Lab 2: MQTT

Elias Berglin

15th December 2021

1 Pre-defined messages

As I am building the broker part of the MQTT system there will be a lot of fixed responses. The complete code can be found in appendix A. The two most important responses will be connection ACK and ping ACK. We need these to connect and keep the connection alive. In figure 1 we can see that all these functions do is create a byte array containing the bytes needed to respond.

```
1  func createConnectionAccept() []byte {
2   var message []byte
3   message = append(message, byte(0b00100000))
4   message = append(message, byte(0b00000010))
5   message = append(message, byte(0x0))
6   message = append(message, byte(0x0))
7   return message
8  }
9  func createPingAck() []byte {
10  var message []byte
11  message = append(message, byte(0b11010000))
12  message = append(message, byte(0x0))
13  return message
14 }
```

Figure 1: Go code for create ACK for connect and ping

2 Parsing incoming message

In figure 2 is the code for parsing a subscribe message. It is important to keep in mind that a subscribe message can contain multiple topics and thus this needs to be checked. The function returns three values. First it returns if the subscription was successful, so we can pass that data to the ACK. The second return parameter is the identifier for the message that is later passed to subscribe ACK. The last return is a list of all the topics that the client subscribed to. The way that I parse the message here is I pop the number of bytes I want from the array. Go does not provide a simple way to do this. If we look at row 9 in figure 2 we can see that I have an assignment of two variables. The variable l in this case is the bytes representing the length of the topic and body is the message body and is overwritten. On the right side of the equals sign I first assign the two bytes to l and then assign the remaining bytes to body, and thus I have popped the front of the array.

```
1  func parseSubscribe(message []byte, c *net.Conn, id string) (bool,
       []byte, []string) {
2   body := message[2:]
3   var l []byte
4   var subscribe string
5   var qos int
6   var subs []string
7
8   for len(body) > 0 {
9    l, body = body[:2], body[2:]
10   subLen := binary.BigEndian.Uint16(l)
11   if len(body) < int(subLen) {
12    return false, message[:2], nil
13   }
14   subscribe, body = string(body[:subLen]), body[subLen:]
15   if len(body) == 0 {
16    return false, message[:2], nil
17   }
18   addSubscription(c, subscribe)
19   subs = append(subs, subscribe)
20   qos, body = int((body[0] & 0b00000011)), body[1:]
21   fmt.Println(DEVIDER)
22   fmt.Println(id+" subscribed to: "+subscribe+" \nWith QOS of:",
         qos)
23   fmt.Println(DEVIDER)
24   fmt.Println()
25  }
26  return true, message[:2], subs
27 }
```

Figure 2: Go code for parsing a subscribe message

3 Broadcaster

To make sure all the connections get the messages they subscribed to I implemented a broadcaster. This function is responsible for delivering published data to all the subscribed clients. To help with this I have a global map with string keys and an array of pointers to connection as value. To know when to send data the broadcaster listens to a channel. A channel in go is a way for threads to communicate with one another. When a publish is received the packet along with information is sent on the channel and is picked up by the broadcaster. When the broadcaster receives the message it can get the list of pointers from the global map and iterate through them and send the packet to each subscriber. The broadcaster code can be found in figure 3.

```
1  func broadcaster(ch chan BroadCastMessage) {
2   for {
3    message := <-ch
4    connections := subscriptions[message.Topic]
5    if len(connections) <= 0 {
6     continue
7    }
8    fmt.Println(DEVIDER)
9    fmt.Println("Broadcasting message to: ",
10    message.Topic, "\ncontaining ", message.Message)
11   fmt.Println(DEVIDER)
12   fmt.Println()
13   for _, c := range subscriptions[message.Topic] {
14    (*c).Write(message.Packet)
15   }
16  }
17 }
```

