CS3102: Practical 1 (P2P File Transfer) - Report

Matric Number: 170002815

11th February 2020

1 Overview

The objective of this practical was to implement a pub-sub system which is as robust and scalable as possible. The system was required to allow the publishing and subscribing of events, as well as protecting itself and its users against many different issues, including out-of-order messages and duplicate messages. My approach for these tasks is discussed in the *Design & Implementation* section of this report.

The testing of the my system, how this testing shows my work satisfies the specification, and how completed extensions were approached are discussed in sections 3, 4 and 5 respectively. The report is completed by an evaluation and conclusion, which reflect back on the learning objectives of the practical, and also evaluate how successful I have been in relation to the specification.

Instructions on how to use my system can be found alongside a bibliography in the appendix of this report.

2 Design & Implementation

2.1 System Interface

For the system's interface with clients, I decided to create a REST API which uses HTTP request methods to transfer data to and from the pub-sub system. The main reason for this decision was that both REST APIs and HTTP request methods are easy to use and integrate into a larger system, especially as HTTP is a very common protocol. Another reason for my decision was that I have never implemented a REST API, and felt it would be beneficial to learn how to do so, as they are a very popular technology [1].

2.1.1 Data Interchange Format

As part of the system's interface, I decided to use JSON for the data interchange format between clients and the pub-sub system, because it has an easy binding into almost every language, and can be transferred over HTTP in the request body as a simple string.

2.1.2 Endpoints

As part of the REST API, I used multiple endpoints to facilitate communication with the pub-sub system. The endpoints used and an overview of the operations which can be completed with each location are listed below. The *README.md* file included in my source code describes each endpoint and its operations in more detail, and the code for the implementation of these endpoints can be found in the *pub-sub.go* file between lines 100 and 1650.

- **/publish:** This endpoint only accepts POST request methods, and is used to publish a message to the system. Including a timestamp or UUID in the request body JSON can avoid problems surrounding out-of-order messages and duplicate messages - this is discussed further in *Section 2.4* of this report.
- **/pull:** This endpoint only accepts GET request methods, and is used for subscribers to pull messages from the system. Including an array of topics in the request JSON can be used to filter which channels should be pulled from, reducing network load by not sending messages which will simply be ignored by clients.

- **/ack:** This endpoint only accepts POST request methods, and is used to acknowledge a message. This is useful to know when to remove a message from the system, and also if a push attempt needs to be retried after a timeout.
- **/topics:** This endpoint accepts GET, POST and DELETE request methods, allowing clients to list the available topic channels, add a new topic channel, and delete an existing topic channel respectively. Including a subscriber's endpoint in the GET request JSON will return the topics which that subscriber is subscribed to.
- **/subscribers:** This endpoint accepts GET, POST and DELETE request methods, allowing clients to list, create, and delete subscribers respectively.
- **/subscriptions:** This endpoint accepts POST and DELETE request methods, allowing clients to subscribe and unsubscribe from topic channels.
- **/generate:** This endpoint only accepts GET request methods, and is used to generate UUIDs compliant with the system.

2.2 Internal Data Structures

The three main data structures used in my system relate directly to the conceptual objects of a pub-sub system: messages, subscribers, and channels. Subscribers and channels each contain a slice (Go's dynamic array) of messages, and while this is extra overhead, it reduces the number of look-ups required to complete a variety of operations, including saving which messages have been received by a subscriber, and limiting the number of messages in a channel - this is discussed further in the *Extensions* section of this report.

2.3 Programming Language & Packages

I decided to approach the implementation of my system with Google's Go language, because it is popular for middleware and networking, with companies including Amazon, Google, and Adobe integrating it in their businesses [2], and it also allowed me to learn a brand new programming language while completing university coursework.

To help me in developing my pub-sub system in Go, I used multiple 3rd party packages, including gorilla/mux [3] for request routing, Google's Go UUID package [4] for UUIDs, and BoltDB [5] for persistent key-value storage.

2.4 Issues Considered

- **R1) Subscribing and publishing events:**
  My system offers both pull and push subscriptions to send events (messages) to subscribers, and also allows the publishing of events (messages) to the system at the `/publish` endpoint.
- **R2) Lookup, discovery, and access of event channels:**
  The `/topics` endpoint gives clients the ability to lookup, discover, create, and delete event channels. The publishing and subscribing of events gives clients access to events in the event channels.
- **R3) Dynamic message queue management:**
  The creation and deletion of messages in my system is handled through clients publishing and subscribing to events, and Go's dynamic arrays (slices) and maps are used to dynamically manage the size of each message queue (channel) within the system's memory.
- **R4) Temporary interruptions of connections:**
  Minor connection interruptions will be dealt with by HTTP, as it built upon TCP, which offers reliable data transfer.

  More significant connection interruptions for publishers can be dealt with by waiting for a 200 OK status response code, and retrying unsuccessful requests.

  The acknowledgement system for subscribers means that until they have received a message and sent an acknowledgement, the message in question will not be deleted from the system, which prevents temporary interruptions of connections being an issue for subscribers.
- **R5) Crashing queues:**
  With my system's message queues being implemented using Go's in-build dynamic arrays, they are unlikely to crash; however in the event of a crash (e.g. memory overflow), the system can be

restarted without any data loss, as it is persisted - this is discussed further in the *Extensions* section of this report.

