Outline

RPC overview

Writing an RPC server in Go

MapReduce: fault tolerance and optimizations
Outline

RPC overview

Writing an RPC server in Go

MapReduce: fault tolerance and optimizations
Remote Procedure Call
Remote Procedure Call

Calling a procedure on a remote process as if it were local

Request-response interface

Request: arguments to remote procedure

Response: return values of remote procedure

Examples: client-server, master-worker, peer-peer
Example: Master-Worker

Master {
  func LaunchTasks() {
    for worker in workers {
      // want to call Worker.RunTask(...)
    }
  }
}

Worker {
  func RunTask(index) result {
    // ...
  }
}
Example: Master-Worker

Master {
    func LaunchTasks() {
        for worker in workers {
            index = worker.Index
            address = worker.Address
            request = MakeRequest(index)
            response = sendRPC("RunTask", address, request)
            result = response.Result
            handleResult(result)
        }
    }
}

Worker {
    func RunTask(index) result {
        // ...
    }
}
Asynchronous RPC

Await RPC response in a separate thread

Multiple ways to implement this:

Pass a *callback* to RPC that will be invoked later
Asynchronous RPC

Await RPC response in a separate goroutine

Multiple ways to implement this:

Pass a \textit{callback} to RPC that will be invoked later

\begin{verbatim}
func handleResponse { ... }
sendRPC("RunTask", address, request, handleResponse)
\end{verbatim}
Asynchronous RPC

Await RPC response in a separate thread

Multiple ways to implement this:

- Pass a *callback* to RPC that will be invoked later
- Use *channels* to communicate RPC reply back to main thread
Asynchronous RPC

Await RPC response in a separate thread

Multiple ways to implement this:

Pass a $callback$ to RPC that will be invoked later

Use $channels$ to communicate RPC reply back to main thread

```go
func(address, request string) {
    channel <- sendRPC("RunTask", address, request)
} (some_address, some_request)
handleResponse(<-channel)
```
What’s an example application where we would want asynchronous RPCs?
Outline

MapReduce: fault tolerance and optimizations

RPC overview

Writing an RPC server in Go
Go RPCs

Implementation in built-in library net/rpc requires 3 steps:

1. Write stub receiver methods of the form:

   ```go
   func (t *T) MethodName(args T1, reply *T2) error
   ```

2. Register receiver methods

3. Create a listener (i.e., server) that accepts requests
func makeRequest(input string, serverAddr string) (map[string]int, error) {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    err = client.Call("WordCountServer.Compute", args, &reply)
    if err != nil {
        return nil, err
    }
    return reply.Counts, nil
}
**Go example: Word count client**

```go
func makeRequest(input string, serverAddr string) (map[string]int, error) {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    err = client.Call("WordCountServer.Compute", args, &reply)
    if err != nil {
        return nil, err
    }
    return reply.Counts, nil
}
```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    err = client.Call("WordCountServer.Compute", args, &reply)
    if err != nil {
        return nil, err
    }
    return reply.Counts, nil
}
Go example: Word count server

```go
type WordCountServer struct {
    addr string
}

type WordCountRequest struct {
    Input string
}

type WordCountReply struct {
    Counts map[string]int
}

func (*WordCountServer) Compute(request WordCountRequest, reply *WordCountReply) error {
    counts := make(map[string]int)
    input := request.Input
    tokens := strings.Fields(input)
    for _, t := range tokens {
        counts[t] += 1
    }
    reply.Counts = counts
    return nil
}
```

18
func (*WordCountServer) Compute(request WordCountRequest, reply *WordCountReply) error {
    counts := make(map[string]int)
    input := request.Input
    tokens := strings.Fields(input)
    for _, t := range tokens {
        counts[t] += 1
    }
    reply.Counts = counts
    return nil
}

