This is a preprint version of a paper published in Distrib. Comput. (2003) 16:239–247,
Digital Object Identifier (DOI) 10.1007/s00446–003–0096–6

Appraising Two Decades of Distributed Computing Theory Research

Michael J. Fischer[1], **Michael Merritt**[2]

[1] Department of Computer Science, Yale University, New Haven, CT 06520–8285, USA, e-mail: fischer-michael@cs.yale.edu
[2] AT&T Labs—Research, 180 Park Ave., P.O. Box 971, Florham Park, NJ 07932–0971, USA, e-mail: mischu@research.att.com

May 14, 2003

This paper is based on an invited lecture presented by the first author at the International Symposium on Distributed Computing, Bratislava, Slovac Republic, September 27, 1999.

Summary. The field of distributed computing started around 1970 when people began to imagine a future world of multiple interconnected computers operating collectively. The theoretical challenge was to define what a computational problem would be in such a setting and to explore what could and could not be accomplished in a realistic setting in which the different computers fell under different administrative structures, operated at different speeds under the control of uncoordinated clocks, and sometimes failed in unpredictable ways. Meanwhile, the practical problem was to turn the vision into reality by building networks and networking equipment, communication protocols, and useful distributed applications. The theory of distributed computing became recognized as a distinct discipline with the holding of the first *ACM Principles of Distributed Computing* conference in 1982. This paper reviews some of the accomplishments of the theoretical community during the past two decades, notes an apparent disconnect between theoretical and practical concerns, and speculates on future synergy between the two.

1 Introduction

Theoretical computer science had its beginnings in mathematical logic through the attempt to formalize the notion of *effective procedure*. Turing machines [27], general recursive functions [16], and lambda calculus [5] were among the proposed models. They were eventually all shown to be equivalent in the sense of defining the same class of functions (cf. [6]), now known as the *Turing-computable* functions [24]. Church's thesis states that all reasonable models of computation compute only the Turing-computable functions [4].

The Turing machine model and the notion of simulation upon which the equivalence proofs were based was extremely useful in modeling early computers. Computers then typically consisted of a single processing element operating deterministically. They were generally run in batch mode, meaning that the inputs for each job were made available at the beginning of the job, the job was expected to run to completion, and the only thing that mattered about a completed job was the output that it produced during its run. Computers were thus deemed to be devices for evaluating functions. Since they were subject to the simulation methods used to establish the equivalence of the early models of effective procedure, they too could only compute Turing-computable functions. The only real questions of interest about a computable function were how fast could it be computed and by what methods? These questions were the genesis of the present-day fields of complexity theory and algorithms.

The practical picture began to change as early as the 1960's. Parallel hardware such as DMA channels began to be incorporated into computers [15]. Multiprogramming and multitasking operating systems were introduced to make better use of this new hardware [8]. Models of computation adequate for operating systems had to address issues of concurrency and non-terminating computations [25], both of which remain at the heart of distributed computing research today. Concurrency, within computer systems and between such systems and their environments, and the uncertainty it introduces, is fundamental to the understanding of distributed systems. Liveness properties replace termination conditions in many interactive problems.

The field of distributed computing theory distinguished itself from the related areas of operating systems and parallel computing when it began to focus on the intrinsic characteristics of physically distributed systems: the distinguishing elements of true concurrency (as opposed to the "pseudo-concurrency" that arises from task-switching in uniprocessor operating systems), asynchronous computation, communication delays, failures of communication or processors, and decentralized administration.

The original goals of this new field were ambitious: To build a mathematical theory of distributed computing that would shed light on distributed computing systems just as Turing machine theory had done for sequential computers. The hope was to find an abstract distributed model that would capture the salient features of real distributed systems while suppressing distracting and unenlightening details of the physical world. The theory to be constructed was to be elegant, general, and powerful.

Twenty-plus years later, these goals seem hopelessly naïve. The theory of distributed systems is immensely more complex than its sequential counterpart. Every one of the distinguishing elements has seemingly endless plausible variations. Elegant theoretical assumptions such as pure asynchrony (no timing assumptions whatsoever) and Byzantine faults (no assumptions limiting faulty behavior) lead to pessimistic results that do not jibe well with real-world experience.

Even finding precise specifications for the problems to be solved by distributed systems has proven to be much more difficult than expected. Many tasks of interest in the real world include philosophically subtle interplays of knowledge, cooperation, and competition in systems that can perhaps best be described as multiagent games. The particular input-output behavior exhibited by the system in isolation may be of less interest than global properties about the interaction of the system with its environment.

