**Modeling and Simulation**

**Take Home Midterm Exam**

*I pledge my honor that I have abided by the Stevens Honor System.* - Joshua Schmidt4/13/20

Please write the honor pledge at the top of your exam and sign your name (it can be printed not handwritten).

This midterm will focus on practicing what we learned in class, applied for a simulation example. The questions will guide you step by step through the process of modeling, simulation, and interpretation of results. This will be the base for the final exam, which will be a Final Project Report.

**Step1 (Problem 1).** Look at the world around you and select an example system that could be modeled as a network of queues. If you can find an example from your previous projects/experiences as an electrical/computer engineering that would be great, but any real world example is good. Since we are picking an example of a network of queues we assume that the objective of the simulation is to minimize the delays experienced by the customers in the system.

Describe your system in words and draw the network of queues using classic queueing symbols. Explain your assumptions for the system.

**Example:** For the Pret a Manger fast food place, customers align to pick up food items that are in the display window, than they join a line to pay. The network of queues drawing will be as follows (assuming there is a line to pick up food – another model could be that each customer is its own server if the place is not crowded, in that case, first queue has an infinite number of servers):



For MailPear, a start-up that I am currently involved in, we needed to create a collaborative editing platform for dynamic emails. This feature is similar to a something like google forms, where different users can edit the same email at the same time, and can add different input objects – short