Course Overview

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: *Computing Systems and Concurrency*
Lecture 2

Marco Canini

Philosophy and Recurring Themes

- Keep it real! This is the real world:
  - Things break. Components fail.
  - Latency matters. Can’t beat speed of light.
  - Certain things are impossible. Need work arounds.
- How do we build systems that work at **very large scale** and **tolerate failures**?
- Given systems span many nodes, how do we enable different nodes to **agree** on “things” (e.g., time, order of operations, state of the system)?

Learning Objectives

- Reasoning about concurrency
- Reasoning about failure
- Reasoning about performance

- Building systems that correctly handle concurrency and failure

- Knowing specific system designs and design components

Course Goals

- Gain an understanding of the principles and techniques behind the design of modern, reliable, and high-performance systems
- In particular learn about distributed systems
  - Learn general systems principles (modularity, layering, naming, security, ...)
  - Practice implementing real, larger systems that must run in nasty environment
- One consequence: Must pass exams and projects independently as well as in total
  - Note, if you fail either you will not pass the class

Keep the Big Picture in Mind

- Course: many topics, grouped around key areas
- Might feel like lectures are disconnected…
- ... and first need to cover some background

- **Big Picture:**
  - real systems have complex requirements that span the concepts of multiple topics
  - E.g., we want fault tolerance, consistency and scalability

Course Organization

http://sands.kaust.edu.sa/classes/CS240/F21/

Learning the material: People

- Lecture
  - Professor Marco Canini
  - Slides available on course website
  - Office hours: right after lecture or by appointment
- TAs
  - Arnaud Dethise: M 10:00-11:30, 1-4409-WS18
- Main Q&A forum: www.campuswire.com
  - No anonymous (to instructors) posts or questions
  - Can send private messages to instructors

Learning the Material: Books

- Lecture notes!
- No required textbooks
- References on website available in the Library:
  - Programming reference:
    - *The Go Programming Language.* Alan Donovan and Brian Kernighan
  - Topic reference:
    - *Distributed Systems: Principles and Paradigms.* Andrew S. Tanenbaum and Maaten Van Steen
    - *Guide to Reliable Distributed Systems.* Kenneth Birman

Grading

- Four programming assignments (50% total)
  - 10% each for 1 & 2
  - 15% each for 3 & 4

- Two exams (50% total)
  - Midterm exam on October 6 (15%)
  - Final exam during exam period (35%)

Exams

- Test learning objectives mostly using designs covered in lectures
- And test knowledge of specific design patterns and designs
- Open book (but if you don’t study it will create time pressure)
- Recipe for success:
  - Attend lecture and actively think through problems
  - Ask questions during lecture and afterwards in my office hours
  - Actively work through problems
  - Complete programming assignments
  - Study lecture materials for specific design patterns and designs
  - Run the system designs in your mind and see what happens

About Assignments

- Systems programming somewhat different from what you might have done before
  - Low-level (C / Go)
  - Often designed to run indefinitely (error handling must be rock solid)
  - Must be secure - horrible environment
  - Concurrency
  - Interfaces specified by documented protocols
- TAs' Office Hours
- Read: Dave Andersen's "Software Engineering for System Hackers"
  - Practical techniques designed to save you time & pain

Why use Go?

- Easy concurrency w/ goroutines (lightweight threads)
- Garbage collection and memory safety
- Libraries provide easy RPC
- Channels for communication between goroutines

Where is Go used?

- Google, of course!
- Docker (container management)
- CloudFlare (Content delivery Network)
- Digital Ocean (Virtual Machine hosting)
- Dropbox (Cloud storage/file sharing)
- … and many more!

About Assignments

- Reinforce / demonstrate all learning objectives!

- 1: Sequential Map/Reduce (due September 16)
- 2: Distributed Map/Reduce (due September 23)
- 3-1: Raft Leader Election (due November 16)
- 3-2: Raft Log Consensus (due December 2)
- 4: Key-Value Storage Service (due December 9)

Programming Assignments

- Recipe for disaster
  - Start day assignment is due
  - Write code first, think later
  - Test doesn’t pass => randomly flip some bits
  - Assume you know what program is doing

Programming Assignments

- Recipe for success
  - Start early (weeks early)
  - Think through a complete design
  - Progressively build out your design (using tests to help)
  - Checkpoint progress in git (and to gitlab) frequently
  - Debug, debug, debug
    - Verify program state is what you expect (print it out!)
    - Write your own smaller test cases
    - Reconsider your complete design
  - Attend office hours

Policies: Collaboration

- Working together important
  - Discuss course material
  - Work on problem debugging
- Parts **must** be your own work
  - Midterm, final, programming assignments
- What we hate to say: we run cheat checkers… **they work surprisingly well**
- Please *do not* put code on *public* repositories

Policies: Write Your Own Code

Programming is an individual creative process. At first, discussions with friends is fine. When writing code, however, the program must be your own work.

