**San Jose State University**

**CMPE-277**

**Project-1: LifeStream**

**Team**

Manoj Dhoble([manoj.dhoble@gmail.com](mailto:manoj.dhoble@gmail.com))

Prateek Sharma ([sharmaprateek84@gmail.com](mailto:sharmaprateek84@gmail.com))

Srinath Seshanarayanan ([srinath.seshanarayanan@gmail.com](mailto:srinath.seshanarayanan@gmail.com))

Sugandhi Sugandhi([sugandhi.sharma@gmail.com](mailto:sugandhi.sharma@gmail.com))

Content

[Abstract 1](#_Toc352767811)

[Introduction 1](#_Toc352767812)

[Candidate Technologies 1](#_Toc352767813)

[Problem Statement Breakdown 2](#_Toc352767814)

[Design 2](#_Toc352767815)

[System Components 3](#_Toc352767816)

[Multi Server Start Up 3](#_Toc352767817)

[Configuration File 3](#_Toc352767818)

[Health Monitor 4](#_Toc352767819)

[Message Routing 4](#_Toc352767820)

[Event Driven Communication and Asynchronous Message handling 5](#_Toc352767821)

[Geo Spatial Data Handling 6](#_Toc352767822)

[Proto File 6](#_Toc352767823)

[JPA 6](#_Toc352767824)

[Clients 6](#_Toc352767825)

[Limitations of Solution 6](#_Toc352767826)

[GPB 7](#_Toc352767827)

[Future Enhancements 7](#_Toc352767828)

[Failure Handling 7](#_Toc352767829)

[Desktop Client 7](#_Toc352767830)

[Data Replication 7](#_Toc352767831)

[Collaboration 7](#_Toc352767832)

[Conclusion 8](#_Toc352767833)

[References 9](#_Toc352767834)

# Abstract

Project LifeStream is a system intended to store and stream images in a distributed system. The distributed nature of the system makes it easy to scale and also provides in built redundancy to take care of node failures and node overload. Due to the distributed nature of the system and the size of data that would be exchanged, a technology that would transfer data without adding much overhead and also do this asynchronously would be required. Search is another key use case for the system and users could search the images based on different metadata parameters like the location where the image was taken, the time when the image was taken or in general for images uploaded by a given user.

# Introduction

The Design and Architecture for the had to take into account the Scalability, Redundancy, Asynchronous handling of User Request, Ability to Store Geo Spatial Data, Ability to Handle a multitude of client who could be implemented using different languages, and the possibility of querying images based on different metadata parameters. Based on market research and considering the non-functional and functional requirements of the system, a set of candidate technologies were selected that would best fit the desired implementation.

Since this would the first phase of the project, the emphasis is on building a scalable backend system that would use a simple Java, Python or C++ Client to perform the various operations ranging from user create, remove and query to image upload, remove and query.

# Candidate Technologies

Image Upload and Download is one of the main use cases for the system and hence using a data exchange format that was light, efficient and easy to use was required. Google’s Protobuf was a good fit, since it exchanges data in a binary format and is amongst one of the lightest payload transfer technologies available in the market.

Scalability and handling the large number of user registration and image upload was another key use case for the system. For designing a system that is elastic in nature and can be scaled up or down based on user load meant that the system had to be distributed and should be able to communicate easily with each other. To handling asynchronous communication, Netty server was a good option. Netty is a Java library and API primarily aimed at writing highly concurrent networked and networking applications and services.

For storing Geo Spatial data we have used PostGIS DB. Post GIS is based on lightweight geometrics and indexes optimized to reduce disk and memory footprint. It also lets user perform queries with longitude and latitude information associated with data.

User data information storage and retrieval is done using JPA to make it independent of the underlying storage infrastructure and to make it compatible with any future storage technology decision we may make. Image storage and retrieval is not done using JPA since, it won’t able to handle the geometric information associated with the image.

We also intend to create a Python and C++ library to showcase the versatility of the solution and the ability of system to be a platform for clients to build application for desktop or mobile devices, as desired.