answer areas, multiple choice questions, check boxes, image selectors, etc. – for the recipient of the email to fill out. With further features like form validation, project and email organization and management, and additionally analytics to show who opened the emails, when and for how long, we hope to differentiate our product and eliminate long, cumbersome email-chains.

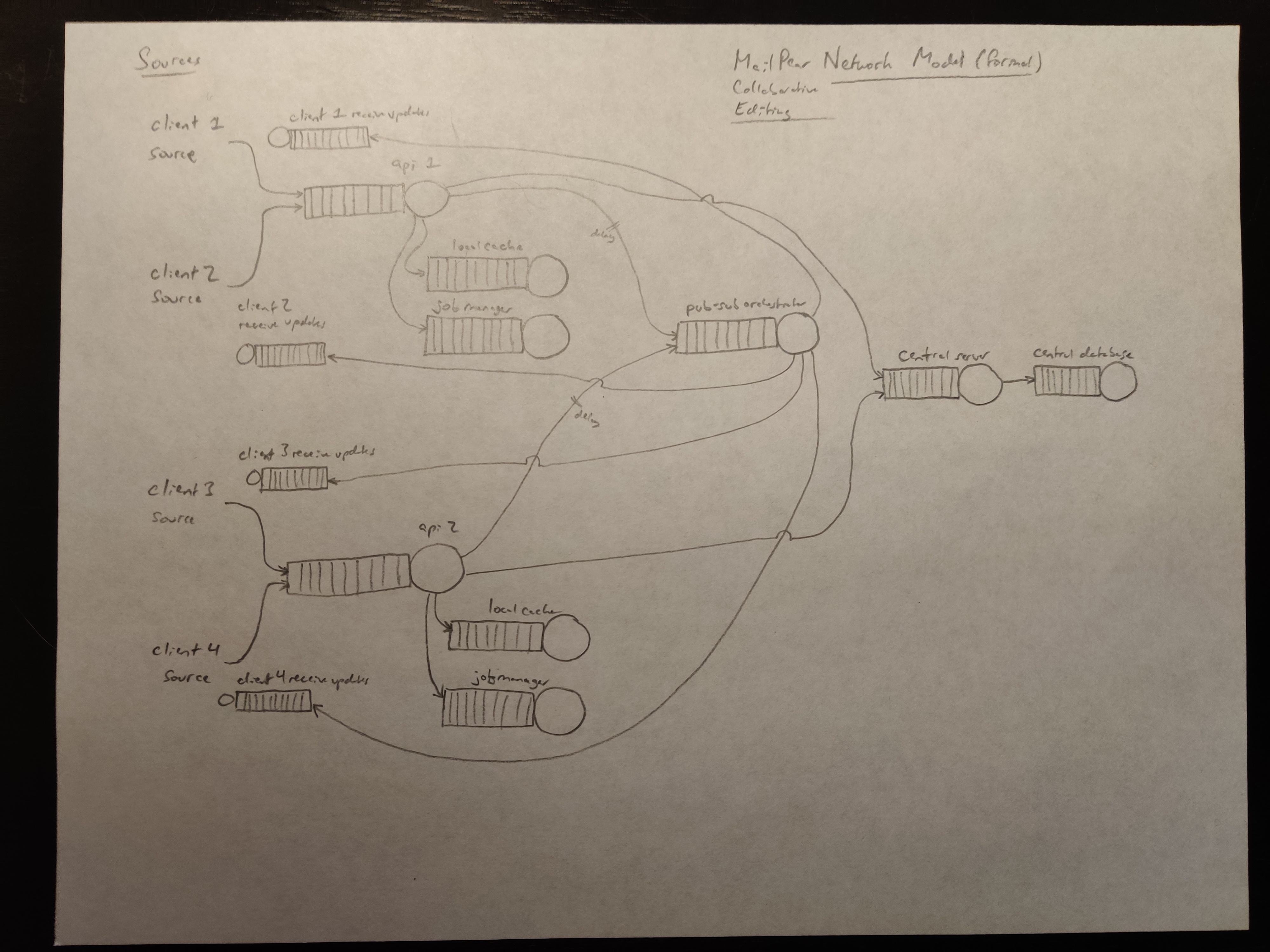
For this project, it is not feasible to analyze the entire product, so that is why I will be focusing on specifically the collaborative editing tool for creating the emails. The api is written in the graphql framework, which is similar to SQL but for HTTP requests – i.e. the client uses the graphql language to query for specific data. Graphql includes a subscription model using websockets, allowing for long-term bi-directional communication. After clients are subscribed to the form editing data, they can send json objects to edit different parts of the email, and see the updates to their local web application view. The web socket server will save editing data to a local caching server running Redis. Additionally, two timeout jobs are created – one to send the updates to all connected clients, and one to update the central server. The update job for the clients relays the updates that one user made to all users who are editing that particular email, over their respective websocket connections (subscriptions to email updates). The second timeout job updates the single source of truth – the central database – with the most up-to-date edits for the application. The timeouts for these two jobs are different. The clients are updated every few seconds after there is an edit, while the central server is updated every twenty seconds after. These timeouts ensure that our servers are not inundated with extraneous messages.

