Front End - Back End Server Architecture With Apache Kafka Distributed Logging System

Artidoro Pagnoni Henrique Vaz

Abstract

In this paper we analyze the problem of client side persistent connections. Persistent connections, as opposed to single use connections, have become standard in all modern browsers and are necessary for many applications. However, the classical client - server architecture does not take into account possible failures on the server side, which would reset the connection. We propose a front end - back end server architecture with the intermediary of a distributed logging system to solve this problem. In our implementation, we decided to adopt Apache Kafka (Wang et al., 2015) as logging system. Our experiments show that the proposed architecture is tolerant to back end server failures. It also has the added benefit of introducing an abstraction layer between the client facing functionalities and their implementation. The rest of the paper is outlined as follows. In Section 1, we introduce the problem of persistent client connections in more detail and present related work on this topic. We also describe Apache Kafka and explain some key features of the system. In Section 2, we give a detailed motivation of our design choices and explain how we implemented them. Finally, in Section 3 we present the experiments we ran and the results we obtained along with a discussion of the implications of such a design. All the code for the front end - back end server architecture that we describe in this paper is publicly available here.

1. Introduction

Persistent connections, also called HTTP keep-alive, come from the idea of using a single TCP connection to send and receive multiple HTTP requests/responses, as opposed to opening a new connection for every new request/response pair. The adoption of this type of connections provides several benefits as documented by Mogul (1995). First, it reduces the latency of subsequent requests, as there is no handshaking required. Fewer round-trips and no handshaking also means reduced CPU usage. Persistent connections also make it possible to have HTTP pipelining, which consists in sending multiple requests on the same connection without waiting for a server response. Furthermore, having fewer TCP connections reduces network congestion, which is an important feature in distributed systems with limited networking resources. Finally, there is only a marginal extra cost for reporting errors or other types of secondary messages, and in particular there is no extra penalty for closing a connection. Persistent connections do come with a cost on the server side. When a client does not close the connection after all of the data it needs has been received, the resources needed to keep the connection open on the server side will be unavailable for other clients. This will affect the server availability. In many applications, however, the drawback of persistent connections is outweighed by the added performance benefits. They are therefore often preferred to single use connections.

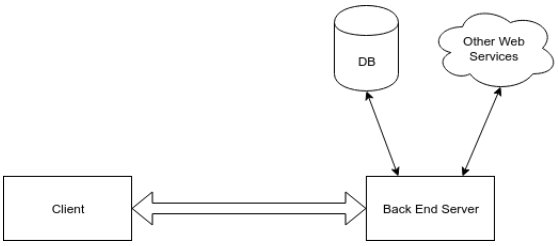

Figure 1. Classical client - server architecture.

Applications that rely on persistent connections depend on the assumption that the connections will not be broken. This fact justifies the importance of minimizing the probability that a client - service persistent connection is broken.

The traditional client - server architecture in the context of persistent connections does not take into account sever side failures. We show in Figure 1 such an architecture. In this setting, the client opens a connection directly with the back end server. This connection might be a persistent connection, but it would be dropped in the eventuality of either ends encountering a failure. We assume that the client side failures are not under control of the service provider,

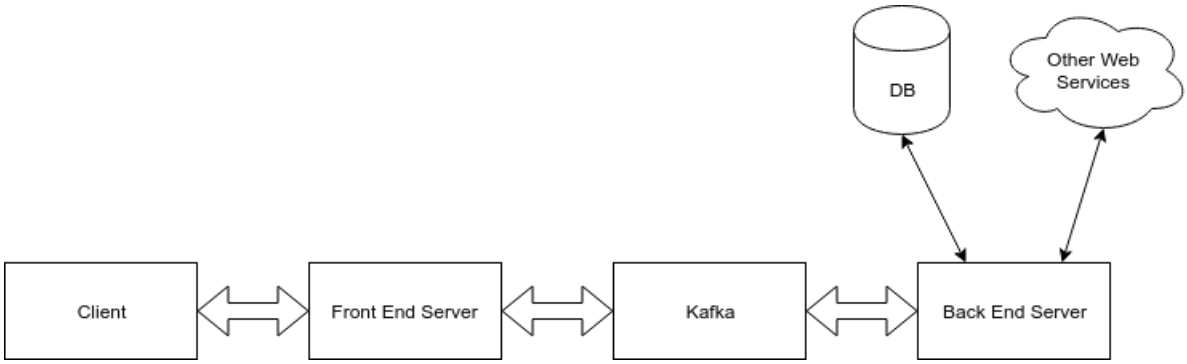

Figure 2. Front End - Back End Architecture, example with one client.

