Distributed Computing Theory To Date (Part I: Models and an Example)

Marios Mavronicolas[*]

January 16, 2001

Abstract

Distributed Computing Theory has been undergoing a series of spectacular developments in the last two decades or so; such developments have turned it into one of the most vivid and challenging fields of modern Theoretical Computer Science. In this four-part survey, we attempt to highlight some of the most thrilling developments, both by now classical and more recent, in this young field. Our presentation is structured around three main axes:

- canonical problems in the field, and results concerning them that have spawned a whole bunch of subsequent research activities and findings (part II);
- theoretical and mathematical concepts, techniques and tools that have contributed to the establishment of the formal foundations of the field (part II);
- some of the currently major research directions within the field, that are being observed to promote its frontier and are hoped to lead to even more significant developments in the future (part III).

In this first introductory part, we introduce two common examples of formal models of distributed computation *message-passing* and *shared memory,* and we discuss a very simple example of an algorithm to achieve symmetry in a specific message-passing system.

[*]Department of Computer Science, University of Cyprus, P. O. Box 20537, Nicosia CY-1678, Cyprus. Supported by funds for the promotion of research at University of Cyprus, and by the research program *Efficiency and Performance of Distributed Systems: Capabilities and Limitations,* Research Promotion Foundation, Nicosia, Cyprus & General Secretariat for Research and Technology, Athens, Greece – Joint Program of Scientific and Technological Collaboration between Greece and Cyprus. Email: mavronic@ucy.ac.cy

1 Introduction

Distributed Computing Theory investigates the computability and complexity properties of theoretical models for distributed systems, much in a corresponding way that (classical) Theory of Computation studies the computability and complexity properties of sequential computers. However. while Theory of Computation has been drawing its problems among those arising in the everyday practice of computation, such as *sorting, searching* and *matrix multiplication* (see, e.g., [3] for a wealth of successes), the problems on which Distributed Computing Theory focuses have been of a completely different flavor: they are of the sort arising in a distributed computer system; examples of such problems include processor coordination, communication, and robustness to faults.

Distributed computer systems are extremely common nowadays to the extent that they need no particular introduction. Examples of distributed systems range from the Internet, to airline reservation and management systems, and to shared memory multiprocessor architectures; see [4] for many more examples. The motivation for introducing and using distributed computer systems is to share computing resources more efficiently, enhance communication, and improve performance.

The fundamental computing unit of a distributed system usually goes by the name *process* or *processor*. For our purposes, a *process* is but a sequential *thread*: a piece of code that includes instructions. Some of the instructions may involve access to *communication objects,* such as registers or channels. Processors interact through these communication objects in a certain, in general unpredictable manner. The two major interprocessor communication models are *message-passing* and *shared memory*. In the first, processors interact by sending messages to each other through a *communication graph*; in the second, communication is achieved through a set of *shared variables*.

Each processor is an independent processing unit equipped with local memory that runs a local program. The local programs contain both computation and communication instructions. A distributed algorithm is but a collection of local programs, one for each processor. Executions of an algorithm are produced by running the local programs independently (perhaps though under some restrictions in some models).

Two important characteristics of a distributed system that affect the possibility and the complexity of solving problems are the degree of *synchrony* and the degree of *faults*. As expected, the more synchrony available, the more problems can be solved and more efficiently. The model making the strongest assumptions about synchrony is the *synchronous* model, where computation proceeds in rounds and all processors take a step in each round. At the other

extreme lies the *asynchronous model*, where arbitrary interleaving of steps is allowed. The most benign type of faults are *crash* failures where a processor stops executing from some point on in an execution. At the other extreme, a *Byzantine* fault allows a processor to arbitrarily deviate from its local program. There is a great deal of literature on the impact of these two characteristics, synchrony and faults, on distributed computability and complexity; we will discuss some of the related results in later parts of this survey.

Space limitations have prohibited an attempt to a complete treatment in this survey of all topics that have been investigated within the framework of Distributed Computing Theory. Thus, many important topics will have to be introduced very briefly or even remain unaddressed. Even for the topics that we will touch, the list of references will by no means be complete; we hope that the included references will serve as initial pointers to a substantial body of literature on distributed computing.

The rest of the first part of this survey paper is organized as follows. Section 2 presents the essentials for models of distributed computation. An example algorithm is discussed in Section 3. We conclude, in Section 4, with some initial remarks and pointers to additional information.

2 Two Models of Distributed Computation

In this section, we provide the basic elements of a formal model of a distributed system. These elements are essential for following the rest of this survey, although somehow more detailed than needed for the sake of providing an example of a formal model of a distributed system. Our presentation partially follows [2, Sections 3.1 & 4.1]. The model itself is a simplification of other formal models, such as the I/O automaton [11]. We use the terms processors and processes interchangeably.

