James Jacobson
180030848
15 April 2019
CS3301: Component Technology
Practical 2 Report

Overview, Design and Implementation

For this practical we were asked to build a robust, scalable, pub-sub message passing system as a backend for Uber. After seeing the group presentations in the lectures, I decided to use RabbitMQ to complete this assignment. RabbitMQ comes along with a nice HTTP API that allows you to view message publish and delivery rates in real time, which was very helpful in the testing of my system.

My system consists of a consumer and a producer class that simulate the passing of several different types of messages that might be included in Uber's backend systems. I use 5 different types of simulated messages that are simply marked by strings shown below:

String	**Simulated message type in Uber backend**
"DtoC"	Messages from Driver to Customer
"DtoS"	Messages from Driver to Uber's servers
"CtoD"	Messages from Customer to Driver
"CtoS"	Messages from Customer to Uber's servers
"Payments"	Messages carrying payment information.

These strings act as the routing/binding key for the simulated message, and each of these different message types get routed to a different queue, resulting in 5 different queues/services that are able to be looked up and discovered in the base case of my system. I know that there are probably hundreds of different queues in the actual Uber backend, but this simple setup was sufficient for simulating the tasks that this practical seeks to address. I began with a simple base case with one consumer process and one producer process, each set to consume/produce 1000 messages per second. As shown below, we can see all five queues, each with randomly generated

names and each actively running at 200 messages/second, just as expected.

Overview			Messages			Message rates			+/-
Name	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
amq.gen-cxNekG1VAZa7NNIOqp5DAw	Excl	running	0	0	0	200/s	200/s	0.00/s	
amq.gen-eAhTiYPdWlJ8syml6PPi_w	Excl	running	0	0	0	200/s	200/s	0.00/s	
amq.gen-oRLyEX4dkp1GwuZiMWfO3g	Excl	running	0	0	0	200/s	200/s	0.00/s	
amq.gen-ryBTLqt20044JYRk2LpntA	Excl	running	0	0	0	200/s	200/s	0.00/s	
amq.gen-wr4iKvXRgjHAzweXBnpVdw	Excl	running	0	0	0	200/s	200/s	0.00/s	

Testing and Question

The main part of this practical was demonstrating that my implementation could react to common errors that message oriented middleware often experiences and must be equipped to deal with as quickly as possible without any drop in performance. I mainly aimed to extend the functionality of my MOM by dealing with errors in the messages themselves, including dropped messages, messages that are out of order, and duplicated messages. To do this, I simulated an "expected order" of messages with a simple integer added onto the end of all of my messages.

I believe this is a fair assertion - a good message oriented backend should anticipate certain things about the flow of messages throughout the system including the ordering of the messages it is built to handle and deliver. In an actual backend for Uber, for example, I might set an expected order of events in any particular ride to be something like:

1. Customer requests trip
2. Closest driver is located
3. Driver is notified of trip request
4. Driver accepts trip request
5. Customer is notified of driver's acceptance of trip
6. Customer is notified of driver's location periodically
7. Driver arrives at pickup location
8. Customer is picked up by driver
9. Periodic location updates
10. Arrival at destination
11. Customer reviews driver and leaves a tip
12. Trip complete

Based on this expected order of events, I can assume that if a queue receives message 10 before it receives message 4, then something is wrong with my middleware and some intermediate messages didn't get sent correctly. Similarly, if I receive message 4 for the second or third time within a single trip, I know that this must be a duplicate message. This is exactly the methodology I used in building my middleware to handle problems with message delivery. I attached a simple integer onto the end of all of my messages as they were passed through my producer process, and used this integer index to process any potential errors in message delivery.

Shown below is the logic that is used to handle these message errors, which handles four different cases of message ordering.

```
# Dealing with out of order messages, missed messages, and du

# Simulated expected-order of message
str_m = body.decode('utf-8')
mo = str_m[-3:]

if int(mo) == 0:
    messages_master_c.append(0)

last_processed = max(messages_master_c)
diff = int(mo)-int(last_processed)

 if int(mo) == 502:
     pdb.set_trace() #use for debugging


# If diff = 0 or we've already processed the message, then we
if diff == 0 or int(mo) in messages_master_c:
    print("This message is a duplicate of another message.")
    if self.auto_duplicate_drop == False:
        a = input('Would you like to drop this duplicated mes
        if (str(a) == 'Y'):
            print("Dropping duplicated message")
    if self.auto_duplicate_drop == True:
        print("Dropped duplicate message at index " + mo)

# If diff > 1 then we missed a message in our expected orderi
elif diff > 1 and not int(mo) in potential_dropped_messages:
    print("Possible missed message at or before index " + mo
    to_find_l = max(messages_master_c)
    for x in range(to_find_l, int(mo)):
        potential_dropped_messages.append(x)

# We are dealing with messages received out of the expected o
elif diff > 1 and int(mo) in potential_dropped_messages:
    print("Expected message "+str_m+" at index " + mo)
    if self.auto_reorder == False:
        a = input('Would you like to move this message to its
        if (str(a) == 'Y'):
            print("Moving message to its expected index")
    if self.auto_reorder == True:
        print("Moving message to its expected index")

# If diff == 1 then everything went as expected
elif diff == 1:
    print("Message order confirmed")
    if mo != 0:
        messages_master_c.append(int(mo))
```

- Case 1: The index of the message has already been processed, indicating a duplicate message.
- Case 2: The index is higher than we expected, indicating this message was delivered out of order or some messages have been dropped in between. In this case, we keep track the messages that were potentially dropped so we can correct the error later.
- Case 3: The index is one of the messages that was potentially dropped, and we reorder it back into the expected order.
- Case 4: The index is what we expected and there is no error.

This example of a robust validation and reassignment of message order could be very useful in any middleware that deals with messages that come in a sequence that is predictable in any way at all, which (in my opinion) rules out very few systems. The confirmation messages of my successful tests are shown in the output of the consumer process, like shown below. The tests shown demonstrate

```
Message order confirmed
 [x] Received b'Paymentsmessage496'
Message order confirmed
 [x] Received b'Paymentsmessage497'
Message order confirmed
 [x] Received b'Paymentsmessage498'
Message order confirmed
 [x] Received b'Paymentsmessage499'
Message order confirmed
 [x] Received b'Paymentsmessage497'
This message is a duplicate of another message.
Dropped duplicate message at index 497
 [x] Received b'Paymentsmessage502'
Possible missed message at or before index 502 we will look for this message for you.
 [x] Received b'Paymentsmessage501'
Expected message Paymentsmessage501 at index 501
Moving message to its expected index
```

how my message oriented middleware satisfies the specification and is also extensively built to handle three common errors in message delivery.

Evaluation and Conclusion

Overall, I believe my middleware did a good job of fulfilling the requirements of this practical. My system includes publish - subscribe services to a base number of five different channels/topics that are easily discovered using functionality that can be easily extended and scaled to many more channels. My middleware also extends beyond the compulsory part of the practical to include support for common errors in message delivery including duplicate messages, dropped messages, and messages delivered out of order.

If I had been given more time, I would have liked to finish adding the functionality to kill a message queue and check that all messages were delivered. I spent several hours on this, but was struggling too much with communicating with my producer subprocess to properly kill a message queue. This is one of the things I found most difficult about this practical.

Instruction

Install RabbitMQ
Install pika for RabbitMQ in python
Main functionality

Uncomment “while True” in line 36 of producer.py and indent lines 37-51
Run consumer.py
Run producer.py in a separate console

Testing message error-handling

Comment out “while True” in line 36 of producer.py and unindent lines 37-51
Run consumer.py
Run producer.py in a separate console