CMPE 476 ASSIGNMENT 3 REPORT

In my previous assignment, I've reported my design of a centralized instant messaging app with its requirements, class diagrams, and sequence diagrams. Now, it's time to convert this centralized design into a decentralized instant messaging app.

In centralized design, all data was stored in one single server and every request had to arrive in this single central server. In a decentralized approach, more than one server is in the application structure. Therefore, total data in the application has to be separated into the set of servers in a traceable way. I propose a solution as follows: All channels in the workspaces should be initiated in all servers in the server set. Then, for each message, a timestamp and the message content should be stored in the closest server to be backtraced if required. This allows all messages to be stored independently, yet history of all messages can be retrieved by multicasting an inquiry to all servers in the server set. Hence, all messages can be retrieved by any client.

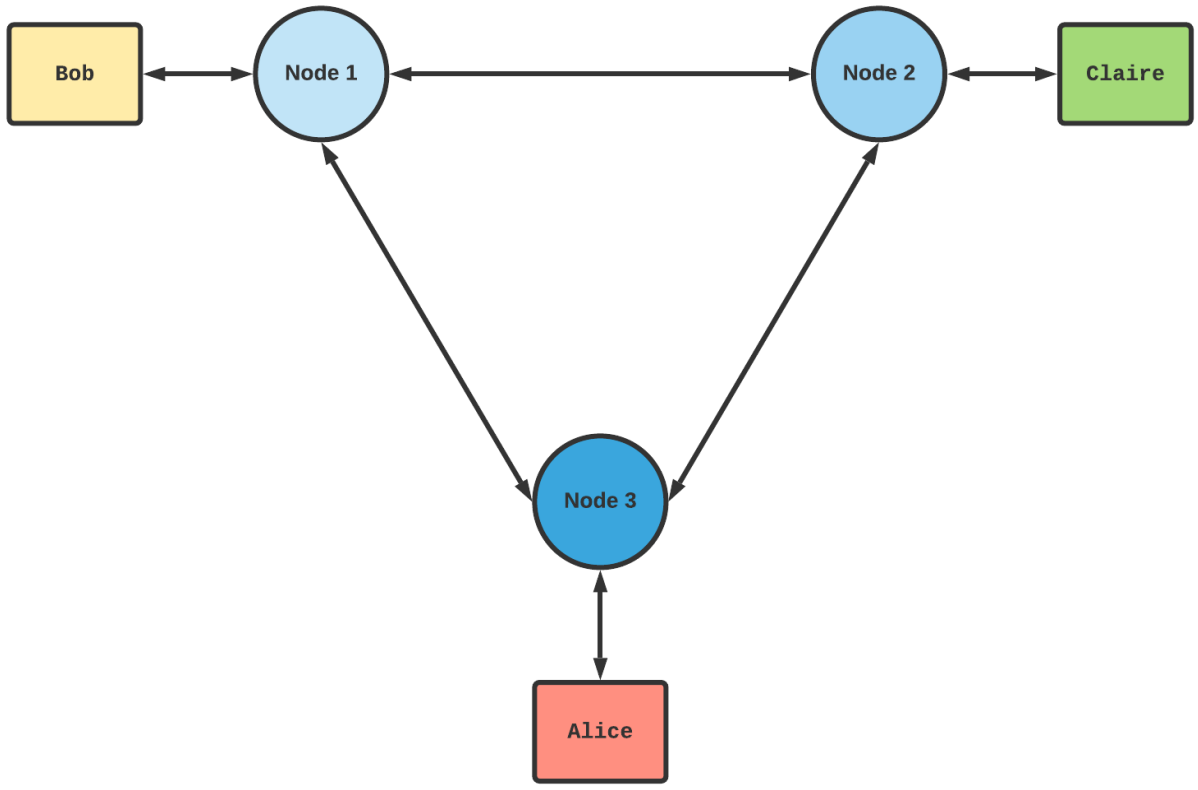

Figure 1: Network structure of the decentralized application with the clients

To elaborate further, I'd like to illustrate the structure with more schematics.

In the graph below, there are three server nodes and each node has three channels which are initiated at the beginning. Each node stores a number of messages in their databases and when a client wants to retrieve the message history of a channel, a multicast signal is produced from a node closest to the requesting client to collect all messages related with the target channel to recreate the message chain based on timestamps. This process is triggered when a client wants to send a message to a channel, which is a kind of a callback.

