Evolution of Distributed Computing Theory
From concurrency to networks and beyond

Michael J. Fischer
Yale University

August 20, 2008
1. The Computing World of the 1970’s
2. The Dawn of Distributed Computing
3. Characteristic Elements of Distributed Computing Theory
4. Conclusion
Background to Distributed Computing:

The Computing World of the 1970’s
How we computed in the 1970’s...
...how we wrote papers...
...and how we did research.
Theoretical computer science grew out of challenges faced by computer pioneers.

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Corresponding theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs and algorithms</td>
<td>• Recursive function/complexity theory</td>
</tr>
<tr>
<td>Computer hardware</td>
<td>• Switching and automata theory</td>
</tr>
<tr>
<td>Programming languages</td>
<td>• Language theory and formal semantics</td>
</tr>
<tr>
<td>Operating systems</td>
<td>• Concurrency theory</td>
</tr>
</tbody>
</table>

Computer networks were just beginning to be developed and hadn’t yet reached the radar screens of theoreticians.
Recursive function theory asks the questions:

- “What is an effective process?”
- “What is computable?”

Three themes.
Church’s Thesis: All “reasonable” models of effective computation are equivalent.

- Recursion equations
- Turing machines
- $\lambda$-calculus
Major theme 2 – negative ("lower bound") results

The halting set $H$ is the set of all programs that halt when fed their own descriptions as input.

**Theorem**

$H$ is undecidable.
**Major theme 3 – relative computability**

*Oracle Turing machine:* Can query an infinite database called an *oracle* during its computation.

Allows computability of different problems to be related.

Problem $X$ is *Turing reducible* to problem $Y$ ($X \leq_T Y$) if $X$ is decidable by a Turing machine with a $Y$ oracle.
Friedberg-Muchnik theorem [Fri57]

Theorem

There exist recursively enumerable sets $A$ and $B$ such that $A \not\leq_T B$ and $B \not\leq_T A$, but both are reducible to the halting set $H$.

Thus, there are pairs of incomparable undecidable sets that are both “less undecidable” than the halting problem.
Abstract complexity theory

Refines notion of computability.

Considers not only existence of a solution but also its computational complexity.

**Theorem (Lynch, Fischer, Meyer [LMF76])**

*There exist recursive sets A and B such that both are “hard” to compute, and both remain “hard” in the presence of the other, that is, A is still hard to compute by a Turing machine with access to a B oracle, and vice versa.*

Intuitively, they are hard for different reasons.
Dates back to Claude Shannon’s 1937 master’s thesis [Sha37].

Shannon applied Boolean algebra to the design of electromechanical relay switching circuits.

Later gave rise to the now-familiar Boolean circuit model of computation.
Automata theory

The notion of *state* is essential to computer hardware design.

A *finite automaton* is an abstraction of a state machine.

Kleene [Kle56] developed *regular expressions* to describe sets of sequences.

Rabin and Scott [RS59] made two major advances for which they won the Turing award:

- They introduced the notion of *nondeterministic computation*;
- They showed equivalence between regular expressions, deterministic finite automata, and nondeterministic finite automata.
Language theory

Outgrowth of work on programming languages, compilers, and natural language processing.

Considered various formalisms for describing language syntax:

- Regular languages;
- Context-free languages;
- Context-sensitive languages;
- Finite automata;
- Pushdown automata;
- Linear-bounded automata.
Language theory (cont.)

Positive results:
- Finite automata accept regular languages.
- Pushdown automata accept context-free languages.
- Linear bounded automata accept context-sensitive languages.

Negative results:
- Pushdown automata are more powerful than finite automata.
- Linear bounded automata are more powerful than pushdown automata.
Formal semantics

**Goal:** Give rigorous definition to the meaning of programs.

**Approaches:**

- *Operational semantics:* Relate program construct to actions of an abstract machine model.

- *Denotational semantics:* Map programs to mathematical objects which they denote.

- *Axiomatic semantics:* Define meaning of programs by using formal logic to characterize their properties.

In each case, attempt is to give a finite characterization of the potentially infinite process described by the program itself.
Early operating systems

How were early computers used?

- User takes over entire machine. Requires manual intervention to switch from one task to the next.

