CSE 393 Concurrent and Distributed Algorithms

Annie Liu

February 21, 2015

1 Concurrent and distributed algorithms

Concurrent and distributed algorithms are at the core of concurrent and distributed systems.

- Concurrent systems are systems that consist of multiple separate processing elements. These elements may be computers, processors, cores, threads, etc. They generally are for running the code or program.
- Distributed systems are concurrent systems that consist of also multiple separate storage elements. These elements may be disks, memories, buffer, etc. They generally are for holding the data or state.

The distinction can be conceptual, like in multi-player games: players can be physically and geographically distributed but connected to a server and seeing the shared state of the game, or physically together but communicate following certain rule and not seeing each other's state. Indeed a system can have both aspects at the same time, with some data shared, and some data private.

In general, it is helpful to think of these systems as multi-agent systems. What they need fundamentally, going beyond centralized sequential systems, is the ability to communicate and synchronize.

Application domains

There are many applications of concurrent and distributed systems, in many domains even after grouping by categories. We list them under three meta categories:

1. Real-world multi-agent applications, with physically separate processing units, and physically separate data storage. Examples are systems for enterprise operations, air traffic control, supply chains, as well as cloud computing services and data centers in general.
2. Applications for better quality of life, even if the original problem does not involve physically separate processing units or data storage. For example, applications may use parallelization, with multiple processing units, to increase performance, or use replication, with multiple storage, to enhance availability.

3. Application for ease of modeling, even if the original problem does not have quality-of-life measures. For example, if a number of tasks are independent, modeling them as separate parallel tasks as opposed to sequential tasks may help one to utilize their independence in correctness validation, even if parallelization is not needed.

Example problems

We give example problems that require concurrent or distributed algorithms. For each problem, the solutions need to answer three usual questions:

1. What are the algorithms? That is, what steps should the processes take?

2. Why are the algorithms correct? That is, what properties do they have, and how do they satisfy the requirements?

3. How efficient are the algorithms? That is, how much time, space, and other resource do the algorithms consume?

Some problems are impossible to solve, and in such cases, the usual questions are: what are the proofs, and what relaxed versions of the problems can be solved?

Mutual exclusion. The problem is that multiple processes need to access a shared resource, for example, a printer, and need to access it mutually exclusively, in what is called a critical section (CS), i.e., there can be at most one process in a critical section at a time. The problem exists both when the processes need to communicate using shared memory, and when they need to communicate by passing messages.

What should the processes do? For example, when should a process read or write what shared variables, or send or wait for what messages? Why do the processes satisfy mutual exclusion, and what other properties do they have? For example, is it acceptable that no process is ever in a CS? What efficiency can be achieved? For example, how long must a process wait before going into a CS, and how many variables or messages are needed?

Leader election. A collection of processes, each with a unique id, need to elect a single process as the leader, so that the leader process knows that it is the leader, and all other processes know that too. Assume that the processes are connected to form a ring, and each process can only pass messages to its two neighbors.

What algorithm to use? For example, which process should be elected as the leader, and how should the algorithm even start? Why is a solution correct, or even possible? For example, what if the processes were in an anonymous ring, that is, had no ids? How long does it take to elect a leader, and how many messages are needed? In addition, what are the best case, the worse case, and average case?

Prisoners problem. A prison has 100 prisoners, a ward, and a room with a light switch that can be turned on or off, initially off. The prisoners will be called into the room one at a time, and eventually all will have been called at least once. At any time, any prisoner can

tell the ward that all have been called; if it is correct, then all will be freed; otherwise, all will be killed. The prisoners have a few minutes to discuss what to do.

How can the prisoner get themselves all freed? In particular, the light switch gives them only 1-bit for communication. Why is the algorithm correct? Could they solve the problem if the initial state of the switch was unknown? How long do the prisoners need to wait to be all freed? They certainly want to achieve that as soon as possible.

Coordinated attack. Two generals overlooking enemies in a valley from two sides of the valley. Both know that, if both attack at the same time, they will win, but if only one attacks, they will loose. So one will only attack if one is sure that the other will attack at the same time. They can communicate by sending messengers, but messengers might be captured by enemies.

What can the generals do to ensure that they attack at the same time? If one general sends a messenger with the attack time of, say, 3am, will he attack at 3am? Suppose the other receives the message and sends a messenger with an acknowledge, will he attack at 3am? Is there anything that the generals can do to ensure that they attack at the same time? This problem is also known as the two generals problem. It is an instance of the general problem of distributed consensus.

Muddy children. Three children, some of whom have mud on their foreheads. Each one can see others but not oneself. Dad says "I see that some of you have mud on your forehead. Do you know whether you have mud on your forehead?" Dad asks once, no one answers; asks twice, no one answers; asks a third time, all three say "Yes I have".

What happened? You may solve this by starting thinking with the case that only one child has mud on the forehead. However, why did Dad need to say the first sentence? All children can see that for a fact. Solving this problem requires reasoning with knowledge and understanding common knowledge.

