Modern System Administration with Go and Remote Procedure Calls (RPC)

KELSEY HIGHTOWER

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration and distributed systems.
kelsey.hightower@gmail.com

The datacenter is the new computer and it's time to look past the UNIX shell for building tools and utilities. While the programming environment outside the shell is different, the UNIX philosophy is still applicable: the tools and utilities you build should have a single purpose and support composition through clean inputs and outputs that allow users to build larger systems and custom workflows.

In the early days of UNIX, stdin and stdout streams allowed us to chain specialized tools and compose various workflows to suit our needs. For example, processing HTTP logs was as simple as running the following command:

```
$ grep 'html HTTP' /var/log/apache.log | uniq -c
```

What an easy way to build a data pipeline with very little code, but there are a few minor problems. The above solution only works for a single machine running specific versions of the UNIX utilities used in the pipeline. Running the same command on another flavor of UNIX is not guaranteed to work, or even worse, might yield different results. On top of everything else, the data between grep and uniq is often unstructured, which means ad hoc text parsing will be required to extract specific fields before data processing can continue.

To overcome these challenges, a programming language with a little more power, such as Go, can be used to model data using modern serialization formats such as JSON, which can improve interoperability between command line tools and services over a network. Expanding beyond a single system does introduce another set of challenges, such as invoking code over a network and handing failures without introducing too much overhead or complexity. One way of doing this is to use remote procedure calls (RPCs) between clients and servers.

Go and its robust standard library provide everything you need to build tools ranging from simple command line utilities to microservices that scale horizontally across a cluster of machines. The remainder of this article will focus on Go's native syscall interface, RPC mechanisms, and standard libraries you can use to ship robust sysadmin tools in little to no time.

Remote Procedure Calls (RPC)

When creating system administration tools that need to scale beyond a single host, RPC should be strongly considered. While there are other platforms for building services, I feel that RPC maps closest to task originated tools built by most system administrators and provides better performance by avoiding the unnecessary overhead required by protocols such as HTTP.

What Are Remote Procedure Calls?

As the name implies, RPC is about calling procedures (functions) remotely. RPC aims to ease the development of client-server applications by reusing native-language semantics and sharing code between both client and server. The learning curve for RPC is relatively low because there is no need to learn new ways of interacting with remote services outside of the native function calling conventions and error handling of the language you're programming in.

gls: A Distributed ls Service

To demonstrate the simplicity of Go and RPC for system administration tasks, we are going to reimplement the classic UNIX tool ls—with a twist. gls is a distributed tool for collecting file attributes for a given file system on a remote system.

The remainder of this article will walk you through building gls from the ground up. The source code for gls is hosted on GitHub [1], but I encourage you to type the commands by hand as you follow along—of course, this assumes you have a working Go installation [2].

The gls Package

At the heart of the gls server is the gls package, which holds common code shared by the gls server and client. Create the gls package directory under the GOPATH. We'll get into the details later, but type exactly what you see here for now:

```
$ mkdir -p $GOPATH/src/github.com/kelseyhightower/gls
```

Next, change into the gls package directory and save the following code snippet to a file named gls.go:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
$ vim gls.go
package gls

import (
	"os"
	"path/filepath"
)

type Files []File

type File struct {
	Name    string
	Size    int64
	Mode    string
	ModTime string
}

type Ls struct{}

func (ls *Ls) Ls(path *string, files *Files) error {
	root := *path
	err := filepath.Walk(*path, func(path string, info
os.FileInfo, err error) error {
		if err != nil {
			return err
		}

		file := File{
			info.Name(),
			info.Size(),
			info.Mode().String(),
			info.ModTime().Format("Jan _2 15:04"),
		}
		*files = append(*files, file)
		if info.IsDir() && path != root {
			return filepath.SkipDir
		}
		return nil
	})
	if err != nil {
		return err
	}
	return nil
}
```

Let's walk through the gls package to see what's happening.

First, we declare the gls package and import the `os` and `filepath` packages from the standard library:

```
package gls

import (
	"os"
	"path/filepath"
)
```

Next, we define two types, a File type, which holds file metadata, and a Files type, which holds a list of File objects:

```
type Files []File

type File struct {
	Name    string
	Size    int64
	Mode    string
	ModTime string
}
```

Finally, we define the Ls type for the sole purpose of defining the Ls method, which is responsible for gathering metadata from files under a specific directory path. For each file found, the name, size, permissions, and last modified time are captured and appended to a files list that will ultimately be returned to the caller.