- **Batch processing.** User’s program takes over most of the machine. Resident “monitor” program transfers control to next task (like present-day cluster computers).

- **Multiprogramming.** Much like modern virtual machines – multiple tasks share memory and CPU. Goal: Keep expensive CPU busy when one task is I/O bound.

- **Time-sharing.** Interactivemultiprogramming. Goal: Give each user the illusion of her own machine. Requires “fair” scheduling.
Process abstraction and pseudo-concurrency

Desire to run multiple tasks on a single computer led to the development of the \textit{process} abstraction.

Each process operates as if it were the only program running on the system.

On a uniprocessor system, there is no true concurrency among processes – steps of different processes are \textit{interleaved}.

Provides an illusion of concurrency called \textit{pseudo-concurrency}.
True concurrency

*True concurrency* exists when multiple hardware devices operate in parallel. Multiple CPUs were rare in early computers, but true concurrency existed at the hardware level between processor and I/O channel, for example.

Not always obvious how to model concurrent actions on shared physical devices.

Assuming *atomicity* of primitive actions sidesteps the problem and allows the interleaved execution model to be used.

This is not always appropriate in real-life situations.
The Dawn of Distributed Computing
Concurrent processes must coordinate their activities to avoid interference.

The *mutual exclusion problem* is to provide exclusive access to a *critical region* of code.

Often solved in early OS’s by disabling interrupts before entrance to the critical section and re-enabling them afterwards.

(This does not work with true concurrency and led to expensive refactoring of operating systems in the 1990’s in order to support multiple processors.)
Dijkstra’s paper (1965)

Dijkstra encountered the mutual exclusion problem while designing the *THE multiprogramming system* [Dij68]. He formulated it abstractly, along with a solution and informal proof of correctness [Dij65]:

- Interprocess communication: *atomic reads and writes* to shared memory.
- *Asynchronous processes.*
- Solution tolerates stopping faults outside of the critical region.
- Solution avoids deadlock.

Starvation possible.
Dijkstra’s paper sparked much interest.

- Hyman [Hym66] proposed a “simplification” for the 2-processor case.
- Knuth [Knu66] gave a counter example to Hyman’s solution, showed that Dijkstra’s solution was subject to starvation, and introduced the notion of “fairness”.
- de Bruijn [dB67] improved Knuth’s fairness bound from exponential to quadratic in the number of processors.
- Eisenberg and McGuire [EM72] improved the fairness bound to linear.
My introduction to concurrency with Albert Meyer (1971)

Entry Code

\[ W_i = 1 \quad ("i \text{ wants resource}"). \quad p \text{ points to the selected process.} \]
The idea:

- Shared variable $p$ points to the currently enabled process.
- Each process when leaving its critical section chooses its successor and sets $p$ to point to it.
- $W_p$ should always be true when there are active processes waiting to enter their critical sections.
- The “$W_p$” tests allow the first active process to initialize $p$ to point to itself.
**Subtle bug:** Testing “$W_p$” is not atomic; requires two steps:

1. Fetch $p$ and store value in local temporary $p'$.
2. Fetch $W_{p'}$ and test value read.

If $p$ and $W_{p'}$ change value between steps, the wrong result can be obtained.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$W_1$</th>
<th>$W_2$</th>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(in critical section)</td>
<td>fetches $p$ ($p' = 1$)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>selects $P_2$ as successor</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>sets $W_1 = 0$ and exits</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>
The bug was discovered by Irene Greif four years later when I presented the unpublished algorithm in her concurrency seminar at the University of Washington.

I learned that concurrent algorithms were hard to understand and harder still to get right!

Gary Peterson began working on the problem with me and coauthored my first paper in distributed computing [PF77].
I mentioned this vexing mutual exclusion problem to Nancy Lynch.

She got interested in it and invited me to spend a week to work on it with her and her students at Georgia Tech in January 1977.

It was a most productive week!