As part of another extension, I have included a configuration file, which allows users to change settings for the system, including capping the number of messages, subscribers, and topics in the system, as well as giving each message a lifetime before it is removed. By doing so, users can prevent memory overflow, reducing the potential of message queues crashing even further.

- **R6) Crashing customers:**
  Similar to the details discussed in point **R4**, my system copes with crashing customers by using the acknowledgement system for subscribers, preventing messages being lost, and publishers can submit the same request, and can avoid duplicates by including a UUID (see **R11**).

- **R7) Increasing number of queries**
  The gorilla/mux [3] package I used for request routing acts as a wrapper around Go's `http` library, which uses a goroutine for each HTTP request [6]. Goroutines are concurrent, and therefore my system handles each request concurrently, meaning it scales well for an increasing number of queries.

  When implementing database storage for my system (discussed in the *Extensions* section of this report), I used read-only accesses of the database where possible, as BoltDB allows multiple concurrent read-only accesses [5], therefore improving the scalability of my system.

- **R8) Long delays in network traffic**
  As my system uses HTTP and TCP, most network issues are catered for, including long delays in network traffic; however my system also includes features to cater for these such issues.

  The acknowledgement system for subscribers prevents the deletion of messages in the event of a network timeout, and publishers can re-publish the same message with an option of avoiding a duplicate (see **R11**).

  As an extension to this practical, I included a configuration file. This allows for the adjustment of acknowledgement windows and the choice between pull and push subscriptions, which caters for situations where known network delays may be in place.

- **R9) Dropped messages**
  Similar to the other networking considerations, HTTP and TCP will avoid dropping almost all messages over the network, and the acknowledgement system for subscribers will prevent the system dropping unreceived messages.

  The configuration file implemented as an extension offers a cap on the number of messages in the system and each message queue, meaning that new messages will either have to be rejected or replace currently-stored messages. The configuration file offers the choice between rejecting new messages and removing the oldest message in the system - this is discussed further in the *Extensions* section of this report.

- **R10) Out of order messages**
  Network delays, Go/BoltDB's concurrency systems, and the retry of failed message pushes could mean messages are out-of-order when they arrive to the subscribers. Certain use-cases for my pub-sub system may rely heavily on the ordering of messages (e.g. a self-driving car's data), and therefore I have decided to combat this issue.

  Publishers can include a timestamp in the JSON of their PUSH request, and if no timestamp is received by the system, the system will add one on to the message as close to the time of the request as possible. The timestamps on different messages can be compared by subscribers to re-order out-of-order messages.

- **R11) Duplicated messages**
  As discussed in earlier considerations, including **R4**, **R6**, and **R8**, there are a variety of reasons why publishers may publish the same message twice.

  To avoid duplicate messages being inputted into the system, publishers can make a GET request at the `/generate` endpoint to receive a system-compliant UUID, which can be sent in the body of a message being published. The system will reject the publishing of any message using the same UUID as a message currently in the system. This gives the option for publishers to avoid publishing a duplicate message.

  As my system uses at-least-once delivery to transfer messages to subscribers, they also need a way to work around duplicate messages. If the publisher of the message does not include a UUID, the system will generate one and add it to each message. Every message's UUID is unique by definition, and is sent to subscribers, allowing them to detect duplicate messages by comparing the received UUIDs.

2.5 Pub-Sub-Related Decisions

For most pub-sub systems, architects will make many decisions relating to the core functionality of the system, which will improve it for the specific use case it is being created for.

Some of these decisions will include whether the system pushes messages to subscribers or if the subscribers pull messages from the system, if there is a maximum number of messages, subscribers or topics in the system, and how to deal with overflowing queues.

As the specification provided no specific use case, I decided to make my pub-sub system use case agnostic by implementing multiple functionalities and allowing users to choose how the system works by editing a configuration file. This is discussed further in the *Extensions* section of this report.

2.6 System Design Draft

Figure 1 below shows a digital representation of the sketch I created before implementing my program.

