```
1  //
2  // *** a simple application using the lock service
3  //
4
5  func main() {
6    primary_port := os.Args[1]
7    backup_port := os.Args[2]
8    clerk := lockservice.MakeClerk(primary_port,
9                                   backup_port)
10
11   for clerk.Lock("car keys") == false {
12     time.Sleep(100 * time.Millisecond) // wait
13   }
14
15   // it's my turn to drive the car...
16
17   clerk.Unlock("car keys")
18 }
19
20 //
21 // *** client.go -- the application calls
22 // these library "stubs"
23 //
24
25 type Clerk struct {
26   servers [2]string // primary port, backup port
27 }
28
29 //
30 // ask the lock service for a lock.
31 // returns true if the lock service
32 // granted the lock, false otherwise.
33 //
34 func (ck *Clerk) Lock(lockname string) bool {
35   args := &LockArgs{}        // RPC arguments
36   args.Lockname = lockname
37   var reply LockReply        // space for RPC reply
38
39   // send an RPC request, wait for the reply.
40   ok := call(ck.servers[0], "LockServer.Lock",
41              args, &reply)
42   return ok && reply.OK
43 }
44
```



```
44 //
45 // *** server.go
46 //
47
48 //
49 // a lock server's state
50 //
51 type LockServer struct {
52   mu sync.Mutex
53   l net.Listener
54
55   am_primary bool // am I the primary?
56   backup string   // backup's port
57
58   // for each lock name, is it locked?
59   locks map[string]bool
60 }
61
62
63 //
64 // server Lock() RPC handler
65 //
66 func (ls *LockServer) Lock(args *LockArgs,
67                            reply *LockReply) error {
68   ls.mu.Lock()
69   defer ls.mu.Unlock()
70
71   locked, _ := ls.locks[args.Lockname]
72
73   if locked {
74     reply.OK = false
75   } else {
76     reply.OK = true
77     ls.locks[args.Lockname] = true
78   }
79
80   return nil
81 }
```

```
1  //
2  // toy RPC library
3  //
4
5  type ToyClient struct {
6    mu sync.Mutex
7    conn io.ReadWriteCloser    // connection to server
8    xid int64                  // next unique request #
9    pending map[int64]chan int // waiting calls [xid]
10 }
11
12 //
13 // client application uses Call() to make an RPC.
14 // client := MakeClient(server)
15 // reply := client.Call(procNum, arg)
16 //
17 func (tc *ToyClient) Call(procNum int, arg int) int {
18   done := make(chan int) // for tc.Listener()
19
20   tc.mu.Lock()
21   xid := tc.xid // allocate a unique xid
22   tc.xid++
23   tc.pending[xid] = done // for tc.Listener()
24   tc.WriteRequest(xid, procNum, arg) // send to server
25   tc.mu.Unlock()
26
27   reply := <− done // wait for reply via tc.Listener()
28
29   tc.mu.Lock()
30   delete(tc.pending, xid)
31   tc.mu.Unlock()
32
33   return reply
34 }
35
36 //
37 // listen for replies from the server,
38 // hand them off to the right caller.
39 // runs as a background thread.
40 //
41 func (tc *ToyClient) Listener() {
42   for {
43     xid, reply := tc.ReadReply()
44     tc.mu.Lock()
45     ch, ok := tc.pending[xid]
46     tc.mu.Unlock()
47     if ok {
48       ch <− reply
49     }
50   }
51 }
```



```
52
53 type ToyServer struct {
54   mu sync.Mutex
55   conn io.ReadWriteCloser // connection from client
56   handlers map[int]func(int)int // procedures
57 }
58
59 //
60 // listen for client requests, call the handler,
61 // send back replies. runs as a background thread.
62 //
63 func (ts *ToyServer) Dispatcher() {
64   for {
65     xid, procNum, arg := ts.ReadRequest()
66     ts.mu.Lock()
67     fn, ok := ts.handlers[procNum]
68     ts.mu.Unlock()
69     go func() {
70       var reply int
71       if ok {
72         reply = fn(arg)
73       }
74       ts.mu.Lock()
75       ts.WriteReply(xid, reply)
76       ts.mu.Unlock()
77     }()
78   }
79 }
80
81 func main() {
82   clientPipe, serverPipe := MakeDualPipe()
83   tc := MakeToyClient(clientPipe)
84   ts := MakeToyServer(serverPipe)
85   ts.handlers[22] = func(a int) int { return a+1 }
86
87   reply := tc.Call(22, 100)
88   fmt.Printf("Call(22, 100) −> %v\n", reply)
89 }
```