State Machine Concept

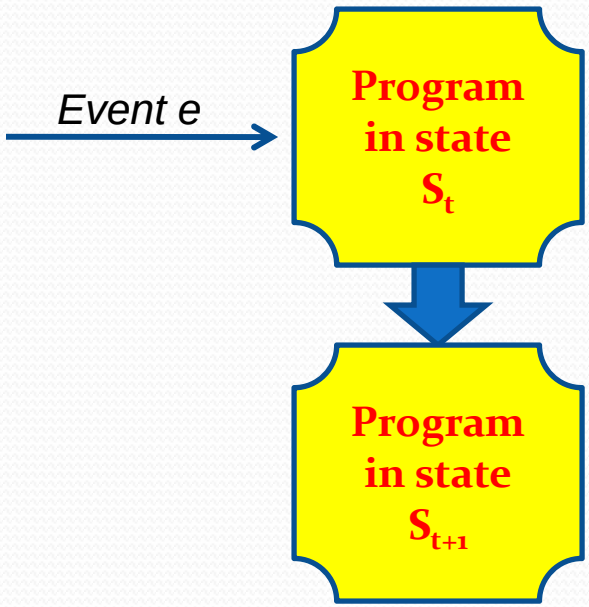
Ken Birman

Cornell University. CS5410 Fall 2008.