This model would be simple to implement if there was only one server, but it becomes more complicated with multiple servers scattered around the world. All servers have to be in communication with one another to determine what users are connected at any given time, and to what email they are subscribed to. To deal with this added complexity, a central publish-subscribe job queue is used, that will orchestrate and send tasks to each server. This model ensures that no user misses real-time updates to the collaborative email because they are connected to a different server.

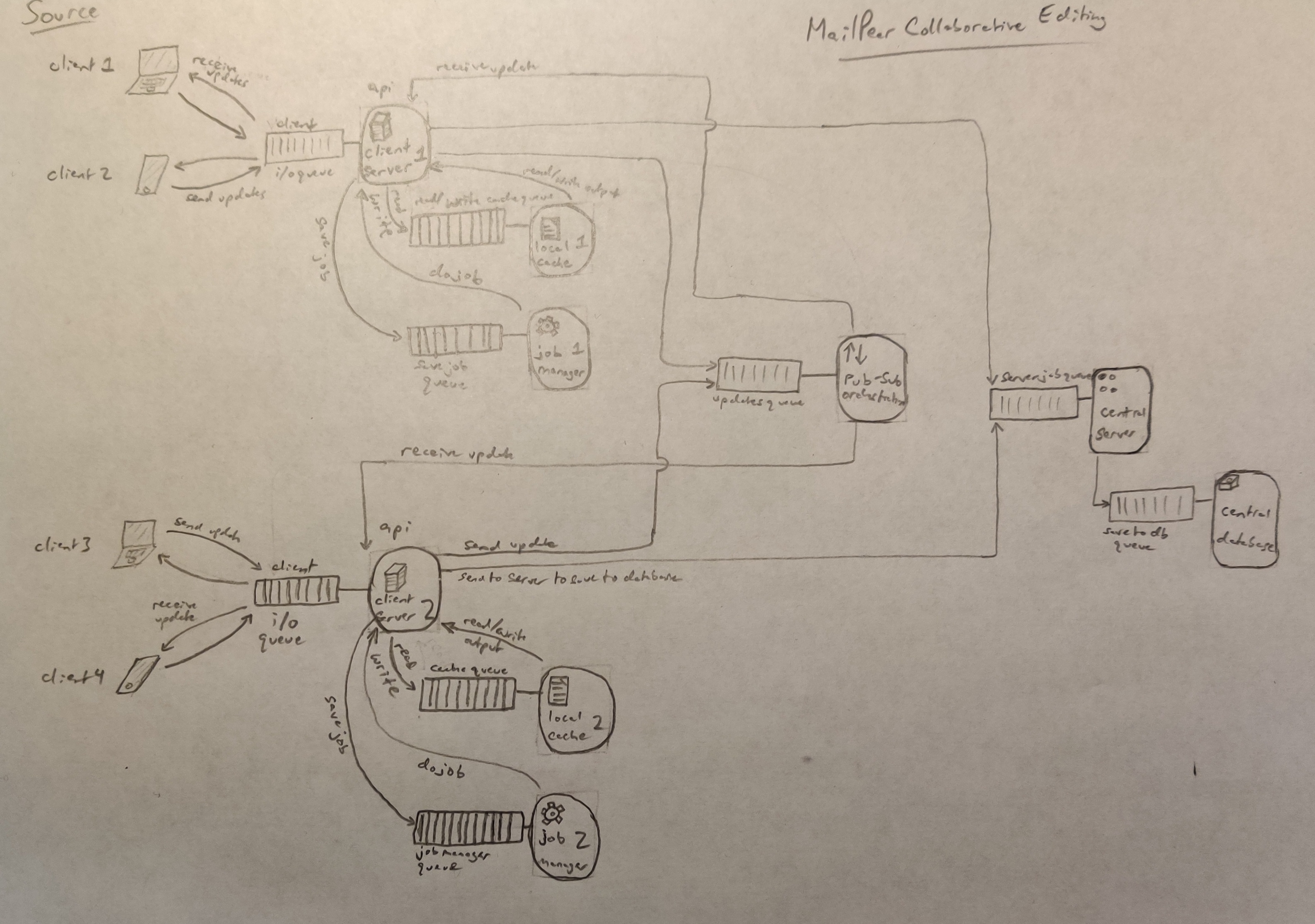
There are many queues and servers involved in this system. The source is obviously the users sending and requesting updates to an email. The first queue is the connection queue, to connect to the email for editing. When an api server is available, the client receives an acknowledgment message and is added to the queue for editing updates. When a user sends an email edit, they are added to an email editing queue, and when the api server is ready it will process and send the edit to another queue for local caching. Then when that the local caching server is ready it will save the update data. The updates timeouts are saved to queues with built-in delays. When these timeouts are hit, the updates are sent to the publish subscribe orchestration job queue, which when ready forwards the updates to the central server and to all client server queues. When the client servers are available, the updates are forwarded to the subscribed clients, to update their web pages. When the central server is available, the central database will receive the given changes and perform updates. The diagrams below illustrate this network.

Some assumptions were made to make this system easier to model. There are only 2 client servers handling 4 clients in this model (2 clients per server), but in reality there can be many more clients and servers. The inter-arrival time for jobs is 0.5 seconds, because it is assumed that client web applications would cache some changes before sending the api request to update other users. The timeout for sending updates to the central is assumed to be 20 seconds, but this value varies with implementation constraints. It is also assumed that there is little to no latency in the networking infrastructure itself, which is seldom the case. Other assumptions for parameters can be found in the omnetpp.ini file screen-shot below.

formal diagram



informal diagram



**Step 2 (Problem 2).** Explain why the simulation for this system will be dynamic discrete-event driven simulation with stochastic components.

* Identify the system entities
* Identify relevant attributes
* Identify events
* Identify state variable
* Why is it a dynamic simulation?
* Why is it a discrete event driven simulation?
* Which are the stochastic variables?

