Web sockets

Web sockets are designed to answer a common problem with web systems: the server is unable to initiate or push content to a user agent such as a browser. Web sockets allow a full duplex connection to be established to allow this. Go has nearly complete support for them.

Introduction

The websockets model will change for release r61. This describes the new package, not the package in r60 and earlier. If you do not have r61, at the time of writing, use `hg pull; hg update weekly` to download it.

The standard model of interaction between a web user agent such as a browser and a web server such as Apache is that the user agent makes HTTP requests and the server makes a single reply to each one. In the case of a browser, the request is made by clicking on a link, entering a URL into the address bar, clicking on the forward or back buttons, etc. The response is treated as a new page and is loaded into a browser window.

This traditional model has many drawbacks. The first is that each request opens and closes a new TCP connection. HTTP 1.1 solved this by allowing persistent connections, so that a connection could be held open for a short period to allow for multiple requests (e.g. for images) to be made on the same server.

While HTTP 1.1 persistent connections alleviate the problem of slow loading of a page with many graphics, it does not improve the interaction model. Even with forms, the model is still that of submitting the form and displaying the response as a new page. JavaScript helps in allowing error checking to be performed on form data before submission, but does not change the model.

AJAX (Asynchronous JavaScript and XML) made a significant advance to the user interaction model. This allows a browser to make a request and just use the response to update the display in place using the HTML Document Object Model (DOM). But again the interaction model is the same. AJAX just affects how the browser manages the returned pages. There is no explicit extra support in Go for AJAX, as none is needed: the HTTP server just sees an ordinary HTTP POST request with possibly some XML or JSON data, and this can be dealt with using techniques already discussed.

All of these are still browser to server communication. What is missing is server initiated communications to the browser. This can be filled by Web sockets: the browser (or any user agent) keeps open a long-lived TCP connection to a Web sockets server. The TCP connection allows either side to send arbitrary packets, so any application protocol can be used on a web socket.

How a websocket is started is by the user agent sending a special HTTP request that says "switch to web sockets". The TCP connection underlying the HTTP request is kept open, but both user agent and server switch to using the web sockets protocol instead of getting an HTTP response and closing the socket.

Note that it is still the browser or user agent that initiates the Web socket connection. The browser does not run a TCP server of its own. While the specification is complex, the protocol is designed to be fairly easy to use. The client opens an HTTP connection and then replaces the HTTP protocol with its own WS protocol, re-using the same TCP connection.

Web socket server

A web socket server starts off by being an HTTP server, accepting TCP conections and handling the HTTP requests on the TCP connection. When a request comes in that switches that connection to a being a web socket connection, the protocol handler is changed from an HTTP handler to a WebSocket handler. So it is only that TCP connection that gets its role changed: the server continues to be an HTTP server for other requests, while the TCP socket underlying that one connection is used as a web socket.

One of the simple servers HHTP we discussed in Chapter 8: HTTP registered varous handlers such as a file handler or a function handler. To handle web socket requests we simply register a different type of handler - a web socket handler. Which handler the server uses is based on the URL pattern. For example, a file handler might be registered for "/", a function handler for "/cgi-bin/..." and a web sockets handler for "/ws".

An HTTP server that is only expecting to be used for web sockets might run by

```
func main() {
        http.Handle("/", websocket.Handler(WSHandler))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}
```

A more complex server might handle both HTTP and web socket requests simply by adding in more handlers.

The Message object

HTTP is a stream protocol. Web sockets are frame-based. You prepare a block of data (of any size) and send it as a set of frames. Frames can contain either strings in UTF-8 encoding or a sequence of bytes.