```
type Ls struct{}

func (ls *Ls) Ls(path *string, files *Files) error {
	...
}
```

There are a couple of things to note here. First, Ls is a method and not a function. Second, Ls takes two arguments and returns a single error value. This is not arbitrary, but a requirement of Go's RPC support, which provides access to exported methods of an object over a network. Only methods that meet the following requirements can be exposed as RPC methods:

- The method's type is exported.
- The method is exported.
- The method has two arguments, both exported (or built-in) types.
- The method's second argument is a pointer.
- The method has return type error.

In the case of the gls package, the exported type is the Ls struct and the exported method is the Ls method. In order to meet the RPC requirements, the Ls method takes two arguments—the path to search for files, and a pointer to a files list to store file metadata—and returns a single error value.

In Go, this is not the typical way methods or functions are written. If the Ls method was not exposed as a RPC method, then it would have been written like this:

```
func (ls *Ls) Ls(path *string) (*Files, error)
```

The set of constraints imposed by Go's RPC support may seem odd at first, but when you think about it, it all makes sense. Requiring all RPC methods to have a similar signature, two arguments and a single return value, means it's much easier to encode and decode the communication between the client and server over the network.

Complex arguments can be expressed using a complex type. For example, if we wanted to include a pattern to limit which files are inspected by the Ls method, we could use the Options type in place of the original path argument.

```
type Options struct {
        Path    string
        Pattern string
}

func (ls *Ls) Ls(options *Options, files *Files) error {...}
```

The gls Server

With the gls package in place, it's time to create the gls server, which is responsible for exposing the Ls method from the gls package over RPC.

Start in the gls package directory created earlier:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
```

Create a new directory named server to hold the gls server binary:

```
$ mkdir server
```

Next, change into the server directory and save the following code snippet in a file named main.go.

```
$ cd server
$ vim main.go
package main

import (
        "log"
        "net"
        "net/rpc"

        "github.com/kelseyhightower/gls"
)

func main() {
        log.Println("Starting glsd..")
        ls := new(gls.Ls)
        rpc.Register(ls)

        l, err := net.Listen("tcp", "0.0.0.0:8080")
        if err != nil {
                log.Fatal(err)
        }

        for {
                conn, err := l.Accept()
                if err != nil {
                        log.Println(err)
                }
                rpc.ServeConn(conn)
                conn.Close()
        }
}
```

Let's quickly walk through what's going on here. First, we import a few packages from the Go standard library, including the `net/rpc` package, which provides support for exposing methods over RPC, and the gls package, which holds the definition of the Ls method.

Before we move on it's important to note the full name of the gls package: `github.com/kelseyhightower/gls`. This name was chosen to match where the gls package will be hosted on the Internet—on GitHub under the username kelseyhightower. Go's tooling has native support for working with packages hosted on remote repositories such as GitHub, and it's common to see packages named using this convention. The package name is important: because we cannot simply import "gls", we must use the complete import path where the gls package lives in relation to the GOPATH or our program will fail to compile. Learn more about Go's import semantics from the official docs [3].

With the gls package imported, we are ready to export the gls.Ls method by registering it using the `net/rpc` package.

```
ls := new(gls.Ls)
rpc.Register(ls)
```

The rest of the code creates a listener which binds to port 8080 on all available network interfaces and waits for RPC requests from clients.

The gls Client

The gls client is responsible for making requests to the gls server and printing the results to stdout. Create the gls client by running the following commands:

Start in the gls package directory created earlier:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
```

Create a new directory-named client to hold the gls client binary:

```
$ mkdir client
```

Next, change into the client directory and save the following code snippet in a file named main.go.