and consider network failures as unavoidable constants. The goal of this work is not to redesign network protocols and reduce network errors, but instead to address ways in which service providers can reduce the probability of client side persistent connections being dropped.

Some applications that currently heavily rely on persistent connections are for example: streaming services, gaming services, and navigation services. In all these examples, the clients continuously receive data from the server. Starting a new connection for each request/response would add a significant performance drawback. It is important to realize that failures on the server side in this context would kill the persistent connection between the client and server and cause negative consequences for user experience. Applications in which clients are connected for a very short period of time, or in which requests are spaced out, do not benefit on a practical level by the use of persistent connections, and are thus not seriously affected by back end failures and connections being dropped.

In this paper we propose a new front end - back end server architecture to solve the problem of broken persistent connections from back end failures. The main objective of this architecture is to decouple the connection point - the front end - and the main processing tasks of the service, which happens in the back end servers. Under this scheme, all client requests are made to the front end servers (connectors), which are then responsible for making RPC calls to the back end servers. The RPC calls are submitted to an intermediary (a distributed queue system or logging system), which is responsible for forwarding the calls to the back end server cluster. There is wide literature on the topic of distributed messaging systems (kaf) (apa) (IBM). All system logic is implemented in the back end servers, which interacts with databases and other web services.

In Figure 2 we illustrate the main components of the front end - back end architecture and how they are connected (the image only shows one client connection). Notice that the client establishes a persistent connection with the front end server, and does not directly connect to the back end. With this architecture, administrators only need to make updates in the back end servers, since the front end servers are always performing the same task. As a result, failures or predicted updates happening in the back end servers do not cause the clients to disconnect from the service. Our proposed architecture does however introduce other possible sources of failure, and latency. We will explore these more in detail in the following sections.

Let us see how this architecture provides benefits in the example cases mentioned earlier beyond the general back end failure case. Gaming services can use this system to update their metagame without having to disconnect players. In other words, players can keep playing the game while its metagame logic is being updated (for instance, the reward you get after accomplishing a specific task). A similar architecture has in fact been suggested (Coplien, a) (Coplien, b) for gaming applications. Streaming services can use this system to update recommendation systems without having to disconnect users from the platform. It could also be applied to updating other features applicable to the service. GPS services like Waze and Google Maps could use this system to update routing algorithms without having to disconnect the users which are currently using them at the time of the update.

Another interesting advantage of our system comes up in the event of app updates. Different users update apps at distinct times. This means that upon releasing an app update, we can use our front end servers to route clients using version x of the app to back end versions that were built specifically to support version x. In other words, the front end servers allow us to match clients and servers accordingly. One could argue that you could execute this matching task using a load balancer. However, such an approach would be extremely painful to maintain and also time consuming. This would involve creating a new domain each time the app is updated to a newer version, a task that gets

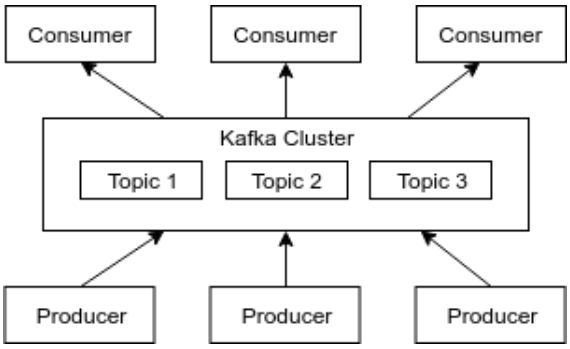

Figure 3. Overview of Kafka Distributed Logging System

even worse if we are using SSL.

