```
6.824 2018 Lecture 2: Infrastructure: RPC and threads

Most commonly-asked question: Why Go?
  6.824 used C++ for many years
    C++ worked out well
    but students spent time tracking down pointer and alloc/free bugs
    and there's no very satisfactory C++ RPC package
  Go is a bit better than C++ for us
    good support for concurrency (goroutines, channels, &c)
    good support for RPC
    garbage-collected (no use after freeing problems)
    type safe
    threads + GC is particularly attractive!
  We like programming in Go
    relatively simple and traditional
  After the tutorial, use https://golang.org/doc/effective_go.html
  Russ Cox will give a guest lecture March 8th

Threads
  threads are a useful structuring tool
  Go calls them goroutines; everyone else calls them threads
  they can be tricky

Why threads?
  They express concurrency, which shows up naturally in distributed systems
  I/O concurrency:
    While waiting for a response from another server, process next request
  Multicore:
    Threads run in parallel on several cores

Thread = "thread of execution"
  threads allow one program to (logically) execute many things at once
  the threads share memory
  each thread includes some per-thread state:
    program counter, registers, stack

How many threads in a program?
  Sometimes driven by structure
    e.g. one thread per client, one for background tasks
  Sometimes driven by desire for multi-core parallelism
    so one active thread per core
    the Go runtime automatically schedules runnable goroutines on available cores
  Sometimes driven by desire for I/O concurrency
    the number is determined by latency and capacity
    keep increasing until throughput stops growing
  Go threads are pretty cheap
    100s or 1000s are fine, but maybe not millions
    Creating a thread is more expensive than a method call

Threading challenges:
  sharing data
    one thread reads data that another thread is changing?
    e.g. two threads do count = count + 1
    this is a "race" -- and is usually a bug
    -> use Mutexes (or other synchronization)
    -> or avoid sharing
  coordination between threads
    how to wait for all Map threads to finish?
    -> use Go channels or WaitGroup
  granularity of concurrency
    coarse-grained -> simple, but little concurrency/parallelism
    fine-grained -> more concurrency, more races and deadlocks

What is a crawler?
  goal is to fetch all web pages, e.g. to feed to an indexer
  web pages form a graph
  multiple links to each page
  graph has cycles
```

```
Crawler challenges
  Arrange for I/O concurrency
    Fetch many URLs at the same time
    To increase URLs fetched per second
    Since network latency is much more of a limit than network capacity
  Fetch each URL only *once*
    avoid wasting network bandwidth
    be nice to remote servers
    => Need to remember which URLs visited
  Know when finished

Crawler solutions [crawler.go link on schedule page]

Serial crawler:
  the "fetched" map avoids repeats, breaks cycles
  it's a single map, passed by reference to recursive calls
  but: fetches only one page at a time

ConcurrentMutex crawler:
  Creates a thread for each page fetch
    Many concurrent fetches, higher fetch rate
  The threads share the fetched map
  Why the Mutex (== lock)?
    Without the lock:
      Two web pages contain links to the same URL
      Two threads simultaneouly fetch those two pages
      T1 checks fetched[url], T2 checks fetched[url]
      Both see that url hasn't been fetched
      Both fetch, which is wrong
    Simultaneous read and write (or write+write) is a "race"
      And often indicates a bug
      The bug may show up only for unlucky thread interleavings
    What will happen if I comment out the Lock()/Unlock() calls?
      go run crawler.go
      go run -race crawler.go
    The lock causes the check and update to be atomic
  How does it decide it is done?
    sync.WaitGroup
    implicitly waits for children to finish recursive fetches

ConcurrentChannel crawler
  a Go channel:
    a channel is an object; there can be many of them
      ch := make(chan int)
    a channel lets one thread send an object to another thread
    ch <- x
      the sender waits until some goroutine receives
    y := <- ch
      for y := range ch
      a receiver waits until some goroutine sends
    so you can use a channel to both communicate and synchronize
    several threads can send and receive on a channel
    remember: sender blocks until the receiver receives!
      may be dangerous to hold a lock while sending...
  ConcurrentChannel master()
    master() creates a worker goroutine to fetch each page
    worker() sends URLs on a channel
      multiple workers send on the single channel
    master() reads URLs from the channel
    [diagram: master, channel, workers]
  No need to lock the fetched map, because it isn't shared!
  Is there any shared data?
    The channel
    The slices and strings sent on the channel
    The arguments master() passes to worker()

When to use sharing and locks, versus channels?