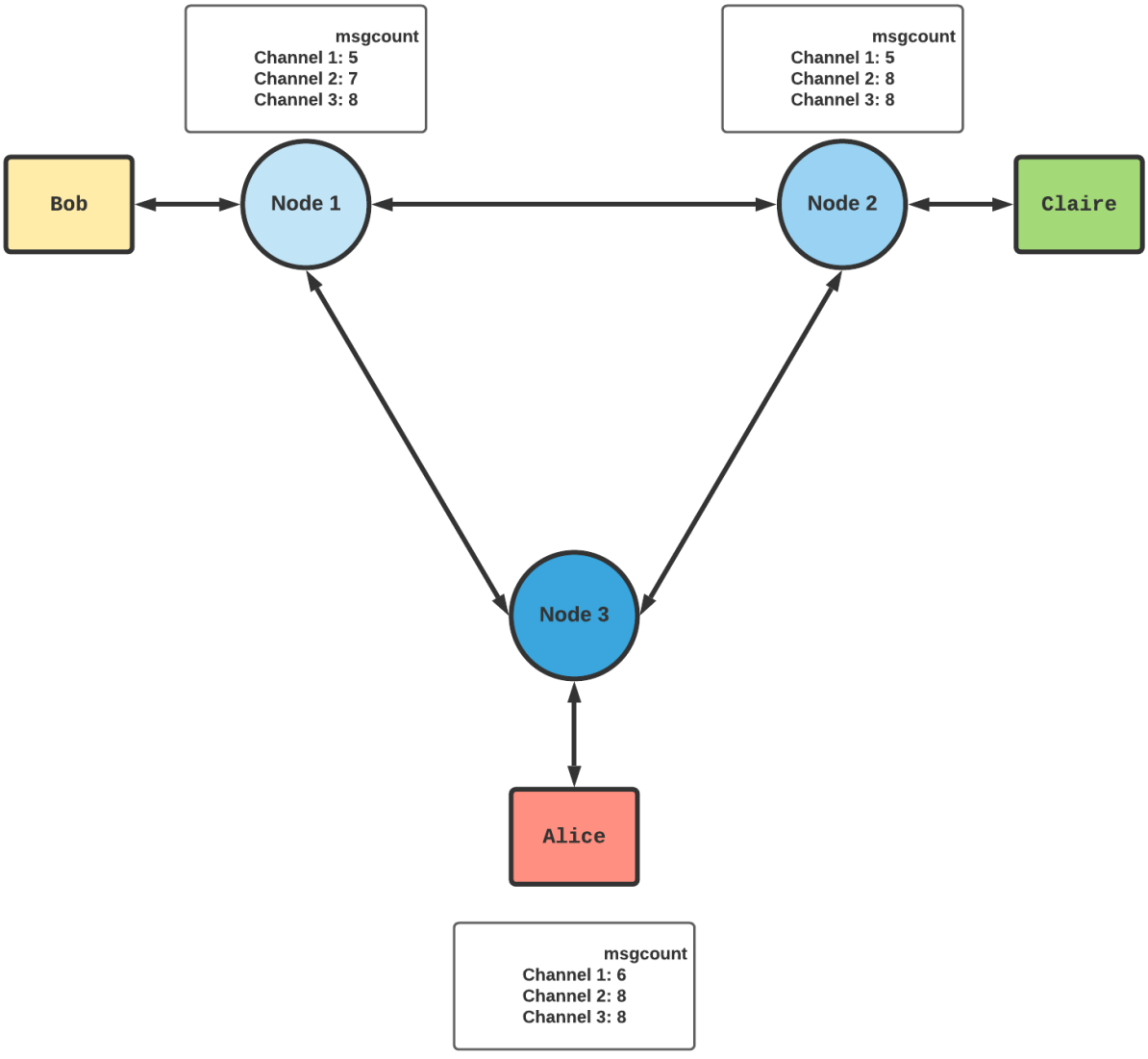

Figure 2: Messaging Channel Structure of the system

This graph shows that there are 16 messages in Channel 1, 23 messages in Channel 2, and 24 messages in Channel 3 when all messages in different nodes are merged. On the other hand, storing all messages by grouping the closest clients of a server node spreads the data storing into multiple points to decrease data storing workload.

Also, by having a vector clock mechanism in all server nodes to track which node is the source of the most recent channel message handles the distributed task scheduling. Any client can send a message in an arbitrary time, thus having a control mechanism to supervise server node states is a necessity.

In addition to those points, I'd like to point out one more: Location of the clients. Clients can be located in an arbitrary continent, therefore all clients have to communicate without naming resolution issues. My solution is to extract the latest closest server node of the recipient client by analyzing the channel messages and storing it in each server node as a key/value map. It would increase the data stored in server nodes, but storing a key/value pair for a client is a very little burden for a server even if the number of clients are huge. By merging this key/value map with the connection table, which is a key/value map for clients who are connected to a server node, the destination address of a message can be composed.

This structure should be achieved by a thin client structure since all messaging data should be stored in the server nodes, therefore client should support only a tiny set of information stored in its browser or system to configure the necessary parts of the application. A fat client would increase the propagation of the overall system because of storing messages in the clients instead of server nodes, but it also increases the synchronization interval of each client in the application drastically.

For storing tasks, I'd like to use the server nodes as replicas of each other since task management should be done more delicately than sending messages. Therefore, I'd like to use the vector clock mechanism for all server nodes to become exact replicas of each other in a very short time after an update occurs. In a case of desynchronization and inconsistency, vector clocks should tell the difference between the almost-concurrent actions to decide the operation order. In Figure 3, Node 1, Node 2 and Node 3 have differences over Taskboard 2 and this inconsistency should be resolved by checking the vector clocks of the server nodes to decide the order of actions. Furthermore, the new state of Taskboard 2 will be multicast among all servers.

