```
6.824 2020 Lecture 1: Introduction

6.824: Distributed Systems Engineering

What is a distributed system?
  multiple cooperating computers
  storage for big web sites, MapReduce, peer-to-peer sharing, &c
  lots of critical infrastructure is distributed

Why do people build distributed systems?
  to increase capacity via parallelism
  to tolerate faults via replication
  to place computing physically close to external entities
  to achieve security via isolation

But:
  many concurrent parts, complex interactions
  must cope with partial failure
  tricky to realize performance potential

Why take this course?
  interesting -- hard problems, powerful solutions
  used by real systems -- driven by the rise of big Web sites
  active research area -- important unsolved problems
  hands-on -- you'll build real systems in the labs

COURSE STRUCTURE

http://pdos.csail.mit.edu/6.824

Course staff:
  Robert Morris, lecturer
  Anish Athalye, TA
  Aakriti Shroff, TA
  Favyen Bastani, TA
  Tossaporn Saengja, TA

Course components:
  lectures
  papers
  two exams
  labs
  final project (optional)

Lectures:
  big ideas, paper discussion, and labs
  will be video-taped, available online

Papers:
  research papers, some classic, some new
  problems, ideas, implementation details, evaluation
  many lectures focus on papers
  please read papers before class!
  each paper has a short question for you to answer
  and we ask you to send us a question you have about the paper
  submit question&answer by midnight the night before

Exams:
  Mid-term exam in class
  Final exam during finals week
  Mostly about papers and labs

Labs:
  goal: deeper understanding of some important techniques
```

```
  goal: experience with distributed programming
  first lab is due a week from Friday
  one per week after that for a while

Lab 1: MapReduce
Lab 2: replication for fault-tolerance using Raft
Lab 3: fault-tolerant key/value store
Lab 4: sharded key/value store

Optional final project at the end, in groups of 2 or 3.
  The final project substitutes for Lab 4.
  You think of a project and clear it with us.
  Code, short write-up, short demo on last day.

Lab grades depend on how many test cases you pass
  we give you the tests, so you know whether you'll do well

Debugging the labs can be time-consuming
  start early
  come to TA office hours
  ask questions on Piazza

MAIN TOPICS

This is a course about infrastructure for applications.
  * Storage.
  * Communication.
  * Computation.

The big goal: abstractions that hide the complexity of distribution.
  A couple of topics will come up repeatedly in our search.

Topic: implementation
  RPC, threads, concurrency control.
  The labs...

Topic: performance
  The goal: scalable throughput
    Nx servers -> Nx total throughput via parallel CPU, disk, net.
    [diagram: users, application servers, storage servers]
    So handling more load only requires buying more computers.
      Rather than re-design by expensive programmers.
    Effective when you can divide work w/o much interaction.
  Scaling gets harder as N grows:
    Load im-balance, stragglers, slowest-of-N latency.
    Non-parallelizable code: initialization, interaction.
    Bottlenecks from shared resources, e.g. network.
  Some performance problems aren't easily solved by scaling
    e.g. quick response time for a single user request
    e.g. all users want to update the same data
    often requires better design rather than just more computers
  Lab 4

Topic: fault tolerance
  1000s of servers, big network -> always something broken
  We'd like to hide these failures from the application.
  We often want:
    Availability -- app can make progress despite failures
    Recoverability -- app will come back to life when failures are repaired
  Big idea: replicated servers.
    If one server crashes, can proceed using the other(s).
    Labs 1, 2 and 3

Topic: consistency
  General-purpose infrastructure needs well-defined behavior.
```

```
    E.g. "Get(k) yields the value from the most recent Put(k,v)."
  Achieving good behavior is hard!
    "Replica" servers are hard to keep identical.
    Clients may crash midway through multi-step update.
    Servers may crash, e.g. after executing but before replying.
    Network partition may make live servers look dead; risk of "split brain".
  Consistency and performance are enemies.
    Strong consistency requires communication,
      e.g. Get() must check for a recent Put().
    Many designs provide only weak consistency, to gain speed.
      e.g. Get() does *not* yield the latest Put()!
      Painful for application programmers but may be a good trade-off.
  Many design points are possible in the consistency/performance spectrum!

CASE STUDY: MapReduce

Let's talk about MapReduce (MR) as a case study
  a good illustration of 6.824's main topics
  hugely influential
  the focus of Lab 1

MapReduce overview
  context: multi-hour computations on multi-terabyte data-sets
    e.g. build search index, or sort, or analyze structure of web
    only practical with 1000s of computers
    applications not written by distributed systems experts
  overall goal: easy for non-specialist programmers
  programmer just defines Map and Reduce functions
    often fairly simple sequential code
  MR takes care of, and hides, all aspects of distribution!

Abstract view of a MapReduce job
  input is (already) split into M files
  Input1 -> Map -> a,1 b,1
  Input2 -> Map ->     b,1
  Input3 -> Map -> a,1     c,1
                    |   |   |
                    |   |   -> Reduce -> c,1
                    |   -----> Reduce -> b,2
                    ---------> Reduce -> a,2
  MR calls Map() for each input file, produces set of k2,v2
    "intermediate" data
    each Map() call is a "task"
  MR gathers all intermediate v2's for a given k2,
    and passes each key + values to a Reduce call
  final output is set of <k2,v3> pairs from Reduce()s

Example: word count
  input is thousands of text files
  Map(k, v)
    split v into words
    for each word w
      emit(w, "1")
  Reduce(k, v)
    emit(len(v))

MapReduce scales well:
  N "worker" computers get you Nx throughput.
