INF6953G: Lab 3
Scaling Databases
and Implementing Cloud Patterns

Foutse Khomh
S. Amirhossein Abtahizadeh
Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
foutse.khomh[at]polymtl.ca
a.abtahizadeh[at]polymtl.ca

Le An, Alexandre Courouble, Mahdis Zolfagharinia

24th March, 2016

1 What were the results of comparing MySQL to MySQL Cluster. Were any of the results surprising, why or why not?

For the sake of comparing MySQL to MySQL Cluster, we set up both databases on AWS EC2 t2.micro instances. The MySQL database was installed on a signle node, while the cluster was setup on a set of four different instances. The first instance would hold the Master and the MySQL nodes while the last three would host the data nodes. We were able to benchmark both databases using Sysbench.

2 When and why should we use The Gatekeeper and Competing Consumers Patterns?

3 Describe your solutions to implement the two Cloud patterns strategies.

3.1 Competing Consumers pattern

3.1.1 Overview

We use 4 AWS t2.micro machines to build the server side of the *Competing consumers pattern*. We install and configure or to the other three data nodes (*slave nodes*). The Competing Consumers pattern only handles `INSERT` requests. After inserting a client request into the specific data node, the master node will reply to the cMySQL in the 4 machines (referred as to data node in the rest of this section). We configure the 4 data nodes as a cluster, where one machine (the *master node*) receives requests from clients and distributes them either to itselflient with a message (*e.g,* "Data inserted into Slave1"). Figure ??? illustrates the workflow of server which implements the Competing consumers pattern.??? In the rest of this section, we elaborate three key points of the implementation of this pattern in terms of the data nodes' configuration, master node's implementation, and socket connection. Our source code in Java can be found at: `???the-link???`.

3.1.2 Configuration of data nodes

We load *Sakila* database into the 4 data nodes. The master node is responsible to balance work loads among the 4 data nodes, *i.e.,* distributing and sending queries to itself or to the slave nodes.

3.1.3 Implementation of the master node

For each client request, the master node will receive a string through the socket connection. The client request string contains the client identifier and her SQL request statement, an example of which is shown as follow:

```
200a3b9b-17a1-4808-b1ba-54d159ea4108+2006||INSERT INTO film (title,
description, release_year, language_id) VALUES ('sample_movie', 'This
is just a test', 2016, 1)
```

In the request, the client identifier and the SQL statement are separated by `||`. The master node parse the all letters and numbers (*i.e.,* `[0-9a-z]`) in the client identifier to convert each of these characters to an ASCII decimal number and sum all ASCII decimal numbers. The result is noted as *DigitSum*. As there are 4 data node in the cluster, to balance the workload in these node, we divide *DigitSum* by 4 and calculate the remainder, noted as *Rem* (*i.e.,*

$DigitSum\%4 = Rem$). Rem is an integer ranged from 0 to 3. The master node will assign the request (by sending the SQL statement), where $Rem = 0$, to itself; the request, where $Rem = 1$, to slave node #1; the request, where $Rem = 2$, to slave node #2; and the request, where $Rem = 3$, to slave node #3.

We use Java Database Connectivity (JDBC) API to send MySQL queries from the master node to slave nodes. We configure the port 3306 for the MySQL communication among the 4 data nodes.

3.1.4 Socket connection

We apply the GlassFish server to implement the message queue, which builds a communication channel between clients and data servers. When a client's request has passed through the message queue, we will receive this request, concatenate the client's UID with its `INSERT` request statement, and send it to the master node by the socket connection.

In the socket connection, on the one hand, the client program establishes a socket connection object, which initialize a `DataOutputStream` object and a `DataInputStream` object. The client uses the `DataOutputStream` object to send the request (the client's UID + a SQL statement) to the master node. It uses the the `DataInputStream` to receive the response sent back by the master node.
On the other hand, the master node acts as the socket server, which runs a dead loop in order to wait for clients' requests. The master node uses a `DataInputStream` object to receive socket packets. One packet represents a client's request. The master node converts each of the packets into a `String` object (*i.e.,* a request), then applied the mechanism described in Section 3.1.3 to decide the request's priority and send it to the corresponding data node. Once a request is well inserted into the data node, the master node will reply to the client through a `DataOutputStream` object.

