```
6.824 2015 Lecture 2: Infrastructure: RPC and threads

Remote Procedure Call (RPC)
  a key piece of distrib sys machinery; all the labs use RPC
  goal: easy-to-program network communication
    hides most details of client/server communication
    client call is much like ordinary procedure call
    server handlers are much like ordinary procedures
  RPC is widely used!

RPC ideally makes net communication look just like fn call:
  Client:
    z = fn(x, y)
  Server:
    fn(x, y) {
      compute
      return z
    }
  RPC aims for this level of transparency

Examples from lab 1:
  DoJob
  Register

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

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

A few details:
  Which server function (handler) to call?
  Marshalling: format data into packets
    Tricky for arrays, pointers, objects, &c
    Go's RPC library is pretty powerful!
    some things you cannot pass: e.g., channels, functions
  Binding: how does client know who to talk to?
    Maybe client supplies server host name
    Maybe a name service maps service names to best server host
  Threads:
    Client often has many threads, so > 1 call outstanding, match up replies
    Handlers may be slow, so server often runs each in a thread

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/net failed just before sending reply
  [diagram of lost reply]

Simplest scheme: "at least once" behavior
  RPC library waits for response for a while
  If none arrives, re-send the request
```

```
    Do this a few times
    Still no response -- return an error to the application

  Q: is "at least once" easy for applications to cope with?

  Simple problem w/ at least once:
    client sends "deduct $10 from bank account"

  Q: what can go wrong with this client program?
    Put("k", 10) -- an RPC to set key's value in a DB server
    Put("k", 20) -- client then does a 2nd Put to same key
    [diagram, timeout, re-send, original arrives very late]

  Q: is at-least-once ever OK?
    yes: if it's OK to repeat operations, e.g. read-only op
    yes: if application has its own plan for coping w/ duplicates
      which you will need for Lab 1

  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 labs 2 and on
    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:
        unique client IDs
        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
      or client agrees to keep retrying for < 5 minutes
        server discards after 5+ minutes
    how to handle dup req while original is still executing?
      server doesn't know reply yet; don't want to run twice
      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?

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

```
Go RPC is "at-most-once"
  open TCP connection
  write request to TCP connection
  TCP may retransmit, but server's TCP will filter out duplicates
  no retry in Go code (i.e. will NOT create 2nd TCP connection)
  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

Go RPC's at-most-once isn't enough for Lab 1
  it only applies to a single RPC call
  if worker doesn't respond, the master re-send to it to another worker
    but original worker may have not failed, and is working on it too
  Go RPC can't detect this kind of duplicate
    No problem in lab 1, which handles at application level
    Lab 2 will explicitly detect duplicates

Threads
  threads are a fundamental server structuring tool
  you'll use them a lot in the labs
  they can be tricky
  useful with RPC
  Go calls them goroutines; everyone else calls them threads

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

Threading challenges:
  sharing data
     two threads modify the same variable at same time?
     one thread reads data that another thread is changing?
     these problems are often called races
     need to protect invariants on shared data
     use Go sync.Mutex
  coordination between threads
    e.g. wait for all Map threads to finish
    use Go channels
  deadlock
     thread 1 is waiting for thread 2
     thread 2 is waiting for thread 1
     easy detectable (unlike races)
  lock granularity
     coarse-grained -> simple, but little concurrency/parallelism
     fine-grained -> more concurrency, more races and deadlocks
  let's look at a toy RPC package to illustrate these problems

look at today's handout -- l-rpc.go
  it's a simplified RPC system
  illustrates threads, mutexes, channels
  it's a toy, though it does run
    assumes connection already open
    only supports an integer arg, integer reply
    omits error checks

struct ToyClient
  client RPC state
```

```
    mutex per ToyClient
    connection to server (e.g. TCP socket)
    xid -- unique ID per call, to match reply to caller
    pending[] -- chan per thread waiting in Call()
      so client knows what to do with each arriving reply

  Call
    application calls reply := client.Call(procNum, arg)
    procNum indicates what function to run on server
    WriteRequest knows the format of an RPC msg
      basically just the arguments turned into bits in a packet
    Q: why the mutex in Call()? what does mu.Lock() do?
    Q: could we move "xid := tc.xid" outside the critical section?
       after all, we are not changing anything
       [diagram to illustrate]
    Q: do we need to WriteRequest inside the critical section?
    note: Go says you are responsible for preventing concurrent map ops
      that's one reason the update to pending is locked

  Listener
    runs as a background thread
    what is <- doing?
    not quite right that it may need to wait on chan for caller

  Back to Call()...

  Q: what if reply comes back very quickly?
     could Listener() see reply before pending[xid] entry exists?
     or before caller is waiting for channel?

  Q: should we put reply:=<-done inside the critical section?
     why is it OK outside? after all, two threads use it.

  Q: why mutex per ToyClient, rather than single mutex per whole RPC pkg?

  Server's Dispatcher()
    note that the Dispatcher echos the xid back to the client
      so that Listener knows which Call to wake up
    Q: why run the handler in a separate thread?
    Q: is it a problem that the dispatcher can reply out of order?

  main()
    note registering handler in handlers[]
    what will the program print?

  Q: when to use channels vs shared memory + locks?
    here is my opinion
    use channels when you want one thread to explicitly wait for another
      often wait for a result, or wait for the next request
      e.g. when client Call() waits for Listener()
    use shared memory and locks when the threads are not intentionally
      directly interacting, but just happen to r/w the same data
      e.g. when Call() uses tc.xid
    but: they are fundamentally equivalent; either can always be used.

  Go's "memory model" requires explicit synchronization to communicate!
    This code is not correct:
      var x int
      done := false
      go func() { x = f(...); done = true }
      while done == false { }
```

```
    it's very tempting to write, but the Go spec says it's undefined
    use a channel or sync.WaitGroup instead

  Study the Go tutorials on goroutines and channels
```