Do not copy another person's programs, comments, README description, or any part of submitted assignment. This includes character-by-character transliteration but also derivative works. Cannot use another's code, etc. even while "citing" them.

Writing code for use by another or using another's code is academic fraud in context of coursework.

Do not publish your code e.g., on Github, during/after course!

Policies: Late Work

- 72 late hours to use throughout the semester
  - (but not beyond December 9)
- After that, each additional day late will incur a 10% lateness penalty
  - (1 min late counts as 1 day late)
- Submissions late by 3 days or more will no longer be accepted
  - (Fri and Sat count as days)
- In case of illness or extraordinary circumstance (e.g., emergency), talk to us early!

Summary

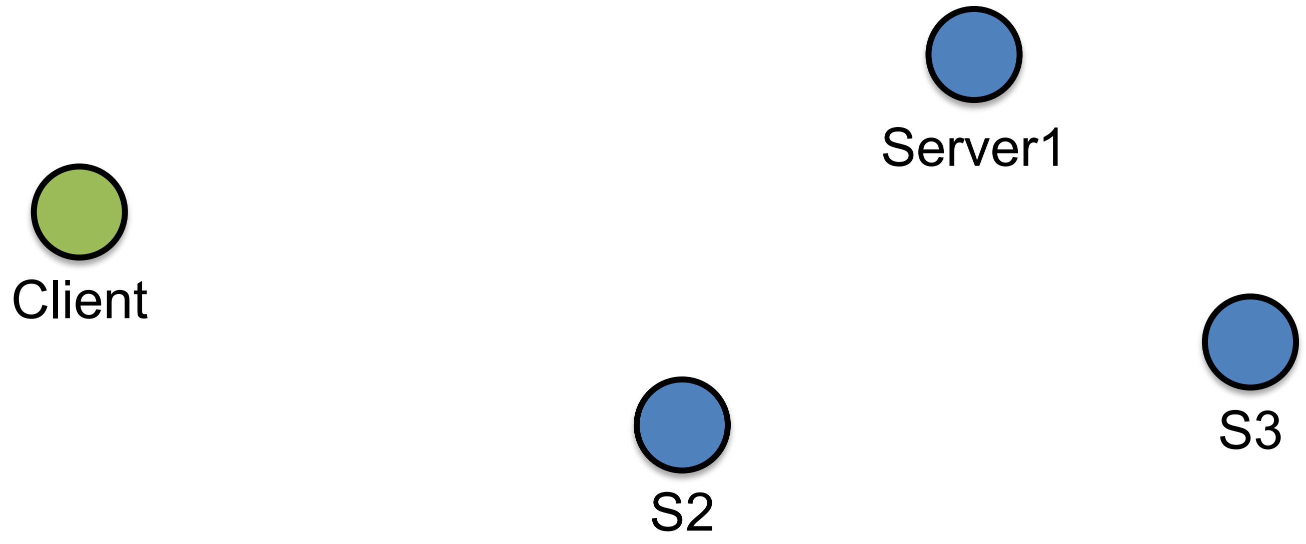
- Attend lecture, attend labs, think actively!
- Start programming assignments early, use the right strategy!

And for some technical bits today …

Distributed Systems and Correctness

Example

- Assume a distributed storage
  - Clients can read and write files



System model

- *N* **processes** $p_1,\ldots,p_N$ in the system (no process failures)
  - Every process executes an algorithm
    - An automation with set of states, set of inputs, set of outputs and a state transition function $S \times I \rightarrow S \times O$