Go example: Word count server

type WordCountServer struct {
    addr string
}

type WordCountRequest struct {
    Input string
}

type WordCountReply struct {
    Counts map[string]int
}
func (server *WordCountServer) Listen() {
    rpc.Register(server)
    listener, err := net.Listen("tcp", server.addr)
    checkError(err)
    go func() {
        rpc.Accept(listener)
    }()
}
Go example: Word count server

```go
func (server *WordCountServer) Listen() {
    rpc.Register(server)
    listener, err := net.Listen("tcp", server.addr)
    checkError(err)
    go func() {
        rpc.Accept(listener)
    }()
}
```
Go example: Word count server

```go
defunc (server *WordCountServer) Listen() {
    rpc.Register(server)
    listener, err := net.Listen("tcp", server.addr)
    checkError(err)
    go func() {
        rpc.Accept(listener)
    }()
}
```
func main() {
    serverAddr := "localhost:8888"
    server := WordCountServer{serverAddr}
    server.Listen()
    input1 := "hello I am good hello bye bye bye bye bye good night hello"
    wordcount, err := makeRequest(input1, serverAddr)
    checkError(err)
    fmt.Printf("Result: %v\n", wordcount)
}

Result: map[hello:3 I:1 am:1 good:2 bye:4 night:1]
Is this synchronous or asynchronous?

```go
func makeRequest(input string, serverAddr string) (map[string]int, error) {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    err = client.Call("WordCountServer.Compute", args, &reply)
    if err != nil {
        return nil, err
    }
    return reply.Counds, nil
}
```
func makeRequest(input string, serverAddr string) chan Result {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    ch := make(chan Result)
    go func() {
        if err != nil {
            ch <- Result{nil, err} // something went wrong
        } else {
            ch <- Result{reply.Counts, nil} // success
        }
    }()
    return ch
}
Making client asynchronous

```go
func makeRequest(input string, serverAddr string) *Call {
    client, err := rpc.Dial("tcp", serverAddr)
    checkError(err)
    args := WordCountRequest{input}
    reply := WordCountReply{make(map[string]int)}
    return client.Go("WordCountServer.Compute", args, &reply, nil)
}

call := makeRequest(...)   
<-call.Done
checkError(call.Error)
handleReply(call.Reply)
```
Exercise: Cristian’s algorithm

Implement a CristianServer that other machines sync their local time to
MapReduce: Fault Tolerance
MapReduce: Fault Tolerance

Synchronization barrier
MapReduce: Fault Tolerance

Synchronization barrier
MapReduce: Fault Tolerance
Launch same task on a different machine

Assumes tasks are deterministic and idempotent
What if server 1 is just **REALLY** slow?

Server 1 is a *straggler*
Use the same idea!

Speculative execution
What should we re-execute?
All mappers might provide inputs to Reduce 2
What should we re-execute?

Write intermediate output to stable storage
What could go wrong?
Key idea: Determine tasks to recompute using *data lineage*, instead of recomputing all tasks.
Lineage is useful for optimizations too
Reusing map outputs

Job 1:

Job 2:
Cristian’s algorithm: Outline

1. Client sends a **request** packet, timestamped with its local clock $T_1$

2. Server timestamps its receipt of the request $T_2$ with its local clock

3. Server sends a **response** packet with its local clock $T_3$ and $T_2$

4. Client locally timestamps its receipt of the server’s response $T_4$
Cristian’s algorithm: Offset sample calculation

Goal: Client sets clock $\leftarrow T_3 + \delta_{\text{resp}}$

- Client samples round trip time $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$

- But client knows $\delta$, not $\delta_{\text{resp}}$

Assume: $\delta_{\text{req}} \approx \delta_{\text{resp}}$

Client sets clock $\leftarrow T_3 + \frac{1}{2}\delta$
Exercise: Cristian’s algorithm

Implement a CristianServer that other machines sync their local time to

```go
func SyncTime(serverAddr string) (time.Time, error)
```

Set local time = $T_3 + RTT/2$, where $RTT = (T_4 - T_1) - (T_3 - T_2)$

Note: You can just build a simplified version where $T_2 = T_3$

Hint: use time.Time’s Sub and Add methods, time.Now()