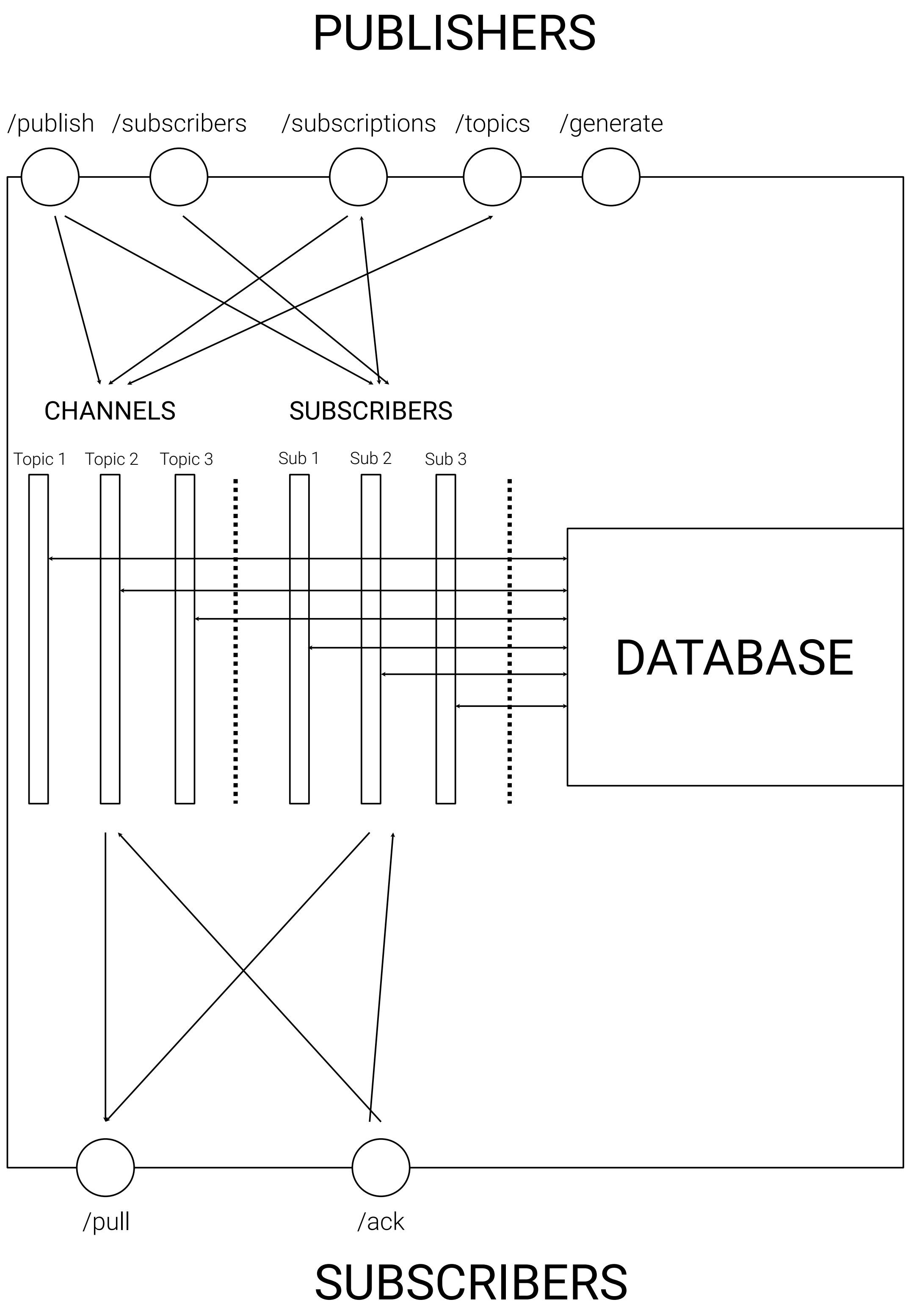


Figure 1: System Design Draft

3 Testing

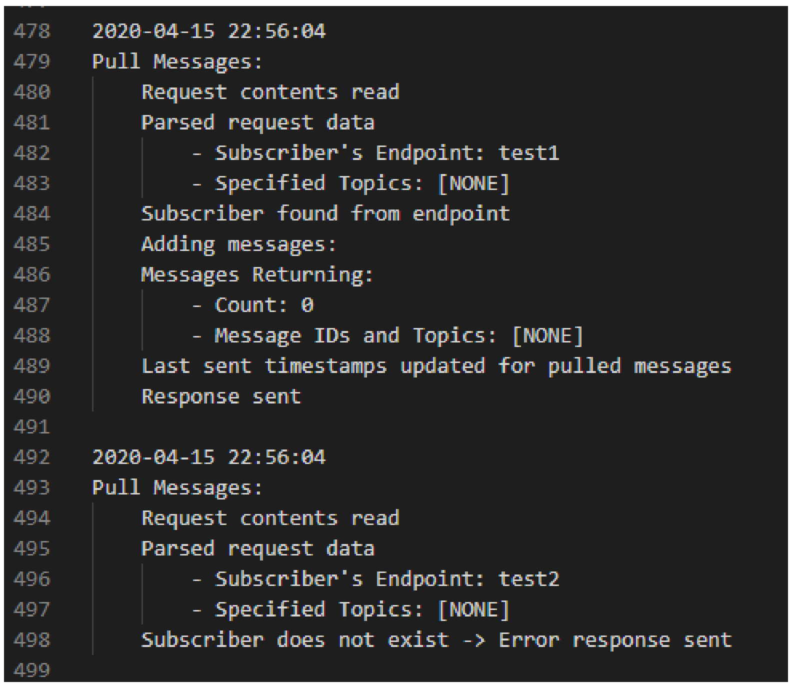
In order to test my system, I used multiple different techniques: log files, Postman, unit tests and an example program. Each of these is discussed in the following subsections.

3.1 Log Files

Early testing of my system mostly involved printing a trace of the program. Rather than printing debug statements to the command line, I decided to contain the output to a log file - `pub-sub/assets/log.txt`.

To avoid code duplication, and promote loose coupling, I created `pub-sub/src/logger.go`, which contains the only code to interact with the log file. The exported functions of `logger.go` are `LogTime()`, which writes the current date and time to the log file, and `LogString(string)`, which writes a given string to the log file.

An example section of the log file can be seen in *Figure 2* below.



