IT/Go 언어 (Golang)

[Golang] 자체 MQTT 브로커 서버 구현 - 2

공대생의 차고 2020. 2. 26. 18:05

기능 상세 설명

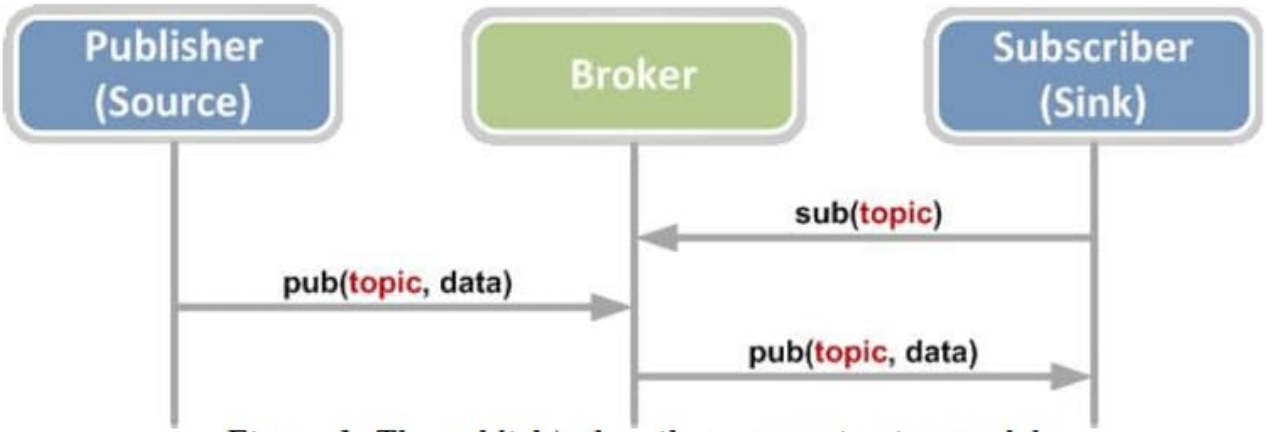


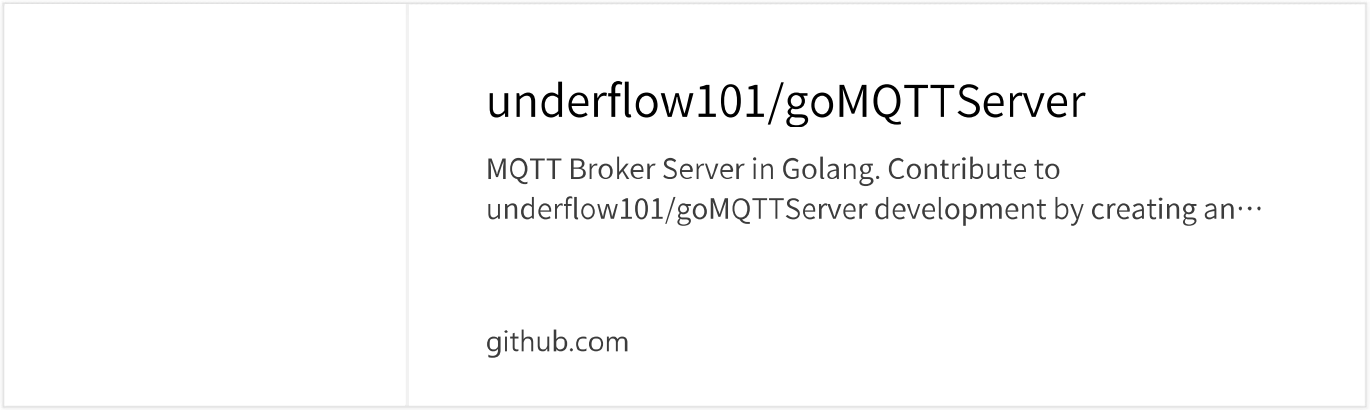
Figure 1: The publish/subscribe communication model

