TuKano: from a Toy Web Application to a Cloud-based Geo-Replicated Social Network

Ricardo Bessa
r.bessa@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Neel Badracim
n.badracim@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

TuKano is a social network inspired in existing video sharing services, such as TikTok or YouTube Shorts. The initial solution proposed in Distributed Systems course (2023-2024) was a local version of a toy application with services for Blobs, Users and Shorts. In order to increase the quality of service of TuKano in a real-world deployment scenario with a global user-base, we propose a Cloud-based version of this social network.

Index Terms - Microsoft Azure, Cloud Computing, Distributed Systems, Scalability, Availability, Quality of Service

1 INTRODUCTION

The original local version of TuKano was very limited in terms of scalability and availability, making it not suitable for a global use scenario. This way, the goal of this project was to make a good use of services available in Cloud Computing Platforms, such as Microsoft Azure, in order to create a Cloud version of TuKano that is scalable, fast and highly available.

The achieved result was the successful transformation of the original centralized solution into a Geo-Replicated one that supports a scenario of a Web Application of high popularity at a regional (continental) scale. The proposed solution also supports the scenario of an application with a global scale that targets clients spanning across multiple continents (Europe and North America).

In the following sections we propose the general architecture of the Cloud-based version of Tukano and then we describe our design choices about the different components of the solution. We present an experimental evaluation by showing the results of a set of performance/load tests on TuKano. Finally, we discuss those results and we present our final conclusions.

2 GENERAL ARCHITECTURE

TuKano was built with a Three-Tier Architecture, including the **Client Tier**, the **Application Tier** and the **Data Tier**. The last one also includes a Caching Layer to improve the data access performance.

The Client Tier is composed by all the client devices that can interact with the social network. Since TuKano is a Web Application, users can access the app using any device connected with the Internet such as personal computers or mobile devices. Clients can access the available services by sending HTTPS requests to the corresponding REST endpoints defined on the TuKano REST API. Services will be running on a Tomcat10 REST Server with a Java 17 implementation, available through a public Cloud URL.

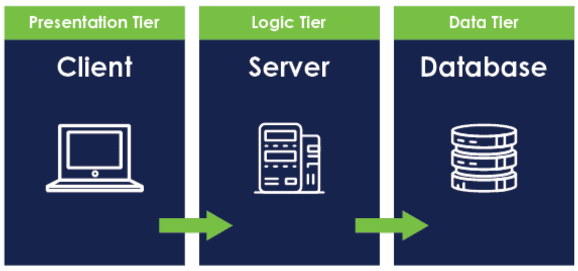

Figure 1: General Architecture

The Application Tier is a centralized implementation that comprises three REST services.

- **Users Service** can be used for managing users' individual information.
- **Shorts Service** can be used for managing the shorts metadata and social networking aspects, such as users' feeds, follows and likes. The social network aspect of TuKano resides on having users follow other users, as the main way for the platform to populate the feed with shorts that each user can visualize.
- **Blob Service** can be used for managing the media blobs that represent the actual videos. TuKano users can upload short videos to be viewed (and liked) by other users of the platform.

The Data Tier is responsible for managing all users, shorts and blobs data. To avoid too many requests to the data layer, our solution improves the performance of the original one by providing a Caching Layer.

3 WEB APPLICATION

For the Web Application we propose a modular solution that includes different implementations of the several services offered by TuKano. Those implementations follow a generic approach, allowing an easy extension of the provided services. The proposed Cloud version of TuKano can easily swap between these different implementations by just activating the right flags. The logic of each component is isolated to achieve a solution that is very compact, robust and and easy to extend. We provide different constructors and init methods to deploy the desired resources in a more automatic fashion, following a similar approach to the provided scripts.

For the Blob Service, we include the original version storing blobs as files in the local filesystem, a version that leverages the Azure Blob Storage and a serverless version that implements the same logic using Azure Functions.

For the Data Tier, we provide the original version with Hibernate and two versions using Azure CosmosDB, one with NoSQL and the other with PostgreSQL. To improve the performance of the Data Tier, we leverage a Redis Cache.

