Introduction to Concurrency

CS 475, Spring 2018
Concurrent & Distributed Systems
Today

• Distributed & Concurrent Systems: high level overview and key concepts

• Relevant links:
  • Syllabus: http://www.jonbell.net/gmu-cs-475-spring-2018/
Course Topics

• This course will teach you **how** and **why** to build distributed systems

• Distributed System is “a collection of independent computers that appears to its users as a single coherent system”

• This course will give you theoretical knowledge of the tradeoffs that you’ll face when building distributed systems
Course Topics

How do I run multiple things at once on my computer?
Concurrency, first half of course

How do I run a big task across many computers?
Distributed Systems, second half of course
Layers

- From hardware
- To OS
- To programming languages
- To networks
- To libraries and middleware
- To developers
Grading

• 50% Homework
  • 5 assignments, ~2 weeks to do each, all done individually
  • Your code will be autograded; you can resubmit an unlimited number of times until the deadline and view your score
  • Also graded by hand for some non-functional issues

• 10% Quizes
  • Pass/fail (Pass if you are in class and submit a quiz, fail if you don’t)
  • Use laptop or phone to complete the quiz in class

• 15% Midterm Exam, 20% Final Exam
Policies

• My promises to you:
• Quiz results will be available instantaneously in class; we will discuss quiz in real time
• Homework will be graded within 3 days of submission
• Exams will be graded within a week
Policies

• Lateness on homework:
  • 10% penalty if submitted UP TO 24 hours after deadline
  • No assignments will be accepted more than 24 hours late
  • Out of fairness: **no exceptions**

• Attendance & Quizzes:
  • You can miss up to 3 with no penalty
  • Again, out of fairness: **no exceptions** beyond this
Honor Code

• Refresh yourself of the department honor code

• Homeworks are 100% individual
  • Discussing assignments at high level: ok, sharing code: not ok
  • If in doubt, ask the instructor
  • If you copy code, we WILL notice (see some of my recent research results in “code relatives”)

• Quizes must be completed by you, and while in class
Course Staff

- Prof Jonathan Bell (me)
  - Office hour: ENGR 4422 Mon & Weds 2:15-3:00 pm or by appointment
  - Areas of research: Software Engineering, Program Analysis, Software Systems

Two hobbies: cycling, ice cream
Course Staff

• GTA: Arda Gumusalan
  • Office Hours: TBA
• UTA: Thanh Luu
  • Office Hours: TBA
• Please, no emails to instructor or TAs about the class: use Piazza
Readings

• Bad news: no single book
• Good news: several free e-books are great references
  • Operating Systems: Three Easy Pieces (Arpaci-Dusseau and Arpaci-Dusseau) http://pages.cs.wisc.edu/~remzi/OSTEP/
Concurrency

- **Goal**: do multiple things, at once, coordinated, on one computer
  - Update UI
  - Fetch data
  - Respond to network requests
  - Improve responsiveness, scalability
- **Recurring problems**:  
  - Coordination: what is shared, when, and how?
Abstractions

• Goal: take something complicated, make it “easy”
• Operating Systems
  • From CPUs and memory to processes and threads
• Distributed Systems
  • From collections of computers to coherent applications
Concurrency & Parallelism

4 different things: T1  T2  T3  T4

Concurrency: (1 processor)

Parallelism: (2 processors)

Time

Processes

• Def: A process is an instance of a running program
• Process provides each program with two key abstractions
  • Logical control flow
    • Each program seems to have exclusive use of the CPU.
  • Private address space
    • Each program seems to have exclusive use of main memory.
• How are these illusions maintained?
  • Process executions interleaved (multitasking)
  • Address spaces managed by virtual memory system
Processes

```java
public class Sample {
    static int i;
    public static void main(String[] args) {
        int k = 10;
        foo(k);
    }
    public static void foo(int in) {
        bar(in);
    }
    public static void bar(int in) {
        i = in;
        System.out.println("bar");
    }
}
```

Active Stack Frame

- stack
  - Sample.main
    - args, k
  - Sample.foo
    - in
  - Sample.bar
    - in, i
- code
- heap data
- files
Threads

• Traditional processes created and managed by the OS kernel
• Process creation expensive - fork system call in UNIX
• Context switching expensive
• Cooperating processes - no need for memory protection (separate address spaces)
Coordination Problems

- Two threads call `increment()` at the same time
- What is the value of `i` afterwards?

```java
static int i = 0;
public static void increment()
{
    i = i + 1;
}
```

Spoiler alert: not guaranteed to be 2
Processes vs Threads

Single-Threaded Process

Multi-Threaded Process
“All non-trivial abstractions, to some degree, are leaky.”

Joel Spolsky

Leaky Abstractions

- Completely hiding the underlying complexity is never possible, usually not desirable
- Example: our first two abstractions (concurrency) - process and thread
Processes vs Threads

• Context Switching

  • Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).

  • When switching processes, all of that data needs to get flushed out (by the OS)

• Threads share the same address space: no need to do this switch
Processes vs Threads

- Although more expensive to switch, OS provides isolation between processes

Only can communicate via specifically exposed memory (private by default)
Processes vs Threads

- Although more expensive to switch, processes OS provides isolation between processes.