```
$ cd client
$ vim main.go
package main

import (
	"fmt"
	"log"
	"net/rpc"
	"os"

	"github.com/kelseyhightower/gls"
)

func main() {
	client, err := rpc.Dial("tcp", "127.0.0.1:8080")
	if err != nil {
		log.Fatal(err)
	}

	files := make(gls.Files, 0)
	err = client.Call("Ls.Ls", os.Args[1], &files)
	if err != nil {
		log.Fatal(err)
	}
	for _, f := range files {
		fmt.Printf("%s %10d %s %s\n", f.Mode, f.Size,
f.ModTime, f.Name)
	}
}
```

As with the gls server, we import a few packages from the standard library and the gls package, which in the case of the gls client provides access to the gls.Files type. Remember the gls.Files type is defined in the gls package:

```
package gls

type Files []File

type File struct {
	Name    string
	Size    int64
	Mode    string
	ModTime string
}
```

In order to communicate with the gls server, we need an RPC client and must establish an RPC connection:

```
client, err := rpc.Dial("tcp", "127.0.0.1:8080")
if err != nil {
	log.Fatal(err)
}
```

Before making the call to the remote Ls method, we must initialize an empty gls.Files slice to hold the results from the gls server:

```
files := make(gls.Files, 0)
```

Now we are ready to make our RPC call and print the results.

```
err = client.Call("Ls.Ls", flag.Args()[0], &files)
if err != nil {
	log.Fatal(err)
}

for _, f := range files {
	fmt.Printf("%s %10d %s %s\n", f.Mode, f.Size, f.ModTime,
f.Name)
}
```

Also, notice how we are using the first positional command line argument identified by `flag.Args()[0]` as the path argument to the Ls method. This will allow us to use the gls client binary like the standard ls UNIX command. For example, to list files in the tmp directory, we can run the gls client like this:

```
$ gls /tmp/
```

The string "/tmp/" will be stored at the first position of the slice returned by the flag.Args() function.

At this point, we are code complete and are ready to build and deploy the gls client and server.

Build and Deployment

Now that we have written and understand the code behind the gls client and server, let's turn our attention to the build and deployment process. Go is a compiled language, which means we must run our source code through a compiler before we can run it.

Building the gls client and server is as simple as running the following commands from the gls package directory:

```
$ cd $GOPATH/src/github.com/kelseyhightower/gls
```

Build the gls client using the `go build` command:

```
$ go build -o gls client/main.go
```

Build the gls server using the `go build` command:

```
$ go build -o glsd server/main.go
```

Running the above commands results in the following binaries:

```
gls
glsd
```

One thing to note about the gls and glsd binaries (and Go binaries in general) is that they are self-contained. This means each binary can be copied to a similar OS and architecture and be run without the need to install Go on the target system. In a future article, I'll cover how cross-compiling in Go works, which allows you to develop applications on one platform (Linux) and compile them to run on another (Windows).

You are now ready to launch the gls server:

```
$ ./glsd
2015/12/23 07:50:06 Starting glsd..
```

At this point the gls server is ready to accept RPC requests on port 8080.

Open a new terminal window and use gls client to get a directory listing of your home directory from the gls server:

```
$ ./gls ~/
drwx------   170  Nov 28 20:23  Applications
drwxr-xr-x   102  Dec 20 01:52  Desktop
drwx------  1122  Dec 20 11:57  Documents
drwx------   340  Dec 22 11:30  Downloads
...
```

The gls client is hardcoded to communicate with the gls server over localhost (127.0.0.1) on port 8080. This is being done because the gls server is not protected by any form of authentication or encryption such as TLS. In a future article, we will revisit extending the gls client and server to support encryption, authentication, and communication over any IP/port combination using a set of command line flags.

Conclusion

The way we think about computers is changing, and this is the perfect time to rethink the way we approach systems programming in general. Go has native RPC support and low-level syscall functionality, which allows us to build enhanced versions of UNIX classics such as ls or new tools that perform tasks that meet the challenges of today while leveraging the timeless UNIX philosophy that has defined computing for decades.

Resources

[1] GitHub for the sources in this column: https://github.com/kelseyhightower/gls.

[2] Installing Go: https://golang.org/doc/install.

[3] Docs for understanding package paths: https://golang.org/doc/code.html#PackagePaths.