The advanced TuKano features were implemented also using Azure Functions and we perform Geo-Replication of the Web Application, the Azure Blob Storage and the Azure CosmosDB. In the Geo-Replicated scenario we include West Europe as the Primary Region and North Central US as the Secondary Region. More details about these components shall be provided in the following sections.

4 BLOB STORAGE

The original solution of TuKano stored video blobs as files in the local file system. This solution is good for the local version of Tukano because, in general, is fast to write to and read from the filesystem. However, for a global application the scalability of this solution is limited to the resources of the machine running the application. The availability is also very limited because users far away from the machine hosting the application will experience high latencies incompatible with the desired quality of service.

For a global scale we leverage Azure Blob Storage to store videos as raw bytes in a storage account. Following the same approach of the original solution, the Cloud version of the Blob Service supports the operations of creating a blob, delete a blob(s) and download a blob. Our solution also provides an Azure Blob Storage-based implementation of the special operation read to a sink. This feature simulates a real-life streaming scenario where, for performance reasons, a video is read progressively in chunks. We also provide a serverless version of the Blob Service based on Azure Functions. These functions implement the same operations of the original service and they are activated by HTTP triggers. The blobs are Geo-Replicated both in West Europe and North Central US. More implementation details shall be provided in the Geo-Replication section.

5 DATABASE STORAGE

The original solution to store user data and shorts data was a relational in-memory Hibernate database. This solution provides great performance because the database is stored in-memory. However, this implementation is also limited by the machine resources, and it don't scale well to a global scenario. We can find here the same problem about availability since far away users will experience high latencies. We can achieve more quality of service by leveraging Cloud databases. This way, we provide two different implementations with Azure CosmosDB, one with PostgreSQL for a database with a relational schema and other with noSQL that stores objects as items in containers. The type of database that TuKano should use can be selected using flags. Our solution supports one type of database for users and shorts, but we have created distinct flags for users database and shorts database to make it easier in the future to extend the solution to a scenario where multiple types of database could be deployed at the same time.

CosmosDB noSQL is an Azure database that stores objects as items with no relational schema. These items are stored in containers and they have an unique ID. CosmosDB supports horizontal scalability based on partition keys. CosmosDB creates logical partitions that can be distributed among the same machine or multiple machines. The ID attribute of an item is an obvious choice for the partition key. This approach was the one followed during our implementation. However, to improve the performance of some queries we would consider to use as partition key other attribute. For example, we could store shorts with the owner ID as partition key to have in the same partition the shorts of a specific user. This way the queries to get the shorts of that user would be more efficient. We organised the data in different containers. We created containers for users shorts, likes and following. One limitation of noSQL is that transactions and joins are only allowed in the same container. This way, queries that need to join information from different containers will have limited performance. In the most cases, the queries logic should be executed only at the database side. However, in this case, due to these noSQL limitations, we need to combine the information from different containers at the application layer, which may introduce some additional overhead. This happens with the query to get feed of an user because we need to join the shorts owned by that user and the shorts of the users he is following. We use ComsosDB with direct mode for a better performance.

CosmosDB PostgreSQL leverages a relational database with a schema which is specified in a properties file. We stablish a connection to a PostgreSQL cluster in CosmosDB. The properties file handles the specification of the tables in the database as well as the items primary keys and foreign keys. We have tables for users, shorts, likes and following. This strong schema makes, in general, this database slower but, on the other hand, the relations between objects makes some queries more efficient. This happens when deleting a user, we can easily delete also his shorts, likes and followings. The PostgreSQL account is linked to a node with burstable computation, 1V Core and 2GB RAM as it was the least expensive version.

6 REDIS CACHE

To avoid too much access to the Database Layer, we leverage a Caching Layer. This happens because each access to the database costs money and resources. This way, to improve the performance of read operations we store the popular items in the Redis Cache. In the most cases, we used a write-through approach, where the updates are executed both in cache and in database. This way, we keep the cache content consistent with the database content.