The system entities are all of the servers, queues, sources, and sinks. In this example, the servers are the 2 client api servers, 2 cache servers (1 for each client api server), 2 job managers (1 for each client api server), the pub / sub orchestration server, central server, and database server. The queues include the 2 input queues for the client api servers, 2 input queues for each of the local cache servers, 2 input queues for the job managers, 1 input queue for the pub / sub orchestrator, 1 input queue for the central server, and 1 input queue for the central database input. The sources include the update requests from each of the 4 clients, but more clients can be added. The sinks include the clients receiving the changes and displaying them on the web-app, the cache being updated, the job manager being updated, and the database being updated.

The relevant attributes include the configured service time for the 2 client api servers and the service time for updating and reading from the 2 local caches. It also includes the delay or timeout for sending the updates to peer clients, and to the central server and database (the source of truth). Relevant attributes also include the maximum size of all of the queues in the system, since there are limits as to how many jobs can be stored before they start to overload the server.

There are many events that occur in this system. The network starts with the initial submit update event, where the client sends an update to the api client server. Then client server triggers an update cache event, update job manager event, send update to pub-sub orchestrator event, and send update to central server event. The central server sends an update central database event, and the pub-sub orchestrator sends a peer client update event to all of the client’s peers. All of these update send events have a corresponding acknowledgment event for receiving the input data and parsing it correctly.

Some state variables include if there was an error with the api client server saving updates to the local cache, or sending the updates to the publish-subscribe server. The number of packets lost on each connection in the network are important state variables. If the current update was successfully sent end-to-end, from client to peer clients and to central update server, is another state variable.

This is a dynamic simulation because the state variables are changing over time. The state is changing in a partially random manner, because as the load on servers increase, parallel processing becomes less efficient and non-deterministic, so the service time increases. The client api server specifically is doing the majority of the computation, so its service times increase dramatically as load increases.

This is a discrete event-driven simulation because each event results in a specific set of subsequent events that do not vary. As more submit update events occur, this results in a 1-1 increase in the number of subsequent events that occur in the system.

The stochastic variables include the amount of time each job spends in queues, on average, and the time it takes for a submit update event to result in a peer client update event. The time it takes on average for an initial submit event to result in an update to the central database is another stochastic variable. The average amount of time each server takes to complete a task, including for the client api server, central database server, and pub-sub server, are additional stochastic variables. The number of packets on average in the api client server queue is another stochastic variable.

Based on all of these factors, this simulation is discrete and event-driven, with stochastic elements (the stochastic variables).

**Step 3 (Problem 3).**

Since we have not yet discussed modeling input variables, assume for now that all your arrivals in the system are Poisson with mean and all the service times for all your servers in the system are exponential random variables with rate

1. What types of queues do you have in your network?
2. Compute the end-to-end delay in your network. Choose appropriate values for arrival rate, service rates, and if applicable, for the routing probabilities between various queues.

Originally, I assumed that the first queue in the network, for accepting submit update events, was M/M/N, with N being the number of api client servers in the network, because there would only be one queue for all of the clients. However, because these clients are geographically split, and there is often only a group of servers deployed in one region with a single outward-facing ip address, it is more correct to say that there is only one queue per api client server. Therefore all of the queues for the api client server inputs are M/M/1. Similarly, the rest of the network has queues with Poisson arrivals and only one server each, so these queues are also all M/M/1. Therefore the only queue type in the network is M/M/1.