2.1 Message Passing

A *system* consists of n processors $p_1, \ldots, p_n$. Processors are located at the nodes of a graph $G = (V, E)$, where $V = [n]$. For simplicity, we identify processors with the nodes they are located at and we refer to nodes and processors interchangeably. Each processor p_i is modeled as a (possibly infinite) state machine with state set Q_i. The state set Q_i contains a distinguished *initial state* $q_{0,i}$. The state set Q_i also includes a subset I_i of *idle* states; we assume $q_{0,i} \notin I_i$. We assume that any state of p_i includes a special component, $buffer_i$, which is p_i's message buffer. A *configuration* is a vector $C = (q_1, \ldots, q_n)$ where q_i is the local state of p_i; denote

$state_i(C) = q_i$. The *initial configuration* is the vector $(q_{0,1}, \ldots, q_{0,n})$. Processes communicate by sending *messages*, taken from some alphabet $\mathcal{M}$, to each other. A *send action* $send(j, m)$ represents the sending of message m to a neighboring process p_j. Let $\mathcal{S}_i$ denote the set of all send actions $send(j, m)$ for all $m \in \mathcal{M}$ and all $j \in [n]$, such that $(i, j) \in E$; that is, $\mathcal{S}_i$ includes all the send actions possible for p_i.

We model computations of the system as sequences of *atomic events*, or simply *events*. Each event is either a *computation event*, representing a computation step of a single process, or a *delivery event*, representing the delivery of a message to a process. Each computation event is specified by $comp(i, S)$ for some $i \in [n]$. In the computation step associated with event $comp(i, S)$, the process p_i, based on its local state, changes its local state and performs some set S of send actions, where S is a finite subset of $\mathcal{S}_i$. Each delivery event has the form $del(i, m)$ for some $m \in \mathcal{M}$. In a delivery step associated with the event $del(i, m)$, the message m is added to $buffer_i$, p_i's message buffer.[1]

Each process p_i follows a deterministic local algorithm $\mathcal{A}_i$ that determines p_i's local computation, i.e., the messages to be sent and the state transition to be performed. More specifically, for each $q \in Q_i$, $\mathcal{A}_i(q) = (q', S)$ where q' is a state and S is a set of send actions. We assume that once a process enters an idle state, it will remain in an idle state, i.e., if q is an idle state, then q' is an idle state. An *algorithm* (or a *protocol*) is a sequence $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of local algorithms.

An *execution* is an infinite sequence of alternating configurations and events

$$\alpha = C_0, \pi_1, C_1, \ldots, \pi_j, C_j, \ldots ,$$

satisfying the following conditions:

1. C_0 is the initial configuration;
2. If $\pi_j = del(i, m)$, then $state_i(C_j)$ is obtained by adding m to $buffer_i$.
3. If $\pi_j = comp(i, S)$, then $state_i(C_j)$ and S are obtained by applying $\mathcal{A}_i$ to $state_i(C_{j-1})$;
4. If π_j involves process i, then $state_k(C_{j-1}) = state_k(C_j)$ for every $k \neq i$;
5. For each $m \in \mathcal{M}$ and each process p_i, let $S(i, m)$ be the set of j such that π_j contains a $send(i, m)$ and let $D(i, m)$ be the set of j such that π_j is a delivery event $del(i, m)$. Then there exists a one-to-one onto mapping $\sigma_{i,m}$ from $S(i, m)$ to $D(i, m)$ such that $\sigma_{i,m}(j) > j$ for all $j \in S(i, m)$.

[1] The system model can be extended to allow arbitrary state change upon message delivery without changing the results; for clarity of presentation, we chose not to do so.

That is, in an execution the changes in processes' states are according to the transition function, only a process which takes a step or to which a message is delivered changes its state, and each sending of a message is matched to a later message delivery and each message delivery to an earlier send. We adopt the convention that finite prefixes of an execution end with a configuration, and denote the last configuration in a finite execution prefix α by $last(\alpha)$. We say that $\pi_j = comp(i, S)$ is a *non-idle step* of the execution if $state_i(C_{j-1}) \notin I_i$, i.e., it is taken from a non-idle state.