The distributed logging system that we use is Apache Kafka (Wang et al., 2015). This distributed system was developed by Linkedin, and is widely used in practice. From previously published work Kafka has been demonstrated to be a distributed messaging system providing fast, highly scalable and redundant messaging through a pub-sub model. Kafkas distributed design gives it several advantages. First, Kafka allows a large number of permanent or ad-hoc consumers. Second, Kafka is highly available and resilient to node failures and supports automatic recovery. In real world data systems, these characteristics make Kafka an ideal fit for communication and integration between components of large scale data systems. Kafka itself is a distributed system: it is run as a cluster on one or more servers that can span multiple data centers.

We now introduce key features of the Kafka system. The Kafka cluster stores streams of records, or messages, in categories called topics. Each record consists of a key, a value, and a timestamp. The entities that write streams of records to Kafka are called producers. They can publish data to the topics of their choice. Any record that is published to Kafka has to have a topic assigned to it. On the opposite end as producers, there are consumers which subscribe to one or more topics and process the stream of records produced to them. Consumers receive any record that was published with a topic to which they have subscribed. Lastly, Kafka, as a distributed system, runs in a cluster. Each node in the cluster is called a Kafka broker. Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data. If a leader fails, a replica takes over as the new leader.

In Figure 3 we show a diagram representing the Kafka architecture. In our setting both front end and back end servers are producers and consumers, but they will write and consume from different topics.

An enhancement of the usual client - server architecture that we presented in Figure 1 is the addition of a load balancer between the client and the back end servers. This is an almost universally accepted practice as it is demonstrated by the large literature on the topic (Cardellini et al., 1999). Load balancers can be misinterpreted as serving the same function as the combination of front end server and the intermediary distributed logging system. In reality they serve complementary functions that can be integrated. The load balancer redirects connection requests to back end servers while ensuring that the back end servers are evenly loaded. As shown in Figure 1 bellow the clients then establish a connection directly with the back end servers that they were assigned to, and the scheme turns into the classical client - server connection architecture. The only difference being that not all clients connect to the same servers, and which servers they connect to is determine by momentarily going through a load balancer. This architecture has obvious benefits of distributing the requests across all servers, but does not answer the problem of ensuring that client - server persistent connections resist to server failures, which is what our work has focused on.

A metaphor that might help better understand load balancers is to compare client server interactions with the immigration lines at United States airports. A load balancer serves the same purpose as the people who direct the flow of incoming non-residents to one of the ten to fifteen lines assigned to an immigration officer. Each person, or client, has only a one time interaction with the load balancer, while the persistent connection is established with the immigration officer, or back end server. If the immigration officer has to leave the position, for whatever reason then the people in that line will be far from pleased and will have to find another line, which we would call reestablish another persistent connection. This explains why the load balancer does not solve the issue that we are focusing on.

In Figure 5 bellow, we show how a load balancer can be integrated with our front end - back end architecture. In a few words, if all the clients established persistent connections to the same front end server it might get overloaded, a load balancer instead can evenly distribute the connections among the available front end servers. This does not affect the benefits that our proposed architecture provides regarding back end server failure tolerance.

2. Design Choices

In the following sections we give a detailed motivation of our design choices and explain how we implemented them. There are five main parts to the complete architecture illustrated in Figure 5: client, front end server, Kafka cluster, and back end server. We will go in each one of them in detail. First however, we have to mention that the various

