Dan Morgan (55086407)
Mazhar Abbass (73877540)
Siddharth Joshi (30807928)

Billing Service Prototype with Kafka Middleware

Objective

The aim is to design a scalable distributed system which provides the functionality of a client-server billing application, using message oriented middleware(MOM). The project involves testing of the system under variable load conditions to evaluate performance. The simulation results of a basic and a scaled version of the system are considered for the evaluation.

Proposed Work

For the stated objective we are proposing to use a Kafka cluster as middleware platform which will be backbone of the whole billing system. We are going to use set of producers which will produce data in certain topics depending on payment type.

Clients which will send a payment request to the proxy servers, which sits between the Kafka cluster and the clients. This configuration simulates the actual configuration what you can find in various applications. Proxy servers read data from proxy topics. Now this topics will contain data published by the backend servers. Backend servers read from the transaction topics and process the messages. Now this messages sent and received as protobuf. As protobuf, provides more advantages over normal XML or JSON format.

Proxy once again reads from the proxy topics which contain message id, and payment confirmation or error message if any. This configuration allows multiple clients to simultaneously do transactions without much delay.

Existing Work

For the case of building a distributed system , the choice of the architecture depends on the application. A typical billing service cares about metrics like scalability reliability and speed of the system . There is a wide range of work already done as well as ongoing research , in the evaluation of need based middleware architectures. Messaging systems like Kafka , RabbitMQ, ActiveMQ , Redis etc are in a fierce competition to satisfy the demands of the customers. RabbitMQ and kafka have persistence and good throughput , ActiveMQ on the other hand is well suited for enterprise level applications whereas Redis is really fast. A survey of different technologies especially RabbitMQ and Kafka answered the question of which technology to use for our prototype[8]. The existing work related to our project revolves around these message brokers. For similar applications like transactions processing, financial services etc these are the top contenders.

Project Approach

The overall goal of the project was to produce a system prototype, not an actual usable billing service; therefore, we mostly aimed to analyze Kafka's performance, scalability, and usability, and less to produce a usable system. Toward that end, we deployed a Kafka broker cluster and our own custom code on Amazon EC2 t2.micro instances, which have the benefit of being free but are very resource-constrained.

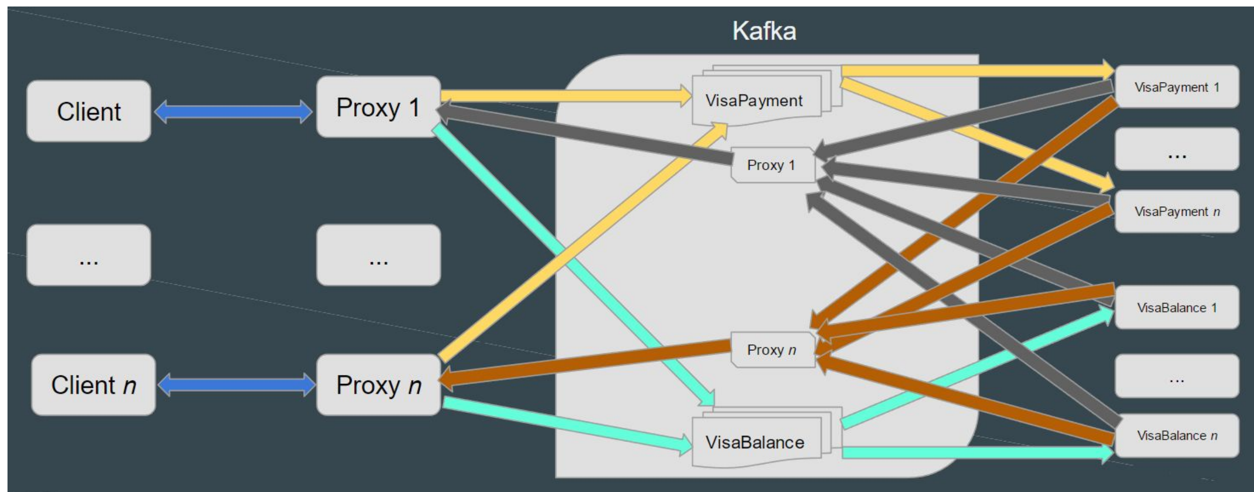
Our custom code was developed in .NET (specifically the .NET Core framework version 1.1), and Java. The .NET client, proxy, and backend "payment" and "balance" processors were developed in Visual Studio 2017 as C# console applications that can run on Windows, Mac, or Linux. .NET code for this project can be found on GitHub here https://github.com/Jensaarai/CS237-billing-service. The Java code can be found here on github https://github.com/sj2753/cs237; Java code includes only a backend "payment processor," which we didn't end up using because the paradigm of "only consume the latest messages" was not as straightforward for the Java Kafka library as it was for C#.

In the .NET applications, a few third-party packages were used as well. gRPC (http://www.grpc.io/) with Protobuf was used for communication between the clients and proxies, and a Protobuf message, serialized to a byte array, was also used as the Kafka payload. Confluent's C# Kafka library was also used (https://github.com/confluentinc/confluent-kafka-dotnet) as the Kafka driver for the .NET applications.