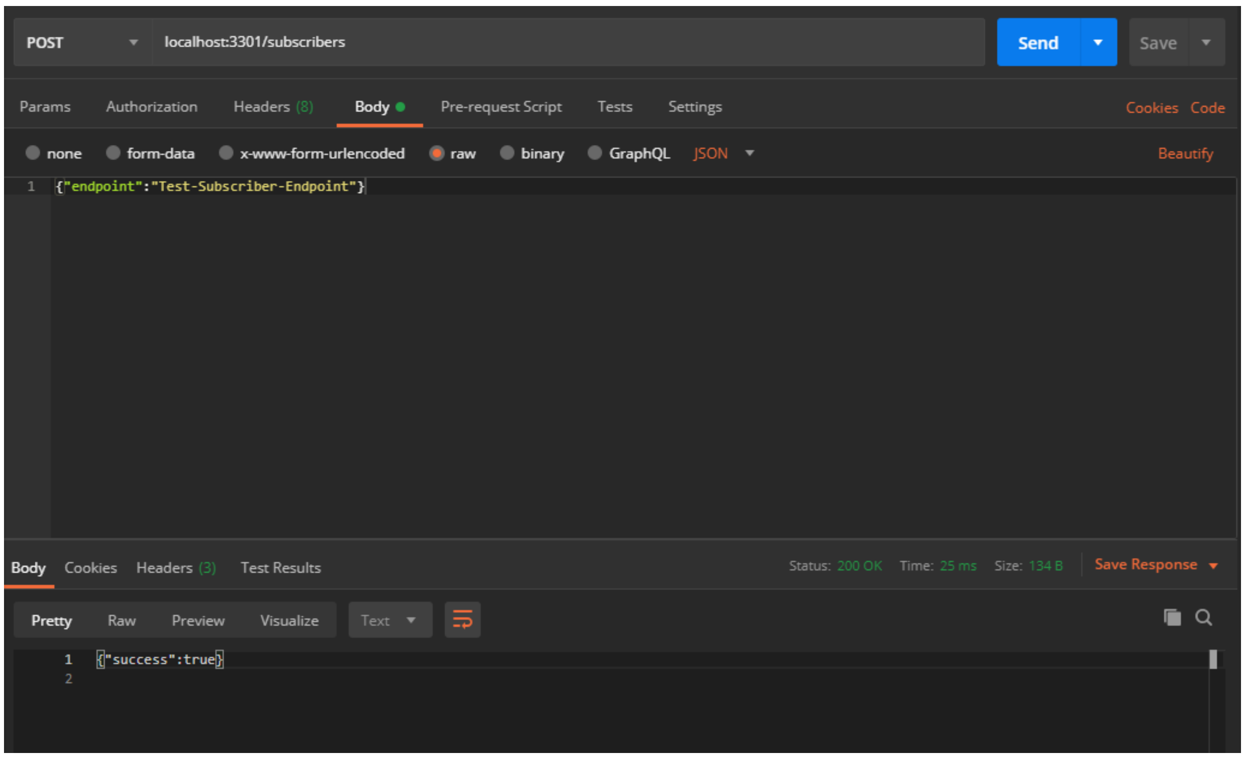
```
2020-04-15 22:56:04
Pull Messages:
    Request contents read
    Parsed request data
        - Subscriber's Endpoint: test1
        - Specified Topics: [NONE]
    Subscriber found from endpoint
    Adding messages:
    Messages Returning:
        - Count: 0
        - Message IDs and Topics: [NONE]
    Last sent timestamps updated for pulled messages
    Response sent

2020-04-15 22:56:04
Pull Messages:
    Request contents read
    Parsed request data
        - Subscriber's Endpoint: test2
        - Specified Topics: [NONE]
    Subscriber does not exist -> Error response sent
```

Figure 2: Log File Excerpt

3.2 Postman

Creating test HTTP requests can be time-consuming and frustrating. In order to speed up the testing process, and view the system's response to different requests easily, I used Postman [7].

An example POST request/response sent by Postman to create a new subscriber can be seen in *Figure 3*, and the same request can be seen in *Figure 4* to demonstrate my system's duplicate subscriber prevention functionality.