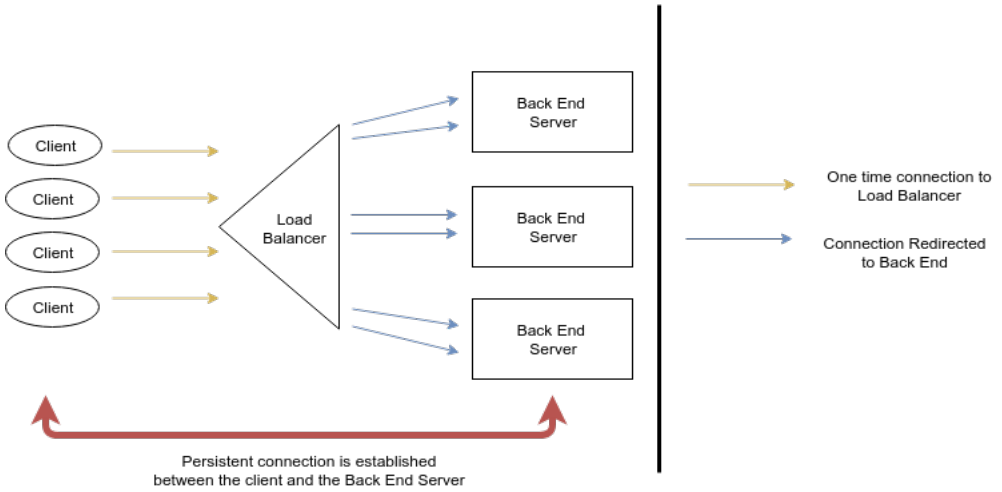

Figure 4. Client Server Architecture with Load Balancing

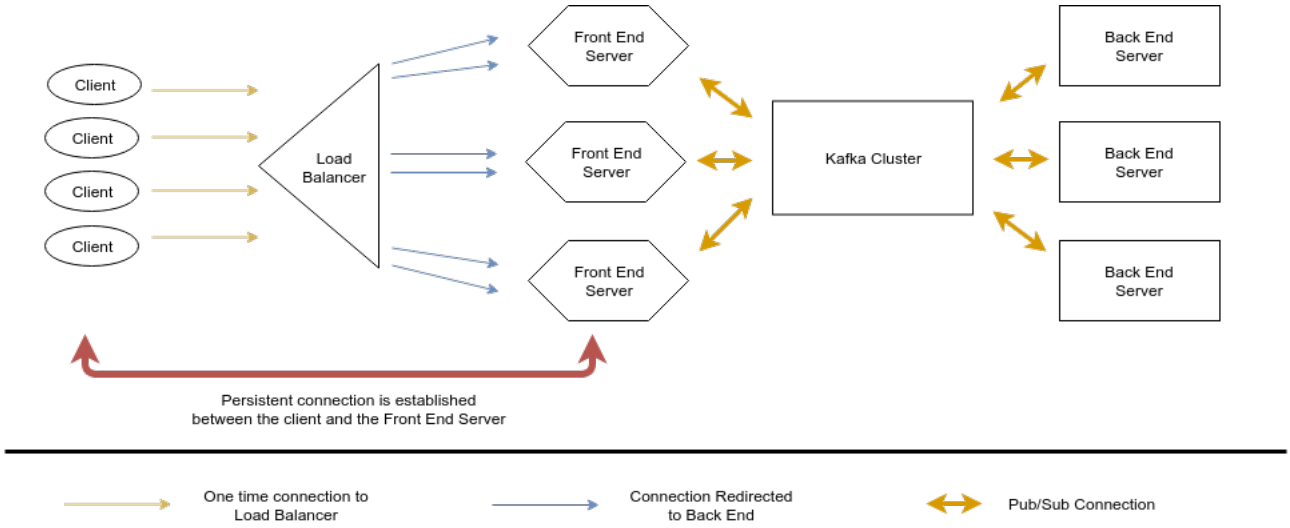

Figure 5. Front End - Back End Architecture integrated with Load Balancing

components of our system communicate with a message protocol defined as follows. A message is composed of the following two parts: message = <header> + <payload>. The header itself is divided into three elements: header = <version> + <payload length> + <opcode>. The specifics of the opcodes and version numbers are included in the code documentation. Defining our own protocol gave us the flexibility to only include only the essential pieces of information that were required in our application.

For the following sections detailing the components of the front end - back end architecture refer to Figure 6 for a detailed schema of the networking diagram.

2.1. Client

The client is very similar to the client found in traditional client - server architectures. The client establishes a persistent connection with the front end server, and exchanges requests and responses with the front end. None of the complexity of this architecture is handled on the client side. The case for persistent connections is already a trade off that prioritizes service availability and performance for the client as opposed to server side resource efficiency.