The remainder of this paper takes a very high-level view of theoretical distributed computing research to date and argues that there has been a disconnect between theory and applications. Much of the theoretical work concerns oversimplified problems on unrealistic models. Theoretical results, which are often clever and sometimes deep, are nevertheless difficult to apply to real-world problems. Attempts to make the models and problems more realistic while retaining theoretical elegance and generality have had limited success. While the theoretical work provides a framework and some good intuition for thinking about real-world distributed systems, it falls short of providing the clarity of understanding that would lead to the broadly-applicable synthesis and analysis tools envisioned early on.

Some of the most important contributions of the theory, which have successfully impacted and illuminated practice, are impossibility results and pessimistic lower bounds. Such results, often in very general and abstract models, justify implementation- or application-specific modeling assumptions. In sequential computing theory, a proof of NP-hardness may justify the search for application-specific heuristic solutions. Similarly, solutions to many distributed computing problems imply a solution to the simple consensus problem—justifying assumptions of synchrony, fault-freedom, failure detectors, use of randomization, or an appeal to strong communication primitives. The theory influences the implementation, but does not contribute directly to the solution.

We argue that the future viability of the field lies in its ability to contribute to solutions of practical real-world problems in theoretically elegant ways. This will require the identification of distinctive distributed computing problem domains and the development of domain-specific system models. While a unique general theory of distributed systems may be unattainable, the practical world of distributed computing is a rich field that is ripe for theoretical development.

2 Early Research

Early work on distributed computing grew out of work on practical problems such as operating system design, reliable network communication, network routing, email systems, concurrent databases, and avionics systems. The early theoretical work focused primarily on problems of concurrency (for shared memory) and fault-tolerance (in message-passing models). Time-slicing in operating systems led naturally to a concurrency model of interleaved atomic actions. Problems fruitfully discussed in this model included synchronization problems such as mutual exclusion, dining philosophers, and readers/writers. In the realm of fault tolerance, the Byzantine generals problem [22] showed the world that achieving fault tolerance was considerably more difficult than simple use of triple redundancy and majority voting would suggest.

2.1 Early Research Agenda

Against this background, the ambitious early research agenda was to develop a theoretical framework for distributed computing systems that supported analysis, synthesis, and verification of distributed protocols. Any computational framework comprises several components:

- Natural abstract *models*.
- Meaningful *performance measures*.
- Comprehensive and comprehensible *problem specifications*.
- *Rules of composition* that enable large systems to be built from simple, well-understood components and that enable complex systems to be decomposed into simpler, more easily understood components.

Once a suitable framework for distributed systems was constructed, the more ambitious goals could be attacked:

- Develop a *complexity theory* of distributed computation. Prove limits on what can and cannot be done by distributed systems. Understand how performance of large systems depends on the performance of their components.
- Develop general *synthesis techniques* to aid in finding a correct distributed protocol for a given problem specification.
- Develop general *verification techniques* to aid in proving the correctness of distributed protocols.
- Create a library of generally useful *primitive building blocks* and *analyze* their properties.
- Develop *domain-specific languages* for describing distributed computations.

The above goals apply to any theory of computation. Additional goals, specific to distributed systems, include:

- Understand and model intuitive notions of *cooperation* and *coordination*.
- Develop general and powerful *fault-masking* techniques.
- Develop techniques for *transforming* concurrent algorithms to distributed settings.

2.2 Accomplishments to date

The two decades of research accomplishments of the field are well documented in the proceedings of the several distributed computing conferences and in various scientific journals.[1] We will not try to summarize them here except to say

[1] The reader is referred to [18] for a comprehensive reference to many of these results.

that much progress has been made toward these goals. Many abstract models have been proposed that reflect interesting aspects of real systems. Some generally-agreed-upon performance measures have emerged. Many non-trivial distributed problems have been analyzed and solved. Many limits have been established on what can be achieved. Most important perhaps has been an exploration of the complexity explosion that results from dealing simultaneously with notions of computation, concurrency and asynchrony operating in a geographically distributed, decentralized environment controlled by agents that are both cooperative and competitive. The landscape of distributed computing has been mapped out to a considerable degree. Theoretical results have challenged previously-held beliefs (such as the sufficiency of triple redundancy for single-fault masking) and exhibited in stark relief strange phenomena that exist in the real world (such as the impossibility of masking even a single process crash failure in as general a way as previously believed). The conceptual framework that has been built is a firm basis for both practical work in distributed systems and also for future theoretical development of the field.

Despite these considerable successes, the field has not achieved the research goals set out early on for a general unified theory of distributed computing. The field has also not had as large an impact on practitioners as the early researchers had hoped it would. In retrospect, the original goals were naïvely optimistic. The real world is vastly more complicated than originally envisioned. General theoretical models often fail to account for significant aspects of real-world problems, and more specific models tend to become unwieldy and difficult to apply.