  Most problems can be solved in either style
  What makes the most sense depends on how the programmer thinks
```

```
    state -- sharing and locks
    communication -- channels
    waiting for events -- channels
  Use Go's race detector:
    https://golang.org/doc/articles/race_detector.html
    go test -race

Remote Procedure Call (RPC)
  a key piece of distributed system machinery; all the labs use RPC
  goal: easy-to-program client/server communication

RPC message diagram:
  Client             Server
    request--->
       <---response

RPC tries to mimic local fn call:
  Client:
    z = fn(x, y)
  Server:
    fn(x, y) {
      compute
      return z
    }
  Rarely this simple in practice...

Software structure
  client app         handlers
    stubs           dispatcher
   RPC lib           RPC lib
     net  ------------ net

Go example: kv.go link on schedule page
  A toy key/value storage server -- Put(key,value), Get(key)->value
  Uses Go's RPC library
  Common:
    You have to declare Args and Reply struct for each RPC type
  Client:
    connect()'s Dial() creates a TCP connection to the server
    Call() asks the RPC library to perform the call
      you specify server function name, arguments, place to put reply
      library marshalls args, sends request, waits, unmarshally reply
      return value from Call() indicates whether it got a reply
      usually you'll also have a reply.Err indicating service-level failure
  Server:
    Go requires you to declare an object with methods as RPC handlers
    You then register that object with the RPC library
    You accept TCP connections, give them to RPC library
    The RPC library
      reads each request
      creates a new goroutine for this request
      unmarshalls request
      calls the named method (dispatch)
      marshalls reply
      writes reply on TCP connection
    The server's Get() and Put() handlers
      Must lock, since RPC library creates per-request goroutines
      read args; modify reply

A few details:
  Binding: how does client know who to talk to?
    For Go's RPC, server name/port is an argument to Dial
    Big systems have some kind of name or configuration server
  Marshalling: format data into packets
    Go's RPC library can pass strings, arrays, objects, maps, &c
    Go passes pointers by copying (server can't directly use client pointer)
    Cannot pass channels or functions

RPC problem: what to do about failures?
```

```
    e.g. lost packet, broken network, slow server, crashed server

  What does a failure look like to the client RPC library?
    Client never sees a response from the server
    Client does *not* know if the server saw the request!
      Maybe server never saw the request
      Maybe server executed, crashed just before sending reply
      Maybe server executed, but network died just before delivering reply
    [diagram of lost reply]

  Simplest failure-handling scheme: "best effort"
    Call() waits for response for a while
    If none arrives, re-send the request
    Do this a few times
    Then give up and return an error

  Q: is "best effort" easy for applications to cope with?

  A particularly bad situation:
    client executes
      Put("k", 10);
      Put("k", 20);
    both succeed
    what will Get("k") yield?
    [diagram, timeout, re-send, original arrives late]

  Q: is best effort ever OK?
     read-only operations
     operations that do nothing if repeated
       e.g. DB checks if record has already been inserted

  Better RPC behavior: "at most once"
    idea: server RPC code detects duplicate requests
      returns previous reply instead of re-running handler
    Q: how to detect a duplicate request?
    client includes unique ID (XID) with each request
      uses same XID for re-send
    server:
      if seen[xid]:
        r = old[xid]
      else
        r = handler()
        old[xid] = r
        seen[xid] = true

  some at-most-once complexities
    this will come up in lab 3
    how to ensure XID is unique?
      big random number?
      combine unique client ID (ip address?) with sequence #?
    server must eventually discard info about old RPCs
      when is discard safe?
      idea:
        each client has a unique ID (perhaps a big random number)
        per-client RPC sequence numbers
        client includes "seen all replies <= X" with every RPC
        much like TCP sequence #s and acks
      or only allow client one outstanding RPC at a time
        arrival of seq+1 allows server to discard all <= seq
    how to handle dup req while original is still executing?
      server doesn't know reply yet
      idea: "pending" flag per executing RPC; wait or ignore

  What if an at-most-once server crashes and re-starts?
    if at-most-once duplicate info in memory, server will forget
      and accept duplicate requests after re-start
    maybe it should write the duplicate info to disk
    maybe replica server should also replicate duplicate info
```

```
Go RPC is a simple form of "at-most-once"
  open TCP connection
  write request to TCP connection
  Go RPC never re-sends a request
    So server won't see duplicate requests
  Go RPC code returns an error if it doesn't get a reply
    perhaps after a timeout (from TCP)
    perhaps server didn't see request
    perhaps server processed request but server/net failed before reply came back

What about "exactly once"?
  unbounded retries plus duplicate detection plus fault-tolerant service
  Lab 3
```