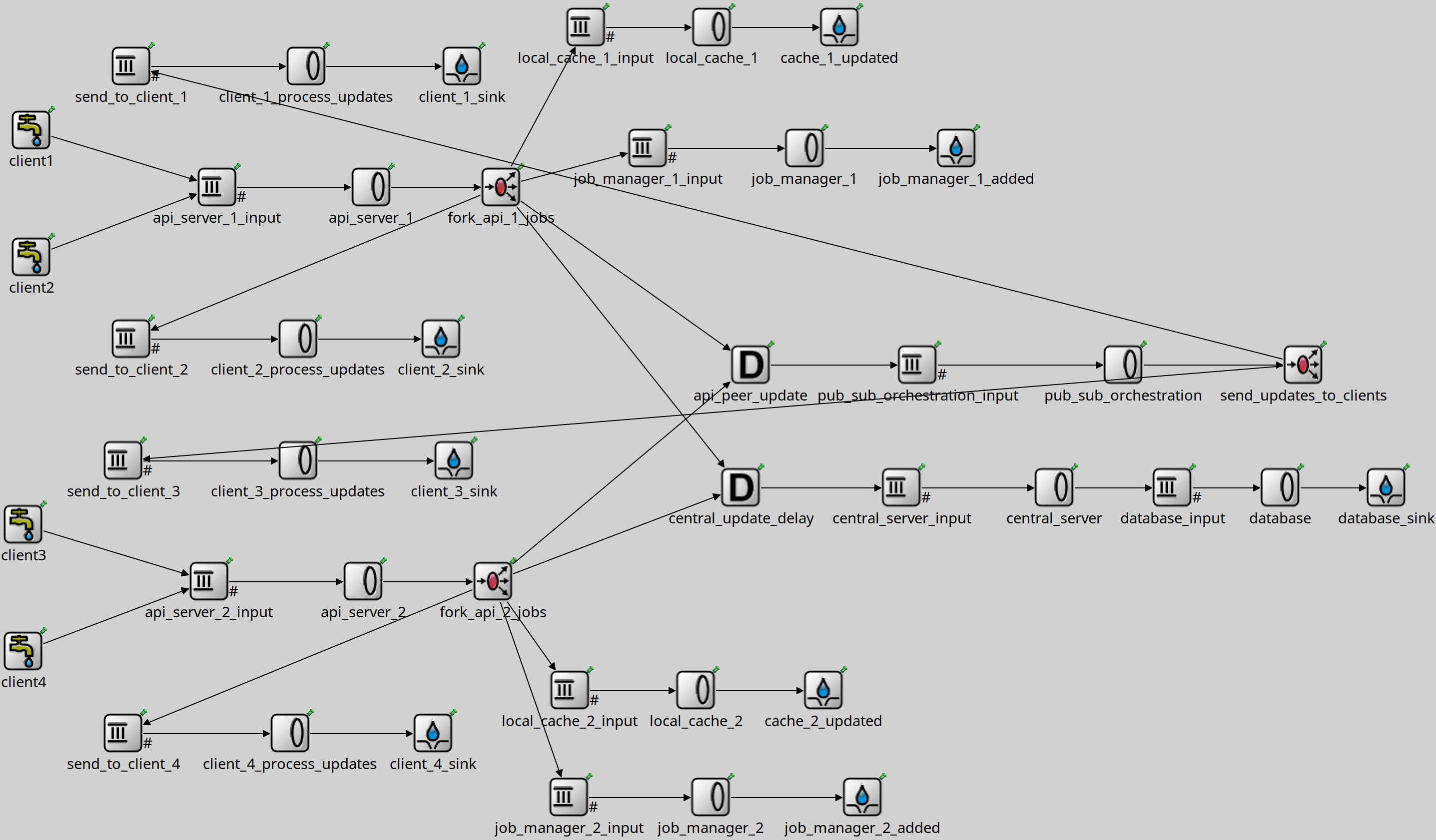
end-to-end delay computation:

for m/m/1:

**Step 4 (Problem 4).**

Simulate the system above using OMNET++ and collect the end-to-end delay (time in system) for 10000 jobs. Compare the results from your simulation with the results from your computation at Step 3.

omnet++ network



omnetpp.ini configuration file

[General]

network = Midterm

*# first client server*

\*\*.api\_server\_1.serviceTime = exponential(.01s)

\*\*.local\_cache\_1.serviceTime = exponential(.01s)

\*\*.job\_manager\_1.serviceTime = exponential(.01s)

*# process updates*

\*\*.client\_1\_process\_updates.serviceTime = exponential(.001s)

\*\*.client\_2\_process\_updates.serviceTime = exponential(.001s)

*# second client server*

*# client server 2*

\*\*.api\_server\_2.serviceTime = exponential(.01s)

\*\*.local\_cache\_2.serviceTime = exponential(.01s)