2.2 Shared Memory

Processes communicate by *b-atomic registers* (also called *shared variables*). Fix some integer $b > 0$ called the *fan-in*. Each shared variable may attain values from a *domain*, a set $\mathcal{V}$ of *values*, which includes a special "undefined" value, $\perp$. Each process p_i has a single *read-modify-write* atomic operation available to it that may read a *shared variable* $\mathcal{R}$, return its value v, and modify $\mathcal{R}$. Associated with each shared variable $\mathcal{R}$, is a set $Access(\mathcal{R})$ that includes the processes which may perform atomic operations on $\mathcal{R}$; we assume that, for each $\mathcal{R}$, $|Access(\mathcal{R})| \leq b$.

A configuration is extended to consist of the states of the processes and the values of the shared variables. Formally, an *extended configuration* $\tilde{C}$ is a vector $\langle q_1, \ldots, q_n, v_1, v_2, \ldots \rangle$, where q_i is the local state of p_i and v_k is the value of the shared variable $\mathcal{R}_k$; denote $state_i(\tilde{C}) = q_i$ and $value_k(\tilde{C}) = v_k$. The *initial configuration* is the configuration in which every local state is an initial state and all shared variables are set to $\perp$.

Each event is a *computation event* representing a computation step of a single process; it is specified by $comp(i, \mathcal{R})$ for some $i \in [n]$. In this computation step, the process, p_i, based on its local state performs an operation on a shared variable $\mathcal{R}$, performs some local computation, and changes to its next state.

Each process p_i follows a deterministic local algorithm $\mathcal{A}_i$ that determines p_i's local computation, i.e., the register to be accessed and the state transition to be performed. More specifically, $\mathcal{A}_i$ determines:

- A shared variable $\mathcal{R}$ as a function of p_i's local state.
- Whether p_i is to modify $\mathcal{R}$ and, if so, the value v' to be written, and p_i's next state as a function of p_i's local state and the value v read from $\mathcal{R}$.

We assume that once a process enters an idle state, it will remain in an idle state. An *algorithm* (or a *protocol*) is a sequence $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of local algorithms.

3 An Example

In this section, we discuss a very simple example of a distributed algorithm for a message-passing distributed system whose underlying communication network has the tree topology. We introduce an algorithm to break symmetry in this system, and we discuss ways to evaluate it. The algorithm is taken from a recent interesting paper by Dinitz, Moran and Rajsbaum [5], investigating the exact communication costs for achieving and breaking symmetry in a tree network.

The *consensus* problem models achieving symmetry in a distributed system. Due to its apparent foundational importance, it has been studied extensively in the literature on distributed computing theory (see [2, Chapter 5] for a collection of results and references). Consensus is a particular case of a *decision task*, where processors start with input values and must eventually decide on output values that satisfy the task's specification. For the sake of presentation, we assume that the inputs are the processors' *id's* (distinct identification numbers), more specifically the least significant bits of them, while the outputs are binary. In the consensus decision task, all processors must decide on the same bit. If all id's are odd, they must output 1, else they must output 0.

A *terminal processor* in a tree network is a processor residing at a leaf; all other processors are called *intermediate processors*. We sketch below (omitting some details) a simple algorithm that enables processors to solve the consensus problem for a tree:

- Upon its wake-up, each terminal processor sends its bit (precisely the least significant bit of its id) to its (unique) neighbor.
- Each intermediate processor p waits for the moment of receiving messages over all links incident to it, except for a single one. It then relays the accumulated maximum bit over this unique remaining link.
- Each processor p waits for the moment of receiving messages over all links incident to it. It then decides on the maximum of all bits it had ever seen and relays its decision in the reverse direction over all links except the one it had previously sent. (Thus, a terminal processor will not send a bit for a second time over its unique incident link.)

It is simple to see that this algorithm is correct. We would like to quantitatively measure the *efficiency* of this algorithm using the two main complexity measures that are used for evaluating (and comparing) distributed algorithms, namely number of messages and time (the latter to be discussed in later parts). The *message complexity* of an algorithm is the number of messages

sent in executions of the algorithm. We usually consider the *worst-case* message complexity which is taken as the maximum number of messages sent over all executions. Sometimes, we distinguish message complexity from *bit complexity,* which is the number of bits sent. This distinction accounts for the *size* of messages and awards algorithms that send as few bits as possible.

The presented algorithm sends exactly two messages with a single bit over each link (in opposite directions). This happens in all possible executions of the algorithm, and not just in the worst-case. Since a tree with n vertices has $n-1$ links, the *bit complexity* of the algorithm is $2(n-1)$, thus $\Theta(n)$.

