case translates into a single round in the synchronous case. If a value is accepted, it will be accepted by all correct processors in the next round. If a value is not accepted by the end of the third round, then all correct processors propose $v := faulty$. In any case, at the end of the fourth round, either a real value or $v = faulty$ or both will be accepted. After a processor P_i is accepted as faulty, at the next round (which may be at the next phase) all processors will accept it as faulty and will ignore it. Thus, ignoring for the moment the problem that a faulty processor may write incorrect values, we have achieved the effect of write-once shared memory with fail-stop.

To deal with the issue of a faulty processor writing values a correct processor would not write, we can check on previous writes and see whether the processor observes its protocol. Here we do not face the difficulty we faced in the asynchronous case: a processor cannot avoid reading a value of a correct processor, and a processor may or may not (either is possible) read a value from a processor that failed. Nevertheless, a processor's value may be inconsistent with the values of correct processors. We notice that correct processors are consistent among themselves. Processors then can draw a "conflict" graph and, by eliminating edges to remove conflicts, declare the corresponding processors faulty. Since an edge contains at least one faulty processor, we can afford the cost of failing a correct processor.

We now argue a lower bound of $f+1$ rounds for consensus for any of our three failure modes. It suffices to prove this bound for fail-stop failure, because fail-stop is a special case of omission failure and of Byzantine failure. Suppose an $(m < t+1)$-rounds consensus algorithm exists for the fail-stop type of failure. We emulate the synchronous system by a 1-resilient asynchronous system in iterated atomic-snapshot shared memory. At a round, there is a unique single processor whose value other processors may fail to read. We consider such a processor to be faulty. We have seen that, without any extra cost, if a correct processor fails to read a value of processor P_j, then all correct processors will consider P_j faulty in the next round, which amounts to fail-stop. Thus, the asynchronous system emulates m rounds of the synchronous system. At each simulated round, there is at most one fault, for a total of at most f faults. This means that the emulated algorithm should result in consensus in the 1-resilient asynchronous system, which is impossible.

We can apply the same logic for set consensus and show that with f faults and k-set consensus, we need at least $\lfloor f/k \rfloor + 1$ rounds. This involves simulating the algorithm in a k-resilient asynchronous atomic-snapshot shared-memory system in which k-set consensus is impossible. (The first algorithm in this section automatically solves k-set consensus in the prescribed number of rounds; see [CHLT93]).

Notice that the above impossibility result for the asynchronous model translated into a lower bound on round complexity in the synchronous model. We now present the failure-detector framework, a framework in which speed mismatch is "transformed" into failure, and thus unifying synchrony and asynchrony.

5 Failure Detectors: A Bridge Between Synchronous and Asynchronous Systems

In the preceding section, we have seen that a 1-resilient asynchronous system looks like a synchronous system in which at each round a single but arbitrary processor may fail. Thus, synchronous systems can achieve consensus because in this setting when one processor considers another processor faulty, it is indeed the case, and one processor is never faulty. In the asynchronous setting, the processor considered faulty in a round is not faulty, only slow; we make a mistake in declaring it faulty.

Chandra and Toueg [CT96] have investigated the power of systems via the properties of their of augmenting subsystem called **failure-detector** (FD) that issues faulty declarations. The most interesting FDs are those with properties called S and $\diamond S$.

FD S can be described as follows:

all processors that take finitely many steps in the underlying computation (faulty processors) will eventually be declared forever faulty by all the local FDs of processors that took infinitely many steps (correct processors), and

some correct processor is never declared faulty by the local FDs of correct processors.

In $\diamond S$, these properties hold eventually. Chandra, Hadzilacos, and Toueg [CHT96] then showed that of all the FDs that provide for consensus, $\diamond S$ is the weakest: any FD that provides for consensus can nonblocking emulate $\diamond S$.

Thus, in a sense, if we take a system with timing constraints and call it synchronous if the constraints provide for consensus, then we have an alternative way of checking on whether the constraints provide for nonblocking emulation of $\diamond S$.