\*\*.job\_manager\_2.serviceTime = exponential(.01s)

*# process updates*

\*\*.client\_3\_process\_updates.serviceTime = exponential(.001s)

\*\*.client\_4\_process\_updates.serviceTime = exponential(.001s)

*# global*

\*\*.pub\_sub\_orchestration.serviceTime = exponential(.1s)

\*\*.central\_server.serviceTime = exponential(.1s)

\*\*.database.serviceTime = exponential(.01s)

*# delays for updating global*

\*\*.central\_update\_delay.delay = 20s

\*\*.api\_peer\_update.delay = 2s

*# job input*

*# 10000 is the default*

\*\*.client1.numJobs = 10000

\*\*.client2.numJobs = 10000

\*\*.client3.numJobs = 10000

\*\*.client4.numJobs = 10000

*# send every half second a new update (on average)*

*# can be faster if typing*

\*\*.client1.interArrivalTime = 0.5s

\*\*.client2.interArrivalTime = 0.5s

\*\*.client3.interArrivalTime = 0.5s

\*\*.client4.interArrivalTime = 0.5s

*# capacity before critical components get overloaded*

\*\*.central\_server\_input.capacity = 2500

\*\*.database\_input.capacity = 50

\*\*.api\_server\_1\_input.capacity = 2500

\*\*.api\_server\_2\_input.capacity = 2500

\*\*.local\_cache\_1\_input.capacity = 1000

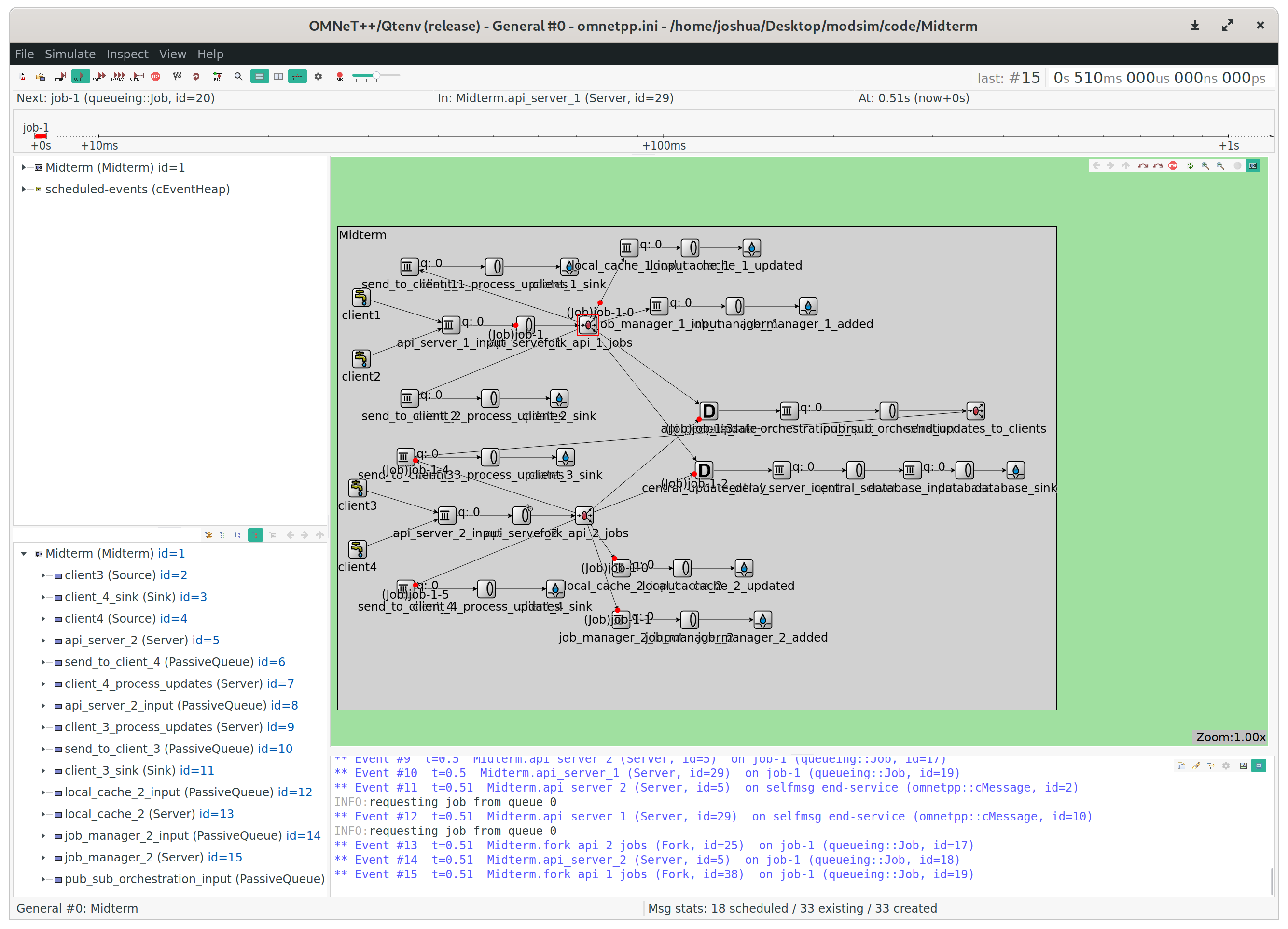
\*\*.local\_cache\_2\_input.capacity = 1000