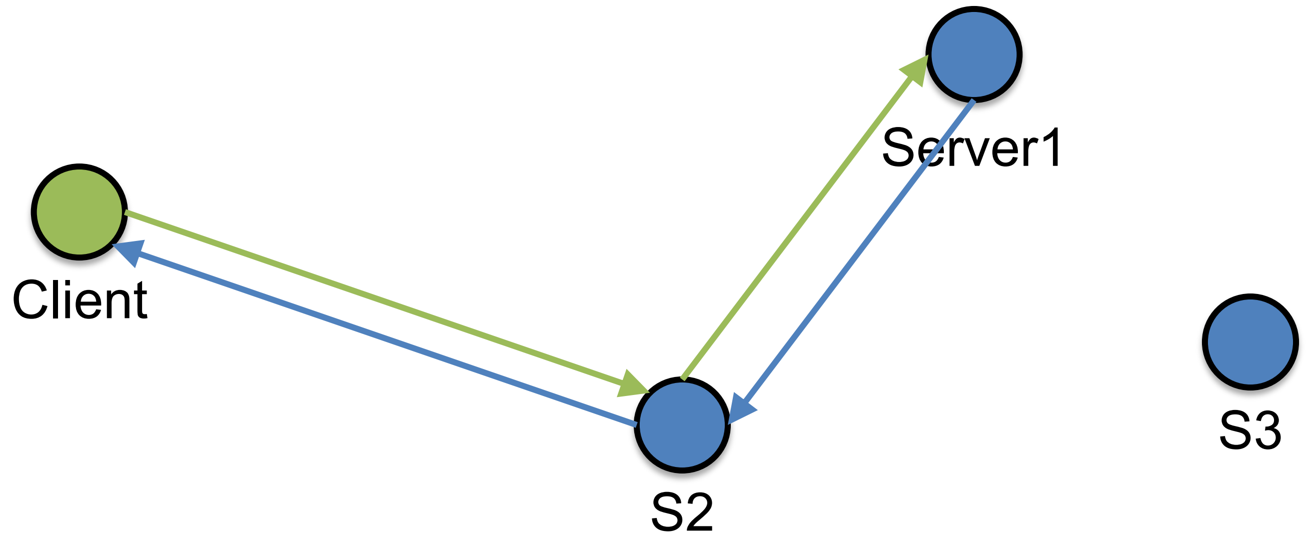
- There are two first-in, first-out, unidirectional **channels** between every process pair p_i and p_j
  - Call them **channel(p_i, p_j)** and **channel(p_j, p_i)**
  - All messages sent on channels arrive intact and in order
  - Channel cannot duplicate, create or modify messages

System model

- Message passing
- No failures (for now)
- Two possible timing assumptions
  1. Synchronous System
  2. Asynchronous System
     - No upper bound on **message delivery**
     - No bound on relative **process speeds**