    Maps()s can run in parallel, since they don't interact.
    Same for Reduce()s.
  So you can get more throughput by buying more computers.

MapReduce hides many details:
  sending app code to servers
  tracking which tasks are done
```

```
    moving data from Maps to Reduces
    balancing load over servers
    recovering from failures

  However, MapReduce limits what apps can do:
    No interaction or state (other than via intermediate output).
    No iteration, no multi-stage pipelines.
    No real-time or streaming processing.

  Input and output are stored on the GFS cluster file system
    MR needs huge parallel input and output throughput.
    GFS splits files over many servers, in 64 MB chunks
      Maps read in parallel
      Reduces write in parallel
    GFS also replicates each file on 2 or 3 servers
    Having GFS is a big win for MapReduce

  What will likely limit the performance?
    We care since that's the thing to optimize.
    CPU? memory? disk? network?
    In 2004 authors were limited by network capacity.
      What does MR send over the network?
        Maps read input from GFS.
        Reduces read Map output.
          Can be as large as input, e.g. for sorting.
        Reduces write output files to GFS.
      [diagram: servers, tree of network switches]
      In MR's all-to-all shuffle, half of traffic goes through root switch.
      Paper's root switch: 100 to 200 gigabits/second, total
        1800 machines, so 55 megabits/second/machine.
        55 is small, e.g. much less than disk or RAM speed.
    Today: networks and root switches are much faster relative to CPU/disk.

  Some details (paper's Figure 1):
    one master, that hands out tasks to workers and remembers progress.
    1. master gives Map tasks to workers until all Maps complete
       Maps write output (intermediate data) to local disk
       Maps split output, by hash, into one file per Reduce task
    2. after all Maps have finished, master hands out Reduce tasks
       each Reduce fetches its intermediate output from (all) Map workers
       each Reduce task writes a separate output file on GFS

  How does MR minimize network use?
    Master tries to run each Map task on GFS server that stores its input.
      All computers run both GFS and MR workers
      So input is read from local disk (via GFS), not over network.
    Intermediate data goes over network just once.
      Map worker writes to local disk.
      Reduce workers read directly from Map workers, not via GFS.
    Intermediate data partitioned into files holding many keys.
      R is much smaller than the number of keys.
      Big network transfers are more efficient.

  How does MR get good load balance?
    Wasteful and slow if N-1 servers have to wait for 1 slow server to finish.
    But 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).
      So faster servers do more tasks than slower ones, finish abt the same time.

  What about fault tolerance?
    I.e. what if a worker crashes during a MR job?
    We want to completely hide failures from the application programmer!
    Does MR have to re-run the whole job from the beginning?
```

```
    Why not?
  MR re-runs just the failed Map()s and Reduce()s.
    Suppose MR runs a Map twice, one Reduce sees first run's output,
      another Reduce sees the second run's output?
    Correctness requires re-execution to yield exactly the same output.
    So Map and Reduce must be pure deterministic functions:
      they are only allowed to look at their arguments.
      no state, no file I/O, no interaction, no external communication.
  What if you wanted to allow non-functional Map or Reduce?
    Worker failure would require whole job to be re-executed,
      or you'd need to create synchronized global checkpoints.

Details of worker crash recovery:
  * Map worker crashes:
    master notices worker no longer responds to pings
    master knows which Map tasks it ran on that worker
      those tasks' intermediate output is now lost, must be re-created
      master tells other workers to run those tasks
    can omit re-running if Reduces already fetched the intermediate data
  * Reduce worker crashes.
    finished tasks are OK -- stored in GFS, with replicas.
    master re-starts worker's unfinished tasks on other workers.

Other failures/problems:
  * What if the master gives two workers the same Map() task?
    perhaps the master incorrectly thinks one worker died.
    it will tell Reduce workers about only one of them.
  * What if the master gives two workers the same Reduce() task?
    they will both try to write the same output file on GFS!
    atomic GFS rename prevents mixing; one complete file will be visible.
  * What if a single worker is very slow -- a "straggler"?
    perhaps due to flakey hardware.
    master starts a second copy of last few tasks.
  * What if a worker computes incorrect output, due to broken h/w or s/w?
    too bad! MR assumes "fail-stop" CPUs and software.
  * What if the master crashes?

Current status?
  Hugely influential (Hadoop, Spark, &c).
  Probably no longer in use at Google.
    Replaced by Flume / FlumeJava (see paper by Chambers et al).
    GFS replaced by Colossus (no good description), and BigTable.

Conclusion
  MapReduce single-handedly made big cluster computation popular.
  - Not the most efficient or flexible.
  + Scales well.
  + Easy to program -- failures and data movement are hidden.
  These were good trade-offs in practice.
  We'll see some more advanced successors later in the course.
  Have fun with the lab!
```