```
6.824 2018 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 distributed?
  to organize physically separate entities
  to achieve security via isolation
  to tolerate faults via replication
  to scale up throughput via parallel CPUs/mem/disk/net

But:
  complex: many concurrent parts
  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 -- lots of progress + big unsolved problems
  hands-on -- you'll build serious systems in the labs

COURSE STRUCTURE

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

Course staff:
  Malte Schwarzkopf, lecturer
  Robert Morris, lecturer
  Deepti Raghavan, TA
  Edward Park, TA
  Erik Nguyen, TA
  Anish Athalye, TA

Course components:
  lectures
  readings
  two exams
  labs
  final project (optional)
  TA office hours
  piazza for announcements and lab help

Lectures:
  big ideas, paper discussion, and labs

Readings:
  research papers, some classic, some new
  the papers illustrate key ideas and important details
  many lectures focus on the papers
  please read papers before class!
  each paper has a short question for you to answer
  and you must 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

Lab goals:
  deeper understanding of some important techniques
  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
  careful: if it often passes, but sometimes fails,
    chances are it will fail when we run it

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, to be used by applications.
  About abstractions that hide distribution from applications.
  Three big kinds of abstraction:
    Storage.
    Communication.
    Computation.
  [diagram: users, application servers, storage servers]

A couple of topics come up repeatedly.

1, Topic: implementation
  RPC, threads, concurrency control.

2. Topic: performance
    The dream: scalable throughput.
      Nx servers -> Nx total throughput via parallel CPU, disk, net.
      So handling more load only requires buying more computers.
    Scaling gets harder as N grows:
      Load im-balance, stragglers.
      Non-parallelizable code: initialization, interaction.
      Bottlenecks from shared resources, e.g. network.
    Note that some performance problems aren't easily attacked by scaling
      e.g. decreasing response time for a single user request
      might require programmer effort rather than just more computers

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

4 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 crash at awkward moments, e.g. after executing but before replying.
      Network may make live servers look dead; risk of "split brain".
    Consistency and performance are enemies.
      Consistency requires communication, e.g. to get latest Put().
      "Strong consistency" often leads to slow systems.
      High performance often imposes "weak consistency" on applications.
    People have pursued many design points in this spectrum.
```

```
CASE STUDY: MapReduce

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

MapReduce overview
  context: multi-hour computations on multi-terabyte data-sets
    e.g. analysis of graph structure of crawled web pages
    only practical with 1000s of computers
    often not developed by distributed systems experts
    distribution can be very painful, e.g. coping with failure
  overall goal: non-specialist programmers can easily split
    data processing over many servers with reasonable efficiency.
  programmer defines Map and Reduce functions
    sequential code; often fairly simple
  MR runs the functions on 1000s of machines with huge inputs
    and hides details of distribution

Abstract view of MapReduce
  input is divided into M files
  [diagram: maps generate rows of K-V pairs, reduces consume columns]
  Input1 -> Map -> a,1 b,1 c,1
  Input2 -> Map ->     b,1
  Input3 -> Map -> a,1     c,1
                   |   |   |
                   |   |   -> Reduce -> c,2
                   |   -----> 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 them to a Reduce call
  final output is set of <k2,v3> pairs from Reduce()
    stored in R output files
  [diagram: MapReduce API --
   map(k1, v1) -> list(k2, v2)
   reduce(k2, list(v2) -> list(k2, v3)]

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 hides many painful details:
  starting s/w on servers
  tracking which tasks are done
  data movement
  recovering from failures

MapReduce scales well:
  N computers gets you Nx throughput.
    Assuming M and R are >= N (i.e., lots of input files and map output keys).
    Maps()s can run in parallel, since they don't interact.
    Same for Reduce()s.
    The only interaction is via the "shuffle" in between maps and reduces.
  So you can get more throughput by buying more computers.
    Rather than special-purpose efficient parallelizations of each application.
    Computers are cheaper than programmers!

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 cross-section bandwidth".
      [diagram: servers, tree of network switches]
      Note all data goes over network, during Map->Reduce shuffle.
      Paper's root switch: 100 to 200 gigabits/second
      1800 machines, so 55 megabits/second/machine.
      Small, e.g. much less than disk (~50-100 MB/s at the time) or RAM speed.
    So they cared about minimizing movement of data over the network.
      (Datacenter networks are much faster today.)

  More details (paper's Figure 1):
    master: gives tasks to workers; remembers where intermediate output is
    M Map tasks, R Reduce tasks
    input stored in GFS, 3 copies of each Map input file
    all computers run both GFS and MR workers
    many more input tasks than workers
    master gives a Map task to each worker
      hands out new tasks as old ones finish
    Map worker hashes intermediate keys into R partitions, on local disk
    Q: What's a good data structure for implementing this?
    no Reduce calls until all Maps are finished
    master tells Reducers to fetch intermediate data partitions from Map workers
    Reduce workers write final output to GFS (one file per Reduce task)

  How does detailed design reduce effect of slow network?
    Map input is read from 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.
      Q: Why not stream the records to the reducer (via TCP) as they are being
         produced by the mappers?

  How do they get good load balance?
    Critical to scaling -- bad for N-1 servers to wait for 1 to finish.
    But some tasks likely take longer than others.
    [diagram: packing variable-length tasks into workers]
    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 work than slower ones, finish abt the same time.

  What about fault tolerance?
    I.e. what if a server crashes during a MR job?
    Hiding failures is a huge part of ease of programming!
    Q: Why not re-start the whole job from the beginning?
    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 requirement for pure functions is a major limitation of
      MR compared to other parallel programming schemes.
      But it's critical to MR's simplicity.

  Details of worker crash recovery:
    * Map worker crashes:
      master sees worker no longer responds to pings
      crashed worker's intermediate Map output is lost
        but is likely needed by every 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
    * Reduce worker crashes.
      finshed 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.

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?
    recover from check-point, or give up on job

For what applications *doesn't* MapReduce work well?
  Not everything fits the map/shuffle/reduce pattern.
  Small data, since overheads are high. E.g. not web site back-end.
  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)
  More flexible systems allow these, but more complex model.

How might a real-world web company use MapReduce?
  "CatBook", a new company running a social network for cats; needs to:
  1) build a search index, so people can find other peoples' cats
  2) analyze popularity of different cats, to decide advertising value
  3) detect dogs and remove their profiles
  Can use MapReduce for all these purposes!
  - run large batch jobs over all profiles every night
  1) build inverted index: map(profile text) -> (word, cat_id)
                           reduce(word, list(cat_id) -> list(word, list(cat_id))
  2) count profile visits: map(web logs) -> (cat_id, "1")
                           reduce(cat_id, list("1")) -> list(cat_id, count)
  3) filter profiles: map(profile image) -> img analysis -> (cat_id, "dog!")
                      reduce(cat_id, list("dog!")) -> list(cat_id)

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!
```