In many systems, this alternative technique is natural. For instance, consider a system in which the pending operations stay pending for at most some bounded but otherwise unknown real time and the processors can delay themselves for times that grow longer and longer without bound (of course, a delay operation is not pending until the prescribed delay time has elapsed). It is easy to see that this system can nonblocking implement $\Diamond S$ and, as a result, achieve consensus. Thus, we have a variety of consensus algorithms from which to choose. In fact, we can transform any synchronous consensus algorithm for omission to an algorithm for $\Diamond S$. The transformation proceeds in two stages. We first transform a synchronous omission algorithm A to an algorithm B for consensus in S. We accomplish this by nonblocking emulating A round by round, where each processor waits on the others until either the expected value is observed or the processor waited on is declared faulty by the FD. Such failure can be considered omission failure. Since in S, one of the processors is never declared faulty, we get that the number of omission faults is less than n, and the emulation results in consensus.

To transform a consensus algorithm A in S to a consensus algorithm B for $\Diamond S$, we use a layer algorithm called Eventual. We assume A is safe in the sense that when running A in $\Diamond S$, the liveness conditions that result in consensus are weakened but the safety conditions that make processors in A over S agree or preserve validity (output has to be one of the inputs) are maintained.

Algorithm Eventual has the property that a processor's output consists of either committing to some input value or adopting one.

1. If all processors start with the same input, then all commit to that input.

2. If one processor commits to an input value, then all other processors commit to or adopt that value.

The consensus algorithm B for shared memory with $\Diamond S$ is to run alternately with A followed by Eventual. The output of A is the input to Eventual, and the output of Eventual is the input to A. When a processor commits to a value in Eventual, it outputs it as the output of B. Algorithm Eventual is simple. Processors repeatedly post their input values and take snapshots until the number of postings exceeds n. If a processor observes only a single value, it commits to that value; otherwise, it adopts a plurality value.

The notion of FDs is very appealing. It unifies synchronous and asynchronous systems. In the FD framework, all systems are of the same type but each system possesses FDs with distinctive properties. Research into the question of how this unified view can be exploited in distributed computing–for example, it might enable understanding of the "topology" of FDs—has hardly begun.

Another direction of possible research is to enrich the semantics of failure-detector. If we take the iterated snapshot system we may consider attaching a separate failure detector subsystem to each layer. A processor which is "late" arriving to a layer is declared faulty by that layer subsystem. We may now investigate the power of such system as a function of the properties of their failure detectors. We can for instance model a t-resilient system as a system in which at most t processors will be declared faulty at a layer. When one considers such a system, the dichotomy of systems between synchronous and asynchronous is completely blurred. The traditional failure detector of Chandra and Toueg cannot capture such a property.

6 Research Issues and Summary

In this chapter, we have considered a body of fairly recent research in distributed computing whose results are related and draw upon one another. We have argued that a unified view of these results derives from the synergy between the application of results from topology and the use of the interpretive power of distributed computing to derive transformations that make topology apply.

Many interesting and important questions, aside from matters of complexity, remain. The most practical of these questions concern computations that are amenable to algorithms whose complexity is a function of the concurrency rather than the size of the system. Such algorithms are usually referred to as **fast**. Examples of tasks that can be solved by application of fast algorithms are numerous, but a fundamental understanding of exactly what, why, and how computations are amenable to such algorithms is lacking. Extension of the theory presented in this chapter to non-terminating tasks and to long-lived objects is next on the list of questions.

7 Defining Terms

t-resilient system A system in which at most t processors are faulty.

0-simplex A singleton set. item[1-dimensional subdivided simplex] An embedding of 1-simplex that is partitioned into 1-simplexes (a path).