The .NET client console application generates many parallel requests of a user-input number that round-robin to configured proxy servers. The client keeps a stopwatch for each request to record the round-trip time for each request as seen by the client. The requests are sent through generated gRPC APIs, *MakePayment* and *CheckBalance,* as described in *billing.proto*. When the proxy receives a request, it reads the requested payment method and publishes a new message to the Kafka cluster to a topic that corresponds to the payment type, e.g. Visa or Mastercard, and the request type, e.g. payment or balance check. For our prototype implementation, only Visa payments and balance checks are sent by the client and handled by the proxy. The message a proxy publishes also includes a "reply-to"-type field, *proxy_topic*, that identifies the proxy-specific Kafka topic that this particular proxy is consuming from (each proxy generates its own custom topic name upon startup). The backend servers consume from their configured topics, currently only "VisaPayment" and "VisaBalance", and Kafka consumer groups are configured for those topics. This tells Kafka to ensure that each message published to the various topic partitions are given to only one consumer in the group; this gives us our parallelization by allowing us to add more backend servers to consume more messages in parallel. After the backend server determines if the payment should be accepted or denied, or what the client's balance is, it publishes a serialized Protobuf response to the topic specified by the proxy, from which the proxy has a consumer. The proxy consumer reads the message, matches it against a transaction ID for the particular client request, and sets the message as the result to a TaskCompletionSource set up earlier. This wakes up the server-side handling of *MakePayment* or *CheckBalance*, which finally returns the response to the client.

Finally, when the client has received responses to all of its requests, it prints the total number of requests, the shortest request time, longest request time, and average request time for its set of payment requests and its set of balance requests, and total time taken for all requests.

The .NET applications were built against the *ubuntu.16.04-x64* runtime and each were deployed to their own Ubuntu Server 16.04 AWS EC2 instance. The .NET Core SDK was installed on each AWS instance as described here https://www.microsoft.com/net/core#linuxubuntu which allowed the applications to be run. Three clients, three proxies, three backend payment processors and three backend balance processors were deployed for a total of twelve Ubuntu instances for our custom code. A system diagram is shown below:



Kafka cluster

We used EC2 instances from Amazon Web Services to host out kafka multi-node broker cluster. We used EC2 free tier eligible t2.micro instances whose specifications are: which use 1 CPU and offer 1 GB of RAM. This are the baseline instances and hence they are not very powerful or performance optimized.
First of all to even run, zookeeper and kafka broker on the dedicated instance we needed to deallocate the default heap size which is required for zookeeper and kafka broker. In all three instances the heap size has been kept to 256 MB. This is very low considering the fact that Kafka when used with right hardware and software configuration it can process data up to 2 million writes per second. But, this instances do not scale up to that level.

The instances mentioned here are tied to the static ip offered by Amazon and each instance of the broker does have its own Zookeeper and Kafka broker running. We used in total 3 instances each with zookeeper and a single kafka node hosted on port 2181 and 9092 respectively.

Zookeeper configuration files have been changed so that it can recognize brokers on different server as well. All the broker configuration files have been changed as well such that default partition value for

any created topic is 5 and leader imbalance check is done frequently. Moreover considering the limits of the machine we have decided to change the log retention period from the default as well. Server configuration files recognize the zookeeper ensemble, zookeeper ensemble is responsible for broker related management tasks. Each broker announces its public ip as advertised host name. This was the basic configuration. Total topics created were for VISA, MASTERCARD and Proxies. System can tolerate up to 2 zookeeper failures. Maximum replication achievable is 3 which is number of brokers. We could have set up multiple brokers on each instance but we haven't only because of limits on heap size and limited capabilities of RAM.

Google's Protocol Buffers

Protocol Buffer (PB) is a platform neutral extensible way of serializing structured data for use in communications protocols, data storage, and more. Google has open sourced this internal development tool as 'Protocol Buffer' in 2008[7]. As compared to xml and other data formats , it is more compact and robust since it is works in an object oriented manner. Once the Protocol Buffer interface is defined, the rest of the implementation process is straightforward. Protocol Buffer saves data in binary format, so there is no overhead for parsing as in XML which leads to fast and easy data transfer over the a communication channel.

Evaluation

To test and evaluate our application, we planned two different approaches. First, we wanted to run some benchmark scripts provided by Kafka itself, and second we wanted to run our own set of tests with the applications we wrote. Unfortunately, we were not able to get the Kafka benchmark scripts fully working, so we only have results for the tests with our own applications.

We ran tests at three different rates of client requests, one set when only one backend processor was consuming from the Visa topic, and another when three backend processors were consuming. In both cases, three clients were running sending requests to three proxies, and all three proxies were publishing to the Visa topic. Therefore, we wanted to see the system's throughput with one backend consumer, and then with three when Kafka distributed the Visa topic's partitions between the three.

