15-440, Fall 2011, Class 06, Sept. 15, 2011
Randal E. Bryant

All code available in:
    /afs/cs.cmu.edu/academic/class/15440-f11/code/class06

Remote Procedure Calls (Tannenbaum 4.1-4.2)

Revisiting Go channel concurrency.

Consider channel of size 0: "Rendezvous

```
c := chan int

var x int = 0

T1:
        x = 1
        <- c

T2:
        c <- 1
        print x

T1: ---- x = 1 --|

T2: .............|-- print 1 ---
```

Channel has single sending thread & single receiving thread:

Whichever one hits its channel operation must wait until other one is ready. Then both atomically see that sender has sent & receiver has received. (In reality, may time slice threads).

What about multiple senders or receivers:

Then OS will select exactly one sender & one receiver, and they will rendezvous.

```
General vocabulary

Middleware: Protocol / software that lives just below Application
providing higher-level services.

Examples:
        HTTP (although now often used as transport protocol)
        LSP from project 1

Persistent vs. Transient communication

Persistent: Protocol holds onto all information until operation
completed.
        E.g., TCP, LSP

Transient: Protocol discards information if fails
        E.g., UDP

Synchronous vs. Asynchronous

Synchronous: Sender blocks until operation completes

Asynchronous: Sender returns from operation immediately
              E.g., Everything we’ve seen so far
```

Remote Procedure call

One way to provide client/server model to model.

Idea: On client, appear to make procedure call, but operation actually performed on server.

How this work:

1. Client application calls function
2. Function is really a "stub" that packages function name & arguments as message ("marshaling")
3. Send message to server
4. Server unpacks message, determines what function is being requested and executes it.
5. Server marshals results back into message and sends it back to client
6. Client stub unmarshals results and returns back to caller

Two versions:

Synchronous: Client must wait for all steps to complete.

Asynchronous: Stub returns after step 3. Some other mechanism provided to pick up result later.

Details:

Marshaling:

Need convention for how to send objects.

Example = JSON. Very general form. Converts struct to named fields. Applied recursively

Example applying it to our sequential buffer:

Add method to bufi:

```
func (bp *Buf) String() string {
        b, e := json.MarshalIndent(*bp, "", "  ")
        if e != nil {
                return e.String()
        }
        return string(b)
}
```

Here’s examples when inserting structures of form

```
type Val struct {
   X interface{}
}
```

```
Empty Buffer: {"Head":null,"Tail":null}

Insert "ABC"  {"Head":{"Val":{"X":"ABC"},"Next":null},
       "Tail":{"Val":{"X":"ABC"},"Next":null}}


Insert "GHI"
{"Head":{"Val":{"X":"ABC"},"Next":{"Val":{"X":"GHI"},"Next":null}},
        "Tail":{"Val":{"X":"GHI"},"Next":null}}
```

(Looks much nicer when use MarshalIndent)

Main point: There are standard ways to convert objects into byte sequences. These are "deep" encodings, meaning that they go all the way into a structure.

RPC Example.

Using Go RPC package.

In general see two styles of RPC implementation:

```
* Shallow integration.  Must use lots of library calls to set things
up:
        - How to format data
        - Registering which functions are available and how they are
          invoked.

* Deep integration.
        - Data formatting done based on type declarations
        - All public methods of object are registered.
```

Go is that latter.

Server side, write each operation as a function

```
func (s *servertype) Operate (args *argtype, reply *argtype) os.Error
```

Function must decode arguments, perform operation, encode reply.

Returns nil if no error.

Then must register servertype. All exported (uppercase names)
operations available.

Client side:

Synchronous call:

Invoke Call, with operation name (as string), and pointers for
arguments and reply.

When Call returns, get result from reply.

Asynchronous call:

Invoke Go, with operation name and pointers for arguments and reply,
and channel for responding.

Function returns immediately.

If want to get result, then receive from channel.

```
RPC Example: An RPC version of an asynchronous buffer

// For passing arbitrary values
type Val struct {
        X interface{}           # Embed in struct.  Not sure if necessary
}

// Server implementation
type SrvBuf struct {
        abuf *dserver.Buf       # Use one of our asynchronous buffers
                                # since needs concurrent access
}

func NewSrvBuf() *SrvBuf {
        return &SrvBuf{dserver.NewBuf()}
}

## Example methods for server

## Note signature.  Pass in arguments + reply location
func (srv *SrvBuf) Insert(arg *Val, reply *Val) os.Error {
        srv.abuf.Insert(*arg)       # Insert object of type Val
        *reply = nullVal()          # Wrapper around nil
        Vlogf(2, "Inserted %v\n", arg.X) # %v useful format type
        Vlogf(3, "Buffer: %s\n", srv.abuf.String()) # JSON marshaling
        return nil
}

func (srv *SrvBuf) Front(arg *Val, reply *Val) os.Error {
        *reply = srv.abuf.Front().(Val)  # Since inserted values of type Val
        Vlogf(2, "Front value %v\n", reply.X)
        Vlogf(3, "Buffer: %s\n", srv.abuf.String())
        return nil   # This means it’s OK
}

...

Here’s the magine
func Serve(port int) {
        srv := NewSrvBuf()
        # Register takes object and makes it’s exported methods available
        rpc.Register(srv)
        # Use HTTP as communication protocol
        rpc.HandleHTTP()
        addr := fmt.Sprintf(":%d", port)
        l, e := net.Listen("tcp", addr)
        Checkfatal(e)
        Vlogf(1, "Running server on port %d\n", port)
        Vlogf(3, "Buffer: %s\n", srv.abuf.String())
        # Set up HTTP server
        http.Serve(l, nil)
}
```

```
Client side

# Really don't need more than provided by RPC package
type SClient struct {
        client *rpc.Client
}

# Wrapper to access Call function
func (cli *SClient) Call(serviceMethod string, args interface{},
        reply interface{}) os.Error {
        return cli.client.Call(serviceMethod, args, reply)
}

# Setup up TCP client
func NewSClient(host string, port int) *SClient {
        hostport := fmt.Sprintf("%s:%d", host, port)
        client, e := rpc.DialHTTP("tcp", hostport)
        Checkfatal(e)
        Vlogf(1, "Connected to %s\n", hostport)
        return &SClient{client}
}

# Making RPC calls

func (cli *SClient) Insert(v Val) {
        var rv Val
        e := cli.Call("SrvBuf.Insert", &v, &rv)
        Vlogf(2, "Inserted %s\n", v)
        if Checkreport(1, e) {
                fmt.Printf("Insert failure\n")
        }
}

func (cli *SClient) Remove() Val {
        av := nullVal()
        var rv Val
        e := cli.Call("SrvBuf.Remove", &av, &rv)
        if Checkreport(1, e) {
                fmt.Printf("Remove failure\n")
                return nullVal()
        }
        Vlogf(2, "Removed %s\n", rv)
        return rv
}
```