In the remainder of this paper, we explore in some detail properties of distributed systems that make them difficult to model and reason about, both as a way to explain the unexpected obstacles encountered during the first two decades of theoretical research, and also to suggest fruitful directions for future research.

3 A Disconnect between Theory and Practice

A topic of conversation at several of the early distributed computing conferences was the question of why theoretical distributed computing research was not having a greater impact on practitioners. One reason is that it takes time for new ideas to percolate across discipline boundaries. But even today, many of the most significant theoretical advances such as protocols for distributed consensus, leader election, and self-stabilization have found only limited practical application. Large-scale distributed systems are still either primarily client-server based, and hence do not face the same problems as what are now known as "peer-to-peer" distributed systems, or they use *ad hoc* protocols and algorithms tailored to the specific application. Programming fully distributed applications remains difficult!

We believe one of the primary reasons for the failure of the theoretical community to have greater impact on the practical world of computing has been a lack of sufficient generality and realism in models, methods, and results. In short, it has failed to achieve the ambitious research goal of creating a general theory of distributed systems! The field is much richer than was originally envisioned. No universal model has emerged on which to build a general theory. Many detailed assumptions are needed to make problems tractable. Finding the "right" abstractions to capture the essence of real-world problems and finding the "right" primitives from which to build a compositional theory have both proven difficult. Easily-applied rules of protocol composition that preserve properties of interest are elusive. Proofs of correctness are still notoriously difficult to produce, despite considerable progress. Without generality in methods or results, it is difficult to apply theoretical results to practical situations, for one or more of the underlying assumptions will often fail to be satisfied.

This is not to say that the theoretical work is misguided or unsuccessful. Theory's greatest value is in providing frameworks for thinking about problems, methods and techniques for finding potential solutions, and paradigms for analyzing the final results. Theory codifies knowledge obtained in one problem domain so that it can be transmitted to others and applied in new problem domains. Distributed computing theory has been quite successful at laying the groundwork for further study, at providing formal models that allow rigorous study of selected problems, and by generating elegant techniques for analyzing particular problems. Getting a firm understanding of the landscape is a necessary prerequisite to achieving the desired level of abstraction and generality—one that is broad enough to encompass interesting new situations yet specific enough to address the crucial issues. Much progress has been made, but more needs to be done.

In the early work, models and problem statements were kept simple in order to remain elegant and tractable, but this has often resulted in an overly-pessimistic view of the world. Even where solutions to particular problems exist, they may be complex to implement and costly to run. To make further progress, one must continually revisit the underlying assumptions with the goal of relaxing those that are unnecessarily restrictive or that conflict with reality. We now examine some of those assumptions.

3.1 Global coordination

Many models of distributed systems include unrealistic or problematical assumptions about global coordination of the system, e.g.,

- All processes start at time zero in a specified initial state.
- All processes terminate after a finite amount of time.

In practice, processes start and stop at different times, and a large distributed computation such as the internet Domain Name System [19] never terminates, even though individual processes are continually starting and stopping.

3.2 Process step timing and scheduling

Any distributed system must somehow account for the fact that processes controlled by independent clocks will necessarily run at different speeds. In the real world, step times of physical processors are affected by variables such as temperature and voltage that vary unpredictably but are likely to remain within reasonable bounds. On the other hand, step times of logical processes that result from time slicing are wildly varying. Most distributed models, in order to remain tractable, have eschewed timing details and assumed instead that processes are either fully synchronous or fully asynchronous.

In the *fully synchronous* model, all processes operate in lockstep, either with respect to the basic process step or with respect to communication with other processes. This is tantamount to assuming that all processes run at exactly the same speed, with no variation permitted. This assumption is difficult to justify in real-world situations where process speeds are determined by independent unsynchronized clocks and communication is subject to variable and unpredictable delays. While a powerful model for proving general lower bounds, it does not support practical algorithms.

In the *fully asynchronous* model, no assumptions are made about the relative speeds of different processes. This also obviates the need to make detailed timing assumptions, but the resulting model admits runs that are highly unlikely to occur in practice, thereby crippling the protocols with unrealistic constraints. For example, not only is there no *a priori* bound on the speed ratio of two processes; there is also no assumption that the ratio is constant within a run. A run in which process 1 takes one step after the first step of process 2, two steps after the second step of process 2, four steps after the third step of process 2, eight steps after the fourth step of process 2, and so forth, is perfectly acceptable. Without any timing assumptions, the commonly-used practical tool of timeouts is inapplicable. Not surprisingly, many problems proved impossible or expensive in the fully asynchronous model have been "solved" in practice.

