2. Distributed Systems Models

Distributed systems

- Distributed systems are hard to design and understand because we lack intuition for them
  – Faults, time, several machines/processes
- We must develop an intuition, so that
  – We can design distributed systems that perform as we intend
  – We can understand existing distributed systems well enough for modification as needs change
- Most of what follows also applies for other areas: Physics, Chemistry, Economics,……
  – Important difference: most of these areas don’t build things

Developing an intuition

Two approaches are conventionally employed:

- **Experimental Observation**
  - We build things and observe how they behave in various settings
  - A body of experience accumulates
  - **Even if we do not understand why something works**, this body of experience enables us to build things for settings similar to those that have been studied
- **Modeling and Analysis**
  - We formulate a model by simplifying the object of study and postulating a set of rules to define its behavior
  - We then analyze the model and infer consequences
  - **If the model accurately characterizes reality**, then it becomes a powerful tool

Tension

- There is an inevitable tension between advocates for "experimental observation" and those for "modeling and analysis"
  - This tension masquerades as a dichotomy between "theory" and "practice"
  - Each side believes that theirs is the more effective way to refine intuition

Theory vs practice

Problems with "theory"

- Practitioners complain they learn little from theory
- A theoretician might simplify too much when defining a model; the analysis of such models will rarely enhance our intuition
- Without experimental observation, we have no basis for trusting our models

Problems with "practice"

- Theoreticians complain that practitioners are not addressing the right problems
- A practitioner might incorrectly generalize from experience or concentrate on the wrong attributes of an object; our intuition does not profit from this, either
- Without models, we have no hope of mastering the complexity that underlies distributed systems

Models are extremely important!

Models

- Basic idea: a *model* is a simplification of an object (= system)
  - Allows us to reason about it
- A ***model*** for an object is a collection of attributes and a set of rules that govern how these attributes interact
  - Also called a **theory**
- Two important facts:
  - There is no single correct model for an object
  - Answering different types of questions about an object usually requires different models

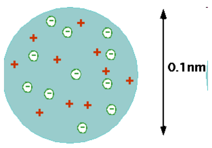
Example of distributed system model

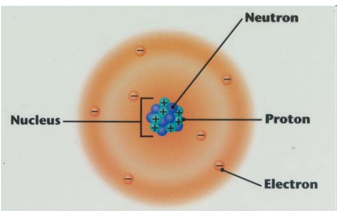
- Two processes, *A* and *B*, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent.

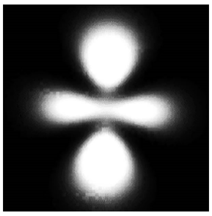
- Simplification, no single model, …

Models of the atom

- Billiard Ball Model (1803) - John Dalton viewed the atom as a small solid sphere.
- Plumb Pudding Model (1897) - Joseph John Thomson
- Solar System Model - Ernest Rutherford
- Electron Cloud Model (1920's) -







Model → Assumptions

- Two processes, *A* and *B*, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent.
- What are the **assumptions** above?
- Is this simple model useful?

DS Models

Good Models

- A model is *accurate* to the extent that analyzing it yields truths about the object of interest
- A model is *tractable* if such an analysis is actually possible
- Defining an accurate model is not difficult; defining an accurate and tractable model is
  - An accurate and tractable model will include **exactly** those attributes that affect the phenomena of interest
  - Selecting these attributes require taste and insight
  - Level of detail is a key issue

Good Models

- In building models for distributed systems, we typically seek answers to two fundamental questions:

1. **Feasibility**. What classes of problems can be solved?
   - Can head-off wasted effort in design, implementation, and testing
2. **Cost**. For those classes that can be solved, how expensive must the solution be?
   - Allows us to avoid designs requiring protocols that are inherently slow or expensive
   - Provides a yardstick with which we can evaluate any solution that we devise

Distributed systems

- Computer science students use models about processes
  - E.g. sequential, uniform memory access
- Distributed systems -> multiple processes communicating over (narrow bandwidth, high-latency) channels, with some faulty processes and channels
  - Additional processes provide more computational power, but require coordination
  - Channel bandwidth limitations mean that inter-process communication is a scarce system resource
- In short: distributed systems raise different concerns and understanding these requires different models

Two main aspects in a distributed system model

1. **Time**: synchronous, asynchronous, …
2. **Failure**: crash, arbitrary, …

- Why are these 2 aspects important?