We keep in the cache **user cookies** to create more efficient user sessions. This way, during the cookie validity of 15 minutes, the user can perform very efficient operations. This happens because during the user session, we assume the user is active and we avoid to make requests to the database to check his credentials. The cookie is the hash SHA-256 of the user password. If the user is authenticated, we avoid to do the same authentication process multiple times in a short period of time. The cookie sas an configurable expiry time that is renewed if the cookie is accessed again before becoming invalid.

The users authentication process is also executed using tokens for the uploading blob operation. To fix the token verification logic, we increased the token validity and replace the user token verification in the operations delete all blobs and delete all shorts by the password verification since these operations manipulate multiple shorts and there was no defined user token to perform them. This way, the password verification is the most suitable option is these operations, also because it's very fast due to the cookies implementation.

In the cache we also keep hot contents such as the **recent shorts**. We leverage special methods to update the cache if a stored short was removed . This may introduce some overhead but our goal was to reach a balance between performance and consistency with the database. The recent shorts are kept in a list with a fixed-size. This way, shorts that are not accessed in a while will be removed from cache due to the list trim operation. This avoids to flood the cache with old content since we want to keep only hot content, in this case shorts that were created or accessed recently to make the get short operation more efficient. In general we assume that a short that was recently created or accessed will be accessed more times in a short period of time following the first interaction.

We store in the cache the results of **complex queries** too. This way, we avoid to preform the same heavy computations on the database multiple times. It's the case of the query to get a feed of an user. This query needs to combine lots of information. To optimize that, we keep the result of that computation in cache under a small expiry time. During that time, the feed owner can see an old version of the feed but the refresh process happens with enough frequency (1 minute) in order to avoid too much inconsistency with the database. If the user creates or deletes some short that he owns, that update is reflected in the cache feed to keep at least some consistency level with the actions executed by the user. However, during the feed validity, the user will not see the shorts published by the users that he follows. This is the consistency price we need to pay to achieve best performance for the get feed operation. The result of counting likes from the database is also stored and updated in a cache counter for each user short. This way, we avoid to access many times the database only to count the number of likes of a short with some associated overhead. When there's no space in cache, the eviction policy LRU will remove the recent least accessed items. If a like counter is removed, next time the count likes operation will be executed at the database side. By keeping the counter in cache, we still execute updates both in cache and datbase for consistency reasons but read operations can be executed at the cache, improving their performance. This applies to a situation where a user publishes a short and check many times in a short period of time the information about that new short, including the number of likes. In each request, we avoid to perform the query multiple times in the database and still return an updated version of the number of likes read form cache counter.

In general, with CosmosDB we get Session Consistency Level and we try to keep the cache also as consistent as possible with the database, without forgetting that we must always sacrifice some consistency in order to provide better performance and availability, which is the main goal of the Cache Layer.

The advanced features were also implemented with the help of cache operations, more details will be provided in the next section.

7 AZURE FUNCTIONS

Our solution leverages Azure Functions to implement a **serverless** version of the Blob Service. Assuming that users will not upload, delete and download blobs all the time, a serverless solution might be more efficient in terms of resources used and money spends. With this implementation, instead of relying on a server continuously running and providing that service, we provide the same operations in Azure Functions that are activated by HTTP triggers. These functions also handle the geo-replication process of blobs.

To implement the advanced feature **counting views**, we added a views counter attribute in shorts objects. Every time a short is downloaded, we update a view counter for that short in the cache. To avoid too many requests to database while still refreshing the views fast enough, we leverage a write-back cache policy. During 1 minute, the views are only updated in a cache counter for the corresponding short. After that, a periodic trigger will activate an Azure Function that collects the accumulated views from the past minute, clean them from cache and write all the updates back to the database to achieve consistency with the cache.

To implement the advanced feature **TuKano Recommends**, we leverage an Azure Function that is triggered periodically. This function executes the views write-back operation to update all views and then selects from the database the short with higher number of views from the shorts created in the last 5 minutes. The chosen short is republished with a special ID by the TuKano Recommends user. A new user automatically follows TuKano Recommends and, this way, he will receive these recommendations. In a period with low user activity, there might be no shorts in the past 5 minutes or in that period the most viewed short maybe was already recommended. In this situation, TuKano will not recommend the same video twice, he only recommends new shorts. This captures the idea of real social network recommendations based on views. A recent video with lots of views will be the best candidate to be republished.