Real systems tend to fall somewhere in between the two extremes of full synchrony and full asynchrony. Coordinated action is generally only possible when processors share a common clock. Thus synchronous models are only really applicable to tightly-coupled parallel computers and not to distributed systems. Full asynchrony is overly general for physical processors whose step time variability is bounded. But how best to capture the amount of variability that can be expected in practice is unclear. Assuming any particular fixed bound on process step times is arbitrary and unpleasing. Parameterizing this bound for the specific implementation to instantiate is dangerous, because incorrect or outdated settings can result in system failure, and non-modular, because bounds for one component may depend on bounds for another, and cycles seem inevitable. Moreover, conservative delay estimates lead to poor performance. Other approaches, such as assuming fixed but unknown bounds (so that the same protocol is supposed to work for all bounds) have been explored, resulting in a complex theory of models and algorithms [9,10]. Work on failure detectors [3] has made significant progress in encapsulating the issues of synchronization and fault-tolerance. But failure detectors (or reliable broadcast primitives, another encapsulation of asynchrony and faults) can bias designs towards a 'consensus bottleneck', where applications are constructed as monolithic state machines with centralized updates. Such designs incur significant performance penalties, in which the advantages of parallelism and fast components are sacrificed to coordination overhead with slow peers.

An alternative approach, that will become increasingly important in the future, is to incorporate clocks and explicit notions of real time into the model. For example, full asynchrony and timeouts can coexist in a model with real-time clocks in which one assumes a fixed upper bound on the step time of any process but no corresponding lower bound. Any asynchronous schedule of process steps remains possible in such a model since processes can run arbitrarily fast. The real-time clock, however, allows a process to observe the passage of time and thereby infer a lower bound on the number of steps taken by any other process.

Of course, once real time is introduced into a model, then one can begin considering real-time properties of the computation—*when* outputs appear may be just as important as *what* results are computed. While real time properties are important for practical applications, they can lead to complex and tedious bookkeeping in both code and analyses and overly-specific solutions that have little or no generality. Partly because of these difficulties, real time models are only beginning to be addressed by the distributed computing community.

3.3 Communication system

The communication system offers even more possibilities for variation than process scheduling. At the one extreme, communication is synchronous and immediate. Each message sent by a synchronous process at the beginning of a round is received at the end of the same round. There is no need for message buffers since every message is received immediately after being sent.

At the other extreme, message systems can be completely asynchronous. In this model, a message sent can be delayed arbitrarily long before eventually being delivered. Messages do not have to be delivered in the order they are sent. The message system can be thought of as an infinite unordered buffer that contains all of the messages in transit. In the most liberal version of this model, a process might receive no messages on a step even though many are waiting in the message buffer for delivery. The only constraint imposed by this model is that every message delivered must have been sent sometime in the past—messages are not spontaneously generated by the message system.

Many intermediate models have been studied. The message buffer might behave like a single queue, so that messages are delivered in the same order as they were sent. The message buffer might act like a set of queues, one for each process. In this case, messages are delivered to each process in the order sent, but messages to different processes can be delivered out of order. Or the sender might have a queue, ensuring that messages from a given sender are delivered in the order sent, but messages from different senders might be delivered to the same recipient out of order.

In practice, we do not have infinite message buffers, and recipients may be unable to receive messages as fast as they are being sent. In this case, messages are lost in transit. This is a kind of "fault" in the message system, but unlike other kinds of faults, this is not due to a malfunction of the communication system but is rather a transient property of the particular order of execution. How this should be modeled in a realistic way is still an open problem. One may of course make the pessimistic assumption that messages can be arbitrarily lost in transit, assuming only some minimal "liveness" property to ensure that sometimes messages do get through. In practice, though, whether or not messages get through depends in complicated ways on the traffic in the network. Interesting protocols such as TCP [26] *require* message loss to adapt gracefully to network congestion. A theory adequate to model such protocols would need a realistic model of network traffic that does not yet exist. In the absence of such a theory, a protocol's performance can be considered for the likely case that communication is failure-free, and the protocol should degrade gracefully as messages are lost.

As with process scheduling, real time and timeouts play important roles in real-world networks by allowing processes to detect and recover from lost packets and other kinds of network faults.

3.4 Fault tolerance

Understanding and controlling system behavior in the presence of faults is much more important for distributed systems than for sequential systems. There are three major reasons for this. Most obviously, redundancy is implicit in the notion of distribution, and redundancy offers opportunities for fault tolerance that simply don't exist in single-processor systems. Secondly, distributed systems are most frequently viewed as operating continuously in a reactive, real-time environment; there is no opportunity to rerun the computation in case of component failures. Finally, the larger the number of components with independent failure modes, the greater the probability of at least one failure actually occurring. In large real-world distributed systems, failures are the rule rather than the exception and so must be dealt with at all levels—problem specification, algorithm design, and analysis.

