Multiplexing

This lesson brings attention to the client-server model that best utilizes the goroutines and channels. It provides the code and explanation of how goroutines and channels together make a client-server application.

WE'LL COVER THE FOLLOWING

- A typical client-server pattern
- Teardown: shutting down the server

A typical client-server pattern

Client-server applications are the kind of applications where goroutines and channels shine. A *client* can be any running program on any device that needs something from a server, so it sends a *request*. The *server* receives this request, does some work, and then sends a *response* back to the client. In a typical situation, there are many clients (so many requests) and one (or a few) server(s). An example we use all the time is the client browser, which requests a web page. A web server responds by sending the web page back to the browser.

In Go, a server will typically perform a response to a client in a goroutine, so a goroutine is launched for every client-request. A technique commonly used is that the client-request itself contains a channel, which the server uses to send in its response.

For example, the `request` is a struct like the following which embeds a `reply` channel:

```
type Request struct {
  a, b int;
  replyc chan int; // reply channel inside the Request
}
```

Or more generally:

```go
type Reply struct { ... }
type Request struct {
  arg1, arg2, arg3 some_type
  replyc chan *Reply
}
```

Continuing with the simple form, the server could launch for each request a function `run()` in a goroutine that will apply an operation `op` of type `binOp` to the ints and then send the result on the `reply` channel:

```go
type binOp func(a, b int) int

func run(op binOp, req *Request) {
  req.replyc <- op(req.a, req.b)
}
```

The server routine loops forever, receiving requests from a `chan *Request` and, to avoid blocking due to a long-running operation, it starts a goroutine for each request to do the actual work.

```go
func server(op binOp, service chan *Request) {
  for {
    req := <-service; // requests arrive here
    // start goroutine for request:
    go run(op, req); // don't wait for op to complete
  }
}
```

The server is started in its own goroutine by the function `startServer`:

```go
func startServer(op binOp) chan *Request {
  reqChan := make(chan *Request);
  go server(op, reqChan);
  return reqChan;
}
```

Here, `startServer` will be invoked in the main routine.

In the following test-example, **100** requests are posted to the server, only after they all have been sent do we check the responses in reverse order:

```go
func main() {
  adder := startServer(func(a, b int) int { return a + b })
  const N = 100
  var reqs [N]Request
  for i := 0; i < N; i++ {
    req := &reqs[i]
    req.a = i
    req.b = i + N
    req.replyc = make(chan int)
    adder <- req // adder is a channel of requests
  }
  // checks:
  for i := N - 1; i >= 0; i-- { // doesn't matter what order
    if <-reqs[i].replyc != N+2*i {
      fmt.Println("fail at", i)
    } else {
      fmt.Println("Request ", i, " is ok!")
    }

  }
  fmt.Println("done")
}
```

The following is the resultant program of combining the above snippets into an executable format:

```go
package main
import "fmt"

type Request struct {
  a, b int
  replyc chan int // reply channel inside the Request
}

type binOp func(a, b int) int
func run(op binOp, req *Request) {
  req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request) {
  for {
    req := <-service // requests arrive here

    // start goroutine for request:
    go run(op, req) // don't wait for op
  }
}
```

```
func startServer(op binOp) chan *Request {
  reqChan := make(chan *Request)
  go server(op, reqChan)

  return reqChan
}

func main() {
  adder := startServer(func(a, b int) int { return a + b })
  const N = 100
  var reqs [N]Request
  for i := 0; i < N; i++ {
    req := &reqs[i]
    req.a = i
    req.b = i + N
    req.replyc = make(chan int)
    adder <- req
  }
  // checks:
  for i := N - 1; i >= 0; i-- { // doesn't matter what order
    if <-reqs[i].replyc != N+2*i {
      fmt.Println("fail at", i)
    } else {
      fmt.Println("Request ", i, " is ok!")
    }
  }
  fmt.Println("done")
}
```

Multiplex Server

This program has been completely described in the introduction above. It has the following basic parts:

- We define the `Request` struct at **line 4**. It has *two* integer fields `a` and `b`, and a channel of *integers* named `replyc`.
- We define a function type `binOp` for any function that takes *two* integers as input and returns an integer (see **line 9**).
- We define the `run()` function, which takes an object of type `binOp` and a pointer to the object of type `Request` (see its header at **line 10**). The `run()` function puts the result of the operation on the reply channel `replyc` of the request struct `req`.
- We define the `server()` function, which takes an object of type `binOp` and a pointer to the channel of type `Request` (see its header at **line 14**). The `server()` function is an infinite for loop that takes a request from the `service` channel (**line 16**), and starts a goroutine at **line 19** with that

service channel (line 16), and starts a goroutine at line 19 with that request.

At **line 30**, in `main()`, the server is started. `startServer()` (from **line 23** to **line 27**) starts the server in its own goroutine, and returns the request channel. From **line 31** to **line 39**, 100 requests are constructed and sent to the request channel. Finally, from **line 41** to **line 47**, we check the correctness of the result of each request, displaying messages in the ok and error cases.

Teardown: shutting down the server

In the previous version, the server does not perform a clean shutdown when main returns; it is forced to stop. To improve this, we can provide a second, `quit` channel to the server:

```
func startServer(op binOp) (service chan *Request, quit chan bool) {
  service = make(chan *Request)
  quit = make(chan bool)
  go server(op, service, quit)
  return service, quit
}
```

Then, the server function uses a `select` to choose between the `service` channel and the `quit` channel:

```
func server(op binOp, service chan *request, quit chan bool) {
  for {
    select {
      case req := <-service:
        go run(op, req)
      case <-quit:
        return
    }
  }
}
```

When a true value enters the quit channel, the server returns and terminates. In `main`, we change the following line:

```
adder, quit := startServer(func(a, b int) int { return a + b })
```

At the end of `main`, we place the line: `quit <- true`. The complete code can be found in the following program, with the same output as the above found in the following program, with the same output as the above:

```
package main
import "fmt"

type Request struct {
  a, b int
  replyc chan int // reply channel inside the Request
}

type binOp func(a, b int) int

func run(op binOp, req *Request) {
  req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request, quit chan bool) {
  for {
    select {
    case req := <-service:
      go run(op, req)
    case <-quit:
      return
    }
  }
}

func startServer(op binOp) (service chan *Request, quit chan bool) {
  service = make(chan *Request)
  quit = make(chan bool)
  go server(op, service, quit)
  return service, quit
}

func main() {
  adder, quit := startServer(func(a, b int) int { return a + b })
  const N = 100
  var reqs [N]Request
  for i := 0; i < N; i++ {
    req := &reqs[i]
    req.a = i
    req.b = i + N
    req.replyc = make(chan int)
    adder <- req
  }
  // checks:
  for i := N - 1; i >= 0; i-- { // doesn't matter what order
    if <-reqs[i].replyc != N+2*i {
      fmt.Println("fail at", i)
    } else {
      fmt.Println("Request ", i, " is ok!")
    }
  }
  quit <- true
  fmt.Println("done")
}
```

Teardown of Multiplex Server

Adding to the explanations of the previous example, the server has a `quit` channel, defined at **line 28**, which `startServer()` returns (see **line 26**). The `server()` function itself, now, contains a `select` clause inside the for (from **line 17** to **line 22**). If there is a request, run it in a separate goroutine (**line 19**). If a value has been put on the `quit` channel (**line 20**), return from the function and stop the program. The value *true* is put on the `quit` channel at **line 52**, so this is an ordered shutdown of our server.

It's essential to have a balance between clients and the server. A considerable number of requests may burden the server. In the next lesson, you'll learn how to limit the number of requests.