The first result from the new collaboration was a paper on the shared memory space required for mutual exclusion (Burns, Fischer, Jackson, Lynch, Peterson [BFJ+78, BJL+82]).
Influence of Leslie Lamport

Two early results of Leslie Lamport greatly influenced my thinking about distributed computing and set the initial direction of the field:

- The “bakery” mutual exclusion algorithm [Lam74].
- The Byzantine Generals problem [PSL80, LSP82].
The “bakery” algorithm [Lam74]

The bakery algorithm contains several innovations:

- It was the first *distributed* solution to mutual exclusion; no central shared hardware. Communication is via 1-writer \( n \)-reader shared registers.

- Registers are *nonatomic* – reads can overlap writes and return arbitrary values.

- Tolerates *stopping faults*. Assumes a special value eventually gets written to memory after a process stops. (Precursor to the idea of a failure detector.)

- Satisfies strong “first-come, first-served” *fairness property*.
The Byzantine agreement problem [PSL80, LSP82]

- Showed majority consensus of 3 processes cannot protect against a single component failure, contrary to the belief at that time.
- Formulated the *consensus problem* and the notion of a Byzantine fault.
- Proved a $3f + 1$ lower bound on the number of processors needed to tolerate $f$ faults.
- Presented the first solution to the consensus problem that tolerated a Byzantine fault.
- The consensus problem became one of the cornerstones of distributed computing research and remains so to this day.
The first PODC conference [POD82] was held in Ottawa in 1982 and was organized by Robert Probert, Nicola Santoro and myself. 30 papers were presented.

Topics ranged from parallel algorithms, concurrency control in database systems, communication mechanisms, real-time systems, semantics of concurrency, and more.

PODC has been held every year since and has come to define both the area of distributed computing theory and the associated community of scientists.

This year’s PODC is number 27.
Characteristic Elements of Distributed Computing Theory
A theory of distributed computing requires making precise many notions that do not arise in the study of sequential computation or do not generalize in obvious ways.

- Communication mechanisms
- Synchrony, nondeterminism, and timing
- Scheduling and fairness
- Autonomy, reliability, and fault-tolerance
Shared memory versus message-passing systems

Processes naturally communicate via atomic multireader multiwriter shared registers in uniprocessor operating systems.

Lamport challenged both the multiwriter and the atomicity assumptions in his bakery algorithm paper [Lam74] but still assumed a shared register model of communication.

*Networks were only beginning to be developed*, and people were not so familiar with message-passing systems.

We now recognize shared memory and message-passing systems as distinct communication models with different formal properties, both worthy of study.
Lynch and Fischer [LF79, LF81] present a general model for describing the behavior and implementation of distributed systems.

- Defines abstract notions of behavior and implementation of a behavior by a system.
- Processes interact through shared registers using atomic test-and-set operations.
- Has two primitive entities – processes and registers.
- Lacks the ability to model direct process interaction.
I/O automata

Lynch and Tuttle [Tut87, LT87] present the I/O automaton framework.

- One type of entity models both process and communication mechanism.
- One type of action – the joint interaction of two entities.
- Subsumes both shared memory models and message-passing models.

I/O automata have been widely influential.
Synchrony

Early shared memory computers often used a global clock and were synchronous. This led to the synchronous message-passing models. Computation takes place in 2-part rounds:

1. Each process simultaneously sends a message to each other process.
2. Each process simultaneously receives all of the messages sent to it in the first part of the round.

This model was used in the early study of the Byzantine Generals problem.
Asynchrony

Parallel systems lacking a global clock are *asynchronous*. Variability in process step times is expected and often unknown.

Modeling asynchrony is tricky. One wants to make no timing assumptions yet assume all processes eventually progress.

Asynchronous execution is often modeled by interleaved sequences of process steps, subject to an overall *fairness condition* to ensure progress.
Time complexity of asynchronous systems

The notion of measuring time in terms of total process step count fails for asynchronous systems, where no upper bound exists on the number of steps that one process can take before another process takes a step.
Two equivalent time complexity measures

1. **Bounded real time measure:** (Peterson, Fischer [PF77]) Successive steps of an execution are labeled by increasing real numbers (representing clock times when the step occurs), constrained by bounding maximum delay between successive steps of the same process.
   - The time complexity is label of last step.

2. **Rounds measure:** (Arjomandi, Fischer, Lynch [AFL81, AFL83]) An interleaved execution is broken into consecutive minimal “rounds” such that each round contains at least one step of each process.
   - The time complexity is the number of rounds.
The order in which processes take steps is determined by a scheduler.