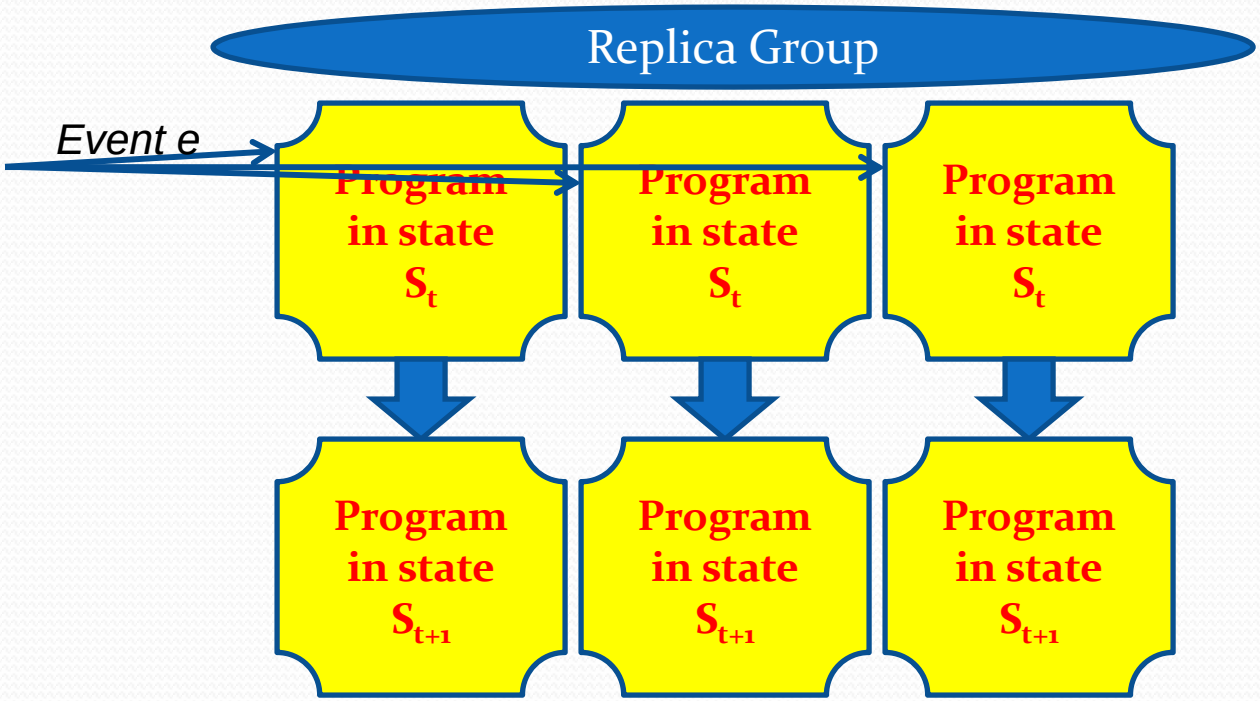
State Machines: History

- Idea was first proposed by Leslie Lamport in 1970's
- Builds on notion of a finite-state automaton
  - We model the program of interest as a black box with inputs such as timer events and messages
  - Assume that the program is completely deterministic
- Our goal is to replicate the program for fault-tolerance
  - So: make multiple copies of the state machine
  - Then design a protocol that, for each event, replicates the event and delivers it in the same order to each copy
  - The copies advance through time in synchrony

State Machine



State Machine



A simple fault-tolerance concept

- We replace a single entity P with a set
- Now our set can tolerate faults that would have caused P to stop providing service
  - Generally, thinking of hardware faults
  - Software faults might impact all replicas in lock-step!

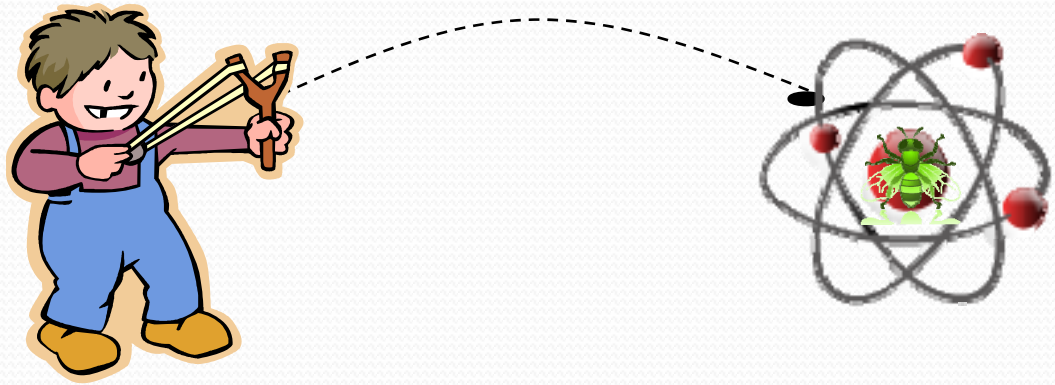
- Side discussion:
  - *Why do applications fail? Hardware? Software?*

(Sidebar) Why do systems fail?

- A topic studied by many researchers
  - They basically concluded that bugs are the big issue
  - Even the best software, coded with cleanroom techniques, will exhibit significant bug rates
  - Hardware an issue too, of course!
- Sources of bugs?
  - Poor coding, inadequate testing
  - Vague specifications, including confusing documentation that was misunderstood when someone had to extend a pre-existing system
  - Bohrbugs and Heisenbugs

(Sidebar) Why do systems fail?

- Bohrbug:
  - Term reminds us of Bohr's model of the nucleus:
    - A solid little nugget
  - If you persist, you'll manage to track it down
    - Like a binary search



(Sidebar) Why do systems fail?

- Heisenbug:
  - Term reminds us of Heisenberg's model of the nucleus:
    - A wave function: can't know both location and momentum
  - Every time you try to test the program, the test seems to change its behavior
  - Often occurs when the "bug" is really a symptom of some much earlier problem

Most studies?

- Early systems dominated by Bohrbugs
- Mature systems show a mix
  - Many problems *introduced* by attempts to fix other bugs
  - Persistent bugs usually of Heisenbug variety
  - Over long periods, upgrading environment can often destabilize a legacy system that worked perfectly well
- Cloud scenario
  - “Rare” hardware and environmental events are actually very common in huge data centers

Determinism assumption

- State machine replication is
  - Easy to understand
  - Relatively easy to implement
  - Used in a CORBA "fault-tolerance" standard
- But there are a number of awkward assumptions
- Determinism is the first of these

- Question: How deterministic is a modern application, coded in a language such as Java?