Figure 3: Go code for the broadcaster function

4 Main function

The main function is responsible to set up the broadcast function, initiate the global video and accept incoming connections. The code can be found in figure 4. In row 8 I initialize the channel that is used by the broadcaster and on line 9 I use the go keyword to run the broadcaster function in a new thread. I use the same keyword on line 22 where I hand over the connection to a new thread and then the function goes back to handle new connections.

```
1  func main() {
2   deviderLen := 50
3   for i := 0; i < deviderLen; i++ {
4    DEVIDER += "-"
5   }
6   subscriptions = make(map[string][]*net.Conn)
7   lastValue = make(map[string][]byte)
8   ch := make(chan BroadCastMessage)
9   go broadcaster(ch)
10  port := ":1883"
11  s, err := net.Listen("tcp", port)
12  if err != nil {
13   panic(err)
14  }
15  defer s.Close()
16
17  for {
18   c, err := s.Accept()
19   if err != nil {
20    panic(err)
21   }
22   go acceptMessage(&c, ch)
23  }
24 }
```

Figure 4: The main go function

5 Flowchart

Figure 5 represents the flowchart for the program.

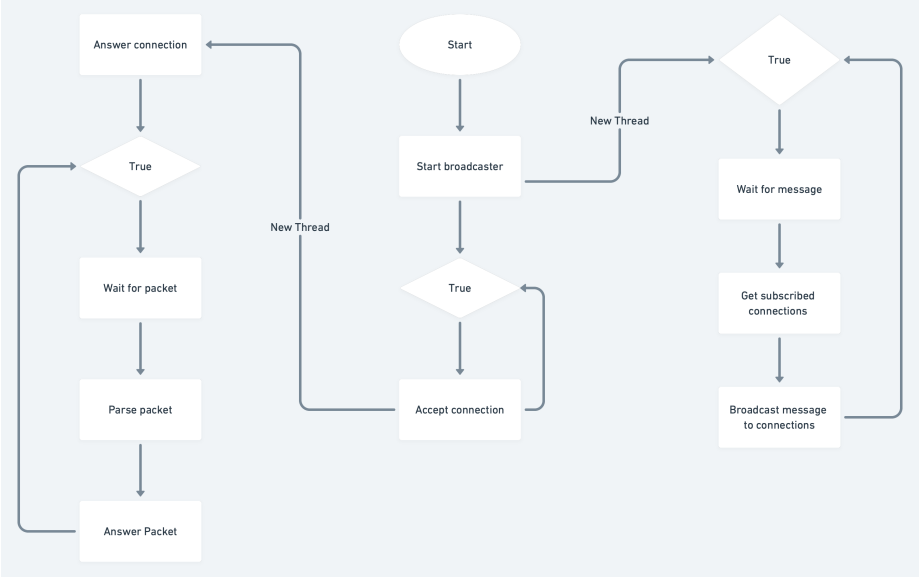

Figure 5: Flow of the MQTT broker

Appendix A Code

```
package main

import (
 "encoding/binary"
 "fmt"
 "net"
 "strconv"
)

var subscriptions map[string][]*net.Conn
var lastValue map[string][]byte
var DEVIDER string

type BroadCastMessage struct {
 Topic   string
 Message string
 Packet  []byte
}

func broadcaster(ch chan BroadCastMessage) {
 for {
  message := <-ch
  connections := subscriptions[message.Topic]
  if len(connections) <= 0 {
   continue
  }
  fmt.Println(DEVIDER)
  fmt.Println("Broadcasting message to: ",
   message.Topic, "\ncontaining ", message.Message)
  fmt.Println(DEVIDER)
  fmt.Println()
  for _, c := range subscriptions[message.Topic] {
   (*c).Write(message.Packet)
  }
 }
}

func addSubscription(c *net.Conn, filter string) {
 if subscriptions[filter] == nil {
  subscriptions[filter] = make([]*net.Conn, 0)
 }
 subscriptions[filter] = append(subscriptions[filter], c)
}

func removeSubscription(c *net.Conn, filter string) {
 conns := subscriptions[filter]
 var newConns []*net.Conn

 for _, conn := range conns {
  if conn != c {
   newConns = append(newConns, conn)
  }
 }
 subscriptions[filter] = newConns
}
```

```
56
57 func removeAllSubs(c *net.Conn) {
58  for topic, conns := range subscriptions {
59   var newConns []*net.Conn
60   for _, conn := range conns {
61    if conn != c {
62     newConns = append(newConns, conn)
63    }
64   }
65   subscriptions[topic] = newConns
66  }
67 }
68
69 func createConnectionAccept() []byte {
70  var message []byte
71  message = append(message, byte(0b00100000))
72  message = append(message, byte(0b00000010))
73  message = append(message, byte(0x0))
74  message = append(message, byte(0x0))
75  return message
76 }
77 func createPingAck() []byte {
78  var message []byte
79  message = append(message, byte(0b11010000))
80  message = append(message, byte(0x0))
81  return message
82 }
83
84 func createSubAck(identifier []byte, success bool) []byte {
85  var message []byte
86  message = append(message, byte(0b10010000))
87  message = append(message, byte(0b00000011))