3.4.1 Fault models

In order to make systems fault tolerant, one must have a model of the kinds of faults to be expected. Almost endless variation is possible. First of all, one must consider which components of the system are likely to fail, processors or the communication systems, and how those faults manifest themselves. A faulty process might simply stop running, or it might continue to run but fail to send certain messages that it should send, or it might continue to run but behave completely erratically, sending incorrect messages and producing incorrect outputs (so-called *Byzantine faults*).

Byzantine faults are particularly difficult to deal with since every process must be distrustful of every message that it receives. The attraction of the Byzantine fault model is that it does not require a detailed characterization of the kinds of faults to be expected. Once a process is deemed to be faulty, it can do anything at all.[2] Weaker fault models always invite the question, "Why does one believe the 'permitted' faults are the only ones likely to occur?" For example, why should one assume that a crashing process stops cleanly without sending bogus messages? Under what circumstances might a processor fail to send messages but otherwise continue correct operation?

In the real world, we often distinguish between *natural faults* and *malicious (Byzantine) faults*. Natural faults are assumed to occur independently of the computation in progress and are usually assumed to result from a random process. Thus, a message might be randomly corrupted (a transient fault), or a processor might randomly overheat and become damaged (causing a persistent fault). Independent random faults can often be detected (with high probability) by simple redundancy techniques such as repeating the computation and comparing results, whereas malicious faults lack such nice properties. The malicious fault model is the worst-case model of fault tolerance. It demands that protocols operate correctly no matter what faults occur. But masking Byzantine faults is often expensive or impossible, so one often settles for lower degrees of fault tolerance.

3.4.2 Adversaries

In considering fault models, it is often convenient to imagine a game situation between the protocol, which is trying to produce a correct result, and an adversary, who is trying to create faults at just the worst possible times so as to cause the protocol to fail. The adversary wins the game if it has a strategy that causes the protocol to fail (with non-negligible probability). The protocol wins if it is immune to the adversary attacks (with high probability).

The adversary corresponding to the Byzantine fault model is a nondeterministic process that can create any sequence of faults (subject to limitations on the number of faulty processors, discussed in Section 3.4.3 below). Protocols that win against a Byzantine adversary are clearly desirable since they tolerate any sequence of Byzantine faults that may occur for whatever reason. However, as mentioned previously, the cost of such protocols may be so large as to make them unattractive to use, and often Byzantine-tolerant protocols do not exist. One is motivated therefore to find weaker adversaries (corresponding to weaker fault models) that nevertheless are capable of exhibiting the fault properties that one expects to see in practice.

One way in which the Byzantine adversary is unrealistically strong is that it ignores the question of how the particular bad sequence of faults that makes the protocol fail might be produced. In this model, all fault sequences receive equal attention. No distinction is made between likely and unlikely, or easily-computed and hard-to-compute, fault sequences.

Real-world faults occur for a reason. Natural faults result from some sort of a randomized physical process that is more or less independent of the computation at hand. Malicious faults result from an agent acting strategically to disrupt the protocol. Either way, the faults can be thought of as resulting from a randomized computational process that is subject to information and computational constraints. Even an intelligent, malicious agent that is trying to compromise a system must still somehow compute the faulty actions necessary to

[2] Actually, even Byzantine-faulty processes are constrained in what they can do. Typically the model does not enable them to prevent non-faulty processes from communicating directly, in contrast to the denial-of-service attacks prevalent on the internet.

bring the system down. Such an agent is considerably weaker than a Byzantine adversary. For example, protocols that use strong cryptographic checksums to assure data integrity are believed to be secure in practice; however, they can be easily defeated by a non-deterministic Byzantine adversary that is able to guess the decryption key.

Once we bring adversaries into our model, we are in the unenviable position of having to define precisely the capabilities and actions of the adversary, and to justify the relevance of those assumptions in practice. In principle, any part of a system might fall under control of an adversary and therefore be considered faulty. Not only can the processors and communication subsystem misbehave, but the adversary might also corrupt underlying infrastructure such as the physical devices that control process scheduling and random number generation upon which the system depends.

The power of the adversary must be somehow limited; otherwise most problems have no solution. Adversaries can be limited in three general ways: static limits on the numbers and kinds of faults produced, limits on information available to the adversary, and computational limits on the adversary's power to process that information. Static limits include the common case of assuming a bound on the number of processes that can ever become faulty, or a bound on the rate at which messages or other faults can occur. Informational limits might prevent an adversary from predicting future random choices that affect the computation. They might prevent an adversary from seeing inside processes it does not control. They might prevent an adversary from seeing traffic on private communication channels. Computational limits assume that the adversary has limited computational power with which to choose its actions.