8 GEO-REPLICATION

Our solution supports a scenario with users spanning in two different continents, Europe and North America. West Europe will be TuKano primary region since we expect the most part of the user-base to be there. However, we must also ensure quality of service for users in the secondary region of North Central US. To achieve that, we geo-replicated the three main components of TuKano.

Web application was deployed on both regions. Users should use the correct URL base on their geographical proximity to one of the deployment regions. For testing purposes we do that by manually assigning the right URL based on a region flag in the app. In a real scenario, we would use an Azure Load Balancer or Traffic Manager accordingly to our research.

CosmosDB was deployed in Spain Central region and then we activated the geo-replication of the database to North Central America, following the instructions in the Azure Portal. All of the described steps about all components replication can be observed in the screenshots provided in the directory geo-replication of the project.

For **Blob Storage**, we activated GRS-RA redundancy level for North Europe, to provide more durability and availability in case there is a region failure. To improve the quality of service of North Central US users, we always write blobs in the primary region and then propagate asynchronously that operation to other Storage Account in the secondary region. This way, the read operation is executed in the closest Blob Storage based on the region flag in the app. The replication logic is provided both to the Azure Blob Storage Service and it is also implemented with Azure Functions in the serverless version of the same service.

9 EXPERIMENTAL EVALUATION

For the experimental evaluation, we prepared six Artillery scripts with different operations. We have **users.yaml** to create users, reading from a CSV file with up to 1000 different users. We have **blobs.yaml** to, for each user, create a short, upload a blob, download a blob and delete all blobs. The blob contains just a couple of random bytes for testing purposes. We have **db.yaml** that executes a mix of write and read operations in the database for each user, including create shorts, follow other random user, like a short, get the user shorts, get the user feed, get the user likes and delete all shorts of the user. We have also **delete.yaml** that deletes all the users from the database. Finally we have **writes.yaml** and **reads.yaml** that execute only write or read operations from the database, including the ones mentioned above.

Total Time (s)	Rate (RPS)	Created Users
30	5	150
30	15	450
30	33	990

Table 1: General statistics of the experiments

We executed our own test scripts only. From the Artillery logs we extracted the mean response time as the general latency of the test. We execute the same tests for three different rates. We tested with arrival rate 5, 15 and 33. The increase of the rate means there are more concurrent requests per second. All tests have the duration of 30 seconds. The tests were executed for the running application deployed in an Azure machine. Due to the free student tier, we only used a more limited machine, which is not best option for real performance tests. However it's enough to extract some interesting metrics about TuKano.

RAM	Cores	Bandwidth	Storage
1GB	Shared (60 CPU minutes/day)	165MB/day	1GB

Table 2: Azure machine specification

For **Blob Storage** we tested the different implementations, including the original one with filesystem. For all tests the app is running in Cloud. For serverless blobs, we needed to modify the blobs URLs to use the Azure Functions URL.