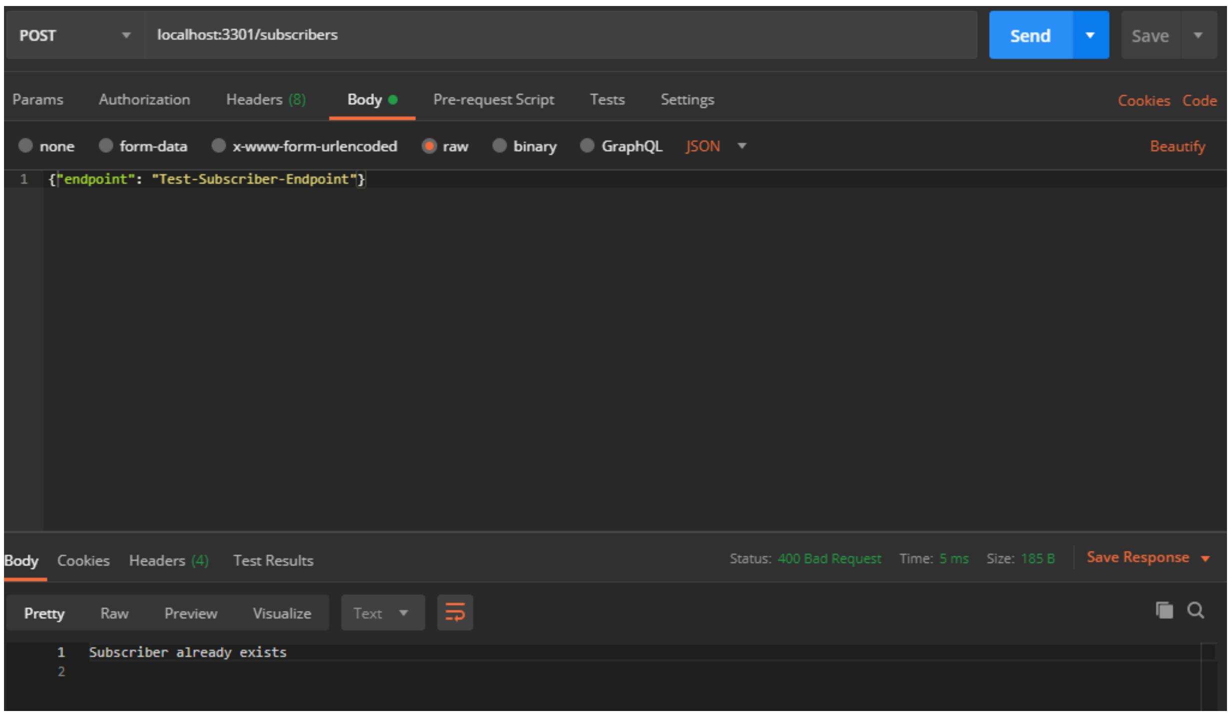
localhost:3301/subscribers

```
{"endpoint":"Test-Subscriber-Endpoint"}
```

Status: 200 OK Time: 25 ms Size: 134 B

```
{"success":true}
```

Figure 3: Postman - New Subscriber Request



localhost:3301/subscribers

```
{"endpoint": "Test-Subscriber-Endpoint"}
```

Status: 400 Bad Request Time: 5 ms Size: 185 B

```
Subscriber already exists
```

Figure 4: Postman - Duplicate Subscriber Request

You can view a set of Postman requests to demonstrate many of the functionalities of my system in the *Tests.pdf* file alongisde this report.

3.3 Unit Testing

The extensions completed involving database access and configuration file parsing were, made my system highly dependent on the code written for these functionalities.

In order to ensure these parts of the program worked correctly, I used Go's `testing` package to write a comprehensive set of unit tests, covering every function in `database-cache.go` and `config-reader.go`. You can view the tests for these files in the `*_test.go` files, and can run the tests by calling `go test` in the `src` directory of the system.

The list of tests for each file is as follows:

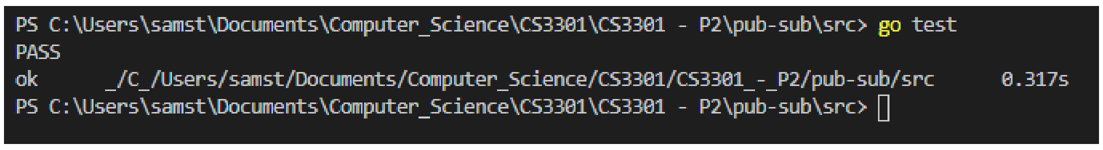
config-reader.go tests

1. A regular configuration file is parsed correctly.
2. A half-complete configuration file is parsed correctly, with defaults being set for missing values.
3. An empty configuration file is accepted, with all defaults being set correctly.
4. A broken configuration file is rejected.

database-cache.go tests

1. The database is initialised correctly and without any errors.
2. A message can be stored, fetched, and deleted from the database correctly.
3. A subscriber can be stored, fetched, and deleted from the database correctly.
4. A topic channel can be stored, fetched, and deleted from the database correctly.
5. All subscribers can be fetched correctly.
6. All channels can be fetched correctly.
7. A system state can be stored and fetched from the database correctly.
8. The database can be closed correctly, and without any errors.
9. A message stored will be persisted between program runtimes.
10. A subscriber stored will be persisted between program runtimes.
11. A topic channel stored will be persisted between program runtimes.
12. A system state stored will be persisted between program runtimes.

Figure 5 shows that all unit tests pass.



```
PS C:\Users\samst\Documents\Computer_Science\CS3301\CS3301 - P2\pub-sub\src> go test
PASS
ok      _/C_/Users/samst/Documents/Computer_Science/CS3301/CS3301_-_P2/pub-sub/src      0.317s
PS C:\Users\samst\Documents\Computer_Science\CS3301\CS3301 - P2\pub-sub\src>
```