To actually impose such limits on an adversary in a precise and rigorous way, one is forced to describe a computational model that includes processes, communication, environment, and adversaries, all of which can interact in complicated ways. For example, a "dynamic adversary" is limited in the total number of processes it can corrupt, but the information available to it depends on previous corruptions. As long as a process is uncorrupted, its memory is hidden from the adversary. However, the dynamic adversary can, at any time, decide to corrupt a particular process. At that time, it gets to see the entire process state and can also take control of the behavior of the process. Or perhaps the adversary gets to see the entire history of the corrupted process. While the dynamic adversary is an appealing abstraction of a well-funded subversive group, it is so complicated that its underlying assumptions become harder and harder to justify in the real world. For example, why should we assume the adversary is limited in the number of processes it can corrupt? Why is it reasonable to assume that it can corrupt any process it wants to whenever it wants to? If the reason the number is limited is that some processes are inherently incorruptible, then that set would presumably be fixed at the beginning and the adversary would not have the freedom to corrupt any process it chooses to.

Adversarial assumptions constrain the safe range of application of a system. Care must be taken in composing systems with different assumptions, and they must be documented and tracked as system components are abstracted and applied in new contexts. (A forgotten assumption, that frequent relocation would force periodic re-boots, led to poor performance when Patriot missiles were statically sited during the 1991 Gulf War [14].)

3.4.3 Tolerance requirements

The above discusses what kinds of faults to expect. There is also an issue with how many faults to tolerate. Intuitively, one wants protocols that tolerate faults as well as possible. In practice, one generally doesn't know how many faults to expect, so one wants the likelihood of incorrect operation to be as small as possible for however many faults actually do occur. Of course, if too many components fail, one expects the protocol to fail, but one desires protocols that make a "best-effort" attempt to mask faults and do "as well as possible", even if a hoped-for threshold is exceeded. For example, both on-board processors simply shut down during the infamous Ariane-5 rocket failure, although the duplicated exception was in a superfluous routine [2]. Surely it would have been better for the second processor to sense it was running in simplex mode and to at least attempt to proceed without the offending process?

By way of contrast, it is typical in theoretical papers to replace the "best-effort" requirement with a threshold requirement. For a threshold t, this states that the protocol must operate correctly if at most t faults occur, but no requirement is imposed if more than t faults occur. (This holds true even for many algorithms which claim to be "gracefully degrading" or "adaptive".) A requirement for gradual degradation of service with increasing faults, as is common for survivable systems, cannot be specified using the threshold model. Moreover, it is sometimes possible for a protocol to exploit the threshold requirement by relaxing its resistance to additional faults as it observes that more and more faults have already occurred, resulting perhaps in improved efficiency but at the great expense of survivability under high fault loads.

3.5 Environments

The environment of a distributed system is the "glue" that binds together the processes, communication system, and external inputs. The environment is generally assumed to control the non-deterministic and randomized aspects of the system. That is, the environment supplies the information that determines a particular run of the system.

Whatever is left to the environment is subject to assumptions that the environment operates correctly. In the case of asynchronous process scheduling, one usually assumes some sort of fairness condition on the environment that at least implies that no processes are starved (that is, at every point in time, every correct process eventually gets to take another step). Does it make sense to talk about faulty environments? For example, a faulty environment might indeed starve a correct process. Is this a kind of fault we are concerned about? Why or why not? Presumably the answer to such a question would arise from an investigation of the physical processes that justified our original assumption. What do we have to assume about the real world in order to believe that every correct process will eventually take another step? Presumably this argument would boil down to a basic understanding of physics and the relative independence of spatially-separated physical devices. The validity of such an analysis must be re-examined as systems are adapted to new uses: an environment failure, or rather the falsification of an environmental assumption, was the root cause of the Ariane-5 rocket failure.[3] Are those assumptions subject to manipulation by an adversary? Maybe or maybe not, but we can at least pose the question.

4 Future Directions

The previous section surveyed some of the issues that make it difficult to find general models for real-world distributed computing situations that are both useful and accurate. As a result, much theoretical distributed computing research has opted for simplicity and elegance rather than practical accuracy. There was good reason to believe initially that a comprehensive general theory would emerge, but our experience to date suggests that a general theory that supports synthesis (as opposed to lower bounds) is not on the horizon. Future research needs to become more specific to the particular application domain and environment characteristics. Models need to be extended in various directions to overcome the limitations discussed previously. Decisions on what features to include and what not to include should be made in the context of the motivating problem domain. The result will be many domain-specific models rather than a universal general model. Generality may eventually emerge as the various domains become better understood, but for now, pragmatism should be emphasized. Carrying out this program will require interdisciplinary research between the theoretical community and the more practical domain-specific research communities.

Below are some of the issues that will be increasingly important in future work.

