CS 237 Project Report

Real-time Chat System based on AWS

Group 9
Yatong Duan, 36369159
Jiashu Xiao, 84404198
Xuan Chen, 69562437

1. Introduction

Nowadays, real-time chat is a popular way of communication and interaction on the website. The first real-time chat system was known as Talkomatic, developed by David R. Woolley and Doug Brown in 1973. Up to now, users in real-time chat systems could transmit the text message but also rich media such as video, audio, emoji, gifs or stickers. Besides, real-time chat comes in many forms which can be one-to-one chat or one-to-many groups chats.

To construct a more scalable and cost effective system, we mainly implement a serverless real-time chat system since it doesn't depend on a big model web server on the back end that users need to maintain. Bidirectional or Duplex communication between the client and the server eliminates the need for the client to poll for new data. What's more, since data will be sent after the connection is constructed, there's low latency in requests and responses as there's no need for sending additional packets for connection establishment.

The chat system we will build represents a complete serverless architecture. In Amazon API Gateway, we could build bidirectional communication applications by using WebSocket APIs without having to provision and manage any servers. We can respond to any API call through invoking a lambda function.

In this quarter project, we plan to mainly use the Amazon API Gateway, AWS Lambda and Amazon DynamoDB to build a real-time chat system. Specifically, we intend to construct the high-level architecture of component that are launched automatically as followed:

1. Amazon S3 bucket is created to store static web page content of chat systems.

2. APIs are implemented by using AWS Lambda functions and exposed as web APIs via the Amazon API Gateway. Meanwhile, API Gateway WebSockets and Lambda functions manage WebSocket routes.

3. By using Amazon DynamoDB, the Lambda functions are designed to be stateless and can use persistence tier to read or write data. Besides, storing WebSocket connection IDs is completed by creating Amazon DynamoDB.

2. Related work

To better understand the serverless system, and help pick the proper category of serverless and database to deal with the request in a real-time chat system, it's necessary for us to first do a survey on appropriate middleware, with detailed analysis on the benefits of these two aspects.

2.1 Serverless

In serverless concepts, the customers are only concerned with the desired functionality of their application and the rest is delegated to the service provider. There are two common categories of serverless. [1] Firstly, serverless was first used to describe applications that significantly or fully incorporate third-party, cloudhosted applications and services, to manage server-side logic and state. These are typically “rich client” applications—think single-page web apps, or mobile apps—that use the vast ecosystem of cloud-accessible databases, authentication services and so on. These types of services have been previously described as “(Mobile) Backend as a Service" (BaaS). Secondly, serverless can mean applications where server-side logic is still written by the application developer, but, unlike traditional architectures, it’s run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party. One way to think of this is “Functions as a Service” (FaaS).

There are some successful commercial implementations of this serverless model such as Amazon Lambda, Google Cloud functions [2] and Microsoft Azure Functions [3]. Compared with BaaS, the use of Function as a Service is making it easier to scale code and providing a highly cost-effective solution to implement microservices. Thus, we pick AWS Lambda [4] as a computing service that lets customers run code without provisioning or managing servers. AWS Lambda runs code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. The customers just need to supply their code in one of the languages that AWS Lambda supports.

Many real-world serverless applications have been proposed in the literature. Real-time collaboration and analytics is one of the applications.

Serverless services provide an attractive platform for real-time collaboration tools such as instant messaging and chatbots. An architecture for chatbot on OpenWhisk [5] was proposed by Yan et.al., [6]. [7] introduced an XMPP-based serverless approach for instant messaging.

In addition, real-time tracking is another instance of collaboration tools that are very suitable for serverless services as these applications don’t mainly rely on the system’s state. Anand et.al., [8], [9] proposed two real-time GPS tracking methods on low-power processors.

Serverless services are also used for data analytics applications [10]. In these applications, various sources stream real-time data to a serverless service. The service gathers, analyses and then represents the data analytics. It is possible to handle concurrent massive data streams using the auto-scaling feature of serverless computing.

2.2 Dynamo

Dynamo [11] uses a synthesis of well known techniques to achieve scalability and availability.