Figure 5: Unit Test Success Output

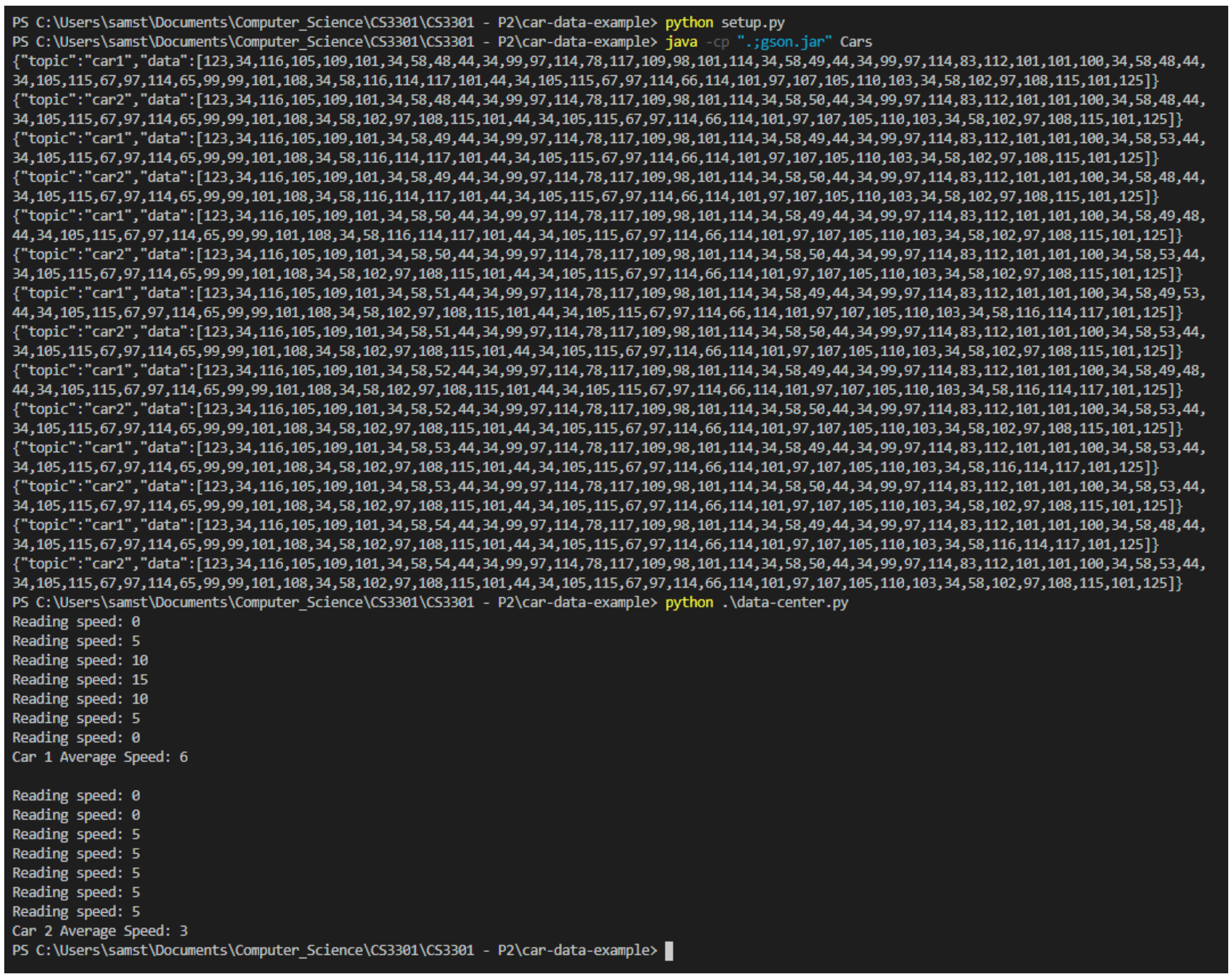
3.4 Example Program

Although Postman is useful for testing specific functionalities of my system, such as duplicate message rejection, it is tedious and time-consuming to insert a volume of messages similar to that of a real application using my pub-sub system.

In order to test my system in a "real" scenario, I created an example application, in which cars produce data, and a data center parses the data. To demonstrate that my system's interface is language independent, I used Java to mimic the cars, and Python to mimic the data-center.

The application works as follows: once the pub-sub system is running, `setup.py` creates two topics, and two subscribers, such that there is a topic for one car, which one subscriber is subscribed to, and one for another car, which the other subscriber is subscribed to. `Cars.java` then publishes data every second to the pub-sub system for each car; the data includes the car's speed, and a few other variables encoded as JSON UTF-8 bytes. When `data-center.py` is executed, it pulls all messages for each topic, and calculates the average speed for each topic, and therefore each car; this data is outputted to the command line.