2.2. Front End Servers

In the proposed architecture it is essential that the front end server is made as simple as possible. It essentially only

behaves as a forwarding agent that transfers requests and responses between the client and back end. This code is made as light as possible in order to reduce the possibilities of failure or added latency. Our architecture works under the assumption that the front end server will not require frequent software changes, at least compared to the back end servers. This is another reason why we build the front end as a forwarding agent.

In general the front end server has four main roles:

- Listen for client side requests
- Transfer client requests to the back end
- Listen for back end responses
- Transfer the response to the client

The client side communication involves establishing a persistent two way connection with the client. This is just the standard persistent connection, similar to the classical client - server connection.

The communication with the back end has to go through the Kafka cluster. This communication goes in both directions, and therefore using the pub-sub vocabulary, the front end is both a producer to and a consumer from Kafka. Both producers and consumers are bound to the topics the write to or to which they subscribe. In this case it is therefore important to define the topics that the front end will consume from and the topics to which the front end will produce to.

First, we assume that each front end server has a unique identifier. As producer, the front end writes all messages with topic defined by its front end unique identifier. This means that all messages produced to Kafka by front end servers are grouped under a topic that keeps track of which front end server they came from. This will be important for the back end server as we will see bellow.

Second, the front end will consume topics corresponding to each unique client connected to it. This means that each client connection will have a unique identifier. This could be a client connection token, a user name, or any other identifier. And the front end will listen to messages that are written under the topic of clients that are connected it. This ensures that a front end server receives messages that are directed to the clients that are connected to it, while other front end severs do not.

Finally, when the client - front end server connection is dropped the front end server stops consuming from the topic of the client that just disconnected.

2.3. Kafka Cluster

The Apache Kafka distributed logging system is used as the main intermediary between the front end servers and the back end servers. This additional layer allows to decouple the connection point between the client and the entities executing the requests. Kafka is the key element that allows the persistent connection between the client and the front end to be tolerant to back end failures.

The two main characteristics of the Kafka system that allow this property are the following (Wang et al., 2015):

- Storage of streams of records in a fault-tolerant durable way
- Real time processing of streams of records as they oc-

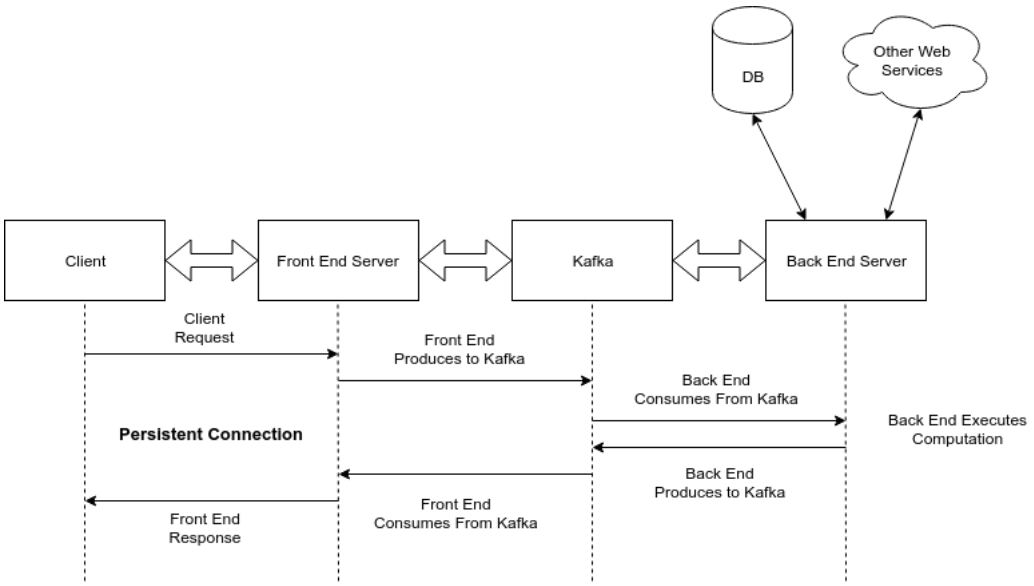

Figure 6. Front End - Back End Architecture Networking Details

cur

Kafka stores all messages, or streams of records, that are published to it. This allows the consumption of the messages to happen in an asynchronous way, and by entities that were not known to the producers of these messages. Clearly, this first characteristic is essential to ensure back end failure tolerance, which requires different back ends to serve the same requests seamlessly.