Since task management includes server replication in itself, I'd like to state the replication for messaging. In my design, messages are stored in the closest server node of the sender client. To include replication, a set of servers can replace one server node, which are exact replicas of each other. Therefore, all message communication between server nodes can be replicated since all messages are replicated in the closest set of servers. Yet, a global disaster can destroy a set of servers in a region, therefore messages of several clients cannot be retrieved. To solve that, each set of servers should select a sister set of servers based on the distance between their geolocations. The farthest set of servers should become the sister set of servers for a set of servers in a region, which increases the survival probability.

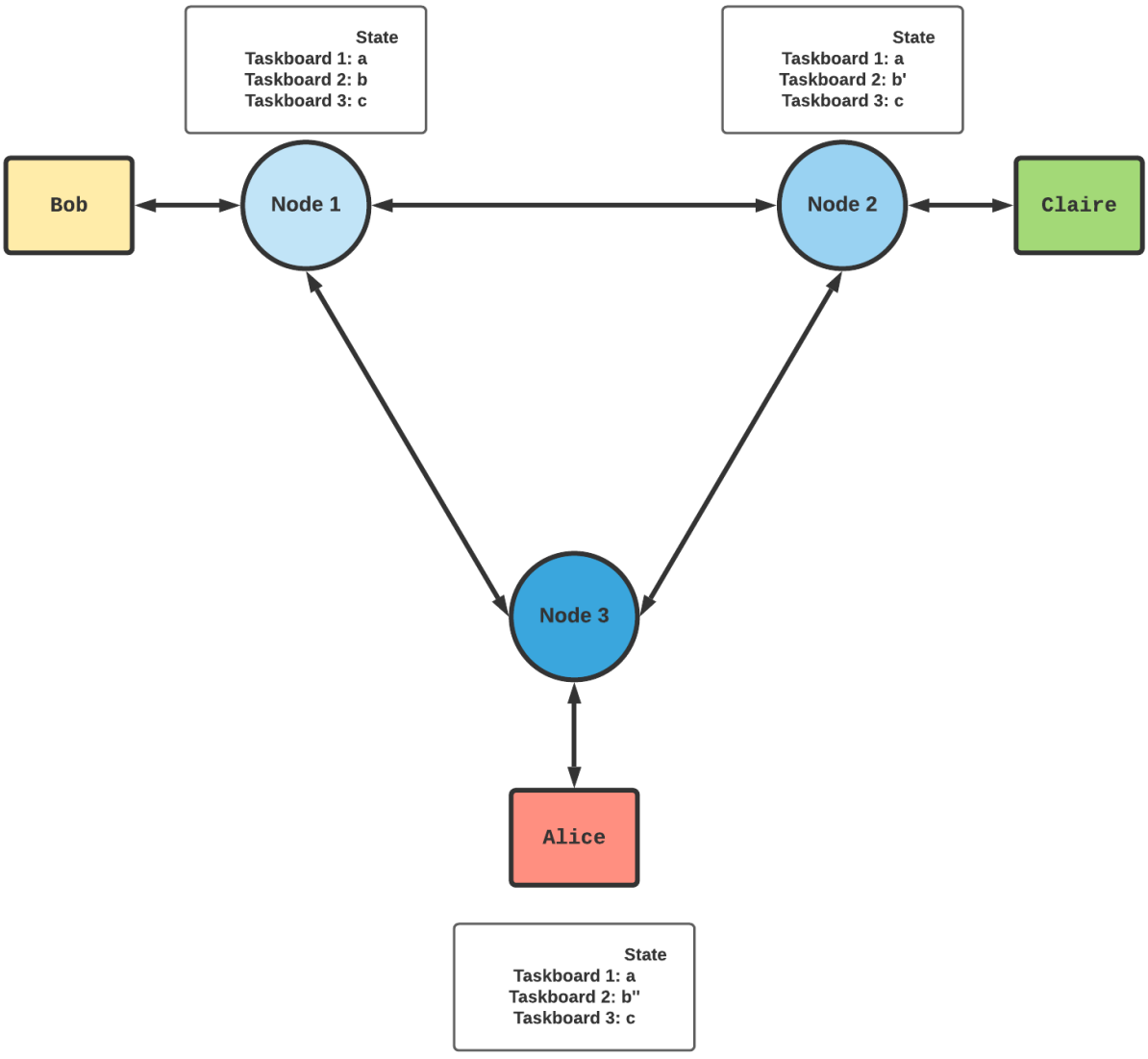

Figure 3: Taskboard structure of the application

As an design approach, MVC should not be used in a decentralized system since views are rendered after controllers server the data from the database and in an inconsistent state of the system, views of multiple clients would render a different set of output which mislead the clients and their further actions. Therefore, MVC will not be utilized in a decentralized system like my design. Other design approaches should be determined and implemented in such a system.