Parna Asadi - 810198498

Report

A simple broadcast system using message queue(broker), server sends notifications and subscribers get them.

Explain

1-a) Sync broker:

```
source := Sync{}
source.wg.Add(1)
broker.messages <- data
source.wg.Wait() // !!!
```

And if message sent to a client, it would call wd.Done().

1-b) Async broker:

```
source := ASync{source: del.Port}
broker.messages <- data
```

And on otherside, dial sender side.

```
server, err := rpc.Dial("tcp", _type.source)

var relpy string
err = server.Call("Receiver.Get", "200: success", &relpy)
```

1-c) Handle overflow: drop extra messages!

```
        if len(broker.messages) == BUFF_COUNT {
                fmt.Println("Message overflow: ", del.Message)
        } else {
                broker.messages <- data
        }
```

1-d) Bi-direction queue:
There are two important variables in broker:

```
- CLIENT_NUM : number of subscribers
- BUFF_COUNT : number of messages in buff
```

- broker:

```
type Broker struct {
        clients  []string
        messages chan Data
        wg       sync.WaitGroup
}
```

- for subscribing:

```
        broker.clients = append(broker.clients, client)
```

- for messaging:

```
        for data := range broker.messages {
                c, err := rpc.Dial("tcp", "0.0.0.0:"+client)
                var relpy string
                err = c.Call("Receiver.Get", data.Message, &relpy)
                data.Type.Send()
        }
```

2-1) Change value of **CLIENT_NUM** to 3 and connect all clients!

3-1) Why using broker?

Message queues provide communication and coordination for these distributed applications. Message queues can significantly simplify coding of decoupled applications, while improving performance,

reliability and scalability.

- Using a message queue system instead of shared data leads to synchronization.With a message-based system, you can think in higher terms of "messages" without having to worry about synchronization issues anymore. (stackoverflow)