88  message = append(message, identifier...)
89  if success {
90   message = append(message, byte(0x00))
91  } else {
92   message = append(message, byte(0x80))
93  }
94
95  return message
96 }
97
98 func createUnsubAck(identifier []byte) []byte {
99  var message []byte
100  message = append(message, byte(0b10110000))
101  message = append(message, byte(0x2))
102  message = append(message, identifier...)
103  return message
104 }
105
106 func parseSubscribe(message []byte, c *net.Conn, id string) (bool,
     []byte, []string) {
107  body := message[2:]
108  var l []byte
109  var subscribe string
110  var qos int
111  var subs []string
```

```
112
113  for len(body) > 0 {
114   l, body = body[:2], body[2:]
115   subLen := binary.BigEndian.Uint16(l)
116   if len(body) < int(subLen) {
117    return false, message[:2], nil
118   }
119   subscribe, body = string(body[:subLen]), body[subLen:]
120   if len(body) == 0 {
121    return false, message[:2], nil
122   }
123   addSubscription(c, subscribe)
124   subs = append(subs, subscribe)
125   qos, body = int((body[0] & 0b00000011)), body[1:]
126   fmt.Println(DEVIDER)
127   fmt.Println(id+" subscribed to: "+subscribe+" \nWith QOS of:",
          qos)
128   fmt.Println(DEVIDER)
129   fmt.Println()
130  }
131  return true, message[:2], subs
132 }
133
134 func parseUnsubscribe(message []byte, c *net.Conn, id string) []
      byte {
135  body := message[2:]
136  var l []byte
137  var unSub string
138  for len(body) > 0 {
139   l, body = body[:2], body[2:]
140   subLen := binary.BigEndian.Uint16(l)
141   unSub, body = string(body[:subLen]), body[subLen:]
142   removeSubscription(c, unSub)
143   fmt.Println(DEVIDER)
144   fmt.Println(id + " unsubscribed from: " + unSub)
145   fmt.Println(DEVIDER)
146   fmt.Println()
147  }
148  return message[:2]
149 }
150
151 func parsePublish(message []byte, retain bool, id string) (string,
      string) {
152  data := ""
153  topic := ""
154  var topicLenBytes []byte
155
156  topicLenBytes, message = message[:2], message[2:]
157  topicLength := binary.BigEndian.Uint16(topicLenBytes)
158  topic, message = string(message[:topicLength]), message[
       topicLength:]
159
160  data = string(message)
161  fmt.Println(DEVIDER)
162  fmt.Println(id + " publshed: ")
163  fmt.Println("Topic: ", topic)
164  fmt.Println("Data: ", data)
```

```
 fmt.Println("Will retain: ", retain)
 fmt.Println(DEVIDER)
 fmt.Println()

 return data, topic
}

func handleConnection(c *net.Conn, ch chan BroadCastMessage, id
    string) {
 fmt.Println(DEVIDER)
 fmt.Println("Accepting connection from: " + id)
 fmt.Println(DEVIDER)
 fmt.Println()
 conAck := createConnectionAccept()
 (*c).Write(conAck)
 for {
  constHEAD := make([]byte, 2)
  (*c).Read(constHEAD)
  messageType := int((constHEAD[0] & 0b11110000) >> 4)
  switch messageType {
  case 12: //Ping
   fmt.Println(DEVIDER)
   fmt.Println("Answering Ping Request from: " + id)
   fmt.Println(DEVIDER)
   fmt.Println()
   pingAck := createPingAck()
   (*c).Write(pingAck)
   break
  case 14: // Disconect
   fmt.Println(DEVIDER)
   fmt.Println("Disconnecting: " + id)
   fmt.Println(DEVIDER)
   fmt.Println()
   removeAllSubs(c)
   (*c).Close()
   return