[Golang] 자체 MQTT 브로커 서버 구현 - 1

MQTT Broker 서버 구현 MQTT 브로커 서버 구현에 앞서, 만약 이론적 배경이 궁금하다면 아래 포스트를 먼저 읽고 오는 것을 추천한다…

underflow101.tistory.com

위 1편에서 설명된 자체 MQTT Broker 서버의 기능 상세 설명을 하려 한다.
이를 위한 소스 코드는 아래 깃허브에서 확인 가능하다.



main.go

최상단 트리에 있는 소스 코드는 실질적으로 main.go 밖에 없다. (나머지는 go.mod나 .gitignore 같은 것들)

main.go의 경우 소스 코드의 길이가 매우 짧다.

```
runtime.GOMAXPROCS(runtime.NumCPU())
```

우선은 위와 같이 runtime의 모든 CPU를 최대 자원으로 끌어와 사용하며, broker에 os.Signal형 chan 자료형이 도착할 때까지 기다린다.
즉, kill signal이 올 때까지 broker를 돌리는 것이 전부이다.

그렇다면 실제 MQTT broker는 broker 폴더 내에 있는 것이 main이라고 할 수 있을 것이다.

broker/broker.go

보통 Golang으로 만든 MQTT Broker는 Benchmark에서 매우 높은 점수를 받게 되는데, 그 비밀은 Go routine의 동시성에 숨어있다.
해당 broker/broker.go 파일에서 보면,

```
func (b *Broker) Start() {
        if b == nil {
                log.Error("broker is null")
                return
        }
```

```
        if b.config.HTTPPort != "" {
                go InitHTTPMoniter(b)
        }

        //listen clinet over tcp
        if b.config.Port != "" {
                go b.StartClientListening(false)
        }

        //listen for cluster
        if b.config.Cluster.Port != "" {
                go b.StartClusterListening()
        }

        //listen for websocket
        if b.config.WsPort != "" {
                go b.StartWebsocketListening()
        }

        //listen client over tls
        if b.config.TlsPort != "" {
                go b.StartClientListening(true)
        }

        //connect on other node in cluster
        if b.config.Router != "" {
                go b.processClusterInfo()
                b.ConnectToDiscovery()
        }

}
```

아예 브로커의 시작부터 Go routine을 활용하여 동시성을 극대화하는 방식을 취한다는 것을 알 수 있다.
이뿐만이 아니라,

```
func (b *Broker) StartClientListening(Tls bool) {
        var hp string
        var err error
        var l net.Listener
        if Tls {
                hp = b.config.TlsHost + ":" + b.config.TlsPort
                l, err = tls.Listen("tcp", hp, b.tlsConfig)
```

```
                log.Info("Start TLS Listening client on ", zap.String("hp", hp))
        } else {
                hp := b.config.Host + ":" + b.config.Port
                l, err = net.Listen("tcp", hp)
                log.Info("Start Listening client on ", zap.String("hp", hp))
        }
        if err != nil {
                log.Error("Error listening on ", zap.Error(err))
                return
        }
        tmpDelay := 10 * ACCEPT_MIN_SLEEP
        for {
                conn, err := l.Accept()
                if err != nil {
                        if ne, ok := err.(net.Error); ok && ne.Temporary() {
                                log.Error("Temporary Client Accept Error(%v),
sleeping %dms",
                                        zap.Error(ne), zap.Duration("sleeping",
tmpDelay/time.Millisecond))
                                time.Sleep(tmpDelay)
                                tmpDelay *= 2
                                if tmpDelay > ACCEPT_MAX_SLEEP {
                                        tmpDelay = ACCEPT_MAX_SLEEP
                                }
                        } else {
                                log.Error("Accept error: %v", zap.Error(err))
                        }
                        continue
                }
                tmpDelay = ACCEPT_MIN_SLEEP
                go b.handleConnection(CLIENT, conn)

        }
}
```