Example execution

- Assume a distributed storage
  - Clients can read and write files

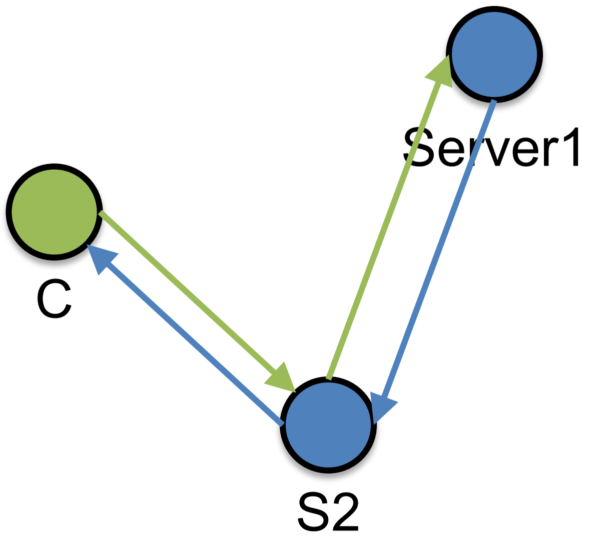


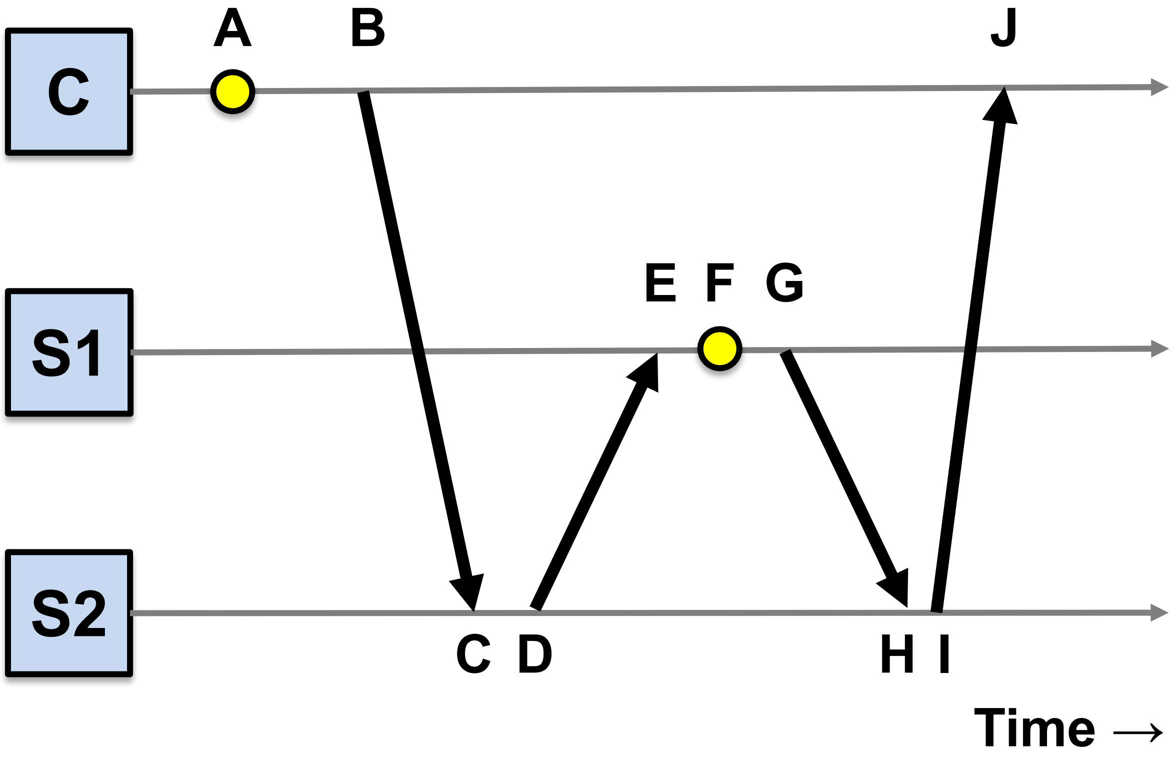
Execution of the system

- Processes execute sequences of events
  - events can be of 3 types: local, send or receive
- An execution (or run) is a sequence of events that respect the system-wide distributed algorithm
  - each process is consistent with the local sequences
  - a message is sent by a process only if its (local) algorithm prescribes it to do it given the preceding sequence of its inputs
  - every received message was previously sent, and no message is received twice

Space-Time diagrams

- A graphic representation of distributed execution





Common failure assumption

- Generally, a failure occurs when a process deviates from the algorithm assigned to it
- A process is *correct* if it never fails

- *crash* failure: the faulty process prematurely stops taking steps of its algorithm

- A typical assumption is that, in every possible execution out of N processes, at most $f < N$ can be faulty

- We call such a system f-*resilient*

Safety and liveness properties

Reasoning about fault tolerance

- This is hard!
  - How do we design fault-tolerant systems?
  - How do we know if we're successful?
- Often use "properties" that hold true for every possible execution
- We focus on **safety** and **liveness** properties

Properties

- **Property**: a predicate that is evaluated over a run of the system
  - “every message that is received was previously sent”
- Not everything you may want to say about a system is a property:
  - “the program sends an average of 50 messages in a run”

Safety properties

- “Bad things” don’t happen, ever
  - No more than k processes are simultaneously in the critical section
  - Messages that are delivered are delivered in causal order
- A safety property is “prefix closed”:
  - if it holds in a run, it holds in every prefix

Liveness properties

- “Good things” eventually happen
  - A process that wishes to enter the critical section eventually does so
  - Some message is eventually delivered
  - Eventual consistency: if a value doesn’t change, two servers will eventually agree on its value
- Every run can be extended to satisfy a liveness property
  - If it does not hold in a prefix of a run, it does not mean it may not hold eventually

Often a trade-off

- “Good” and “bad” are application-specific
- Safety is very important in banking transactions
  - May take some time to confirm a transaction
- Liveness is very important in social networking sites
  - See updates right away

Conclusion

- Attend lecture, attend labs, think actively!
- Start programming assignments early, use the right strategy!
- Super cool distributed systems stuff starts Monday!