The second characteristic is important for real time processing of the requests. For most applications that we mentioned in the introduction, the latency has to be small. So the overhead of using Kafka as an intermediary has to be as little as possible.

Kafka is run as a distributed cluster of nodes. In our setting we choose the nodes to be replicas of the leader. The system handles node failures, including leader reelection in case of leader failure. This means that with $k + 1$ replicas, the Kafka server will be resistant to k node failures. In the future, we envision Kafka nodes to distribute load between topics, through the use of partitions. This functionality is built in Kafka, and can be useful in practice. However, to demonstrate server side failure tolerance, it was not necessary and we decided not to implement it.

2.4. Back End Servers

The back end server is where the main logic of the application will live. All the requests from the client in the proposed architecture are forwarded to the back end through the front end and the Kafka cluster. The back end then has to send the response to the client through Kafka and the front end. This means that the back end, just like the front end needs to be both a producer and a consumer. Here again, we have to define the topics in detail.

Since the back end servers have to listen to all messages from all the front ends, they will consume messages front all front end topics. We will assume that the front end unique identifier are known to the back ends.

The back end will produce to topics corresponding to each user. This means that the messages sent by the front end have to include the unique user identifier. Otherwise the back end cannot know which user the request is coming from, nor to whom to send the response.

2.5. Integration of Load Balancing

We showed earlier that our architecture can be integrated with load balancing and that the two are not mutually exclusive. In our implementation however, we decided not to implement the load balancer. In this project we aimed at achieving back end server failure tolerance, and the implementation of a load balancer would not have helped us demonstrate this property. Furthermore, this functionality can also be requested automatically by web service providers such as Amazon's AWS or equivalent. Therefore, we focused on experimenting with our architecture.

2.6. Failure Scenarios

We now analyze the failure scenarios and how they are handled under our proposed architecture. We already mentioned the possibility of network failures, and argued that they transcend the scope of the present work. The other possible failures are related to components of the distributed system: client, front end servers, Kafka broker nodes, and back end servers.

- Client:
  In this work we focus on the service provider architecture and client side failures, just like networking failures fall in another category of problems. We will therefore not deal with them in our work while beign aware that they constitute a possible problem.

- Front End:
  The front end failures would break the persistent client - front end connection. We argue however that these failures should be less probable than back end failures for the following two reasons: the front end runs very simple code that essentially just transfers messages across in either directions, the back end implements the logic of the application which is subject to changes and updates and thus programmed failures.

- Kafka node:
  The Kafka cluster is run on a set of independent replica nodes governed by a leader. All operations have to pass through the leader. The failure of a replica would therefore have little impact on the functioning of the system, and especially in maintaining the keep-alive connection client-front end. A failure in a leader node would be handled by the Kafka reelection system based on Zookeeper (Hunt et al., 2010). Notice however, that with $k + 1$ nodes in the Kafka cluster, the cluster is k failure resistant.

- Back End:
  The purpose of this architecture is to be back end failure tolerant. We will show in our experiments that this behavior is achieved.

3. Experiment and Results

In this section we present the results of experiments run with our architecture.

Most importantly, we decided to demonstrate the functionality of the proposed architecture by applying it to a chat

application. A chat application is first of all an application in which there are long standing persistent connections between client and server. This task is also simple enough to allow for thorough testing of the implementation. We briefly explain the main functionality of the chat app. The complete details are available in the code documentation. The chat app that we designed has the following functions:

1. Create an account by supplying a unique user name.
2. List accounts of all users.
3. Send a message to a recipient by providing the recipient user name. If the recipient is logged in, deliver immediately; otherwise queue the message and deliver on demand. If the message is sent to someone who isn't a user, return an error message.
4. Deliver undelivered messages to a particular user.
5. Delete an account and handle undelivered messages to deleted accounts.