Challenges

We look at the challenges through four kinds of systems. Again, the distinction can be conceptual. Center to all of them are communication, coordination, and synchronization, often under uncertainty due to lack of knowledge.

Parallel systems. Parallelism arises from inherently independent computations, and clearly can be extremely helpful for improving performance. Communication is usually needed at the start and end of independent computations or subcomputations. The main challenges are tedious setup of multiple processing units for carrying out the independent computations or subcomputations, and coordinating them in combining and using the computation results, with some degree of fault-tolerance.

These computations were once mainly on multi-processor supercomputers. Now they are also on cluster machines and multi-core processors, including million-core supercomputers. These systems can be distributed, like in grid computing, and in cloud computing in general.

Concurrent systems. Concurrency arises from interdependent computations by multiple agents, often communicating through shared memory and thus easily messing up the state of each other. The key challenge is to ensure atomicity, i.e., an operation appearing to the rest of the system to occur instantaneously, and thus isolation, i.e, a transaction appearing to happen in isolation from other concurrent transactions, and serializability, i.e., the outcome of concurrent transactions equaling the outcome of the transactions executed serially.

Concurrent processes collaborating by reading and writing shared memory is well known to be a source of toughest errors—most difficult to debug and even to repeat the errors.

Distributed systems. Concurrent agents with separate storage must communicate by passing messages, leading to inherently asynchronous computations due to lack of shared memory in general and shared clock in particular: there are no bounds on processing speed or message delay, and no accurate failure detector. The key challenge is to provide fault-tolerance in the presence of slow or failed processes or message channels.

As implied by the coordinated attack problem described earlier, consensus is not achievable in asynchronous systems: because acknowledgment of message receipt can be lost just as the original message, a potentially infinite series of messages are required to come to consensus.

Multi-player games. Multi-agent systems can be viewed as multi-player games as studied in game theory, "the study of mathematical models of conflict and cooperation between intelligent rational decision-makers" [Mye91], with a broad range of applications such as economy, biology, as well as computer science. The key challenges are to build accurate mathematical models in which to study algorithms and strategies with exact cost measures and objective functions.

Concurrent and distributed systems correspond to collaborative games. Game theory has studied competitive games more than collaborative games. Competitive games are able to model situations to the finest details, producing accurate results, but much efforts have also been made to connect competitive games and collaborative games.

2 Models and basic issues

Communication is special in concurrent and distributed systems, as can be seen in multi-player games: contrasting single-player games that specify only allowed operations by the player, multi-player games specify also allowed communications among the players. We discuss two main communication models, also called paradigms, followed by basic issues and concepts.

Communication paradigms

The two main paradigms of communication are:

- Shared memory (SM)—reading and writing shared memory. This models concurrent but not distributed computations.

  This is generally realized in imperative programming languages that support multiple threads accessing shared memory. For example, Java, Python, and C# support threads as thread objects.

- Message passing (MP)—sending and receiving messages over a communication network. This models distributed computations.

  This is mostly realized in functional and flow-based languages and as libraries in languages where processes with no shared memory must communicate. For example, Erlang supports message passing, and many languages support Message Passing Interface (MPI) libraries.

These two communication paradigms can also be conceptual instead of physical:

- With distributed shared memory (DSM), as in language Linda with tuple space, physically separate memories can be addressed as one logically shared memory space.

- With message-passing concurrency, as in language Erlang, processes that have physically shared memory communicate by message passing, using the shared memory for message queues.

Order of events

In concurrent and distributed systems, multiple events can happen at the same time. The most important basic issue is to understand and control the order of events.

- In concurrent systems with shared memory, different threads access some shared state. A *race condition* occurs when the system behavior depends on the order of accesses by different threads. Uncontrolled accesses may lead to undesired system state. The solutions are to provide atomicity—using a range of hardware and software mechanisms, such as locks, test-and-set, mutual exclusion algorithms, and non-blocking algorithms—so that shared states are accessed mutually exclusively when needed.

- In distributed systems, different processes have no shared clock. Determining the order of events becomes a key issue. The solutions are to use logical clocks and clock synchronization algorithms, so that the order of events that matters to the system behavior can be determined. Furthermore, different processes have no shared state at all. Determining the global state and global properties of the system becomes a challenge. The solutions include snapshot algorithms and global property detection algorithms.

Locking mechanisms such as semaphores are well-covered in operating systems courses; we do not discuss them here. We discuss other mechanisms such as test-and-set, mutual exclusion algorithms, and non-blocking algorithms later. Logical clocks are not usually covered in common courses; Lamport [Lam78] nicely presents the first logical clock. Snapshot algorithms may use logical clocks or a well-known strategy called flooding; we discuss algorithms that use these later.

Failures and fault-tolerance

With multiple processing units interacting with each other, many kinds of failures are possible. We describe three main kinds:

Process failure. This mainly includes crash failure, meaning that a process failed by simply stopping, and it may or may not recover. This could be because that the process speed is too slow and thus treated as having crashed. A recovered process could be treated as a new process or a new incarnation of the old process.