1-simplex A set consisting of 2 elements.

atomic snapshot An atomic read operation that returns the entire shard memory.

chain-of-runs A sequence of runs in which two consecutive runs are indistinguishable to a processor.

clean-round A round in which no new faulty behavior is exhibited.

communication-closed-layers A distributed program partitioned to layers that communicate remotely only among themselves and communicate locally unidirectionaly.

complete-problem A problem in a model that characterizes the set of all other problems in the model in the sense that they are reducible to it.

complex A set of simplexes that is closed under subset.

consensus-problem A decision task in which all processors agree on a single input value.

election A consensus over the IDs as inputs.

failure-detector An oracle that updates a processor on the operational status of the rest of the processors.

fast-solution A solution to a problem whose complexity depends on the number of participating processors rather than the size of the entire system.

faulty-processor A processor whose view is output finitely many times in a run.

full-information protocol A protocol which induces the finest partition of the set of runs, compared to any other protocol in the model.

immediate snapshots A restriction of the atomic snapshots that achieves a certain closure property that atomic snapshots do not have.

link (of a simplex in a complex) The set of all simplexes in a complex that are contained in a simplex with the given simplex.

non-blocking (emulation) A emulation that may increase the set of faulty processors.

oblivious (to a parameter) Not a function of that parameter.

outputs Map over views which are eventually known to all non-faulty processors.

participating (set of processors) Processors in a run which are not sleeping.

processor A sequential piece of code.

protocol A set of processors whose code refer to common communication objects.

run An infinite sequence of global "instantaneous-description" of a system, such that one element is the preceding one after the application of a pending operation.

safe-bit A single writer single reader register bit whose read is defined and returns the last value written only if does not overlap a write operation to the register.

sleeping (in a run) A processor whose state in a run does not change.

solo-execution A run in which only a single processor is not sleeping.

task A relation from inputs and set of participating processors to outputs.

view A set of runs compatible with a processor's local state.

wait-free (solution) A solution to a non-termination problem in which a processor that is not faulty outputs infinitely many output values.

References

[AAD$^+$90] 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.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[AGMT95] Y Afek, DS Greenberg, M Merritt, and G Taubenfeld. Computing with faulty shared objects. *Journal of the Association of the Computing Machinery*, 42:1231–1274, 1995.

[ALS94] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, July 1994.

[AR96] Attiya and Rajsbaum. The combinatorial structure of wait-free solvable tasks. In *WDAG: International Workshop on Distributed Algorithms, Springer-Verlag*, 1996.

[Att95] Hagit Attiya. Distributed computing theory. In Albert Y. Zomaya, editor, *Handbook of Parallel and Distributed Computing*. McGraw-Hill, New York, 1995.

[BG93] 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.

[BG97] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reason characterization of wait-free computations. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 189–198, 1997.

[BMZ88] 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.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[CHLT93] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. A tight lower bound for k-set agreement. In *34th Annual Symposium on Foundations of Computer Science*, pages 206–215, Palo Alto, California, 3–5 November 1993. IEEE.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[EF82] T. E. Elrad and N. Francez. Decomposition of distributed programs into communicationclosed layers. *Science of Computer Programming*, 2(3), 1982.

[FL82] M. J. Fischer and N. A. Lynch. A lower bound on the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[FLP85] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[GK95] Eli Gafni and Elias Koutsoupias. 3-processor tasks are undecidable. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, page 271. ACM, August 1995.

[Gra78] J. N. Gray. Notes on data base operating systems. In *Springer Verlag (Heidelberg, FRG and NewYork NY, USA) LNCS, 'Operating Systems, an Advanced Course', Bayer,Graham, Seegmuller(eds)*, volume 60. 1978.

[Her91] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Lanugages and Systems*, 11(1):124 – 149, jan 1991. supercedes 1988 PODC version.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[HR95] M. Herlihy and S. Rajsbaum. Algebraic topology and distributed computing — A primer. *Lecture Notes in Computer Science*, 1000:203–??, 1995.

[HS93] 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.

[HS94] 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.

[Lam86] L. Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1157–1199. The MIT Press, New York, N.Y., 1990.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CA, 1996.

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[Spa66] E. H. Spanier. *Algebraic Topology.* Springer-Verlag, New York, 1966.

[SZ93] Michael Saks and Fotios Zaharoglou. Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 101–110, 1993.

8 Further Information

Current research on the theoretical aspects of distributed computing is reported in the proceedings of the annual ACM Symposium on Principles of Distributed Computing (PODC) and the annual International Workshop on Distributed Algorithms on Graphs (WDAG). Relatively recent books and surveys are [Lyn96, Att95, HR95].

Acknowledgement: I am grateful to Hagit Attiya for detailed illuminating comments on an earlier version of this chapter.