Synchronous *vs* Asynchronous Systems

- Asynchronous system: we make no assumptions about process execution speeds and/or message delivery delays
- Synchronous system: we do make assumptions about these parameters.
  - The relative speeds of processes is assumed to be bounded
  - The delays associated with communications channels also
- Postulating that a system is asynchronous is a **non-assumption**: every system is asynchronous
  - This is a compelling argument for studying asynchronous systems. **Why?**

Synchronous *vs* Asynchronous Systems

- Postulating that a system is synchronous constrains how processes and communication channels are implemented
  - Scheduler that multiplexes processors must not violate the constraints on process execution speeds
  - This implies that all processors in the system have access to approximately rate-synchronized real-time clocks
  - Queuing delays, unpredictable routings, and retransmission due to errors must not violate the constraints on channel delays
- In asserting that a system is synchronous, we rule out certain system behaviors
  - This enables us to employ simpler or cheaper protocols than would be required to solve the same problem in an asynchronous system
- An example is the following election problem

An Election Protocol

- A set of processes *P1*, *P2*, ..., *Pn* must select a leader
  - Each process *Pi* has a unique identifier *uid*(*i*)
  - Devise a protocol in which all processes learn the identity of the leader
  - Assume all processes start executing at the same time
  - All communicate using broadcasts that are reliable
- With an asynchronous system model it is possible, but somewhat expensive, to solve:
  - Each process *Pi* broadcasts ⟨*i*, *uid*(*i*)⟩
  - Every process will eventually receive these broadcasts, so each can independently "elect" the *Pi* for which *uid*(*i*) is smallest
  - *n* broadcasts are required for an election

An Election Protocol

- What if:
  - It is assumed that processes can crash?
- What can we do about it?
  - Think about synchrony assumptions…

An Election Protocol

- A set of processes *P1*, *P2*, ..., *Pn* must select a leader
  - Each process *Pi* has a unique identifier *uid*(*i*)
  - Devise a protocol in which all processes learn the identity of the leader
  - Assume all processes start executing at the same time
  - All communicate using broadcasts that are reliable
  - Processes can crash
- With a synchronous system model it is possible to solve it more efficiently:
  - Let τ be a known constant bigger than the largest message delivery delay plus the largest difference that can be observed at any instant by reading clocks at two arbitrary processes
  - Each process *Pi* waits until
    - (i) it receives a broadcast or
    - (ii) $\tau * uid(i)$ seconds elapse on its clock at which time it broadcasts ⟨*i*⟩.
  - *(only 1* broadcast is required for an election)
  - Notion of *communication by time*

Synchronous *vs* Asynchronous Systems

There are other models in the spectrum:

- Partial synchrony
- Timed-asynchronous
- Wormholes
- …

Partial Synchrony

- Partially synchronous system model
  - Processes have clocks but they are not synchronized
  - Initially, the system is asynchronous (no bounds on communications and computations)
  - After some *GST* (Global Stabilization Time) time (which is unknown to the process), the communications and computations delays become bounded (to unknown values) forever

- NOTES:
  - This model defines systems that tend to have unstable and stable periods
  - In practice, the bounds on communication and computation delays must hold until the distributed algorithm finishes its execution
  - This is considered a very realistic system model (Why?)