4.1 Real time

Real-time properties are critical to many real-world situations such as process control, but they arise in some form or other in almost any practical application. It is not enough to eventually complete a task; the result must become available soon enough to become useful. Real-time computing has historically been considered a distinct research area, but timing issues are too pervasive to be viewed as a subspecialty. Various theoretical models of time are possible, ranging from the view of time as a partial ordering on events [17] all the way to considering time as a continuously-varying real number. Introducing continuity into otherwise-discrete models of computation raises a number of interesting issues in its own right [1]. Weaker notions of time that allow modeling discrete clocks and timeouts should also be explored more extensively. Time and timing constraints must also be incorporated into problem specifications and performance measures.

4.2 Communication systems

The assumed underlying communication mechanism has a large effect on the applicability of a distributed computing model. Communication systems have historically been considered a distinct research area (networking), but the ability to accurately model network properties is crucial to many distributed domains. Properties such as performance, survivability, and security of distributed systems essentially depend on the structure of the communications network. Realistic adversarial models such as one needs in modeling internet security may require detailed information on

- connectivity of the underlying graph;
- the sets of communication paths that an adversary may control;
- how much control an adversary has over timing and synchronization;
- how much variability is expected in packet routing.

Moreover, a balance between expected-case and worst-case performance, crucial to practical systems, presumes a good understanding of the expected regime. Collaboration with networking specialists is likely to be particularly rewarding.

4.3 Fault notions

Fault-tolerance has been a cornerstone of much of the early theoretical distributed computing research. Simple fault models that work well for benign faults of nature break down when considering intentional fault behaviors. While intentional deviations from a protocol are often called “malicious faults” and used as justification for the Byzantine fault model, the result is far too pessimistic. Most agents are not malicious but are more accurately described as self-interested, or operating according to internal belief structures that may be irrational according to other plausible belief structures. Such agents are neither benign nor malicious but operate somewhere in between. Many risks are tolerable because most real-world agents do not exhibit extreme Byzantine behavior. To reason about agents, one may need to understand their knowledge and motives. Assuming full honesty is certainly naïve. But assuming the worst precludes many practical solutions.

4.4 Randomized protocols

Randomization is crucial to many distributed protocols. Some of its many uses include breaking symmetry, tagging data with globally unique identifiers, and allowing use of cryptography [13]. Randomization can be introduced into a model either by giving processes explicit access to random number generators or by making probabilistic assumptions about the environment, for example, that random noise is injected into every message sent. Whatever the source of randomness, the effect is to impose a probability distribution on the space of possible behaviors. Certainty properties may be replaced with properties that hold only with high probability.

If explicit random number generators are introduced into the model, a new fault modality must also be considered—randomization faults. Namely, suppose the random number

[3] Flight characteristics of the Ariane-5, specifically horizontal bias measurements, exceeded values observable in safe launches of the Ariane-4. When the Ariane-4 software was ported to the new platform, this crucial environmental assumption was not correlated with the anticipated change in flight characteristics [2].

generator produces outputs with the wrong distribution. This could lead to processes that exhibit correct behaviors but with the wrong probabilities. Detecting and masking randomization faults may be an important consideration in the practical application of randomized protocols since we rarely have confidence in the true randomness of our random number sources.

4.5 Problem specifications

Perfection is difficult in real-world distributed systems. In many cases, absolute correctness properties need to be replaced by probabilistic ones, and the underlying models need to allow for meaningful statements about the distribution of system behaviors.

Along with richer models, we need more refined problem perspectives. Distributed systems cannot be adequately viewed as merely devices for computing prescribed functions from inputs to outputs, even when the notions of inputs and outputs are suitably generalized to account for the on-line nature of distributed computation as described above, using temporal logics or related formalisms [23]. Rather, in many real-world situations, the purpose of a distributed system is to behave "well" in a larger environment. What is of interest are global properties of the system operating within its environment rather than the particular function computed. A simple example is the consensus problem, where the property of interest is the agreement among the (non-faulty) processes, not the particular value agreed upon. Similarly, the requirements for an electronic auction system will include statements about the utility functions of the bidders and the interests of the sellers that fall outside of a narrowly-construed computational model.

4.6 Agents

In the real world, the environment of a distributed system includes a collection of human agents with intentions and motivations. Because people control the processes of the system, it is natural to ascribe intentions and motivations to the processes themselves. A distributed system is then viewed as a large multiagent cooperative/competitive game. A particular system is neither correct nor incorrect, faulty or not. Rather it, together with the agent behaviors, determines a play of the game. The goodness or badness of the result is measured by how well it satisfies the individual and collective goals of the participants. Economic theory suggests that equilibria properties of such systems may be obtainable even where exact characterizations of the behaviors is intractable. Distributed systems can implement very complex games, yet the range of possible behaviors is limited by the computational power of the participants. Studying such systems from a computational perspective may provide fruitful insights that are unattainable in traditional game-theoretic models [28]. Algorithmic mechanism design [11,20], trust-modeling [7], and algorithmic game theory [21] give glimpses into some of the new kinds of results that this approach can yield.