Sources of non-determinism

- Threads and thread scheduling (parallelism)
- Precise time when an interrupt is delivered, or when user input will be processed
- Values read from system clock, or other kinds of operating system managed resources (like process status data, CPU load, etc)
- If multiple messages arrive on multiple input sockets, the order in which they will be seen by the process
- When the garbage collector happens to run
- “Constants” like my IP address, or port numbers assigned to my sockets by the operating system

Non-determinism explains Heisenbug problems

- Many Heisenbugs are just vanilla bugs, but
  - They occur early in the execution
  - And they damage some data structure
- The application won't touch that structure until much later, when some non-deterministic thing happens
- But then it will crash
  - So the crash symptoms vary from run to run
  - People on the "sustaining support" team tend to try and fix the symptoms and often won't understand code well enough to understand the true cause

(Sidebar) Life of a program

- Coded by a wizard who really understood the logic
  - But she moved to other projects before finishing
  - Handed off to Q/A
- Q/A did a reasonable job, but worked with inadequate test suite so coverage was spotty
  - For example, never tested clocks that move backwards in time, or TCP connections that break when both ends are actually still healthy
- In field, such events DO occur, but attempts to fix them just added complexity and more bugs!

Overcoming non-determinism

- One option: disallow non-determinism
  - This is what Lamport did, and what CORBA does too
  - But how realistic is it?

  - Worry: what if something you use "encapsulates" a non-deterministic behavior, unbeknownst to you?
  - Modern development styles: big applications created from black box components with agreed interfaces
    - We lack a "test" for determinism!

Overcoming non-determinism

- Another option: each time something non-deterministic is about to happen, turn it into an event
- For example, suppose that we want to read the system clock
  - If we simply read it, every replica gets different result
  - But if we read *one* clock and replicate the value, they see the same result
- Trickier: how about thread scheduling?
  - With multicore hardware, the machine itself isn't deterministic!

More issues

- For input from the network, or devices, we need some kind of relay mechanism
  - Something that reads the network, or the device
  - Then passes the events to the group of replicas

- The relay mechanism itself won't be fault-tolerant: should this worry us?
  - For example, if we want to relay something typed by a user, it starts at a *single* place (his keyboard)

Implementing event replication

- One option is to use a protocol like the Oracle protocol used in our GMS
  - This would be tolerant of *crash failures* and *network faults*
  - The Oracle is basically an example of a State Machine
  - Performance should be ok, but will limited by RTT between the replicas

Byzantine Agreement

- Lamport's focus: applications that are compromised by an attacker
  - Like a virus: the attacker somehow "takes over" one of the copies
  - His goal: ensure that the group of replicas can make progress even if some limited number of replicas fail in arbitrary ways – they can lie, cheat, steal…
  - This entails building what is called a "Byzantine Broadcast Primitive" and then using it to deliver events

Questions to ask

- When would Byzantine State Replication be desired?
- How costly does it need to be?
  - Lamport's protocol was pretty costly
  - Modern protocols are much faster but remain quite expensive when compared with the cheapest alternatives
- Are we solving the right problem?
  - Gets back to issues of determinism and "relaying" events
  - Both seem like very difficult restrictions to accept without question – later, we'll see that we don't even need to do so

Another question

- Suppose that we take *n* replicas and they give us an extremely reliable state machine
  - It won't be faster than 1 copy because the replicas behave identically (in fact, it will be slower)
  - But perhaps we can have 1 replica back up n-1 others?
  - Or we might even have everyone do 1/n'th of the work and also back up someone else, so that we get n times the performance
  - In modern cloud computing systems, performance and scalability are usually more important than tolerating insider attacks

Functionality that can be expressed with a state machine

- Core role of the state machine: put events into some order
  - Events come in concurrently
  - The replicas apply the events in an agreed order
- So the natural match is with order-based functions
  - Locking: lock requests / lock grants
  - Parameter values and system configuration
  - Membership information (as in the Oracle)
- Generalizes to a notion of “role delegation”