The simplest way of using web sockets is just to prepare a block of data and ask the Go websocket library to package it as a set of frame data, send them across the wire and receive it as the same block. The `websocket`package contains a convenience object `Message` to do just that. The `Message` object has two methods, `Send` and `Receive` which take a websocket as first parameter. The second parameter is either the address of a variable to store data in, or the data to be sent. Code to send string data would look like

```
msgToSend := "Hello"
err := websocket.Message.Send(ws, msgToSend)

var msgToReceive string
err := websocket.Message.Receive(conn, &msgToReceive)
```

Code to send byte data would look like

```
dataToSend := []byte{0, 1, 2}
err := websocket.Message.Send(ws, dataToSend)

var dataToReceive []byte
err := websocket.Message.Receive(conn, &dataToReceive)
```

An echo server to send and receive string data is given below. Note that in web sockets either side can initiate sending of messages, and in this server we send messages from the server to a client when it connects (send/receive) instead of the more normal receive/send server. The server is

```
/* EchoServer
 */
package main

import (
        "fmt"
        "net/http"
        "os"
        // "io"
        "code.google.com/p/go.net/websocket"
)

func Echo(ws *websocket.Conn) {
        fmt.Println("Echoing")

        for n := 0; n < 10; n++ {
                msg := "Hello  " + string(n+48)
                fmt.Println("Sending to client: " + msg)
                err := websocket.Message.Send(ws, msg)
                if err != nil {
                        fmt.Println("Can't send")
                        break
                }

                var reply string
                err = websocket.Message.Receive(ws, &reply)
                if err != nil {
                        fmt.Println("Can't receive")
                        break
                }
                fmt.Println("Received back from client: " + reply)
        }
}

func main() {

        http.Handle("/", websocket.Handler(Echo))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

A client that talks to this server is

```
/* EchoClient
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "io"
        "os"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "ws://host:port")
		os.Exit(1)
	}
	service := os.Args[1]

	conn, err := websocket.Dial(service, "", "http://localhost")
	checkError(err)
	var msg string
	for {
		err := websocket.Message.Receive(conn, &msg)
		if err != nil {
			if err == io.EOF {
				// graceful shutdown by server
				break
			}
			fmt.Println("Couldn't receive msg " + err.Error())
			break
		}
		fmt.Println("Received from server: " + msg)
		// return the msg
		err = websocket.Message.Send(conn, msg)
		if err != nil {
			fmt.Println("Coduln't return msg")
			break
		}
	}
	os.Exit(0)
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

The url for the client running on the same machine as the server should be `ws://localhost:12345/`

The JSON object

It is expected that many websocket clients and servers will exchange data in JSON format. For Go programs this means that a Go object will be marshalled into JSON format as described in Chapter 4: Serialisation and then sent as a UTF-8 string, while the receiver will read this string and unmarshal it back into a Go object.

The `websocket` convenience object `JSON` will do this for you. It has methods `Send` and `Receive` for sending and receiving data, just like the `Message` object.

A client that sends a `Person` object in JSON format is

```
/* PersonClientJSON
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "os"
)

type Person struct {
        Name   string
        Emails []string
}

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "ws://host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := websocket.Dial(service, "",
                "http://localhost")
        checkError(err)

        person := Person{Name: "Jan",
                Emails: []string{"ja@newmarch.name",
"jan.newmarch@gmail.com"},
        }

        err = websocket.JSON.Send(conn, person)
        if err != nil {
                fmt.Println("Couldn't send msg " + err.Error())
        }
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

and a server that reads it is

```
/* PersonServerJSON
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "net/http"
        "os"
)

type Person struct {
        Name   string
        Emails []string
}

func ReceivePerson(ws *websocket.Conn) {
        var person Person
        err := websocket.JSON.Receive(ws, &person)
        if err != nil {
                fmt.Println("Can't receive")
        } else {

                fmt.Println("Name: " + person.Name)
                for _, e := range person.Emails {
                        fmt.Println("An email: " + e)
                }
        }
}

func main() {

        http.Handle("/", websocket.Handler(ReceivePerson))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The Codec type

The `Message` and `JSON` objects are both instances of the type `Codec`. This type is defined by

```
type Codec struct {
    Marshal   func(v interface{}) (data []byte, payloadType byte, err error)
    Unmarshal func(data []byte, payloadType byte, v interface{}) (err error)
}
```

The type `Codec` implements the `Send` and `Receive` methods used earlier.

