Distributed Computing I

Jörn W. Janneck

Computer Science Dept.

Lund University

what we try to accomplish

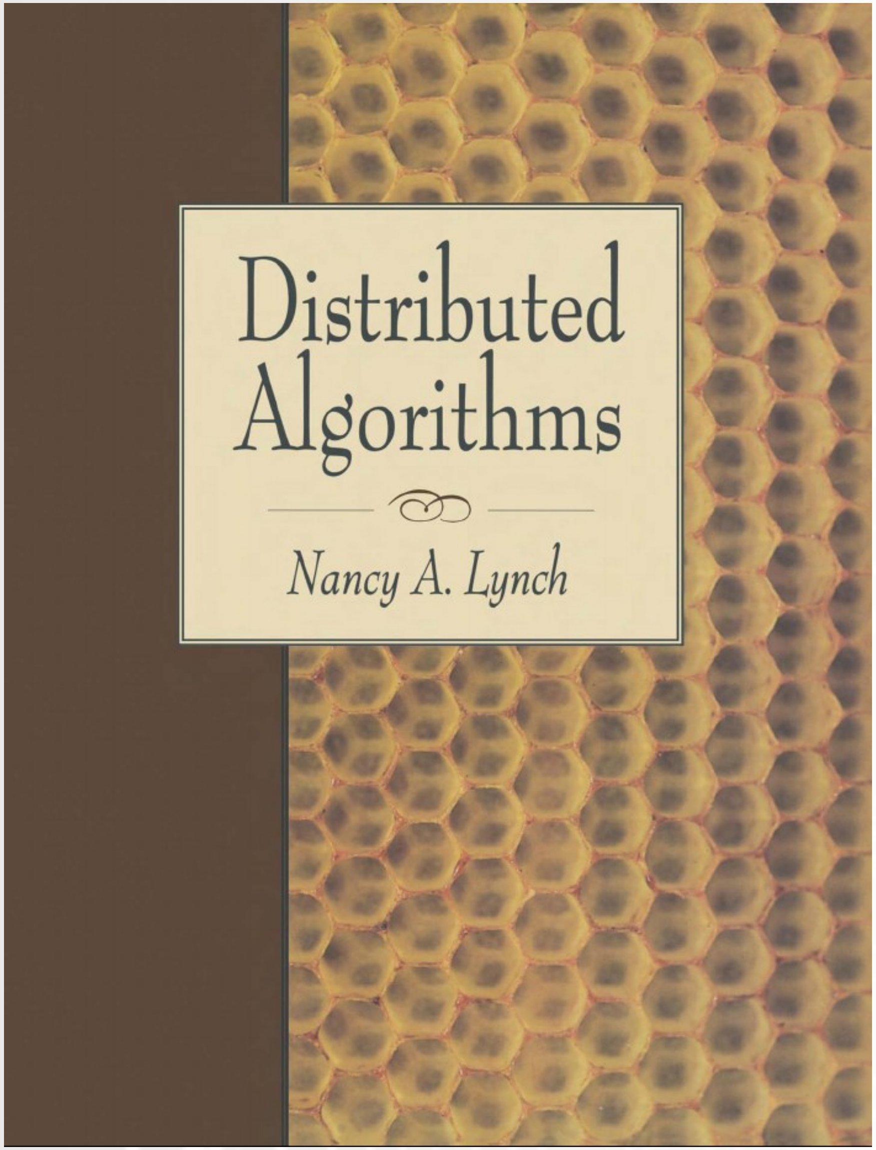
basic terminology in distributed computing

some fundamental concepts and
ways of thinking about distributed systems

some “classical” results and papers

shed some light (?) on the relationship between
distributed systems and cloud computing

Nancy A. Lynch
Distributed Algorithms
Morgan Kaufmann, 1996



distributed computing

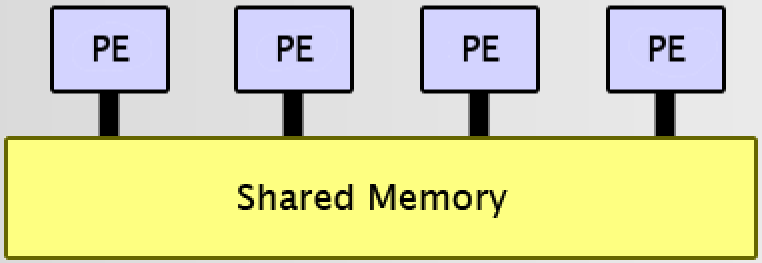
A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