These are the results from our tests. In these tables, the number of requests is the number of simultaneous requests *per client,* so "5,000 requests" is a total of 15,000 simultaneous requests from the three clients round-robining to the three proxies, and either one, or three, backend processor consuming all 15,000.

Number of Requests	One Backend Processor	Three Backend Processors
5,000	Shortest Response: 1.864751s Longest Response: 3.095911s Average Response: 2.378374s	Shortest Response: 1.883761s Longest Response: 3.168537s Average Response: 2.416574s
10,000	Shortest Response: 1.737001s Longest Response: 4.198972s Average Response: 2.673143s	Shortest Response: 1.681944s Longest Response: 4.292545s Average Response: 2.653532s
25,000	Shortest Response: 1.770683s Longest Response: 7.885425s Average Response: 4.65759s	Shortest Response: 1.662826s Longest Response: 7.893259s Average Response: 4.449785s

As we can see, three processors were slightly faster, except in the 5,000 (x3) case, which may be because of batching in the Kafka driver being more efficient at higher loads. At 10,000 (x3), three processors was less than 1% faster, but at 25,000 (x3), throughput was almost 5% higher. Since our throughput is much, much lower than the theoretical Kafka throughput, it's likely that if we could scale out further, and with more powerful machines, we'd see a much larger increase in throughput.

After discussion with Professor Venkatasubramanian, she recommended we try to simulate more real-world behavior in the backend processors to see if that would make a difference in response time and scalability. A random 100-1000ms jitter delay was added to the payment processor, and the balance processor was created to give a second, longer-lived request type that had a random 2-5 second jitter delay. Here, we can see that as we scale up, our throughput increases by a much larger percent: at 10,000 (x3), we're already 9.5% faster, and at 25,000 (x3), we're 15.9% faster. It's feasible that if we could push this farther, we'd see even better results, as we haven't seemed to have hit Kafka's limits.

Number of Requests	One Backend Processor	Three Backend Processors
5,000	Shortest Response: 3.283162s Longest Response: 6.223261s Average Response: 4.533328s	Shortest Response: 3.028235s Longest Response: 6.359679s Average Response: 4.411553s
10,000	Shortest Response: 2.91652s Longest Response: 8.782683s **Average Response: 5.122519s**	Shortest Response: 2.738824s Longest Response: 8.203394s **Average Response: 4.637534s**
25,000	Shortest Response: 3.01406s Longest Response: 17.11741s **Average Response: 9.17651s**	Shortest Response: 2.692739s Longest Response: 14.15711s **Average Response: 7.725362s**

Overall, we exercised the Kafka auto-consumer rebalancing functionality, and showed that it could easily re-distribute load across multiple consumers to provide an increase in throughput.

Conclusion

Kafka works excellently well as message oriented middleware, but in our case its performance is limited by the hardware, if the same system is scaled over large clusters and better hardware we could achieve significant progress in the latency and throughput. Moreover, we are certain that this type of setup with increased sophistication can be used at industrial scale.

Future expansion

In future, we could migrate whole system to hardware platform which provides more storage and speed. As we mentioned in evaluation, increase in partition count and throughput which should have improved latency time affect much. This was mainly because of limitation and we were facing memory full exception which could be because low heap size.

Moreover our backend which is currently in .NET can also work with Java backend. Performance analysis could have been improved. We can also improve incorporated monitoring service, which is useful as cluster size grows. There are many errors which are only found as we scale. Incorporated monitor service on sample data service may help to discover errors early. This are some of proposed changes in existing project.

Works Cited

[1] John, Vineet, and Xia Liu. "A Survey of Distributed Message Broker Queues." *arXiv preprint arXiv:1704.00411* (2017).

[2] Zhang, Tianning. "Reliable event messaging in big data enterprises: looking for the balance between producers and consumers." *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 2015.

[3] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." *Proceedings of the NetDB*. 2011.

[4] Jones, Brett, et al. "RabbitMQ performance and scalability analysis." *project on CS* 4284 (2011).

[5] Wang, Zhenghe, et al. "Kafka and Its Using in High-throughput and Reliable Message Distribution." *Intelligent Networks and Intelligent Systems (ICINIS), 2015 8th International*

Conference on. IEEE, 2015.

[6] Wang, Guozhang, et al. "Building a replicated logging system with Apache Kafka." *Proceedings of the VLDB Endowment* 8.12 (2015): 1654-1655.

[7] Kaur, Gurpreet, and Mohammad Muztaba Fuad. "An evaluation of protocol buffer." *IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the*. IEEE, 2010.

[8] Magnoni, L. "Modern messaging for distributed systems." *Journal of Physics: Conference Series*. Vol. 608. No. 1. IOP Publishing, 2015.

[9] Kleppmann, Martin, and Jay Kreps. "Kafka, Samza and the Unix philosophy of distributed data." *Bulletin of the IEEE CS Technical Committee on Data Engineering* (2015).

[10] Goodhope, Ken, et al. "Building LinkedIn's Real-time Activity Data Pipeline." *IEEE Data Eng. Bull.* 35.2 (2012): 33-45.