It is likely that websockets will also be used to exchange XML data. We can build an XML `Codec` object by wrapping the XML marshal and unmarshal methods discussed in Chapter 12: XML to give a suitable `Codec`object.

We can create a `XMLCodec` package in this way:

```go
package xmlcodec

import (
        "encoding/xml"
        "code.google.com/p/go.net/websocket"
)

func xmlMarshal(v interface{}) (msg []byte, payloadType byte, err error) {
        //buff := &bytes.Buffer{}
        msg, err = xml.Marshal(v)
        //msgRet := buff.Bytes()
        return msg, websocket.TextFrame, nil
}

func xmlUnmarshal(msg []byte, payloadType byte, v interface{}) (err error) {
        // r := bytes.NewBuffer(msg)
        err = xml.Unmarshal(msg, v)
        return err
}

var XMLCodec = websocket.Codec{xmlMarshal, xmlUnmarshal}
```

We can then serialise Go objects such as a `Person` into an XML document and send it from a client to a server by

```go
/* PersonClientXML
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "os"
        "xmlcodec"
)

type Person struct {
        Name   string
        Emails []string
}

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "ws://host:port")
                os.Exit(1)
        }
        service := os.Args[1]
        conn, err := websocket.Dial(service, "", "http://localhost")
        checkError(err)

        person := Person{Name: "Jan",
                Emails: []string{"ja@newmarch.name",
"jan.newmarch@gmail.com"},
        }

        err = xmlcodec.XMLCodec.Send(conn, person)
        if err != nil {
                fmt.Println("Couldn't send msg " + err.Error())
        }
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

A server which receives this and just prints information to the console is

```
/* PersonServerXML
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "net/http"
        "os"
        "xmlcodec"
)

type Person struct {
        Name   string
        Emails []string
}

func ReceivePerson(ws *websocket.Conn) {
        var person Person
        err := xmlcodec.XMLCodec.Receive(ws, &person)
        if err != nil {
                fmt.Println("Can't receive")
        } else {

                fmt.Println("Name: " + person.Name)
                for _, e := range person.Emails {
                        fmt.Println("An email: " + e)
                }
        }
}
```

```
func main() {

        http.Handle("/", websocket.Handler(ReceivePerson))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Web sockets over TLS

A web socket can be built above a secure TLS socket. We discussed in Chapter 8: HTTP how to use a TLS socket using the certificates from Chapter 7: Security. That is used unchanged for web sockets. that is, we use`http.ListenAndServeTLS` instead of `http.ListenAndServe`.

Here is the echo server using TLS

```
/* EchoServer
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "net/http"
        "os"
)

func Echo(ws *websocket.Conn) {
        fmt.Println("Echoing")

        for n := 0; n < 10; n++ {
                msg := "Hello  " + string(n+48)
                fmt.Println("Sending to client: " + msg)
                err := websocket.Message.Send(ws, msg)
                if err != nil {
                        fmt.Println("Can't send")
                        break
                }

                var reply string
                err = websocket.Message.Receive(ws, &reply)
                if err != nil {
                        fmt.Println("Can't receive")
                        break
                }
                fmt.Println("Received back from client: " + reply)
        }
}

func main() {

        http.Handle("/", websocket.Handler(Echo))
        err := http.ListenAndServeTLS(":12345", "jan.newmarch.name.pem",
                "private.pem", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The client is the same echo client as before. All that changes is the url, which uses the "wss" scheme instead of the "ws" scheme:

```
  EchoClient wss://localhost:12345/
```

Conclusion

The web sockets standard is nearing completion and no major changes are anticipated. This will allow HTTP user agents and servers to set up bi-directional socket connections and should make certain interaction styles much easier. Go has nearly complete support for web sockets.

Source: http://jan.newmarch.name/go/websockets/chapter-websockets.html