```
6.824 2015 Lecture 1: Introduction and lab overview

6.824: Distributed Systems Engineering

What is a distributed system?
  multiple networked cooperating computers
  Examples: Internet E-Mail, Athena file server, Google MapReduce, etc.

Why distribute?
  to connect physically separate entities
  to achieve security via physical isolation
  to tolerate faults via replication at separate sites
  to increase performance via parallel CPUs/mem/disk/net

But:
  complex, hard to debug
  new classes of problems, e.g. partial failure (did server accept my e-mail?)
  advice: don't distribute if a central system will work

Why take this course?
  interesting -- hard problems, non-obvious solutions
  active research area -- lots of progress + big unsolved problems
  used by real systems -- driven by the rise of big Web sites
  hands-on -- you'll build a real system in the labs

COURSE STRUCTURE

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

Course components:

Lectures about big ideas, papers, labs

Readings: research papers as case studies
  please read papers before class
    otherwise boring, and you can't pick it up by listening
  each paper has a question for you to answer
  and you should think of a question you would like to have answered
  submit question&answer before class, one or two paragraphs

Mid-term exam in class, and final exam

Labs: build increasingly sophisticated fault-tolerant services
  First lab is due on Monday

For PhD students, you can substitute a small research project for 5th lab
  talk to us

TAs: Steven Allen, Rohan Mahajan, Steven Valdez
  answer questions about material
  help you with labs
  will post office hours

MAIN TOPICS

Example:
  a shared file system, so users can cooperate, like Athena's AFS
  lots of client computers
  [diagram: clients, network, vague set of servers]

Topic: architecture
  What interface?
    Clients talk to servers -- what do they say?
    File system (files, file names, directories, &c)?
```

```
    Disk blocks, with FS in client?
    Separate naming + file servers?
    Separate FS + block servers?
  Single machine room or unified wide area system?
    Wide-area more difficult.
  Transparent?
    i.e. should it act exactly like a local disk file system?
    or is it OK if apps/users have to cope with distribution,
      e.g. know what server files are on, or deal with failures.
  Client/server or peer-to-peer?
  All these interact w/ performance, usefulness, fault behavior.

Topic: implementation
  How to simplify network communication?
    Can be messy (msg formatting, re-transmission, host names, &c)
    Frameworks can help: RPC, MapReduce, &c
  How to cope with inherent concurrency?
    Threads, locks, &c.

Topic: performance
  Distribution can hurt: network b/w and latency bottlenecks
    Lots of tricks, e.g. caching, concurrency, pre-fetch
  Distribution can help: parallelism, pick server near client
  Idea: scalable design
    Nx servers -> Nx total performance
  Need a way to divide the load by N
    Divide data over many servers ("sharding" or "partitioning")
    By hash of file name?
    By user?
    Move files around dynamically to even out load?
    "Stripe" each file's blocks over the servers?
  Performance scaling is rarely perfect
    Some operations are global and hit all servers (e.g. search)
      Nx servers -> 1x performance
    Load imbalance
      Everyone wants to get at a single popular file
      -> one server 100%, added servers mostly idle
      -> Nx servers -> 1x performance

Topic: fault tolerance
  Big system (1000s of server, complex net) -> always something broken
  We might want:
    Availability -- I can keep using my files despite failures
    Durability -- my files will come back to life someday
  Availability idea: replicate
    Servers form pairs, each file on both servers in the pair
    Client sends every operation to both
    If one server down, client can proceed using the other
  Opportunity: operate from both "replicas" independently if partitioned?
  Opportunity: can 2 servers yield 2x availability AND 2x performance?

Topic: consistency
  Assume a contract w/ apps/users about meaning of operations
    e.g. "read yields most recently written value"
  Consistency is about fulfiling the contract
    despite failure, replication/caching, concurrency, &c
  Problem: keep replicas identical
    If one is down, it will miss operations
      Must be brought up to date after reboot
    If net is broken, *both* replicas maybe live, and see different ops
      Delete file, still visible via other replica
      "split brain" -- usually bad
  Problem: clients may see updates in different orders
    Due to caching or replication
    I make 6.824 directory private, then TA creates grades file
```

```
    What if the operations run in different order on different replicas?
  Consistency often hurts performance (communication, blocking)
    Many systems cut corners -- "relaxed consistency"
    Shifts burden to applications

LABS

focus: fault tolerance and consistency -- central to distrib sys
  lab 1: MapReduce
  labs 2 through 5: storage servers
    progressively more sophisticated (tolerate more kinds of faults)
      progressively harder too!