The execution and output of my example program can be seen in *Figure 6*, and the code can be found in the `car-data-example` directory of this practical.



```
PS C:\Users\samst\Documents\Computer_Science\CS3301\CS3301 - P2\car-data-example> python setup.py
PS C:\Users\samst\Documents\Computer_Science\CS3301\CS3301 - P2\car-data-example> java -cp ".;gson.jar" Cars
{"topic":"car1","data":[123,34,116,105,109,101,34,58,48,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,48,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,116,114,117,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,48,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,48,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car1","data":[123,34,116,105,109,101,34,58,49,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,116,114,117,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,49,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,48,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,116,114,117,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car1","data":[123,34,116,105,109,101,34,58,50,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,49,48,
44,34,105,115,67,97,114,65,99,99,101,108,34,58,116,114,117,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,50,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car1","data":[123,34,116,105,109,101,34,58,51,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,49,53,
44,34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,116,114,117,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,51,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car1","data":[123,34,116,105,109,101,34,58,52,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,49,48,
44,34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,116,114,117,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,52,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car1","data":[123,34,116,105,109,101,34,58,53,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,116,114,117,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,53,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
{"topic":"car1","data":[123,34,116,105,109,101,34,58,54,44,34,99,97,114,78,117,109,98,101,114,34,58,49,44,34,99,97,114,83,112,101,101,100,34,58,48,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,116,114,117,101,125]}
{"topic":"car2","data":[123,34,116,105,109,101,34,58,54,44,34,99,97,114,78,117,109,98,101,114,34,58,50,44,34,99,97,114,83,112,101,101,100,34,58,53,44,
34,105,115,67,97,114,65,99,99,101,108,34,58,102,97,108,115,101,44,34,105,115,67,97,114,66,114,101,97,107,105,110,103,34,58,102,97,108,115,101,125]}
PS C:\Users\samst\Documents\Computer_Science\CS3301\CS3301 - P2\car-data-example> python .\data-center.py
Reading speed: 0
Reading speed: 5
Reading speed: 10
Reading speed: 15
Reading speed: 10
Reading speed: 5
Reading speed: 0
Car 1 Average Speed: 6

Reading speed: 0
Reading speed: 0
Reading speed: 5
Reading speed: 5
Reading speed: 5
Reading speed: 5
Reading speed: 5
Car 2 Average Speed: 3
PS C:\Users\samst\Documents\Computer_Science\CS3301\CS3301 - P2\car-data-example>
```

Figure 6: Example Program Output

4 Questions

I believe the tests shown in the *Testing* section of this report combined with the *Tests.pdf* file demonstrate that all solutions to issues discussed in *Section 2.4* of this report are implemented properly, meaning my testing provided is sufficient to indicate that my program satisfies the specification.

5 Extensions

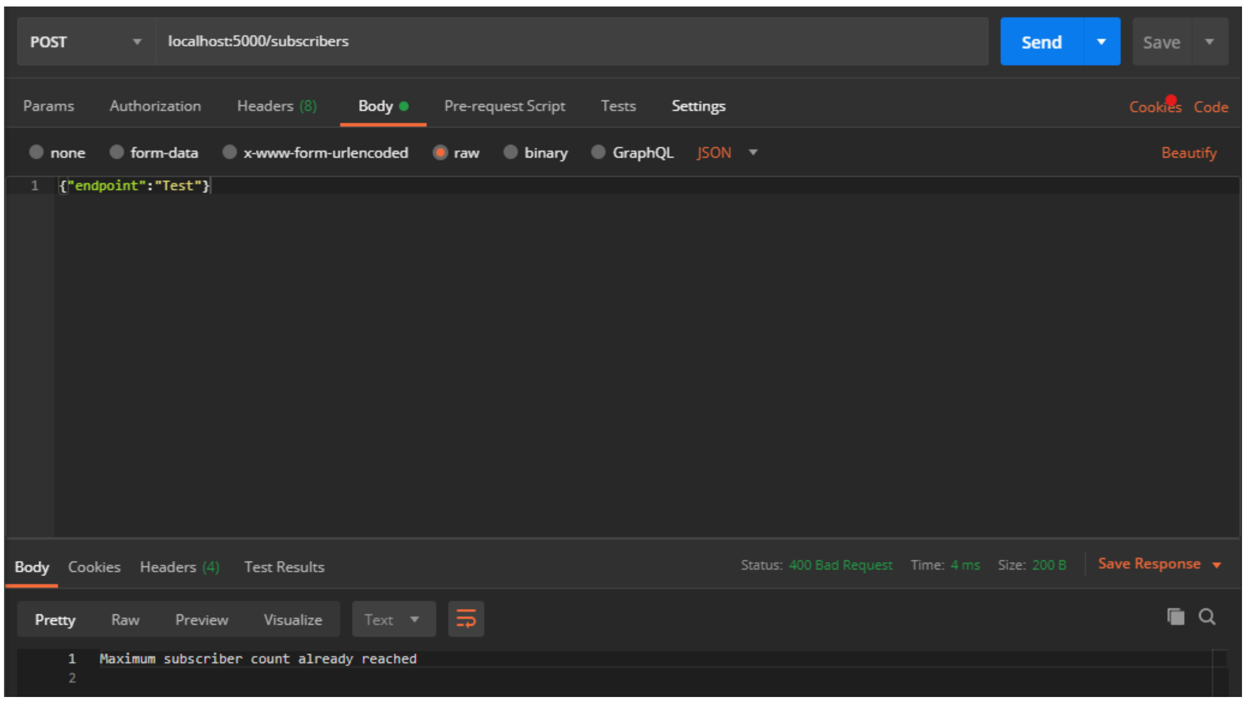
5.1 Configuration File

For my first extension, I decided to implement a configuration file. As discussed earlier in the report, particularly in *Section 2.5*, I decided to avoid making decisions which could impact the effectiveness of the system for different use cases by implementing a configuration file.

The configuration file can be found inside the `pub-sub/assets` directory, and is named `options.conf`. The file contains 14 options, each with a comment to describe the effect which is caused by adjusting the value. Examples of options include the port number on which the pub-sub system is hosted, the message overflow type for the system, and the time before a message is removed from the system.