1. Partitioning algorithm: Dynamo must satisfy scale incrementally, which requires the dynamically partition the data over a set of nodes. Dynamo ‘s partitioning scheme depends on the consistent hashing to distribute loading to multiple storage hosts.

2. Data versioning: faced with multiple versions of the same object, Dynamo represents the vector clocks to capture causality different versions of them. In vector clocks, Dynamo will store ten states of pairs at most by deleting the oldest one.

3. Handling temporary failures: instead of using traditional quorum approach, Dynamo uses sloppy quorum hinted handoff to ensure the read and write operations are not failed because of temporary node or network failure.

4. Handling permanent failures: Dynamo implements an anti-entropy protocol to keep the replicas synchronized. By using Merkle trees, Dynamo could quickly detect the inconsistencies between replicas and minimize the amount of transferred data.

5. Membership and failure detection: a gossip based distributed failure detection and membership protocol. Dynamo is a highly decentralized system which maintains a globally consistent view of failure state.

Dynamo provides infinite read/writes per seconds to make users not worry about the capacity, up-time, back-up. Dynamo maintains high scalability to handle three parameters increasing in our project.

3. System implementation

In this section, we will introduce the architecture and implementation of this serverless real-time chat system. After comparing various serverless computing platforms and databases, we choose AWS Lambda and DynamoDB to implement our system.

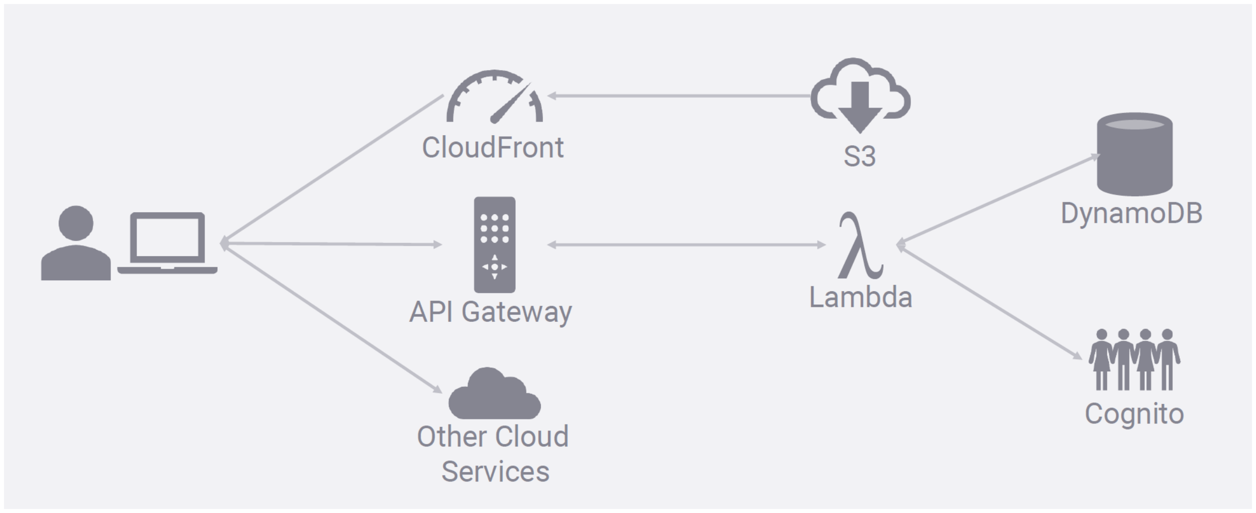


Figure 1. System architecture

Figure 1 shows the main architecture of our chat system. We present the main functions and some setup details of Amazon web service.

S3 (Simple Storage Service) - We store static HTML files in the S3 bucket and vend it to users' browsers.

Lambda - We create six lambda functions to store and retrieve data in the chat system. Chat-Messages-GET/POST functions are responsible for reading chat messages from DynamoDB and writing information to DynamoDB. Similarly, Chat-Conversations-GET/POST and Chat-Users-GET functions are used to store and retrieve corresponding information.

API Gateway - It presents a well-modeled API to the clients and automatically generates client-side code to communicate with it in the system.