Crash failure is common in concurrent and distributed systems, essentially so in distributed systems, and thus must be handled to provide fault-tolerance, especially in distributed systems.

Link failure. This mainly includes message loss in distributed systems, anywhere on the communication channel. This could be because that the channel is too slow, and thus the messages are treated as lost. A process suffering a crash failure can also be viewed as if all messages from that process are lost.

Message loss is common in distributed systems, and thus must be addressed by practical distributed algorithms to provide fault-tolerance.

Byzantine failure. This refers to when arbitrary bad things can happen. For example, a process or channel can misbehave, including sending wrong values and initiating security attack.

This does not happen as commonly as the two kinds above, is more costly to address, and thus is usually only handled at the boundary of distributed systems, not in algorithms that run common tasks inside the boundary of the systems.

There are many methods and techniques for handling failures and making systems fault-tolerant. We separate them into three main categories:

1. Fault isolation—determining and separating out parts of the system that are affected by faults, so that those parts can be fixed while unaffected parts can carry on their operation.

2. Checkpointing and message logging—storing intermediate system states and message histories, so that fixing of parts affected by faults can be faster by starting from the stored records.

3. Replication—creating replicated processes and storage and ensuring mutual consistency, so that failures will not affect the operation of the system.

To help detect possible error states and failures, algorithms have been developed for tasks such as the following: termination detection—detecting that all processes are passive and have finished the computations intended; deadlock detection—detecting that all processes are passive but have not finished the computations intended; failure detection—detecting that some processes have died or crashed leading to crash failures; and inconsistency detection—detecting that replicated processes and storage are not mutually consistent.

Correctness and efficiency

Correctness criteria. Concurrent and distributed algorithms generally have three main correctness criteria:

1. Safety—bad things will not happen. For the mutual exclusion example, this corresponds to mutual exclusion—at most one requesting process will be in CS at a time. Safety is the most basic requirement; all systems must satisfy safety.

2. Liveness—good things will happen to the system. For the mutual exclusion example, this corresponds to deadlock freedom—if some processes are requesting, some processes will be in CS. Liveness guarantees that the system can always make progress; it is not always possible to satisfy.

3. Fairness—good things will happen to each process. For the mutual exclusion example, this corresponds to starvation freedom—if a process is requesting, that process will be in CS. Fairness can also have different degrees; stronger fairness is harder to satisfy.

Efficiency measures. Efficiency measures for concurrent and distributed algorithms include the usual time and space complexities for each process but more importantly the overall throughput by the set of processes. Distributed algorithms also use two main new measures—round complexity and message complexity.

- Throughput is the number of requested operations that are performed by the system per unit of time.

- Round complexity measures the number of round of messages, in synchronous systems and loosely in asynchronous systems, an algorithm takes to perform a requested operation.

- Message complexity, also called communication complexity, measures the number of messages an algorithm sends to perform a requested operation.

For some problems, including in particular the distributed consensus problem, impossibility results have been well studied.

3 Bibliographical notes

There is a large literature, including many textbooks, on concurrent and distributed algorithms and systems.

Raynal's many books [Ray88, Ray10b, Ray10a, Ray13a, Ray13b] cover a large variety of concurrent and distributed algorithms; they also use the clearest pseudocode notations I found in algorithm textbooks.

Lynch's book [Lyn96] also covers a wide variety of concurrent and distributed algorithms, with formal descriptions using I/O automata (a state-machine model with Input and Output actions) for many of them, and with extensive correctness and complexity proofs.

Lamport's book [Lam02] describes rigorous specifications of algorithms and systems, especially concurrent and distributed systems, using TLA+ (a language for writing specifications based on the Temporal Logic of Actions).

For concurrent algorithms, Taubenfeld's book [Tau06] is an excellent source. Michael and Scott's article [MS98] nicely describes their well-known non-blocking and blocking concurrent queue algorithms.

For distributed algorithms, other good books include Tel [Tel00], Garg [Gar02], Attiya and Welch [AW04], Guerraoui and Rodrigues [GR06], and Fokkink [Fok13]. Wikipedia pages, `http://en.wikipedia.org/wiki/Distributed_computing` and `http://en.wikipedia.org/wiki/Distributed_algorithm`, provide some helpful overviews.

References

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley, 2nd edition, 2004.

[CR79] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[Fok13] Wan Fokkink. *Distributed Algorithms: An Intuitive Approach.* MIT Press, 2013.

[Gar02] Vijay K. Garg. *Elements of Distributed Computing.* Wiley, 2002.

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming.* Springer, 2006.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[MS98] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[Mye91] Roger B Myerson. Game theory: Analysis of conflict. *Harvard University Press*, 1991.

[Ray88] Michel Raynal. *Distributed Algorithms and Protocols*. Wiley, 1988.

[Ray10a] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan & Claypool, 2010.

[Ray10b] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Morgan & Claypool, 2010.

[Ray13a] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013.

[Ray13b] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[Tau06] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice Hall, 2006.

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.