# Problem Statement Breakdown

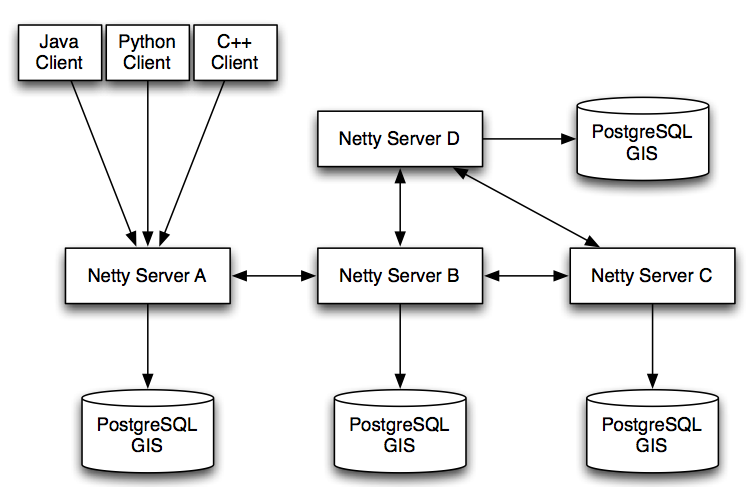
The requirement statement was quite open and a lot was left out for users design choices. The first challenge we faced was with understanding the problem statement and relating it with the technology choices and technologies we learned in class. The unfamiliarity with most of the technologies that was suggested added to the confusion. In order to break down the problem, we decided to take a bottom up approach, dealing with the requirements we were clear on first and then slowly work towards improving the understanding level and then work on the other components.

The following steps were followed to understand the requirements clearly and come up with the design for the system that would be future proof and take care of the non-functional requirements as well.

1. Understand the core Netty Project that was provided and figure out how Netty works.
2. Figure out the role of queuing to improve load handling ability of the server.
3. The different use cases with a queue and explore rabbitMQ implementation.
4. Understand GPB and how it can be tuned to meet our use case requirements
5. Ability and the features of Post GIS when dealing with Geospatial data.
6. How does a C++ client provide a value add and how does it help with image processing
7. Handling user session management and would memcached be a good fit for this
8. What would be the best way to keep the servers in sync and would Zookeeper be a good fit.
9. Also, later on in the project as we understood about JPA and Hibernate, how and where it would help to use JPA.

# Design

Researching on the candidate technologies, we came with a system design that would best fit out requirement.



# System Components

Since we decided to follow a bottom up approach with the implementation of the system, we decided to start with design aspects that we were familiar with and then proceed step by step.

## Multi Server

For achieving a multi-server design, we had to get multiple servers running and them communicating with each other. The design also needed to be flexible such that the multiple servers could be started in the same host or different hosts with port and host information that could be controlled outside the server code. We decided to have a central configuration file that could track this information and it could be kept in sync between multiple server instances by the system administrator or by using tools like Zookeeper, if needed.

## Configuration File

The server configuration file had to take care of storing information regarding multiple servers. On top of this it also acted as a central location to track the all the messages that the client could send to the server. This gave the flexibility of each server to handle or not handle a particular message type in future, if we find that system is bombarded with a certain message type and handling this is causing delay in handling other higher priority messages.

The configuration file retained information regarding the routing between the servers, since each server even though is identical in design, is not identical in behaviour. With this flexibility we could design serves with only a single point of entry or multiple point of entry for any client.

The connection parameters of the various database schemas is also maintained in the same centralized configuration file and based on the changes we make to this section, we can make all the servers to use different or same database.

The parsing of this configuration file and making this information available easily for the server programmer is done with the help of JAXB and the JSON Encoder/Decoder.

## Health Monitor

This is a key component in our design. This piece of the system takes care of monitoring the other nodes and the always ensuring the nodes are up before sending the messages to the other node.

We implement the Circuit Breaker pattern here, which keeps checking the health of the other nodes in the system on a regular basis, the duration of which is configurable, and retains the state in a local data structure on the server. Before forwarding a message to another node, the status of the node is checked in the data structure to avoid unwanted failures and cascading effect of the failure.

The design is simple and yet quite powerful in the bigger picture of the system.

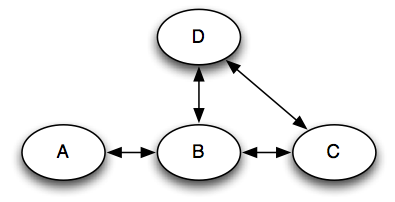
## Message Routing

With any system design with multiple nodes, load balancing and message routing between these nodes is key. Since we don’t have an external load balancer in our system, the servers needed to have the intelligence for load balancing and message routing. An overlay network is created for the nodes to control the message routing and the sever implements health monitoring also based on the node visibility.