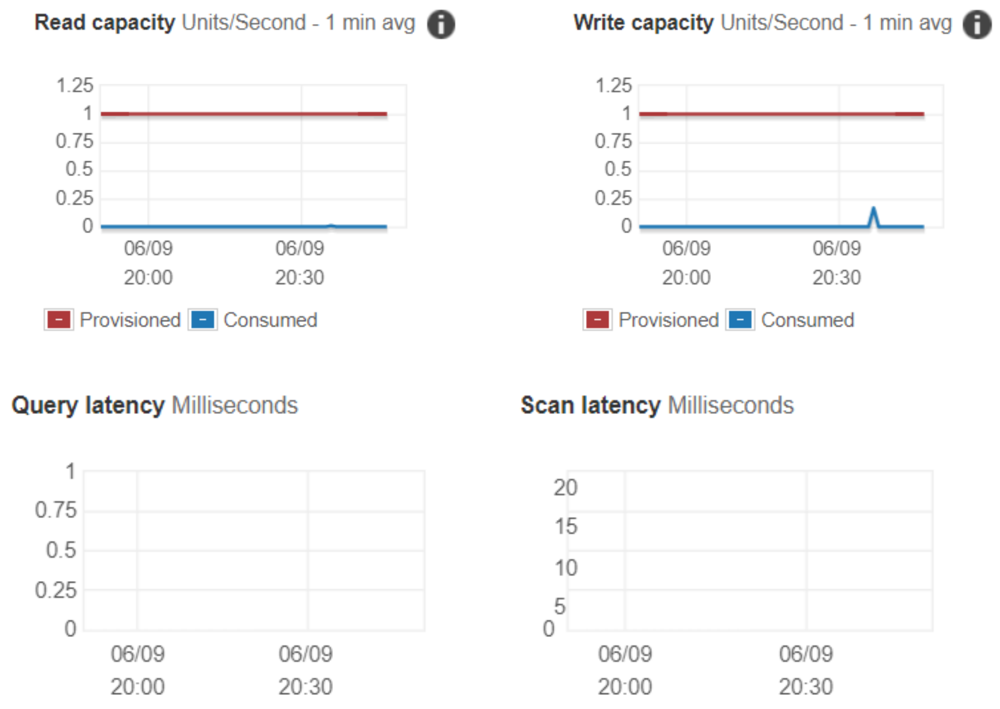
DynamoDB - We use Dynamo to store and retrieve data at scale. We create two tables: Chat-Conversations and Chat-Messages. The partition key of the Chat-Conversations table is conversation id and the sort key is username. In the Chat-Messages table, we use conversation id as partition key and set up timestamp as sort key.

Cognito - We create a user pool using Cognito to manage users, account creation, and logins securely.

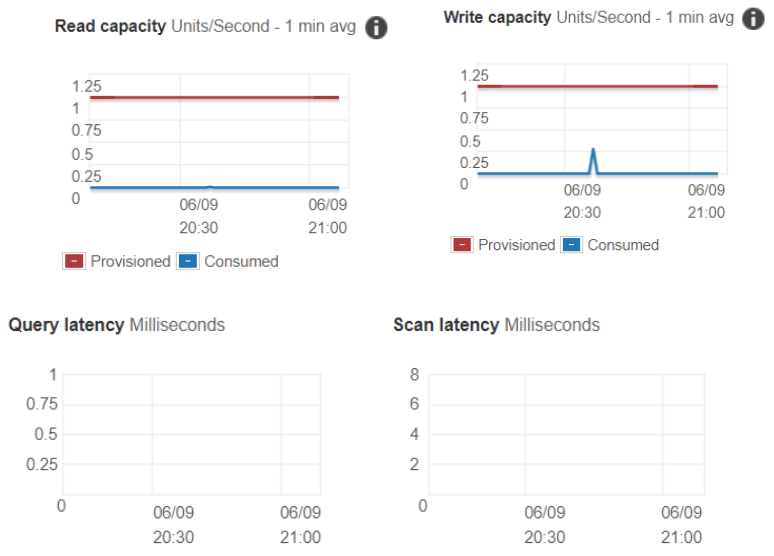
IAM (Identity and Access Management) - It secures access to the services.