Similar to the real web environment, if there is a socket connection error, the client will receive an error message. The client can decide whether to re-send a request to the master node.

3.2 Gatekeeper pattern

3.2.1 Overview

We use 3 AWS t2.micro instances to build the server side of the *Gatekeeper* pattern. The first instance refers as to the *gatekeeper*, which filters out malicious information from all clients' requests. The second instance refers as to the *trusted host*, which receives safe requests from the gatekeeper and send them to the database. The third instance refers as to the *sensitive data node*, which handles clients' requests and send the result back to the trusted host. To ensure

the confidentiality of the database, the sensitive data node can be only reached by the trusted host. And the trusted host can be only reached by the gatekeeper and the sensitive data node. Therefore, one client's request targeted to the sensitive data node will pass through the gatekeeper, trusted host, and handled by the the sensitive data node; then the results will be sent back by the sensitive data node through the trusted host and the gatekeeper to the client. Figure ??? illustrates the workflow of server which implements the Gatekeeper pattern.???

In the rest of this section, we elaborate three key points of the sensitive data node's configuration, the trusted host's configuration and implementation, and the gatekeeper's implementation. Our source code in Java can be found at: `???the-link???`.

3.2.2 Configuration of the sensitive data node

We configure the security rules (through AWS security group) and allow the machine of the sensitive data node to only accept the inbound TCP requests from the IP of the trusted host through the port 3306. Thus, other machines cannot directly send queries to the sensitive database.

3.2.3 Configuration and implementation of the trusted host

We configure the security rules and allow the machine of the trusted host to only accept the inbound requests from the IP of the gatekeeper through the port of 6666. Thus, only the gatekeeper can directly send a query to the trusted host, which can then transfer the query to the sensitive database.

The trusted host applies a socket connection to exchange data with the gatekeeper. In this connection, the trusted host acts as the socket server, which runs a dead loop to wait for the gatekeeper's queries. While the gatekeeper acts as the socket client, which send safe SQL queries to the trusted host and send back the response to clients.

The trusted host uses JDBC API to send MySQL queries, which are received from the gatekeeper, to the sensitive data node. The JDBC API will wait and receive the response from the sensitive data node, and send it back the gatekeeper.

3.2.4 Implementation of the gatekeeper node

The gatekeeper node receives `SELECT` or `INSERT` requests from clients. As in Section 3.1.3, the gatekeeper node parses a client request to extract her UID and the SQL query. It considers the requests, in which the UID with the first character as letters (*i.e.,* `[a-z]`, ASCII code from 97 to 122) as safe requests, and will transfer ot to the trusted host. Other requests are considered as malicious requests, and will be rejected by the gatekeeper node.

Table 1: Application execution time of the Competing Consumers pattern.

# of clients	Run1	Run2	Run3	Run4	Run5	Average
25	207	208	206	203	207	206.2
50	404	403	406	410	416	407.8
75	602	604	608	604	605	604.6
100	795	801	808	834	806	808.8

To communicate with clients, there is another socket connection between the gatekeeper node and the client program. In this socket connection, the gatekeeper node acts as a socket server, while the client program acts as a socket client.

4 Describe the implementation of multi-threaded application scenarios.

5 What were the results of comparing the patterns in terms of performance? Were any of the results surprising, why or why not?

5.1 Setup

We intend to compare the performance between the Competing Consumers pattern and Gatekeeper pattern. We use the following scenario to evaluate the performance of the pattern applications.

- We simulate respectively 25, 50, 75, and 100 concurrent clients for the scenario. The GlassFish server is deployed in an AWS instance, which connects with the master node (in the Competing Consumers pattern) and with the gatekeeper node (in the Gatekeeper pattern).
- For the Competing Consumers pattern, each of the clients send 5 `SELECT` queries to the database, then wait for 5 seconds; she send 100 `SELECT` queries to the database, then wait for 5 seconds; she finally send 5 `INSERT` queries to the database. For the Gatekeeper pattern, the clients only send `INSERT` queries to the database.
- We repeat the scenario 5 times for each number of clients.

