```
15-440, Fall 2012, Class 07, Sept. 18, 2012
Randal E. Bryant

All code available in:
    /afs/cs.cmu.edu/academic/class/15440-f12/code/class07
Remote Procedure Calls (Tannenbaum 4.1-4.2)

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.

Natural way to express interaction:

client-->server: Do this for me (call)

Server does some work

server-->client: Here’s my response (return)

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.

History: Developed by Bruce Nelson & Andrew Birrell, ca. 1980. At the time, Nelson was PhD student at CMU, although much of the work done at Xerox PARC. Nelson went on to being very successful, including as CTO at CISCO. Died in 1999. Recognized via Bruce Nelson chair (held by Manuel Blum).

Let’s think about an application of RPC.

How about a distributed password cracker. Similar to that of Project 1, but designed to use RPC

Three classes of agents:

1. Request client. Submits cracking request to server. Waits until server responds.

2. Worker. Initially a client. Sends join request to server. Now it should reverse role & become a server. Then it can receive requests from main server to attempt cracking over limited range.

3. Server. Orchestrates whole thing. Maintains collection of workers. When receive request from client, split into smaller jobs over limited ranges. Farm these out to workers. When finds password, or exhausts complete range, respond to request client.

So, here’s the RPC version:

Request-->Server-->Request: Classic synchronous RPC

Worker-->Server. Synch RPC, but no return value.

"I’m a worker and I’m listening for you on host XXX, port YYY."

Server-->Worker. Synch RPC? No that would be a bad idea. Better be Asynch.

Otherwise, it would have to block while worker does its work, which misses the whole point of having many workers.

```
Why is RPC different from a regular procedure call?

1. Running on different machines:
   A. Don’t have shared memory space.
      Must ship argument data from client to server, and results back to client
      If need any data structures, must send them over, too.
   B. Possibly, two machines differ (e.g., byte ordering), and client & server
      may be written in different languages, or running different OS’s

2. Need to communicate with each other
   A. May need to locate server.  Simple: Have IP address.  More elaborate: locate
      server based on service being requested.
   B. Need some kind of network connection between them.
   C. Need conventions for how to encode ("marshal") data, send,
      and decode (unmarshal) it at other end.

3. Things don’t always  work right.
   A. Packets get dropped, duplicated, or mangled.
   B. Client or server may die, before or during call

4. Might be worried about security
   A. Need way to have client & server authenticate to each other
   B. Need way to keep communications secret
```

Lets look at some 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:

Data structures declared as:

```
// Linked list element
type BufEle struct {
        Val interface{}
        Next *BufEle
}

type Buf struct {
        Head *BufEle         // Oldest element
        tail *BufEle         // Most recently inserted element
        cnt int              // Number of elements in list
}
```

(Note that only upper case names get marshaled.)

Add method to bufi:

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

Here’s examples when inserting strings into the buffer:

Empty buffer
```
{
  "Head": null
}
```

After inserting "pig", "cat", "dog":
```
{
  "Head": {
    "Val": "pig",
    "Next": {
      "Val": "cat",
      "Next": {
        "Val": "dog",
        "Next": null
      }
    }
  }
}
```

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. Beware of trying to do this with circular
data structures!

Other encoding methods:

```
gob: Used by Go RPC.
XML:
```

```
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
        - (Almost) all public methods of object are registered.

Go is the latter.
```

Server side, write each operation as a function

```
func (s *servertype) Operate (args *argtype, reply *argtype) 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.  Gob wants it this way.
}

// 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) Error {
        srv.abuf.Insert(arg.X)      # Insert object of type interface{}
        *reply = nullVal()          # Wrapper around nil
        return nil
}

func (srv *SrvBuf) Front(arg *Val, reply *Val) Error {
        *reply = Val{srv.abuf.Front()}
        return nil   # This means it’s OK
}

...

Here’s the main incantation

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)
        # 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)
        return &SClient{client}
}

# Making RPC calls

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

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

What about bigger data structures?