3.1. Test Environment and Infrastructure

For setting up a reliable test environment that works independently of the underlying system, we used Docker. We have four different types of Docker images:

i. One image for the Kafka brokers, which also incorporate all necessary setup and configurations for Zookeeper. We used spotify/kafka image provided in the Docker Hub.
ii. One image for the front end servers, which we built ourselves
iii. One image for the back end servers, which we developed
iv. One image for the clients, which was also built by us

In order to simulate the behavior of a distributed system, we created a docker network which is shared among all docker containers. Using that, we are allowed to expose the containers as being independent hosts with their own IPs, which is of great help for running more reliable tests locally.

Another simple but very convenient tool we added to make the testing pipeline simpler and more robust were some bash scripts. Each one is responsible for spawning a different part of our system. Using that, we can easily kill and create different components of the system as often as desired, which is particularly important for testing the effectiveness of our system.

3.2. Main Experiment: Back End Failure

In our main experiment we test whether our architecture is back end failure tolerant. To do this, we set up several clients connected to a front end server. We also start up a Kafka cluster and a back end server. The hypothesis that we are trying to verify is whether the clients stay connected to the messaging server even in the eventuality of a back end server failure, or programmed update. To do this, we actually kill the back end server. After waiting varying amounts of time we restart a new back end server.

The result from the experiment are positive and the clients remain connected to the front end servers with a persistent connection. This experiment therefore shows that we have achieved back end failure tolerance.

3.3. Front End Failure

As already mentioned earlier, front end failures do brake the persistent connection between the clients and the application. This was expected, but we already argued why this is an improvement over the traditional architecture.

3.4. Kafka Broker Failure

Another experiment involved killing a Kafka node. When Kafka is only run on one node this breaks the internal logic of the application and produces errors. When Kafka is run as a cluster of nodes, there was no perturbation in terms of functionalities of the system.

We did not however do an analysis of performance overhead in the scenario of node failures. This would be interesting to explore in future work.

4. Conclusion

In conclusion we propose a front end - back end server architecture to provide server side failure tolerance. This helps reduce the risk of persistent connection failure with the additional cost of having an intermediary between the client and the server. This is not to be confused with the advantages provided by a load balancer, which should be integrated with the architecture that we are introducing.

We demonstrated that our architecture has the added benefits of introducing an abstraction layer between the connection point and the execution layer, possibly facilitating updates through the system that do not reach all servers concurrently.

Furthermore, we provide an implementation of the proposed system in the context of a messaging app. We show that the app achieves the desired back end fault tolerance.

To extend this work to be readily used in practice, it would be important to add the load balancing feature, and to ensure internal load balancing between Kafka broker nodes.

References

Apache activemq messaging and integration platform. URL http://activemq.apache.org/.

Kafka distributed streaming platform. URL https://kafka.apache.org/.

Cardellini, Valeria, Colajanni, Michele, and Yu, Philip S. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999. doi: 10.1109/4236.769420. URL https://doi.org/10.1109/4236.769420.

Coplien, James. Pattern: Map-centric game server, a. URL https://gameserverarchitecture.com/2015/10/pattern-map-centric-game-server/.

Coplien, James. Pattern: Distributed network connections, b. URL https://gameserverarchitecture.com/2015/10/pattern-distributed-network-connections/.

Hunt, Patrick, Konar, Mahadev, Junqueira, Flavio Paiva, and Reed, Benjamin. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010. URL https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems.

IBM. Ibm messaging platform. URL https://www.ibm.com/us-en/marketplace/secure-messaging.

Mogul, Jeffrey C. The case for persistent-connection HTTP. In *Proceedings of the ACM SIGCOMM 1995 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Cambridge, MA, USA, August 28 - September 1, 1995.*, pp. 299–313, 1995. doi: 10.1145/217382.217465. URL http://doi.acm.org/10.1145/217382.217465.

Wang, Guozhang, Koshy, Joel, Subramanian, Sriram, Paramasivam, Kartik, Zadeh, Mammad, Narkhede, Neha, Rao, Jun, Kreps, Jay, and Stein, Joe. Building a replicated logging system with apache kafka. *PVLDB*, 8(12):1654–1655, 2015. URL http://www.vldb.org/pvldb/vol8/p1654-wang.pdf.