Failure Models

- A variety of failure models have been proposed in connection with distributed systems
- All are based on assigning responsibility for faulty behavior to the system's components
  - Processors
  - Communication channels
- We count faulty components, not occurrences of faulty behavior
  - In classical work on fault-tolerant computing systems, it is the occurrences of faulty behavior that are counted
- We speak of a system being *t*-fault tolerant when that system will continue satisfying its specification provided that no more than *t* of its components are faulty
  - We also use *f* instead of *t*

Example failure models

- **Failstop**. A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed is detectable by other processors
- **Crash**. A processor fails by halting. Once it halts, the processor remains in that state
  - Unless the system is synchronous, it is not possible to distinguish between a very slow processor and one that has halted due to a crash failure
- **Crash+Link**. A processor fails by halting. Once it halts, the processor remains in that state. A link fails by losing some messages, but does not delay, duplicate, or corrupt messages
- **Byzantine Failures**. A processor fails by exhibiting arbitrary behavior

- Important: these are **models** => **assumptions**

Example distributed system model

- Model in plain English:
  - Two processes, *A* and *B*, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent.

- Model in a more rigorous form:
  - System = two processes A and B
  - Communication model: message passing, bidirectional channel
  - Failure model: processes do not fail; communication can have omissions (lose messages)
  - Synchrony model: asynchronous (no statements about time)

Which Model When?

- Theoreticians have good reason to study all of the models we have discussed
  - Each idealizes some dimension of real systems, and it is useful to know how each system attribute affects the feasibility or cost of solving a problem
- Theoreticians also may have reasons to define new models

Which Model When?

- Dilemma faced by practitioners: deciding between models when building a system
  - Assume that processes are asynchronous or synchronous, failstop or Byzantine?
- One way to regard a model is as an interface definition
  - i.e., a set of assumptions that programmers can make about the behavior of system components
  - Programs are written to work correctly assuming the actual system behaves as prescribed by the model
  - When the system behavior is not consistent with the assumed model, then no guarantees can be made
- Examples:
  - Assume Byzantine: safe, but expensive solutions
  - Assume crash: cheaper solutions (but is it safe?)
  - Assume synchronous: expensive to enforce constraints (and is it safe?)

Which Model When?

- Systems are not constructed as a single monolithic entity
  - Rather, a system is structured by implementing abstractions
  - Each abstraction builds on other abstractions, providing some added functionality, providing **a new system model**
- Example:
  - Physical communication channels corrupt packets
  - Using CRCs (Cyclic Redundancy Checks) we have an abstraction of channels that make omissions (but not corruptions)
  - Using retransmissions we have an abstraction of channels that are reliable
  - What is the model in each case?
  - Which of the models is ***stronger*** (makes stronger assumptions that require more effort to satisfy)?

Which Model When?

- Models are limiting cases: the behavior of a real system is bounded by our models
- Understanding the feasibility and costs associated with solving problems in these models can give us insight into the feasibility and cost of solving a problem in some given real system whose behavior lies between the models

Assumptions and Coverage

Coverage

- Models => assumptions
  - Wrong assumptions…
- **Coverage**
  - Given a fault in the system, the coverage is the probability that it will be tolerated
  - Think of coverage of 1, 0, 0.9, 0.99, etc.
- Objective is to minimize the probability and number of failures
- Therefore, an important part of the equation is: **assumption coverage**

Assumption coverage

There is an important separation of concerns to be made in the design of a system:

- Environmental assumptions
  - The assumptions concerning the behavior of the environment where the system will run (infrastructure, networks, hardware, etc.), namely its faulty behavior
- Operational assumptions
  - The assumptions concerning the behavior of the system itself, or how the system will run (programs, algorithms, protocols, etc.), under a given set of environmental assumptions

Assumption coverage

- Environmental assumption coverage (Pre)
  - Conditional probability of a set of assumptions (H) holding, given any occurrence of a fault
  - Examples: clock rate of drift, network datagram delivery delay, omission error degree, number of component failures
- Operational assumption coverage (Pro)
  - Probability that a given algorithm (A) solves a problem, given the assumed set of environmental assumptions H
  - In fault-tolerant algorithms, this denotes the coverage of the error processing mechanisms
- **Total coverage = Pro x Pre**

Assumption coverage

- **Total coverage = Pro x Pre**
- If the algorithm and its implementation are proven correct, we expect a coverage Pro = 1
- Pre is always an upper bound on total coverage
- What happens
  - With wrong assumptions about faults or synchrony?
  - With the wrong assumption that the algorithm is correct and correctly implemented?