\*\*.job\_manager\_1\_input.capacity = 1000

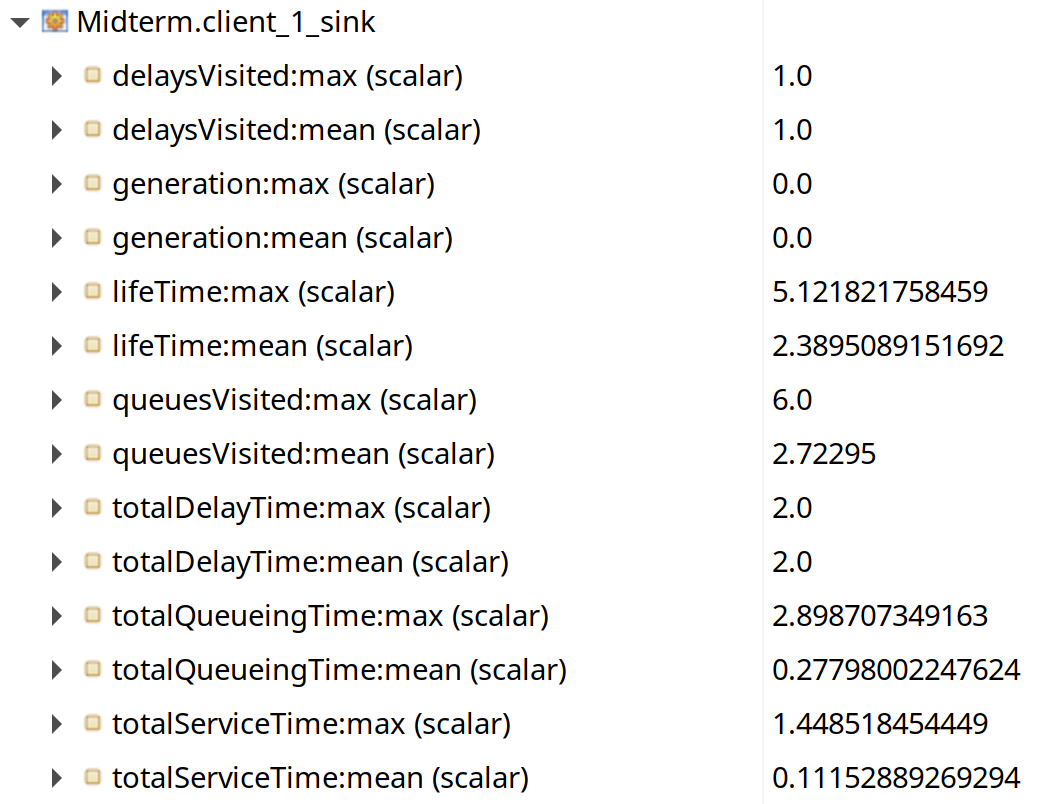
\*\*.job\_manager\_2\_input.capacity = 1000

