EECS 345 Distributed Systems

To do ...

- Distributed systems
- This class
- Next: A brief introduction to networking
What is a **distributed system**?

A set of independent, interconnected processors that communicate and coordinate their action by exchanging messages

… appears to its users as a single coherent system: DNS, ‘The Web’, BitTorrent, FB, Twitter, Google, …

- This means …
  - Concurrent execution is the norm
  - No global clock
  - Independent failures
Why do you want one?

- To connect/organize physically separate people or entities
- ... get fault tolerance via replication
- ... scale up throughput via parallel CPUs/memory/disks/networks
- ... achieve security via isolation
But they hard to build (right)

- Making resources available/accessible
- Security – comes with sharing
- Heterogeneity – networks, HW, OS, programming languages, …
- Scalability in numbers and geographic span
- Asynchrony
- Failures detecting, masking/recovering …
- Concurrency and consistency
- False assumptions*
  - The network is reliable, secure, and homogeneous; topology does not change
  - Latency is zero, bandwidth is infinite, transport cost is zero
  - There is one administrator

*Based on Peter Detusch (1994) and James Gosling (1997)
Why taken this course?

- Interesting; hard problems, powerful solutions
- Used by real systems
- Active research area
- As usual at NU, a serious systems component
Place in the curriculum

- **213 – Systems programming**
  - How to use the OS interface
  - How to be a better systems programmer

- **343 – Operating systems**
  - OS abstractions and implementations
  - How to make a single computer work reliable, efficiently

- **340 – Networking**
  - How to connect computers together
  - Networks can be seen as distributed systems

- **345 …**
  - How to make multiple computers work reliable, efficiently despite failures of components
Logistics/organization

- Course website
  - Slides, pointers to piazza, canvas, papers …
- Piazza
  - Main Q&A forum
- Canvas
  - Projects descriptions
  - Homework assignments
  - Grades
Course components

- Lectures
- Reading
- Projects
- Homework assignments
- Final exam
Lectures

- Big ideas
  - Slides available in the course website

- Paper discussions
  - Read the papers *before* class
  - Participate!

- Lab introductions (TA sessions)
Reading

- Not great book in this area
  - One listed as reference

- A series of papers
  - Some classic, some new
  - Starting with Keshav’s “how to read a paper” (under Materials’ tab)
Projects

- Goals
  - Deeper understanding of some important ideas
  - Experience with distributed systems

- Using Go
  - A language well suited for DS
  - To ensure the hard problems have to do with DS (e.g., no compiler, libraries, or pointers)

- Four projects
  - On MapReduce and achieving consensus

- Projects done in teams of 2-3 students (not 1)
On projects – Code walkthrough

- For every project, code walkthrough by randomly selected groups with the staff

  To get full credit for your project you must be able to carry the walkthrough, showing you understand your own code

- A few points
  - Sampling with replacement, i.e., you may do it multiple times
  - To be done in first week after submission
  - All team member must be there, everyone drives at one point
Homework assignments

- 4 problem sets
- To be done individually
- Short answer questions
Exam

- In-class final
  - One-page of notes (1 or 2-sides)
  - Papers from reading

- Example questions
  - With asynchronous RPC, a client is blocked until its request has been accepted by the server. To what extent do failures affect the semantic of asynchronous RPC?
On cheating/plagiarism

- Collaboration is good, cheating is a very serious offense

- It's OK to
  - Meet with colleagues
  - Study for exams together
  - Discuss assignments with them

- But, *what you turn in must be your own work*
  - Do not copy code, solution sets, etc. from other people or any other sources (*Web included!*)
  - Do not make your code available for other (even future) students (*e.g., do not post in github, bitbucket, …*)
On class participation

- A Wong-Baker’s like scale

  - Contributes to the discussion (A)
  - Seem to follow the discussion (B)
  - I’ve seen her in class (C)
  - She’s in class, but probably on FB (D)
  - Is he actually in this class? (F)
Main topics covered

- Abstractions that hide distribution from applications
- Three big classes of abstractions
  - Storage
  - Communication
  - Computation
- A few topics come up repeatedly
  - Performance – $N_x$ servers $\rightarrow N_x$ performance
  - Fault tolerance – Design for failure
  - Consistency – Fault tolerance | consistency | performance
Go on, take 5'
MapReduce as a case study

“… a programming model and associated implementation for processing and generating large datasets”

– Google’s Jeff Dean and Sanjay Ghemawat, 2004 OSDI paper

- Several implementations, including Hadoop
- A good illustration of the course main topics
- And the focus of your first two projects
MapReduce motivation

- Many special purpose computations
  - Conceptually straightforward, but
  - running over large input that has to be distributed over 1000s of machines
  - e.g. crawled documents, web requests, …

- Challenges: how to parallelize it, distribute the data, handle failures, …
  - Large complex code just to handle this
Main contribution