In an ideal external load balanced environment, the client would be exposed a VIP and the load balancer would route the messages to different servers based on the routing and persistence strategy. We wanted to retain a similar logic without an external load balancer. The client would be exposed to a single node, and then based on the key in the incoming request (hash of the user\_id), the message is routed to one of the available servers. This is a simple implementation of the routing and doesn’t add any unwanted computation to slow down the request. The beauty of this implementation being, the key i.e the user\_id would be associated with all incoming requests, be it user creation or document upload. Also, for a given user the hash would be the same always and hence the same server and hence the same database will take care of the request for a given user. This also helps us avoid database replication.

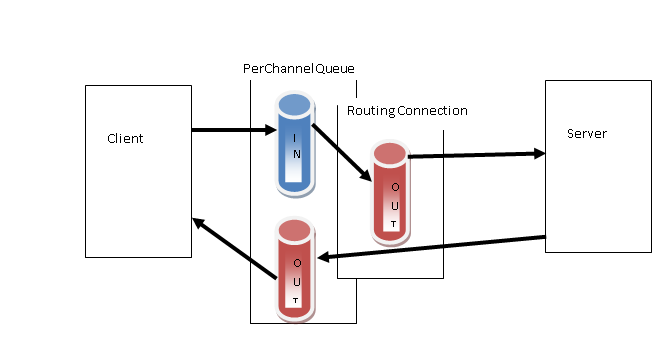
The second challenge to tackle was to make the routing efficient and avoid unwanted hops to reach the destination. We wanted to compute the shortest route and do in such a way that even if in the future, all nodes are not treated equally and the spatial separation between them ends up being different, the routing mechanism wouldn’t have to be tweaked. So we decided to use a proven algorithm that is quite often used for networking problems, the Dijkstra’s Algorithm for finding the shortest path We create a Graph for the nodes during server start-up and when a user request arrives at the server, shortest path between that node and the destination node computed based on the hash is calculated. Once we know the destination node that has to handle the request, we find the next node that has to handle the request and forward the request to that node. Each server has this intelligence and would end up computing the same shortest path and hence the route information doesn’t need to be passed on with any forwarded request.

The Overlay network that we have implemented as a proof of concept is given below. The design can be scaled to add more nodes and implement more complex network layouts.



## Event Driven Communication and Asynchronous Message handling

LifeStream uses event driven communication and asynchronous message handling to process requests and provide responses. When we get a request, a pipeline is created which associates with an ServerHandler to handle specific event types. The ServerHandler handles the events asynchronously. We implemented proactor pattern to allow the asynchronous communication.

Each channel consists of an inbound queue and an outbound queue. Whenever message received event occurs, the handler en-queues the request to the inbound queue. The request is added to the inbound queue of the server that get gets the request. When the routing is found, the request gets en-queued to the outbound queue of the server the request is routed to, where multiple workers can work on the same queue. Inbound worker takes the message from this queue and starts processing it. Now, when the processing of the request finishes, it is added to the outbound queue of the initial server. The outbound worker picks the message from the outbound queue and writes it to the channel.

## Geo Spatial Data Handling

The geo-spatial information related to the document (image) is stored in the PostGIS database using the special “geometry” data-type. The latitude and longitude values of the location are passed to the database service layer using a “Point” proto object, which is then inserted into the DB as geometry entry. This makes the possibility of the queries endless. If a user wants to search for images around a particular location we can do that, if the user wants to query in the radius of a point, use can do that. A PostGIS handle gives us the capability to perform such queries with a very simple script.

## Proto File

The proto file determines message flow. The file has to be generic enough that we avoid the overhead of defining multiple messages that are similar, but has to be specific enough to ensure there is no loophole in user login or image upload. The proto file that was used in the core Netty implementation, had placeholders for most of the client server communication that was required for this project. Having a clear understanding of the use cases, we were able to modify the proto file to handle them all. Finalizing the proto file was key before we could take care of the backend storage and clients implementation. There were changes to be done on this file frequently as the flow of events became clearer. Though we wanted to take an approach of contract first and then implement based on the contract, it turned out that we ended up changing it many times as the understanding of the use cases improved.