A fairness condition constrains what the scheduler is allowed to do.

Fairness is often a property of infinite schedules, e.g., that every process must be scheduled infinitely many times.

Infinite sequences greatly complicate formal reasoning.

Questions: What is a scheduler? How does a fair schedule arise?
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Token circulation

Consider an oriented ring of 3 or more processors, two of which contain a token. A token advances to the next node whenever the node containing it takes a step.
Question: Will the two tokens ever meet?

- No, in the presence of an adversarial scheduler who can alternately schedule the red and blue tokens forever.
- Yes, if the processes are “self-scheduled” so the choice of when a process takes a step is “independent” of the location of the other token.

Which reflects “reality”? 

Answer determines appropriate fairness condition.
Local fairness

*Local fairness* is when every action that is enabled infinitely often is taken infinitely often.

The scheduler that alternates between red and blue tokens satisfies local fairness since every token moves infinitely many times.

Many impossibility results exploit an adversarial scheduler constrained only by local fairness.
Global fairness

*Global fairness* (Angluin et al. [AAD⁺04, AAD⁺06]) requires the scheduler to be fair with respect to the global context. Every action that is enabled infinitely often *in a given context* must be taken infinitely often *in that context*.

The alternating schedule that prevents tokens from colliding is not globally fair since the closer token is never scheduled.

A simple randomized scheduler that chooses the next action at random generates a globally fair schedule with probability 1. Using it, the tokens will eventually collide with probability 1.
Differences between parallel and distributed systems

An early concern in the PODC community was how to distinguish distributed computing from parallel computing.

The models are similar but the outlook is not.

*Parallel computing* is primarily concerned with how to solve large problems faster.

*Distributed computing* is primarily concerned with how to deal with uncertainty:

- in scheduling and timing;
- in the correctness and reliability of system components;
- in the intentions and behaviors of other processors.
Setting the stage

Dijkstra’s mutual exclusion problem set the stage for dealing with uncertainty in scheduling.

Lamport’s Byzantine Generals problem set the stage for dealing with failures and malicious behavior.

Lamport, Lynch and many others developed formal machinery for proving the correctness of distributed algorithms and for proving impossibility results.
Conclusion

1. Distributed computing theory grew from the attempt to extend sequential theory of computation to encompass parallelism and concurrency.
2. The goal of finding a single universal model of distributed computing akin to the Turing machine has remained elusive.
3. Early work focused on two problems: mutual exclusion and consensus.
4. The difficulty of reasoning correctly about distributed systems motivated the development of formal models and proof techniques.
5. We’re not done yet!
Congratulations, Nancy, on 30+ years of distributed computing research.

We look forward to another productive 30 years!
Thank you!

~ finis ~
[AAD⁺04] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta.  
Computation in networks of passively mobile finite-state sensors.  

[AAD⁺06] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta.  
Computation in networks of passively mobile finite-state sensors.  
A difference in efficiency between synchronous and asynchronous systems.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch.
Efficiency of synchronous versus asynchronous distributed systems.
Shared data requirements for implementation of mutual exclusion using a test-and-set primitive. 

Data requirements for implementation of $N$-process mutual exclusion using a single shared variable. 
<table>
<thead>
<tr>
<th>Reference</th>
<th>Author</th>
<th>Title</th>
<th>Journal</th>
<th>Volume Issue Pages</th>
<th>Year</th>
</tr>
</thead>
</table>


Representation of events in nerve nets and finite automata.  

[Knu66] Donald E. Knuth.  
Additional comments on a problem in concurrent programming control.  

[Lam74] Leslie Lamport.  
A new solution of Dijkstra’s concurrent programming problem.  


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
</table>
Chairman-Robert L. Probert and Chairman-Michael J. Fischer and Chairman-Nicola Santoro.

Reaching agreement in the presence of faults.

[RS59] M.O. Rabin and D. Scott.
Finite automata and their decision problems.
[Sha37] Claude E. Shannon.  
A symbolic analysis of relay and switching circuits.  

Hierarchical correctness proofs for distributed algorithms.  