Start() 함수에서 Call하는 StartClientListening(Tls bool) 함수 내부에서도 go b.handleConnection(CLIENT, conn)을 통해 Go routine을 돌린다.

broker/broker.go 소스 코드에서 중요한 함수 중에 하나로는 아래 함수가 있다.

```
func (b *Broker) handleConnection(typ int, conn net.Conn)
```

위 func (b *Broker) handleConnection(typ int, conn net.Conn) 함수에서는 MQTT로 전송 받은 packet(MQTT message)를 받아 해당 패킷(메시지)이 nil인지, 혹은 깨진 패킷인지를 검사하고, 해당

패킷에 오류가 없다면 패킷 내부에 있는 clientID와 msg를 반환한다.

이 때 테스팅, 혹은 디버깅을 위해 (로그를 제외한) 두 개의 콘솔 메시지가 뜨게 되는데,
첫 번째 콘솔 메시지는 아래와 같다.

```
$ Terminal

$ 2. message get:  CONNECT: dup: false qos: 0 retain: false rLength: 50
protocolversion: 4 protocolname: MQTT cleansession: true willflag: false WillQos:
0 WillRetain: false Usernameflag: true Passwordflag: true keepalive: 15 clientId:
ESP32Client-6ae3 willtopic:  willmessage:  Username: ESP32client Password: Connect
$ {"level":"info","timestamp":"2020-02-
26T13:46:36.747+0900","logger":"broker","caller":"broker/broker.go:279","msg":"rea
d connect from ","clientID":"ESP32Client-6ae3"}
```

이것은 사용자가 ESP32나 Arduino를 통하여,

```
// Arduino IDE Source Code

client.connect(clientId.c_str(),"ESP32client", "Connect")
```

위 코드로 접속을 시도하면 뜨게 되는 메시지이다.
필자는 이를 Connection 메시지라고 규정하여 "2. message get: (msg)" 로 콘솔창에서 확인할 수 있게 코드를 구성했다.

위 메시지에서는 연결된 디바이스의 유저 네임, 프로토콜, QoS, 등이 있다.

두 번째 콘솔 메시지는 아래와 같다.

```
$ Terminal

$ 1. message get: PublishPacket
$ PUBLISH: dup: false qos: 0 retain: false rLength: 42 topicName: test/test
MessageID: 0 payload: Hello, MQTT World! I'm Bear! :)
$ 1. message get: PublishPacket
$ PUBLISH: dup: false qos: 0 retain: false rLength: 28 topicName: test/test
MessageID: 0 payload: Hey, I'm Bear! :)
```

위 메시지는 사용자가 ESP32나 Arduino를 통하여,

```
// Arduino IDE Source Code

client.publish(mqtt_topic, "Hello, MQTT World! I'm Bear! :)");
```

```
delay(2000);
client.publish(mqtt_topic, "Hey, I'm Bear! :)");
```

위 코드로 메시지를 발행(publish)하면 MQTT 브로커가 받아서 콘솔창에 띄워주게 되는 메시지이다. 해당 메시지를 필자는 main 메시지라고 규정하여 "1. message get: (TypeOf(msg.packet).String()) ₩n msg.packet.String()"으로 콘솔창에서 확인할 수 있게 코드를 구성했으며,

위 메시지를 통해 알 수 있는 정보는 QoS가 0인지 1인지, 토픽명은 무엇인지, 그리고 payload 뒤에 메시지 내용이 있다.

이외에도,

```
func (b *Broker) removeClient(c *client)
```

함수로 클라이언트를 삭제시킨다던지,

```
func (b *Broker) PublishMessage(packet *packets.PublishPacket)
```

발행받았던 MQTT 메시지를 subscriber(구독자) 디바이스들에 다시 발행한다던지 하는 기본적인 기능이라던지,

```
func (b *Broker) OnlineOfflineNotification(clientID string, online bool)
```

클라이언트가 몇 시 몇 분 몇 초에 온라인, 오프라인이 되었는지에 대한 정보를 $SYS/broker/connection/clients/ + clientID 토픽에 저장해놓는다던지 추후 확장 개발에 용이성이 있다.