- A new abstraction to let you express the simple computation and hide the messy details
- Programmer defines *Map* and *Reduce* functions
  - Computation takes a set of input key/value pairs and produces a set of output key/value pairs
  - Sequential code; often fairly simple
- MapReduce runs the functions
  - On 1000s of machines, over large inputs, hiding details of distribution
Programming model

- **Map** takes input pair, outputs a set of intermediate key/value pairs
- MapReduce library groups all intermediate values with the same intermediate key $I$ and passes them to **Reduce**
- **Reduce** takes an intermediate key $I$ and a set of values and merges those values
- **Types**
  - Map $(k1, v1) \rightarrow \text{list}(k2, v2)$
  - Reduce $(k2, \text{list}(v2)) \rightarrow \text{list}(v2)$
Example – word count

- Input: a large collection of documents
- Output: list of words with counts
- Pseudocode

```java
Map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

Reduce(String key, Iterator values):
    // key: word
    // values: list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```
Partition data into splits; start copies of the program: one master, many servers
- $M$ map tasks, $R$ reduce tasks
- Master picks idle workers, assigns an $M$ or $R$ task

A map worker, reads, parses k/v pairs, passes each to Map function
- Intermediate k/v pairs produced by Map are buffered in memory

Periodically, buffered pairs to local disk, partitioned into $R$ regions by partitioning function
- Locations on local disk sent to Master that forwards to Reduce workers
When reduce worker is told about locations, use RPC to read buffered data

- Reads all, sorts, iterates over and for each unique intermediate key, passes the key and set of intermediate values to Reduce
- Output of Reduce is appended to final output file

When all map and reduce tasks have completed, master wakes up user program, MapReduce returns back to user code
Output is in the $R$ output files
Hiding pesky and painful details

- Starting software on servers
- Tracking which tasks are done
- Moving data around
- Recovering from failures
Scalability

- MapReduce scales well – $N$ computers, $N^x$ throughput
  - Assuming $M$ and $R$ are $\geq N$, i.e., enough files to run $Maps()$ in parallel, same for $Reduce()$
  - Only interaction is via shuffle between phases

- So more throughput with more computers
  - Cheaper than special-purpose efficient parallelism
  - Computers are cheaper than programmers
Limiting performance

- The thing to optimize
  - CPU? memory? disk? network?
- In 2004, limited by "network cross-section bandwidth"
  - All data goes over the network during Map→Reduce shuffle
  - Switch capacity per node
    - 100-200Gbps of aggregate bandwidth at the root
    - 1800 machines, so ~7-14 MBps per machine
    - << disk or RAM speed
  - So try minimizing movement of data over the network
How ... 

- Map input is read from Google file system (GFS) replica on local disk, not over network
- Intermediate data goes over network just once
  - Map worker writes to local disk, not GFS
- Intermediate data partitioned into files holding many keys
Load balance in MapReduce

- Some tasks likely take longer than others
- **Solution** – Many more tasks than workers
  - Master hands out new tasks to workers who finish previous tasks
  - So no task is so big it dominates completion time (hopefully)
  - And faster servers do more work than slower ones, finish at the same time
- **Similar idea with stragglers**, the few slow workers that hold everyone back
  - Execute in parallel on another worker, a race to complete
Fault tolerance in MapReduce

- Master pings worker periodically
  - What if a server crashes during a MR job?
- Hiding failures is a huge part of ease of programming!
- MR re-runs just the failed Map()s and Reduce()s
  - MR requires them to be pure functions:
    - they don't keep state across calls
    - they don't read or write files other than expected MR inputs/outputs
    - there's no hidden communication among tasks
  - So re-execution yields the same output
- The pure functions requirement, a major limitation of MR compared to other parallel programming schemes
  - But it's critical to MR's simplicity
Details on worker crash recovery

• Map worker crashes
  – Crashed worker's intermediate Map output is lost but is likely needed by Reduce task
  – Master re-runs, spreads tasks over other GFS replicas of input
  – some Reduce workers may already have read failed worker's intermediate data
    • here we depend on functional and deterministic Map()
  – master need not re-run Map if Reduces have fetched all intermediate data though then a Reduce crash would then force re-execution of failed Map
Details on worker crash recovery

- Reduce worker crashes
  - Finished tasks are OK -- stored in GFS, with replicas. Master re-starts worker's unfinished tasks on other workers.

- Reduce worker crashes in the middle of writing its output.
  - GFS has atomic rename that prevents output from being visible until complete
  - So it's safe for the master to re-run the Reduce tasks somewhere else.
When you don’t want MapReduce

- Not everything fits the map/shuffle/reduce pattern …
- Small data, since overheads are high
- Small updates to big data, e.g. add a few documents to a big index
- Unpredictable reads (neither Map nor Reduce can choose input)
- Multiple shuffles, e.g. page-rank (can use multiple MR but not very efficient)
Distributed systems – cool, challenging, relevant!

Case study: MapReduce
- Single-handedly make big cluster computation popular
- Not the most efficient or flexible
- But scales well and is easy to program

Next: An short intro to networking

TODO
- Take the Go tour if you haven’t!
- Join Piazza
- Look for a team