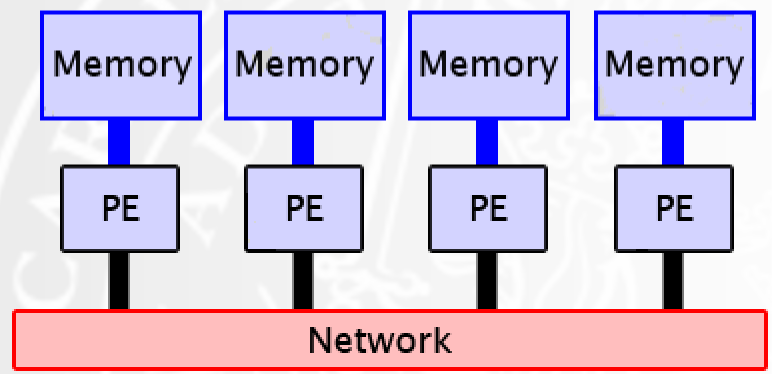
Leslie Lamport
email communication
1987

Distributed computations are concurrent programs in which processes communicate by message passing.

Gregory R. Andrews
“Paradigms for Process Interaction in Distributed Programs”
ACM Computing Surveys 23(1), 1991

concurrent programs





the trouble with distributed computing

"fallacies of distributed computing"

(1) The network is reliable.

(2) Latency is zero.

(3) Bandwidth is infinite.

(4) The network is secure.

(5) Topology doesn't change.

(6) There is one administrator.

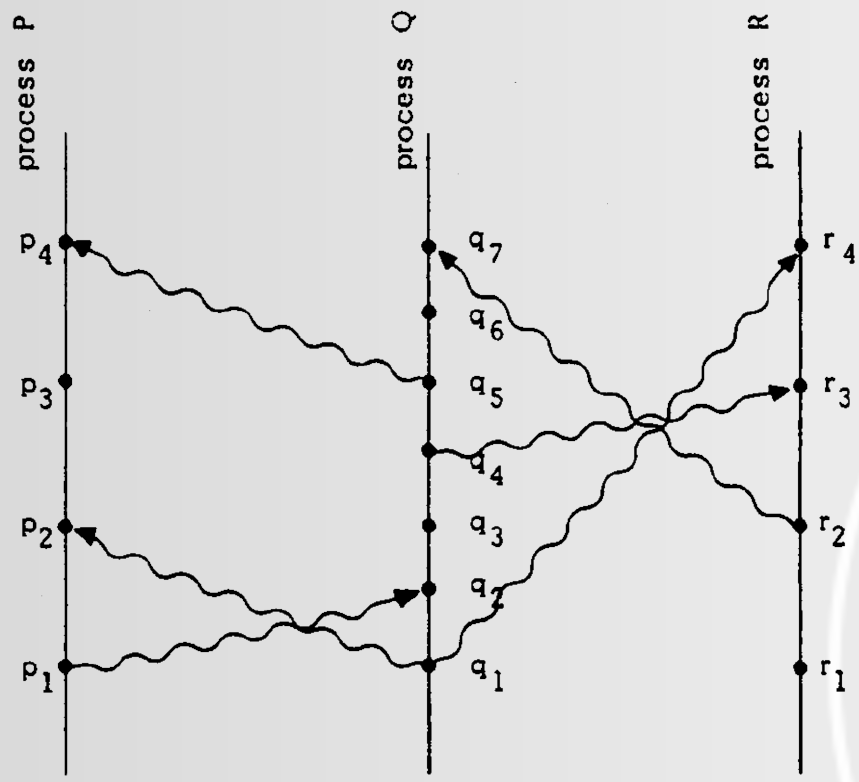
(7) Transport cost is zero.

(8) The network is homogeneous.