로그 차원에서 zap 패키지를 사용했는데,
zap 패키지는 [zap-GoDoc] 문서에도 나와있듯이, 속도가 매우 critical하고 json 등의 반정형 로깅에도 유리하고, MQTT 브로커 특성상 많은 로그 데이터를 생성해야 하지만, 이것이 application을 느리게 하는 결과물을 초래하지 않아야 하기 때문에 선택했다.

broker/client.go

client.go에서는 클라이언트 디바이스들에게서 발행/구독 메시지를 받았을 시에 브로커가 취해야할 행동에 대해 프로그래밍 되어있다.

특히 client.go에서는 QoS에 민감한데, 예를 들어,

```
func (c *client) processRouterPublish(packet *packets.PublishPacket) {
        if c.status == Disconnected {
```

```
                return
        }

        switch packet.Qos {
        case QosAtMostOnce:
                c.ProcessPublishMessage(packet)
        case QosAtLeastOnce:
                puback := packets.NewControlPacket(packets.Puback).
(*packets.PubackPacket)
                puback.MessageID = packet.MessageID
                if err := c.WriterPacket(puback); err != nil {
                        log.Error("send puback error, ", zap.Error(err),
zap.String("ClientID", c.info.clientID))
                        return
                }
                c.ProcessPublishMessage(packet)
        case QosExactlyOnce:
                return
        default:
                log.Error("publish with unknown qos", zap.String("ClientID",
c.info.clientID))
                return
        }
}
```

와 같이 QoSAtMostOnce(QoS = 0), QoSAtLeastOnce(QoS = 1), QoSExactlyOnce(QoS = 2)로 구분하여 처리하고 있다.

이외에도 subscribe와 publish를 처리하는 함수 등이 있어 클라이언트 디바이스가 브로커에 요청하는 행위를 처리할 수 있게 되어있다.

comm.go, config.go, usage.go

comm.go에서는 자주 쓰이는 상수나 함수를 정의하고 있다. (QoS 상수, CONNECT, CONNACK, PUBLISH 등, 그리고 equal() 같은 함수들)

config.go에서는 설정으로 자주 쓰이는 구조체나 함수를 정의하고 있다. (Port, Host, Auth, Verify 등)

usage.go에서는 터미널에서 아래와 같은 명령어를 치면

```
$ Terminal

$ ./main -h
$ ./main --help
```

도움말이 나올 수 있도록 도움말을 설정해주는 파일이다.

broker/lib/topics

이 폴더 내에 있는 go 소스코드들은 모두 broker의 토픽 관리를 위한 파일들이다.

```
const (
        /*******************************
         * MWC: Multi-Level Wildcard
         * SWC: Single-Level Wildcard
         * SEP: Serparator
         * SYS: System-Level Topics
         * _WC: All Wildcards
         *******************************/
        MWC = "#"
        SWC = "+"
        SEP = "/"
        SYS = "$"
        _WC = "#+"
)
```

위 상수들은 토픽의 와일드카드와 구분자, SYS 토픽을 관리하기 위한 상수이다.

```
type TopicsProvider interface {
        Subscribe(topic []byte, qos byte, subscriber interface{}) (byte, error)
        Unsubscribe(topic []byte, subscriber interface{}) error
        Subscribers(topic []byte, qos byte, subs *[]interface{}, qoss *[]byte)
error
        Retain(msg *packets.PublishPacket) error
        Retained(topic []byte, msgs *[]*packets.PublishPacket) error
        Close() error
}
```

```
type Manager struct {
        p TopicsProvider
}
```

토픽 프로바이더와 매니저는 각각 interface, struct로 규정되어 있어 topics.go 파일에서 register, unregister, subscribe, unsubscribe, retain, close 등 토픽과 관련된 함수를 다룬다.