5 Summary

The 20-year old goals for a general theory of distributed systems are just as seductive now as they originally were:

- universal model
- general specification techniques
- general verification methods
- general protocol synthesis techniques
- general notions of cooperation and coordination
- general and meaningful performance measures

Experience over the past two decades has shown that achieving these goals in practically-relevant ways is much more difficult than originally imagined. Future research will need to become more domain specific and to cross disciplinary boundaries. It should identify problem domains of practical interest and strive to construct realistic models for those limited domains. For example, one might distinguish between cooperative distributed environments such as the Grid [12], where participants are assumed to be non-malicious, and hostile environments such as the Internet at large, where assumptions of benign intent are unfounded.

We conclude with some goals for future research:

- Identify and characterize additional interesting domains of practical importance.
- Find increasingly realistic distributed models and problem specifications for specific domains.
- Explore and understand properties of the resulting models and their relationship to competing models.
- Identify unrealistic assumptions and replace them with practically justifiable ones.
- Establish correctness properties in the gray area between absolute perfection and catastrophic failure.
- Understand and apply knowledge from related disciplines such as real-time systems, computer networking, programming languages, and economics.
- Model and understand realistic agents in multiparty cooperative/competitive environments.

Acknowledgements. Our goal in this paper is to be succinctly provocative, rather than encyclopædic and bibliographic. In the interest of inciting debate and novel investigations, our tone is deliberately critical rather celebratory. But both authors have been appreciative members of a vibrant and productive research community since its inception. We have included a few key references, but any overview of a only a few pages must omit work of great significance, beauty, and depth. Moreover, we have recommended areas and directions for future research, but do not mean to suggest that these areas are wholly unexplored. Indeed, several reviewers have helped us sharpen our arguments by rebutting with references to extant bodies of work.

We are grateful to Hagit Attiya, Sergio Rajsbaum, and Vassos Hadzilacos for their encouragement and patience during the writing of this paper. We thank Walid Taha for many insightful comments and for the "domain-specific" terminology. We thank the referees for pointing out many weaknesses in an earlier draft of this paper that were even more egregious than the ones that remain.

References

1. Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *Proceedings of the 30th Annual Symposium on Foundations*

of Computer Science, pages 164–169. IEEE Computer Society Press, 1989.

2. [Ariane]. Inquiry board traces Ariane 5 failure to overflow error. *SIAM News*, 29(8), October 1996.
3. Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
4. Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.
5. Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.
6. Martin Davis, editor. *The Undecidable*. Lippincott Williams & Wilkins (formerly Raven Press), New York, 1965.
7. Zoë Diamadi and Michael J. Fischer. A simple game for the study of trust in distributed systems. *Wuhan University Journal of Natural Sciences*, 6(1–2):72–82, March 2001. Also appears as Yale Technical Report TR–1207, January 2001.
8. Edsger W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.
9. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.
10. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
11. Joan Feigenbaum and Scott Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13. ACM Press, 2002.
12. Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, 1998.
13. Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*, volume 17 of *Algorithmics and Combinatorics*. Springer-Verlag, Berlin, Heidelberg, 1999.
14. Government Accounting Office. *Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia*, GAO/IMTEC-92-26, February 1992.
15. International Business Machines Corporation. *Reference Manual: IBM 709 Data Processing System.*
16. Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(5):727–742, 1936.
17. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
18. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.
19. Paul Mockapetris. Domain names—concepts and facilities. *Internet Requests for Comments (RFCs)*, 1034:55, November 1987. Available from `http://www.rfc-editor.org/rfc/rfc1034.txt`.
20. Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35(1–2):166–196, April 2001.
21. Christos Papadimitriou. Algorithms, games, and the internet. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 749–753. ACM Press, 2001.
22. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the Association for Computing Machinery*, 27(2):228–234, April 1980.
23. Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 1981.
24. Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
25. Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., sixth edition, 2001.
26. [TCP]. Transmission control protocol. *Internet Requests for Comments (RFCs)*, 793:85, September 1981. Available from `http://www.rfc-editor.org/rfc/rfc1034.txt`.
27. Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (series 2)*, 42:230–265, 1936–7. Corrections, Ibid, 43:544–546, 1937.
28. J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

Michael J. Fischer is currently a professor of Computer Science at Yale University in New Haven, Connecticut.

Michael Merritt is currently head of the Network Optimization and Analysis Research Department at AT&T Labs–Research, in Florham Park, New Jersey.