Core functionality

- Anything that can be expressed in terms of an event that gets "applied" to the state and causes a new state
  - Locking: events are lock requests/release
  - Parameter changes: events are new values
  - Membership changes: events are join/failure
  - Security actions: events change permissions, create new actors or withdraw existing roles
  - DNS: events change <name><ip> mappings
- In fact the list is very long. Reminds us of "active directory" or "dynamic DNS" (aka "Network Info Svc")

Fancier uses

- Castro and Liskov use a state machine to "manage" files actually stored in an offline store
  - They call this Practical Byzantine Replication
- The state machine tracks which copies are current and who has them: a small amount of meta-data
  - And they use Byzantine Agreement for this
- The actual file contents are *not* passed through the state machine, so it isn't on the critical path

Role Delegation

- New concept for a very sophisticated way of thinking about state machine replication
- Starts with our GMS perspective of state machine as an append-oriented log
- Then (like we did) treats this as a set of logs, and then as a set of logs spread over a hierarchy of state machines

Role Delegation

- Now think about this scenario:
  - Initially, the "lock" for the printer resided at the root
  - Then we moved it to cs.cornell.edu
  - Later we added a sub-lock for the printer cartridge

- Notice similarity to human concept of handing a role to a person:
  - John, you'll be in charge of the printer
  - [John]: OK, then Sally, I want you to handle the color ink levels in the cartridge

Role Delegation

- We can formalize this concept of role delegation
  - Won't do so in cs5410
- Basic outline
  - Think of the log as a "variable"
  - Work with pairs: one has values and one tracks the owner of the log. Appending to the ownership log lets us transfer ownership to someone else
  - Think of decisions as functions that are computed over these variables

Role Delegation

- In this way of thinking, we can understand our GMS as a big role delegation and decision-making tool
- It can handle any decision that occurs in a state machine where all the needed variables are local
- But it can't handle decisions that require "one shot" access to variables split over multiple GMS services

Example?

- Suppose the FBI handles all issues relating to agents. Mulder and Scully work at the FBI

Cornell handles all issues relating to campus access

- After reading a Daily Sun article ("Zombies Kill Six Near Bell Tower"), Mulder and Scully leap on a plane

Humans vs Zombies

Our State Machine Challenge

- Should Cornell give Mulder access to student records?
- Think of this as a computer science question…

Grant Access?

- Issue is a multi-part decision
  - Are Mulder and Scully legitimate FBI agents?
  - Is this a real investigation?
  - What are Cornell policies for FBI access to student records?
  - Are those policies “superceded” by the Zombie outbreak?
- Very likely decision requires multiple sub-decisions, some by FBI.gov and some by Cornell.edu, in their respective GMS services!

Options

- Break decision into parts
  - Issue: what if outcome leaves some form of changed state behind (a side-effect)
  - Until we know the set of outcomes, we don't know if we should update the state
- Collect data at one place
  - But where? FBI won't transfer all its data to Cornell, nor will Cornell transfer data to FBI!

Can't always solve such problems

- If a decision splits nicely into separate ones, sure…
- … but many don't

- If a decision requires one-shot access to everything in one place, we need a kind of database transaction
  - Allows atomicity for multi-operation actions
  - Would need to add these functions to our GMS and doing so isn't trivial

Performance worries

- Last in our series of “yes, but” warnings
- Recall that with a GMS, we send certain kinds of decisions to the GMS and it reports results back
- This means that decision making is “remote”
  - May sound minor, but has surprisingly big costs
  - Especially big issue if load becomes high

Summary

- State machine concept is very powerful
- But it has limits, too
  - Requires determinism, which many applications lack
  - Can split application (GMS) up using role delegation, but functions need to be disjoint
- Scalability
  - If one action sometimes requires sub-actions by multiple GMS role holders, we would need transactions
  - But due to indirection, and nature of protocol, state machines are also fairly slow