4 Conclusion and Further Information

In the first part of this survey, we have reviewed the message-passing and shared-memory models of distributed computation, and provided a simple example to convey the flavor of a distributed algorithm. In the next (second) part, we will survey significant concepts and results related to the central problems of mutual exclusion, leader election and consensus. These problems carry much of the flavor of the problems arising in a distributed system, and they have been studied extensively since the early days of distributed computing. They have been the vehicles of the earliest algorithms and impossibility results in distributed computing theory, and have lent themselves as convenient test-beds for important ideas, tools and techniques, such as randomization, formal reasoning and impossibility proofs, that have played a significant role for the development of the field. They are expected to continue drawing the attention of the research community, delimiting the formal foundations and theoretical principles of the field, setting the stage for more applications-oriented research to benefit from and be built on top of them, and motivating and inspiring younger researchers.

Current and on-going research on the theory of distributed computing is reported in the proceedings of the annual *ACM Symposium on Principles of Distributed Computing (PODC),* and the annual *International Symposium of Distributed Computing (DISC),* which evolved from the former *International Workshop on Distributed Algorithms (WDAG).* Some other broader conferences of Theoretical Computer Science that are covering research on the theory of distributed computing are the annual *International Colloquium on Automata, Languages and Programming (ICALP),* the annual *Symposium on Theoretical Aspects of Computer Science (STACS),* the annual *International Colloquium on Structural Information and Communication Complexity (SIROCCO),* the annual *IEEE Symposium on Foundations of Computer Science (FOCS),* the annual *ACM Symposium on Theory of Computing (STOC),* and the annual *ACM Symposium*

on Parallel Algorithms and Architectures (SPAA). The annual *IEEE International Conference on Distributed Computing Systems (ICDCS)* is a major forum for complementary research on distributed systems. A similar forum is the annual *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Distributed Computing is a specialized journal, which is exclusively devoted to research articles that deal with distributed computing. Distributed Computing is also a featured research area for the *Journal of the ACM*. Many other major journals of Theoretical Computer Science have traditionally been hosting research articles in distributed computing theory; we refer to *SIAM Journal on Computing, Information and Computation, Journal of Computer and System Sciences, Journal of Algorithms, Theoretical Computer Science, Theory of Computing Systems, Information Processing Letters, Algorithmica* and *Chicago Journal of Theoretical Computer Science*, to mention a few. Papers on distributed computing appear also in journals of a somehow more practical orientation, such as *IEEE Transactions on Parallel and Distributed Systems* and *Journal of Parallel and Distributed Computing*. Online bibliographies for many of these journals and the conferences listed above can be found on the World Wide Web.

Two excellent, authoritative books on the theory of distributed computing have been written by Attiya and Welch [2] and Lynch [10]. A representative book on distributed systems is the one by Coulouris, Dollimore and Kindberg [4]. Two relatively recent surveys on distributed computing theory have been written by Attiya [1] and Gafni [7]. The latter places a strong emphasis on issues of fault-tolerance. An earlier survey was written by Lamport and Lynch [8].

A nice survey talk on lower bounds in distributed computing has been given by Fich [6]. Impossibility results in distributed computing were earlier surveyed by Lynch [9].

References

[1] H. Attiya, "Distributed Computing Theory," in *Parallel and Distributed Computing Handbook,* A. Y. Zomaya ed., pp. 127–160, McGraw Hill, 1996.

[2] H. Attiya and J. L. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* McGraw Hill, 1998.

[3] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms,* McGraw-Hill and The MIT Press, 1990.

[4] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems, Concepts and Designs,* Second Edition, Addison-Wesley, 1994.

[5] Y. Dinitz, S. Moran and S. Rajsbaum, "Exact Communication Costs for Consensus and Leader in a Tree," *7th International Colloquium on Structural Information and Communication Complexity,* in *Proceedings in Informatics 7,* Carleton Scientific Press, pp. 63–77, June 2000.

[6] F. Fich, *Lower Bounds in Distributed Computing,* slides from invited talk at *13th International Symposium on Distributed Computing,* Bratislava, Slovakia, September 1999.

[7] E. Gafni, "Distributed Computing: a Glimmer of a Theory," in *Algorithms and Theory of Computation Handbook,* M. J. Atallah ed., CRC Press, 1999.

[8] L. Lamport and N. Lynch, "Distributed Computing: Models and Methods," in *Handbook of Theoretical Computer Science,* Volume B (*Formal Models and Semantics*), Chapter 19, pp. 1157–1199, MIT Press, 1990.

[9] N. Lynch, "A Hundred Impossibility Proofs in Distributed Computing," *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing,* pp. 1–28, August 1989.

[10] N. Lynch, *Distributed Algorithms,* Morgan Kaufmann, 1996.

[11] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly,* Vol. 2, No. 3, 1989.