We use the application execution time to measure their performance. The lower the execution time indicates the higher performance of the pattern application.

Table 2: Application execution time of the Gatekeeper pattern.

# of clients	Run1	Run2	Run3	Run4	Run5	Average
25	204	205	203	205	205	204.4
50	405	403	403	402	402	403
75	603	605	605	605	650	613.6
100	793	805	806	806	805	803

5.2 Results

Table 1 and Table 2 show the execution time of the Competing Consumers pattern and the Gatekeeper pattern according to the scenario. Figure 1 illustrates the average execution time (in the 5 runs) of the two patterns. We observe that the execution time tends to linearly increase with the increase of the clients. The two cloud design patterns have very similar execution time for each number of clients.

These results are not surprising, because for the two patterns, their scenario and concurrent clients are generated by the same machine. They also use the same GlassFish server to transform REST calls to the queries. Therefore, the two pattens need the same time to simulate concurrent client requests.
In addition, the speed of the network connection between clients and data nodes is much slower than the response speed of data nodes. Although in the Competing consumers pattern, there are four data nodes handling the SQL queries, while there are only one data node handling the SQL queries in the Gatekeeper pattern, the total network connection time between clients and data nodes decides the execution time of an application of a cloud design pattern.
Hence, when the network connection becomes the bottleneck of the speed of the service, higher number of concurrent clients needs higher number of total requests, which need a larger amount of time to be executed.

6 What were the results of measuring energy consumption in scenarios?

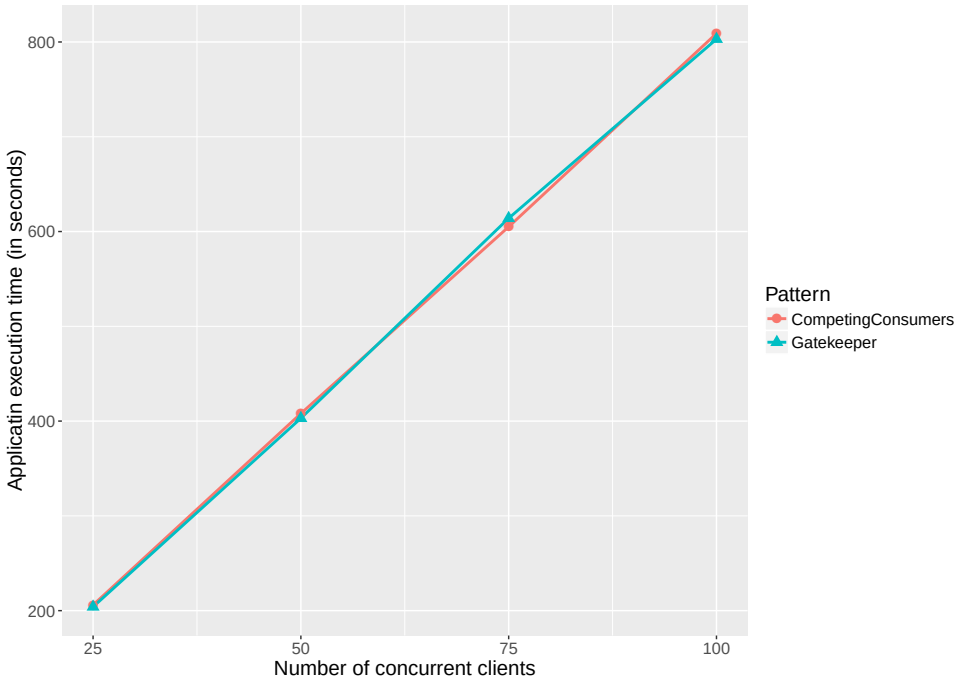

Figure 1: Comparison of the execution time (in seconds) of the Competing Consumers pattern against the Gatekeeper pattern.