The configuration file is parsed into the program by `config-reader.go` file inside the `pub-sub/src` directory. The file has one exported function, named `ReadConfigFile(string)`, which parses a configuration from the specified configuration file and stores it into the exported variable named `Config`, allowing other files to access the configuration as Go variables. The parsing takes into account commented lines, and fills in gaps using default values.

The configuration file extension is tested thoroughly using Go's unit testing package, `testing`, and is discussed further in *Section 3.3* of this report. An example of the port number being set to 5000, and the maximum number of subscribers being set to 1 when adding a second subscriber is shown in *Figure 7.*



```
POST localhost:5000/subscribers
{"endpoint":"Test"}
```

Status: 400 Bad Request Time: 4 ms Size: 200 B

```
Maximum subscriber count already reached
```

Figure 7: Postman - Special Configuration on Port 5000

5.2 Persistent Data & Caching

For my second extension, I decided to implement persistent storage of messages, topics and subscribers in the pub-sub system, along with a caching mechanism for regular requests.

To persist data, I used BoltDB's key-value store [5], which has a strong binding into the Go language. Messages are stored using their UUID as a string, topic channels are stored using their topic as a string, and subscribers are stored using their endpoint as a string. To avoid collisions between endpoints, topics, and message IDs, I created a separate bucket for each. The system state is stored using the zero-UUID as a key in its own bucket.

The code used to interact with the database is confined to `database-cache.go`, which promotes loose coupling. There are 4 unexported methods which use BoltDB code directly: `CreateBucket(string)`, `StoreKeyValue(string, string, byte[])`, `FetchValue(string, string)`, and `DeleteValue(string, string)`, each of which has an implicit name. There are then 13 exported functions, which wrap around the original 4, and allow one-line calls to store, fetch, or delete a message, subscriber, channel, or system state.

To improve performance of my system, each store call caches the retrieved value in a respective cache, and each fetch call checks the cache before fetching from the database. Caches are implemented using Go's in-built maps, and are transparent to other code using the store and fetch functions.

This extension is tested thoroughly using Go's `testing` package. This is discussed further in *Section 3.3* of this report.

6 Evaluation & Conclusion

Overall I believe I have been successful with this practical - I have implemented the basic specification to a high standard, and have completed both testing and extensions to accompany my work.

Reflecting on the learning objectives for this practical listed in the specification, I have learned a lot about pub-sub systems, including message and message queue structures, and which interfaces should be exposed to publishers and subscribers. I have gained knowledge on QoS, and understand the benefits of using a REST API with HTTP to create a strong QoS.

Difficulties I encountered during this practical included scope creep, encoding JSON within JSON in Go, and designing sufficient testing for a system with a near-infinite number of uses.

Given more time, I would enjoy implementing more flexible endpoints, for example listing all subscriptions, as well as implementing a proper application with this system.

Instructions

System Setup

1. Ensure Go is installed on the machine you are using.
   You can install Go here: *https://golang.org/doc/install*

2. Execute the following commands in a terminal to install the package dependencies for the system:

   - `go get github.com/boltdb/bolt`
   - `go get github.com/google/uuid`
   - `go get github.com/gorilla/mux`

3. Open the submission directory in the terminal, and recursively set permissions using the following command:

   - `chmod 777 * -R`

Pub-Sub System

1. Open the `pub-sub/src` directory in a terminal.
2. Execute the following command to start the pub-sub system:
   - `go build -o ../bin/<executable_name>`

   Note 1: *`<executable_name>` should be replaced by `pub-sub`, followed by your system's executable suffix, e.g. `.exe` for Windows, or nothing for Linux.*

   Note 2: *You can find the Windows and Linux/MacOS executables in the `bin` directory if you do not wish to compile the code yourself. These can be run without having Go installed.*
3. Change into the `bin` directory alongside the `src` directory.
4. Execute the executable named `pub-sub` followed by your system's executable suffix.
   Note: *Run the executable with flag `-h` to view the flags available, and how to use them.*

Unit Tests

1. Open the `pub-sub/src` directory in a terminal.
2. Execute the following command to run all tests:
   - go test

   Note: *You can view the testing code in the files ending in `_test.go`.*

Example Program

1. Start the pub-sub system using the steps noted in the *Pub Sub System* subsection above.
2. Open the `car-data-example` directory in the root directory of the practical submission folder in a terminal. You will need Java and Python3 installed.
3. Call the following commands:
   - `pip install httplib2`
   - `python setup.py`
   - `java Cars` , being sure to include the *gson.jar* file in your classpath.
   - `python data-center.py`

References

[1] Guy Levin. The rise of rest api, Feb 2018.
https://blog.restcase.com/the-rise-of-rest-api/ [Accessed April 2020].

[2] Golang. golang/go.
https://github.com/golang/go/wiki/GoUsers [Accessed April 2020].

[3] gorilla/mux.
https://github.com/gorilla/mux [Accessed April 2020].

[4] google/uuid.
https://github.com/google/uuid [Accessed April 2020].

[5] Boltdb.
https://github.com/boltdb/bolt [Accessed April 2020].

[6] Golang - http.
https://golang.org/pkg/net/http/ [Accessed April 2020].

[7] Postman — the collaboration platform for api development.
https://www.postman.com/ [Accessed April 2020].