memtopics.go 파일에서는 위에서 래핑되어 있는 함수들의 기능이 더 자세하게 다뤄져있으며,
func ValidQos(qos byte) bool 함수로 QoS 검사 등도 수행한다.
특히 memtopics.go에서의 함수들에는 self.smu.Lock()이나 self.smu.Unlock() 등으로 상호배제(Mutual Exclusion)를 자주 하는 것을 볼 수 있는데,
이는 Go Routine으로 병행처리를 함에 있어 Race Condition이나 Deadlock을 피하기 위함이다.

broker/lib/sessions

이 폴더 내에 있는 go 소스코드들은 모두 broker의 세션 관리를 위한 파일들이다.

```
type SessionsProvider interface {
        New(id string) (*Session, error)
        Get(id string) (*Session, error)
        Del(id string)
        Save(id string) error
        Count() int
        Close() error
}

type Manager struct {
        p SessionsProvider
}
```

세션 프로바이더와 매니저는 각각 interface, struct로 규정되어 있어 sessionsProvider.go 파일에서 register, unregister, new, get, del, save, count, close, sessionID 등의 함수를 다루며,
session.go 파일에서 update, retainmessage, addtopic, removetopic, ID, willflag, setwillflag, cleansession 등의 함수를 다룬다.

나머지는 broker/lib/topics의 소스 코드들과 유사한 기능을 한다.

logger/logger.go

logger.go 파일에서는 로깅에 관련된 함수를 처리한다.

디버그, info용으로 사용되는 logging이며, zap 패키지를 활용한다.

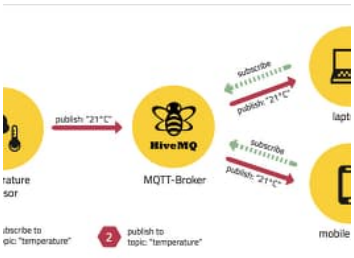
zap 패키지는 많은 양의 로그를 time-critical하게 처리하기에 많은 양의 로그를 처리해야 하는 MQTT Broker에서는 필수이다.

공감 구독하기

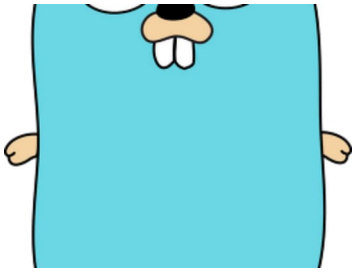
'IT > Go 언어 (Golang)' 카테고리의 다른 글

[Golang] 자체 MQTT 브로커 서버 구현 - 1 (2)	2020.02.26
[Golang] 고루틴과 채널(chan)을 이용한 피보나치 수열 (0)	2020.01.12
[Golang] Go Routine (고 루틴) - Golang의 꽃 [기본] (0)	2020.01.12
[Golang] Go Routine (고 루틴) - Golang의 꽃 [심화] (0)	2020.01.12
[Golang] 문자열과 문자열의 종류 (0)	2019.12.22

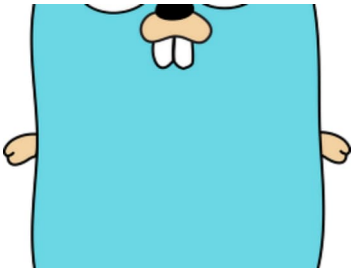
'IT/Go 언어 (Golang)' Related Articles



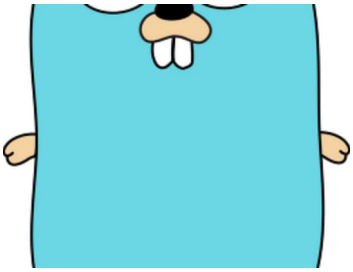
[Golang] 자체 MQTT 브로커 서버 구현 - 1



[Golang] 고루틴과 채널(chan)을 이용한 피보…



[Golang] Go Routine (고 루틴) - Golang의 …



[Golang] Go Routine (고 루틴) - Golang의 …

댓글 0

이름

비밀번호

내용을 입력하세요.

등록