  case 8: // Subscribe
   remainder := int(constHEAD[1])
   messageArr := make([]byte, remainder)
   (*c).Read(messageArr)
   success, code, subs := parseSubscribe(messageArr, c, id)
   subAck := createSubAck(code, success)
   for _, s := range subs {
    if len(lastValue[s]) == 0 {
     continue
    }
    (*c).Write(lastValue[s])
   }
   (*c).Write(subAck)
   break
  case 10: // Unsubscribe
   remainder := int(constHEAD[1])
   messageArr := make([]byte, remainder)
   (*c).Read(messageArr)
   (*c).Write(createUnsubAck(parseUnsubscribe(messageArr, c, id)))
   break
  case 3: // Publish
```

```
221     remainder := int(constHEAD[1])
222     messageArr := make([]byte, remainder)
223     retain := int(constHEAD[0] & 0b00000001)
224     qos := int((constHEAD[0] & 0b00000110) >> 1)
225     if qos != 0 {
226      fmt.Println("Unsuported QoS type")
227      removeAllSubs(c)
228      (*c).Close()
229      return
230     }
231     (*c).Read(messageArr)
232     data, topic := parsePublish(messageArr, retain != 0, id)
233     ch <- BroadCastMessage{
234      Message: data,
235      Topic:   topic,
236      Packet:  append(constHEAD, messageArr...),
237     }
238     if retain != 0 {
239      lastValue[topic] = append(lastValue[topic],
240       append(constHEAD, messageArr...)...)
241     }
242     break
243    default:
244     fmt.Println(DEVIDER)
245     fmt.Println("Got message type: " + strconv.Itoa(messageType))
246     fmt.Println("This is not handled by brooker")
247     if messageType == 0 {
248      fmt.Println("Message type 0 could be a forced disconect")
249     }
250     fmt.Println(DEVIDER)
251     fmt.Println()
252     removeAllSubs(c)
253     (*c).Close()
254     return
255    }
256   }
257  }
258
259  func acceptMessage(c *net.Conn, ch chan BroadCastMessage) {
260   constHEAD := make([]byte, 2)
261   (*c).Read(constHEAD)
262   packetType := int((constHEAD[0] & 0b11110000) >> 4)
263   remainder := int(constHEAD[1])
264   if packetType != 1 {
265    panic("Not a connection")
266   }
267   message := make([]byte, remainder)
268   (*c).Read(message)
269   mqttStringLen := binary.BigEndian.Uint16(message[:2])
270   if mqttStringLen != 4 {
271    (*c).Close()
272    panic("Wrong protocoll")
273   }
274   mqttString := string(message[2:6])
275   if mqttString != "MQTT" {
276    (*c).Close()
277    panic("Wring protocoll")
```

```
278  }
279
280  version := message[6]
281  if version != 4 {
282   panic("Protocoll not supported")
283  }
284  var payload []byte
285  if len(message) > 10 {
286   payload = message[10:]
287  }
288  id := "Null Identifier"
289  if len(payload) > 0 {
290   id = string(payload)
291  }
292  handleConnection(c, ch, id)
293 }
294
295 func main() {
296  deviderLen := 50
297  for i := 0; i < deviderLen; i++ {
298   DEVIDER += "-"
299  }
300  subscriptions = make(map[string][]*net.Conn)
301  lastValue = make(map[string][]byte)
302  ch := make(chan BroadCastMessage)
303  go broadcaster(ch)
304  port := ":1883"
305  s, err := net.Listen("tcp", port)
306  if err != nil {
307   panic(err)
308  }
309  defer s.Close()
310
311  for {
312   c, err := s.Accept()
313   if err != nil {
314    panic(err)
315   }
316   go acceptMessage(&c, ch)
317  }
318 }
```