# Remote Procedure Calls with RPC

This lesson covers how Go lets two different machines (a server machine and a client machine) communicate with another using remote calls.

Go programs can communicate with each other through the `net/rpc-package`; so this is another application of the client-server paradigm. It provides a convenient means of making function calls over a network connection. Of course, this is only useful when the programs run on different machines. The `rpc` package builds on the package `gob` to turn its encode/decode automation into transport for method calls across the network.

A server registers an object, making it visible as a *service* with the type-name of the object. This provides access to the exported methods of that object across a network or other I/O connection for remote clients. It is all about exposing methods on types over a network. The package uses the Http and TCP protocol, and the `gob` package for data transport. A server may register multiple objects (services) of different types, but it is an error to register multiple objects of the same type.

Here we discuss a simple example:

Environment Variables

| Key: | Value: |
| --- | --- |
| GOROOT | /usr/local/go |
| GOPATH | //root/usr/local/go/src |
| PATH | //root/usr/local/go/src/bin:/usr/local/go... |

```
package main
import (
        "net/http"
        "log"
        "net"
        "net/rpc"
        "time"
        "rpcobjects"
)
```

```
func main() {
        calc := new(rpcobjects.Args)

        rpc.Register(calc)
        rpc.HandleHTTP()
        listener, e := net.Listen("tcp", "0.0.0.0:3001")
        if e != nil {
                log.Fatal("Starting RPC-server -listen error:", e)
        }
        go http.Serve(listener, nil)
        time.Sleep(1000e9)
}
```

> **Remark**: If you're running locally, then replace **line 15** with `listener, e := net.Listen("tcp", "localhost:1234")`

This is the server-side code. In **main.go**, we need the package `net/rpc`, which is imported at **line 6**. We also import the package `rpcobjects` (see **line 8**). At **line 12**, we construct a new instance of type `rpcobject.Args`, called `calc`. At **line 13**, this is registered as an `rpc` method.

**Line 14** tells `rpc` to handle the HTTP protocol. Then, at **line 15**, we make an RPC server listener with `net.Listen`. From **line 16** to **line 18**, we perform the usual error-handling. **Line 19** starts the RPC server in a goroutine, and then the `main()` routine waits for a second.

`rpcobjects.go` is a package we made and is imported at **line 8**. This server-side code defines the type `Args` as consisting of two exported integers `N` and `M` (from **line 3** to **line 5**). It also defines a method `Multiply` on Args (from **line 7** to **line 10**) that takes pointers to `Args` and the reply, which is the *product* of the integers.

Here is the client code:

```
package main
import (
"fmt"
"log"
"net/rpc"
"./rpc_objects"
)

const serverAddress = "localhost"
func main() {
  client, err := rpc.DialHTTP("tcp", serverAddress + ":3001")
  if err != nil {
    log.Fatal("Error dialing:", err)
```

```
  }
  // Synchronous call
  args := &rpc_objects.Args{7, 8}
  var reply int
  err = client.Call("Args.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Args error:", err)
  }
  fmt.Printf("Args: %d * %d = %d", args.N, args.M, reply)
}
```

RPC Client

> **Remark**: If you're running locally, then change **line 11** as `client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")`

This is the client-side code, that goes with the preceding server code. The address of the server (can be its name or IP address) is stored in the constant `serverAddress` at **line 9** (for easy testing, we use localhost here).

The client machine has to know the definition of the object type (`Args` in our case) and its methods (`Multiply` in this example). It calls `rpc.DialHTTP()` at **line 11**, with error-handling from **line 11** to **line 14**.

At **line 16**, `args := &rpc_objects.Args{7, 8}`, the client initializes an `Args` struct with the values 7 and **8**. When the client connection is made, remote methods can be invoked upon it with `client.Call("Type.Method", args, &reply)`, as done at **line 18**, where `Type.Method` here becomes `Args.Multiply`.

Now, let's run the client and server.

Again, after error-handling (from **line 19** to **line 21**), the reply comes in from the remote server and is displayed at **line 22**.

Environment Variables

| Key: | Value: |
| --- | --- |
| GOROOT | /usr/local/go |
| GOPATH | //root/usr/local/go/src |
| PATH | //root/usr/local/go/src/bin:/usr/local/go... |

```
package main
import (
"fmt"
```

```
"log"
"net/rpc"
"./rpcobjects"
)

const serverAddress = "localhost"
func main() {
  client, err := rpc.DialHTTP("tcp", serverAddress + ":3001")
  if err != nil {
    log.Fatal("Error dialing:", err)
  }
  // Synchronous call
  args := &rpcobjects.Args{7, 8}
  var reply int
  err = client.Call("Args.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Args error:", err)
  }
  fmt.Printf("Args: %d * %d = %d", args.N, args.M, reply)
}
```

First, start the server, and then a client process. Click the **RUN** button and wait for the terminal to start. Then, open a separate console-window for a client process, which is started by performing the following steps on the terminal:

- Type `cd usr/local/go/src` and press ENTER.
- Type `go run client.go` and press ENTER.

> **Remark:** If you're running it locally then only perform the second step: `go run client.go`

This gets the following result:

```
Args: 7 * 8 = 56
```

This call is *synchronous*, so it waits for the result to come back. An asynchronous call can be made as follows:

```
call1 := client.Go("Args.Multiply", args, &reply, nil)
replyCall := <- call1.Done
```

If the last argument has a value of *nil*, a new channel will be allocated when the call is complete.

If you have a Go server running as root and want to run some of your code as

a different user, the package go-runas by *Brad Fitz* uses the `rpc` package just to accomplish this.

---

Now, you have gained a lot of knowledge on web servers. In the next lesson, you'll learn how to send an email with Go.