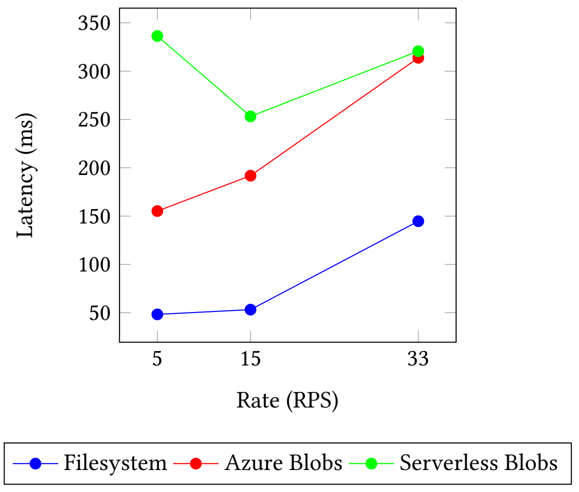

Figure 2: Latency vs Rate for different Blobs

As we were expecting, the latency increases with the rate of requests. we can observe that the filesystem is the fastest solution. However it is not feasible for a global scenario since it provides very limited scalability and availability to far away users. Serverless blobs in general are slower than the other options. It's hard to estimate their latency since the code needs to be uploaded and executed somewhere in West Europe where the functions were deployed. This can take some time and it's hard to study. During our tests, we observed many different fluctuations in latency of serverless blobs but, in general, we can conclude that they can be a bit slower than the classical server solution. However, this latency variation may be not very significant and serverless blobs continue to be a valid option because the money and resources are only allocated when there's a need for that and in the right amount. Azure server blobs can be faster but during periods of low activity, we are spending money and resources continuously to provide a service that can have low periods of activity. As for the best choice, in the end the serverless blobs follow a modern trend of serverless computing and they can be a good option here although the server blobs may have more predictable response time.

For testing the different **database** solutions, we started by using the script db.yaml to test a mix of write and read operations.

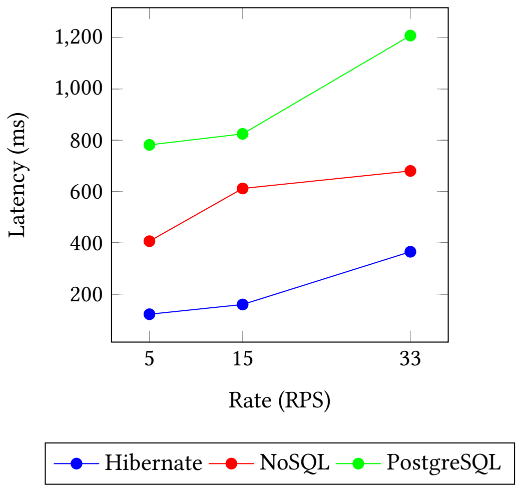

Figure 3: Latency vs Rate for Different Databases

As it can be observed, the Hibernate solution is the fastest one because the database is in-memory. But this solution scalability is limited by the machine resources and the availability is also very limited for far away users. The best option appears to be CosmosDB NoSQL because the data objects are treated as items stored in containers, following a very light logic and no strict schema. The PostgreSQL solution is slower because it forces a strong relational schema, which introduces some additional overhead.

However, there are trade-offs to consider. For example, complex queries that need to combine different information, like fetching a user's feed that includes their own shorts and the shorts of users they follow, are challenging in NoSQL databases. Deleting a user also implies deleting their shorts, which requires multiple queries in NoSQL. In contrast, relational databases like PostgreSQL execute these operations with a more simple and efficient approach by leveraging primary and foreign key relationships. The following chart shows the latency for the delete user operation.

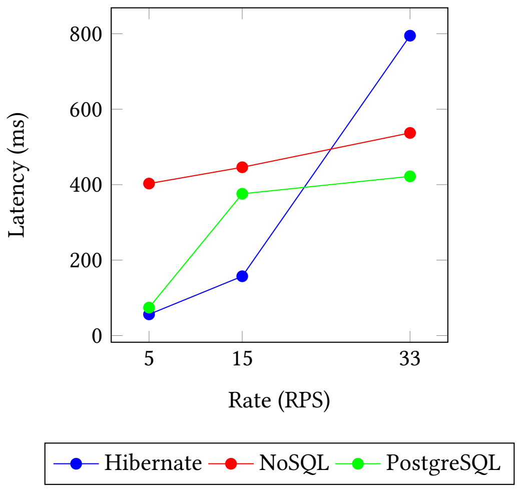

Figure 4: Delete User Latency vs Rate for Different Databases

As we can perceive from the chart, for the delete user operation the PostgreSQL solution is faster than noSQL solution as we were expected. Same applies to the get feed operation. In the end, both Cloud databases are valid options. NoSQL is generally more fast but PostgreSQL is more efficient for complex queries due to the relational schema that is imposed.