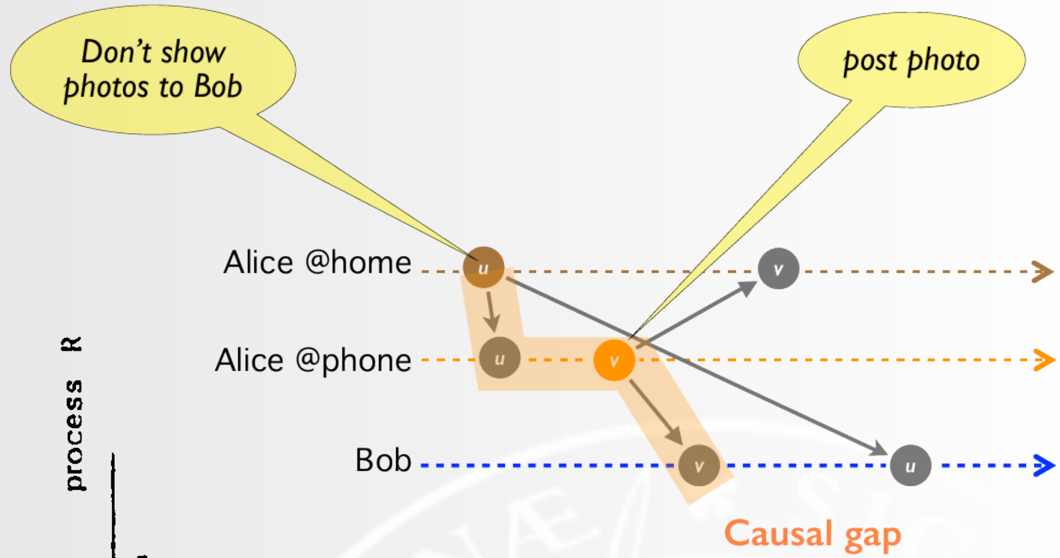
Many things can go wrong in a distributed system.

(1) How to detect that stuff went wrong?
(2) What to do about it?
(3) Can we characterize the *resiliency* of a system?

message passing



(source: Leslie Lamport)



(source: Marc Shapiro)

distributed computing

things being looked at

- the algorithm(s) of the processes
- the messages
- order, causality
- whether delivery is reliable
- whether processes crash (and how)
- whether processes are “nice”

things that usually aren't

- the nature of the interconnect
- time / speed [*]
- location of processes
- data formats

synchronous vs asynchronous (systems)

synchronous systems:
known upper bounds on time for computation and message delivery
or access to global clock
or execution in synchronized rounds

asynchronous systems:
no upper bounds on time for computation and message delivery

partially synchronous systems:
anything in between, e.g.

- *unknown* upper bounds on time for computation and message delivery
- almost-synchronized clocks
- bounded-drift local clocks
- approximate bounds (on execution/message delivery time)
- bound on message delay, bound on relative process speeds
- bound on the delay ratio between fastest and slowest message at any time (Θ-model)

ways in which things go wrong

failure classes (partial)

- crash-failstop
- crash-recover
- omission
- timing
- Byzantine

failure detector (example of distributed algorithm)

A **failure detector** is (part of) a process that determines whether other processes have failed.

Strong completeness
: Every faulty process is eventually permanently suspected by every non-faulty process.

Weak completeness
: Every faulty process is eventually permanently suspected by some non-faulty process.

Strong accuracy
: No process is suspected (by anybody) before it crashes.

Weak accuracy
: Some non-faulty process is never suspected.

Eventual strong accuracy
: After some initial period of confusion, no process is suspected before it crashes. This can be simplified to say that no non-faulty process is suspected after some time, since we can take end of the initial period of chaos as the time at which the last crash occurs.

Eventual weak accuracy
: After some initial period of confusion, some non-faulty process is never suspected.

source: James Aspnes, Yale

some kinds of distributed algorithms

- failure detectors
- consensus
- leader election
- synchronizers
- resource allocation, mutual exclusion

today's double feature

Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER
Yale University, New Haven, Connecticut

NANCY A. LYNCH
Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON
University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols–*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation–*parallelism*; H.2.4 [**Database Management**]: Systems–*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

1. *Introduction*

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many

Editing of this paper was performed by guest editor S. L. Graham. The Editor-in-Chief of JACM did not participate in the processing of the paper.

This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

Shubh: Lamport 1978 — Antonio: FLP 1985