    patterned after real systems, e.g. MongoDB
    end up with core of a real-world design for 1000s of servers

what you'll learn from the labs
  easy to listen to lecture / read paper and think you understand
  building forces you to really understand
  you'll have to do some design yourself
    we supply skeleton, requirements, and tests
    you'll have substantial scope to solve problems your own way
  you'll get experience debugging distributed systems
    tricky due to concurrency, unreliable messages

we've tried to ensure that the hard problems have to do w/ distrib sys
  not e.g. fighting against language, libraries, &c
  thus Go (type-safe, garbage collected, slick RPC library)
  thus fairly simple services (mapreduce, key/value store)

grades depend on how many test cases you pass
  we give you the tests, so you know whether you'll do well
  careful: if it usually passes, but occasionally fails,
    chances are it will fail when we run it

code review
  look at someone else's lab solution
  perhaps learn about another approach
  send feedback
  and receive feedback

Lab 1: MapReduce
  framework for parallel programming on 1000s of computers
  help you get up to speed on Go and distributed programming
  first exposure to some fault tolerance
    motivation for better fault tolerance in later labs
  motivating app for many papers
  popular distributed programming framework
    with many intellectual children

MapReduce computational model
  programmer defines Map and Reduce functions
  input is key/value pairs, divided into splits
    perhaps lots of files, k/v is filename/content
  Input Map -> a,1 b,7 c,9
  Input Map ->     b,2
  Input Map -> a,3     c,7
               |   |   |
                   |   -> Reduce -> c,16
                   -----> Reduce -> b,9
  MR framework calls Map() on each split, produces set of k2,v2
  MR framework gathers all Maps' v2's for a given k2,
    and passes them to a Reduce call
  final output is set of <k2,v3> pairs from Reduce()

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))

  What does MR framework do for word count?
    [master, input files, map workers, map output, reduce workers, output files]
    input files:
      f1: a b
      f2: b c
    send "f1" to map worker 1
      Map("f1", "a b") -> <a 1> <b 1>
    send "f2" to map worker 2
      Map("f2", "b c") -> <b 1> <c 1>
    framework waits for Map jobs to finish
    workers sort Map output by key
    framework tells each reduce worker what key to reduce
      worker 1: a
      worker 2: b
      worker 2: c
    each reduce worker pulls needed Map output from Map workers
      worker 1 pulls "a" Map output from every worker
    each reduce worker calls Reduce once for each of its keys
      worker 1: Reduce("a", [1]) -> 1
      worker 2: Reduce("b", [1, 1]) -> 2
                Reduce("c", [1]) -> 1

  Why is the MR framework convenient?
    * programmer only needs to think about the core work,
      the Map and Reduce functions, does not have to worry
      network communication, failure, &c.
    * the grouping by key between Map and Reduce fits
      some applications well (e.g., word count), since
      it brings together data needed by the Reduce.
    * but some applications don't fit well, because MR
      only allows the one type of communication between
      different parts of the application.
      e.g. word count but sort by frequency.

  Why might MR have good performance?
    Map and Reduce functions run in parallel on different workers
      Nx workers -> divide run-time by N
    But rarely quite that good:
      move map output to reduce workers
      stragglers
      read/write network file system

  What about failures?
    People use MR with 1000s of workers and vast inputs
    Suppose each worker only crashes once per year
      That's 3 per day!
    So a big MR job is very likely to suffer worker failures
    Other things can go wrong:
      Worker may be slow
      Worker CPU may compute incorrectly
      Master may crash
      Parts of the network may fail, lose packets, &c
      Map or Reduce or framework may have bugs in software

  Tools for dealing with failure?
    retry -- if worker fails, run its work on another worker
    replicate -- run each Map and Reduce on *two* workers
```

```
    replace -- for long-term health
    MapReduce uses all of these

  Puzzles for retry
    how do we know when to retry?
      can we detect when Map or Reduce worker is broken?
      can we detect incorrect worker output?
      can we distinguish worker failure from worker up, network lossy?
    why is retry correct?
      what if Map produces some output, then crashes?
        will we get duplicate output?
      what if we end up with two of the same Map running?
    in general, calling a function twice is not the same as calling it once
      why is it OK for Map and Reduce?

  Helpful assumptions
    One must make assumptions, otherwise too hard
    No bugs in software
    No incorrect computation: worker either produces correct output,
      or nothing -- assuming fail-stop.
    Master doesn't crash
    Map and Reduce are strict functions of their arguments
      they don't secretly read/write files, talk to each other,
      send/receive network messages, &c

  lab 1 has three parts:
    Part I: just Map() and Reduce() for word count
    Part II: we give you most of a distributed multi-server framework,
             you fill in the master code that hands out the work
             to a set of worker threads.
    Part III: make master cope with crashed workers by re-trying.

  Part I: main/wc.go
    stubs for Map and Reduce
    you fill them out to implement word count
    Map argument is a string, a big chunk of the input file

  demo of solution to Part I
    ./wc master kjv12.txt sequential
    more mrtmp.kjv12.txt-1-2
    more mrtmp.kjv12.txt

  Part I sequential framework: mapreduce/mapreduce.go RunSingle()
    split, maps, reduces, merge

  Part II parallel framework:
    master
    workers...
    shared file system
    our code splits the input before calling your master,
      and merges the output after your master returns
    our code only tells the master the number of map and reduce splits (jobs)
    each worker sends Register RPC to master
      your master code must maintain a list of registered workers
    master sends DoJob RPCs to workers
      if 10 map jobs and 3 workers,
      send out 3, wait until one worker says it's done,
      send it another, until all 10 done
    then the same for reduces
    master only needs to send job # and map vs reduce to worker
      worker reads input from files
    so your master code only needs to know the number of
      map and reduce jobs!
      which it can find from the "mr" argument
```

```
Thursday:
  master and workers talk via RPC, which hides network complexity
  more about RPC on Thursday
```