All heap data is shared between threads

Multi-Threaded Process

- code
- heap data
- files
- stack
- stack
- stack
Processes vs Threads

• Example: browsers launching tabs in their own process
More Abstractions

• Process + Thread -> one computer
• How can we abstract many computers working together?
• What does that even look like?
Distributed Systems

Model:
Many servers talking through cloud
Distributed Systems

Model: Servers and Clients talking through cloud
Distributed Systems

Model:
Many clients talking through cloud
Distributed Systems

Model:
Two clients talking through cloud
Why expand to distributed systems?

• Scalability
• Performance
• Latency
• Availability
• Fault Tolerance

“Distributed Systems for Fun and Profit”, Takada
Distributed Systems Goals

- **Scalability**
- Performance
- Latency
- Availability
- Fault Tolerance

“the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.”

“Distributed Systems for Fun and Profit”, Takada
Distributed Systems Goals

• Scalability
• **Performance**
• Latency
• Availability
• Fault Tolerance

“is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.”
Distributed Systems Goals

• Scalability
• Performance
• **Latency**
• Availability
• Fault Tolerance

“*The state of being latent; delay, a period between the initiation of something and the it becoming visible.*”
Distributed Systems Goals

- Scalability
- Performance
- Latency
- **Availability**
- Fault Tolerance

"the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable."

Availability = uptime / (uptime + downtime).

Often measured in “nines”

<table>
<thead>
<tr>
<th>Availability %</th>
<th>Downtime/year</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>&gt;1 month</td>
</tr>
<tr>
<td>99%</td>
<td>&lt; 4 days</td>
</tr>
<tr>
<td>99.9%</td>
<td>&lt; 9 hours</td>
</tr>
<tr>
<td>99.99%</td>
<td>&lt;1 hour</td>
</tr>
<tr>
<td>99.999%</td>
<td>5 minutes</td>
</tr>
<tr>
<td>99.9999%</td>
<td>31 seconds</td>
</tr>
</tbody>
</table>
Distributed Systems Goals

- Scalability
- Performance
- Latency
- Availability
- Fault Tolerance

“What kind of faults?

- Disks fail
- Power supplies fail
- Networking fails
- Security breached
- Power goes out
- Datacenter goes offline

“ability of a system to behave in a well-defined manner once faults occur”
More machines, more problems

- Say there’s a 1% chance of having some hardware failure occur to a machine (power supply burns out, hard disk crashes, etc)
- Now I have 10 machines
  - Probability(at least one fails) = 1 - Probability(no machine fails) = 1-(1-.01)^{10} = 10%
- 100 machines -> 63%
- 200 machines -> 87%
- So obviously just adding more machines doesn’t solve fault tolerance
More machines, more problems

• PLUS, the network may be:
  • Unreliable
  • Insecure
  • Slow
  • Expensive
  • Limited
Constraints

- Number of nodes
- Distance between nodes
Constraints

- Number of nodes
- Distance between nodes

Even if cross-city links are fast and cheap (are they?)
Still that pesky speed of light…
Recurring Solution #1: Partitioning

All accesses go to single server
Recurring Solution #1: Partitioning

- Divide data up in some (hopefully logical) way
- Makes it easier to process data concurrently (cheaper reads)

Each server has 50% of data, limits amount of processing per server.

Even if 1 server goes down, still have 50% of the data online.
Recurring Solution #2: Replication

All accesses go to single server
Recurring Solution #2: Replication

Entire data set is copied
Recurring Solution #2: Replication

- Improves performance:
  - Client load can be evenly shared between servers
  - Reduces latency: can place copies of data nearer to clients
- Improves availability:
  - One replica fails, still can serve all requests from other replicas
Partitioning + Replication
Partitioning + Replication
**Partitioning + Replication**

The diagram illustrates the concept of partitioning and replication in a database system. Each location (DC, NYC, SF, London) is represented with a set of servers and data partitions.

- **DC**: Shows the initial state with servers and data partitions labeled with ranges [0...100], [A...N], [101...200], [O...Z].
- **NYC**: Shows a replicated state with the same data distribution as DC.
- **SF**: Demonstrates another replicated state with similar data distribution.
- **London**: Shows a third replicated state with the same data distribution.

The diagram emphasizes the use of partitioning to distribute data across different locations and the advantage of replication for data availability and fault tolerance.
Recurring Problem: Replication

- Replication solves some problems, but creates a huge new one: consistency

Set A=5

“OK”!

Read A

“6”!

OK, we obviously need to actually do something here to replicate the data… but what?
How much to hide?

- Completely hiding how distributed a system is may be too much:
  - Communication latencies can't be hidden (pesky speed of light!)
  - Completely hiding failures is **impossible** (we will prove this later in the semester)
    - Can never distinguish a slow computer from one that is crashed
  - Hiding more adds performance costs
Exit-ticket activity

Go to socrative.com and select “Student Login” (works well on laptop, tablet or phone)

Class: CS475
ID is your @gmu.edu email