Suppose we want to return entire buffer contents.

Add to bufi:

```
// Return slice containing entire buffer contents
func (bp *Buf) Contents() []interface{} {
        result := make([]interface{}, bp.cnt)
        e := bp.Head
        for i := 0; i < bp.cnt; i++ {
                result[i] = e.Val
                e = e.Next
        }
        return result
}
```

(Also added field cnt to bufi.Buf, to keep count of number of elements)

Added to dserver:

```
func (bp *Buf) Contents() []interface{}
```

Add to RPC code:

1. Let’s name this data type:

```
type Islice []interface{}
var islice Islice
```

2. Let’s let the server & client know about this type:

```
func NewSrvBuf() *SrvBuf {
        gob.Register(islice)
        return &SrvBuf{dserver.NewBuf()}
}

func NewSClient(host string, port int) *SClient {
        gob.Register(islice)
        hostport := fmt.Sprintf("%s:%d", host, port)
        client, e := rpc.DialHTTP("tcp", hostport)
        Checkfatal(e)
        return &SClient{client}
}
```

3. Let’s implement the server function:

```
func (srv *SrvBuf) Contents(arg *Val, reply *Val) error {
        c := Islice(srv.abuf.Contents())
        *reply = Val{c}
        Vlogf(2, "Generated contents: %v\n", c)
        return nil
}

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

Other issues:

Dealing with failures:

* Network dropped/duplicated/mangled packets
* Client or server dies before or during operation

Typically, want operation required by RPC call to take place EXACTLY
once.

This is hard to guarantee.

Variants:

"At most once": Client sends request to server. Hopefully gets response.

Fails if:
1. Request message doesn’t get to server
2. Server fails
3. Response message doesn’t get to client.

Note that with #1 & #2, call not executed. With #3 call executed,
e.g., could cause state change by server. ("Withdraw $100 from my bank account")

"At least once" Client executes loop: { send request; wait for response }
until either get response or give up.

Same failure modes. But overcomes cases where these failures are not persistent.

Danger: Server gets multiple requests and doesn’t realize they are duplicates.
   (Think of the account withdrawal example)

Solution: Want to make operations "idempotent:" Doing same operation
multiple times has same effect as doing it once.

Example mechanism: Use sequence numbers.

Requires maintaining per-client state at server. (Imagine having 1M clients.)

See example in Project 1 protocol.

LSP is like RPC, in that it serves as middleware between client and
server applications. Deals with failed messages, clients, and
servers. But provides message passing model between clients &
servers, rather than RPC.

```
* Each data message includes sequence number.
  - Can detect duplicate messages (either from network or from resending)

* Each data message acknowledged.
  - Sender knows that it’s been received.

* Sender cannot new send message until previous one acknowledged
  - Prevents lost message in middle of data stream

* Periodic resend of most recent data + acknowlegement.
  - Compensate for dropped messages
  - Indication that machine at other end of connection still alive.
    Same effect as "heartbeat" messages

* Detect failure at other end if no messages for K epochs
  - Independent of application-level activity.
```

Where these mechanisms show up in RPC.

* Typically use TCP or HTTP. Provides reliable transport level that
  eliminates most network problems.

* Typically have sequence numbers to avoid acting on duplicate requests
  (May need to persist across multiple TCP sessions.)

* Don’t (by default) do a very good job detecting failed clients or servers.

Other RPC systems:

ONC RPC (a.k.a. Sun RPC). Fairly basic. Includes encoding standard
XDR + language for describing data formats.

Java RMI (remote method invocation). Very elaborate. Tries to make it look like
can perform arbitrary methods on remote objects.

Thrift. Developed at Facebook. Now part of Apache Open Source.
Supports multiple data encodings & transport mechanisms. Works across
multiple languages.

Avro. Also Apache standard. Created as part of Hadoop project. Uses
JSON. Not as elaborate as Thrift.