4. Test and evaluation

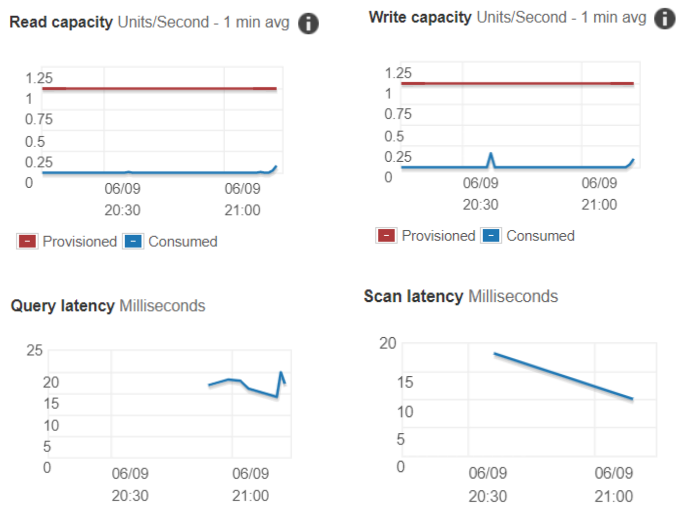
Firstly, we deleted all users, conversations and messages and recorded the state of DynamoDB as follow:



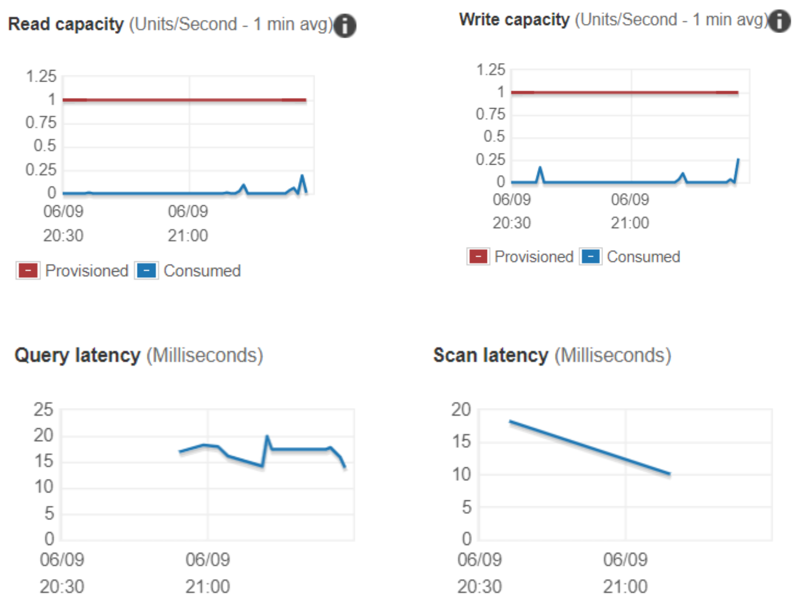
Then we added 10 users to this chat system. We did not save users information in DynamoDB, so we assumed that there should be no change in DynamoDB. The results confirmed our prediction.



After that, we added 5 conversations to this system and got results of Conversation table as following:

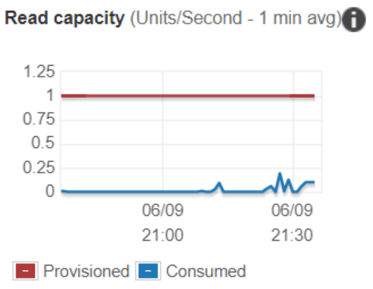


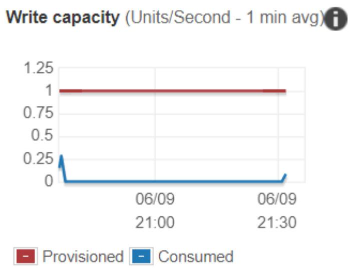
Later, we added another 15 conversations in this system. At this point, we got 20 conversations. States of Conversation table is below:

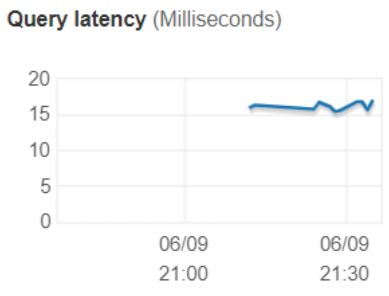


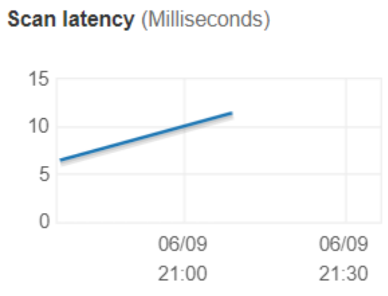
From the observations above, we can see that as the number of conversations increases, the latency does not fluctuate greatly and the capacity always keeps around a small value.

Then we added 5 messages to this system, we got results of Messages table as following:

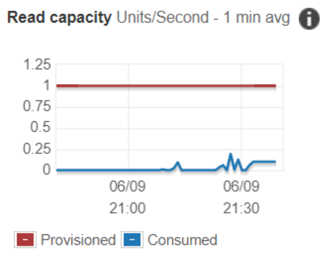


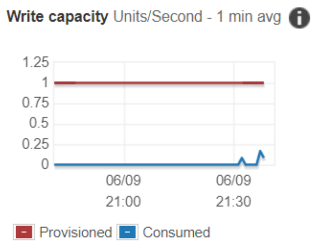


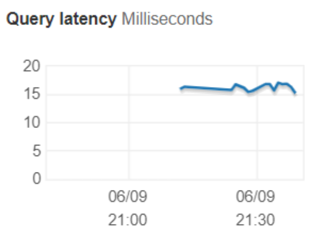


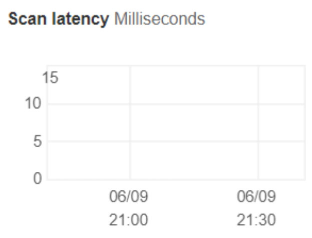


Finally, we added other 15 messages to this system and got results of Messages table as below:









From the observations above, we can conclude that as the number of messages increases, the latency does not fluctuate greatly. Besides, the latency and the capacity always keep around a small value.

Therefore, the capacity and latency of DynamoDB always have a good performance when the number of messages and conversations increase. In other words, DynamoDB has good scalability.

5. Conclusion

In this project, we implement a real-time chat system based on AWS. We design the structure of our system. To build our system, we choose serverless computing and DynamoDB to maintain the scalability and availability. We also simulate the users for testing. After that we compare the result of Query latency, Scan latency, capacity in Dynamodb with a different number of users in our simulated environment, and analyze the results, which agrees with what we expect to see. We can do more about this in the future, for example constructing a group table to implement a group chat model.

Reference

[1] H. Shafiei, *Member, IEEE,* , A. Khonsari, and P. Mousavi, “Serverless Computing: A Survey of Opportunities, Challenges and Applications,” 10.13140/RG.2.2.32882.25286
[2] “Cloud functions,” https://cloud.google.com/functions/, ac- cessed: 2019-10-7.
[3] “Azure functions,” https://azure.microsoft.com/en- us/services/ functions/.
[4] “Amazon lambda,” https://aws.amazon.com/lambda/.
[5] “Openwhisk,” https://openwhisk.apache.org/.
[6] M. Yan, P. Castro, P. Cheng, and V. Ishakian, “Building a chatbot with serverless computing,” in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. ACM, 2016, p. 5.
[7] P. Saint-Andre, “Serverless messaging,” 2018.
[8] S. Anand, A. Johnson, P. Mathikshara, and R. Karthik, “Real-time gps tracking using serverless architecture and arm processor,” in *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE, 2019, pp. 541–543.
[9] “Low power real time gps tracking enabled with rtos and serverless architecture,” in *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 2019, pp. 618–623.
[10] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, “A server- less real-time data analytics platform for edge computing,” *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.
[11] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store.", ACM SIGOPS 2007.

We showed how to chat by this system in this demo video:
https://drive.google.com/file/d/1zWEvS2LENGwMsQhFvI0KxJ1ML6ums6H7/view?usp=sharing