Next we look into the performance of read and write operations in CosmosDB noSQL with and without a Redis Cache. The purpose of this experiment was to evaluate the impact on the performance while using a Redis Cache and what kind of operations are affected. The experiments were all conducted under rate of 5 RPS and tests of 30 seconds in total.

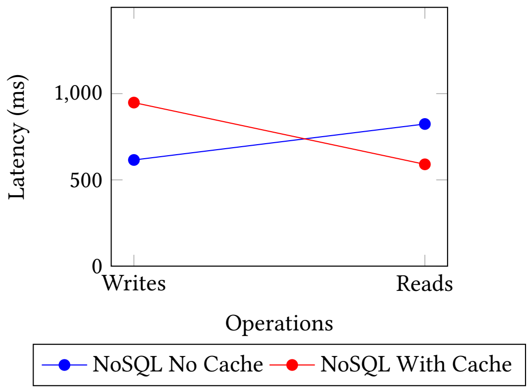

Figure 5: Latency (ms) for Read and Write Operations

We can conclude from the chart that the read operations performance was highly increased by use of the Redis Cache as expected. However the write operations were much slower with the cache because of the overhead of writing both in cache and in database. Our assumption is that in a social network the read operations will be more frequent than the write operations and, in this scenario, using a Redis Cache could be a good option to increase the quality of service of the application.

Up to this point all tests were executed from the perspective of a user in Western Europe where all the resources were deployed. Next we take into account the scenario of a global use with users also in North Central America. To evaluate the perspective of those users, we allocated an Azure Ubuntu VM in West US 2, closest available region of North Central America for our price tier. The VM has a Standard D2s v3 (2 vcpus, 8 GiB memory), a virtual network and a public IP. We can access that IP by doing SSH with the defined credentials. First we connected to the machine by using a Shell environment provided by the Azure Portal. We swapped to privilege mode (sudo -i) and install in the VM the npm and the Artillery dependencies. Next we moved our Artillery scripts to the machine by doing scp of those scripts.

Finally, we ran the scripts to obtain the latencies from the perspective of a far away user. The next step was to compare that latencies with the ones after the geo-replication was activated. To do this, we deployed TuKano in North Central US, changed the base URI of the Artillery scripts to use the secondary region URI and change the app region flag to North Central America. We also activated the flag to do the blobs asynchronous replication to the secondary region and we activated the CosmosDB replication to North Central US as well. Now we ran the tests again and collected the results. In both scenarios we conducted the tests of 30 seconds with rate 5 and using the scripts blobs.yaml and db.yaml. For the database we used noSQL without cache.

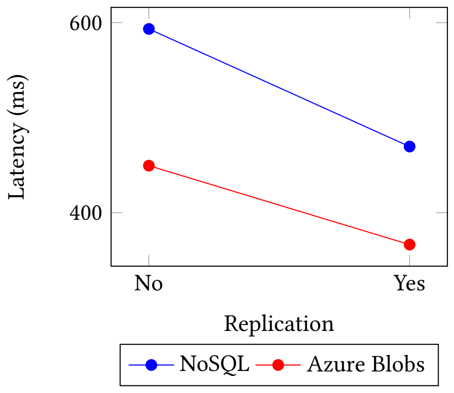

Figure 6: Impact of geo-replication on US Latency (ms)

From the chart we can perceive the positive impact of geo-replication in the latency perceived by US users. The region to where we send the requests has a great importance in the quality of service of users. For example, in our initial logs we tested the latency of users in Europe when they try to directly contact the application running in North Central US and obtained much higher values as expected. Some screenshots of the process of running the tests in the US VM will attached to the report directory at the GitHub repository.

10 CONCLUSIONS

By our experimental evaluation we can conclude that the goal of increase the quality of service of TuKano in a real-world deployment scenario with a global user-base was achieved with success. We provide a Cloud-based solution that makes a good use of Microsoft Azure services to give the original TuKano levels of scalability and availability that are compatible with a real life scenario deployment with a global user base.

At our personal side, we believe that we fulfilled our high expectations and we feel proud of the result that was achieved.