## JPA

The storage of retrieval of user information and validation of user during login required storing the user information in database. Since PostGIS was used for image storage retrieval, it made sense to store user information also in PostGRES database. Its quite possible that later on, changes in requirements would require us to use some other storage infrastructure. With the use of JPA, this implementation is made independent of the underlying storage mechanism. It also provides a better mapping between the server implementation and the data storage template. We decided not to go with a EJB based approach as it would require an extra application server, which we felt was an overhead for the current system. Hibernate provided enough support. We also have a POJO based implementation of the Namespace operations.

## Clients

To indicate the versatility of the solution and to encourage future development of web or mobile-based applications, we are providing non-Java clients, primarily Python and C++ clients. Use of GPB’s simplifies this process as it makes the data transport agnostic of the implementation language. A proto file can be complied using a GPB compiler to generate Java, C++ or Python client and server serialize/ deserialize API’s.

# Limitations of Solution

The solution is quite robust and scalable, but does come with some limitation due to some of the technology choices.

## GPB

GPB is quite light and easy to use. The exchange of data in binary format makes sure that out payload is the smallest possible, but this inhibits the readability of the payload. When we have to implement a Web browser client, we might have to introduce another layer in between, as JavaScript won’t be able to deal with GPB payload.

GPB also enforces strict rules on the data. Later changes on the message structure makes dealing with backward compatibility a bit tricky.

# Future Enhancements

The system being the first implementation is a proof of concept and not a full-blown solution. There are enhancements that can be done to system that would make it a better and complete implementation

## Failure Handling

The system right now has minimal failure handling. The use of circuit breaker makes sure that nodes don’t get bombarded with request when they are down. We can make system more robust by adding edge condition handling and testing out scenarios where nodes keep rebooting. We can also make user session management more robust by handling user sessions using memcached.

Database failures are not handled by the current system and that is also an area to be improved upon.

## Client

Current Implementation uses a simple java or python client for data insert and query. The system can be further enhanced to use a Web browser based client or even a mobile app.

## Data Replication

Even though we are using multiple databases in our implantation, they are not in sync with each other. If a users request gets wrongly routed to a different database than the one he was associated with or if the database in which particular users information was stored goes down, the user will have a loss of service. Replication between databases will handle such situations more gracefully.

## Dynamic Configuration Update

Current system loads server configuration during start up and this is not updated unless the server is bounced. We could make the server handle dynamic updates to the configuration file and also use Zookeeper to keep the servers in sync during these updates.

# Collaboration

The complexity of the problem statement and the lack of knowledge on the different tools and technologies meant that, we need to coordinate with each other and help each other in improving the system understanding. We tried to meet every alternate week and to work on the project and everyone took up a certain topic or technology and read more about it and provided the rest of the team an overview about it. Since we all work full time, meeting face to face was not possible many times and we coordinated via Google hangout.

W used git repository to collaborate and share code. This made sure everyone was on sync and made the debugging sessions easier logistically. We also created an sql file with list of commands to create user roles, schemas and tables.

Conclusion

Implementation of this project turned out to be more tricky and challenging than anyone of us had anticipated. We were overwhelmed by the technologies that had to be used for this project and were under the impression that mastering the technologies would be the challenge for the project, but it turned out that gaining familiarity with the concepts of distributed systems, mastering the art of controlling multiple servers with simple changes in a configuration file, and creating overlay network that had the power of load balancing and message routing was more challenging than handling with the technology stack.

We all came into the project without having worked with a system that implements asynchronous communication, GPB, PostGres database, or Python language. We were also not familiar with the key design patterns that we have used for the implementation, namely the circuit breaker or proactor pattern. We come out of the project having understood and implemented the art of creating overlay network, using circuit breaker to handle failures better, using proactor pattern to smoothen the in flow of messages to a server node, store data using JPA, deal with geospatial data and code in Python.

# References

|  |  |
| --- | --- |
| [1] | Queue Concepts: http://www.rabbitmq.com/getstarted.html |
| [2] | Netty Server: https://netty.io |

[3] JPA: http://www.objectdb.com/java/jpa/persistence