\*\*.pub\_sub\_orchestration\_input.capacity = 1000

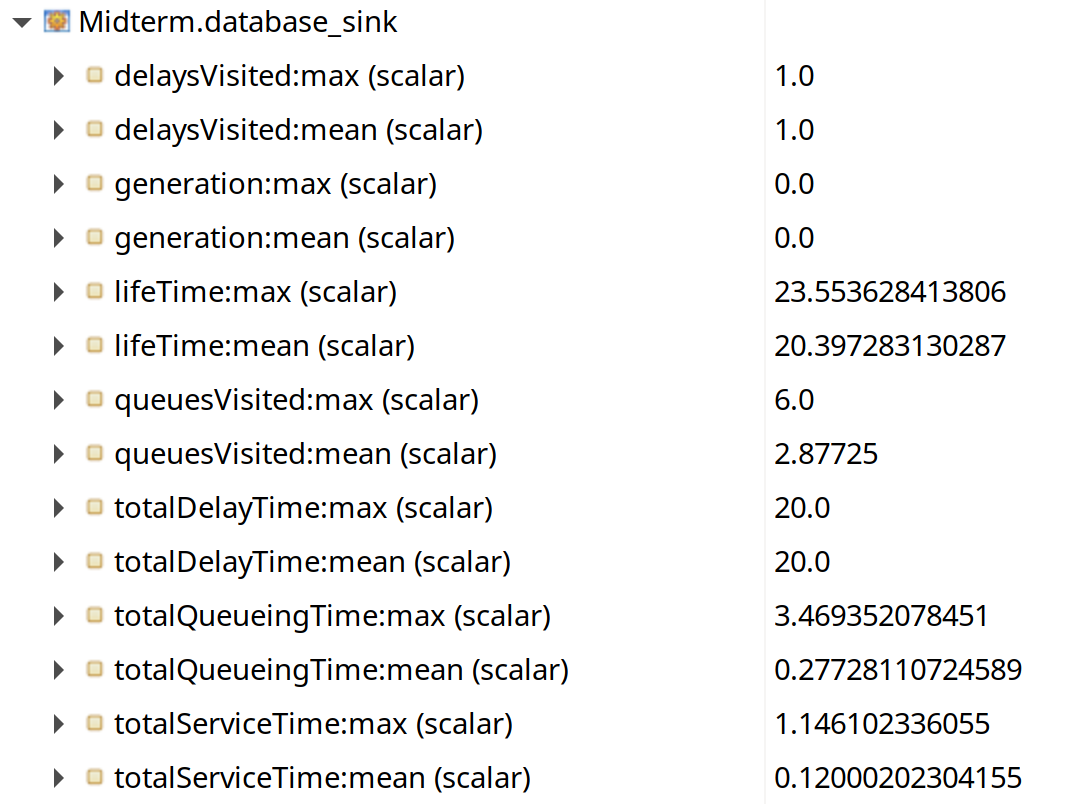
program running



client 1 updates sink



database sink



The time spent in the system on average for client 1, going from the initial update event to the peer update event, was 2.389 seconds. This makes sense because the delay time for peer updates is 2 seconds, and the service time for the pub-sub orchestration server is 0.1 seconds. The service time for the api client server is 0.01 seconds. Theoretical calculations showed that the time spent in the system should be 2.1454 seconds, so the experimental results differ from the theoretical calculations by 0.2436 seconds.

The time spent in the system for update events going to the database is 20.397 seconds on average. This can be mainly attributed to the 20 second delay on sending updates to the central server from individual api client servers. There is a 0.1 seconds of delay for the central server service times, and additionally 0.01 seconds of service time for the client server and central database. The theoretical calculations show the end-to-end delay should be 20.145 seconds, so there is a discrepancy of 0.2516 seconds between theoretical and experimental results.

This discrepancy in both end-to-end delays may be a result of delay from the forks in Omnet++, that were used to split the update job into sub-jobs for updating the database and sending the peer update. There may also be other delays in the Omnet++ model that are not being considered.

**Step 5 (Problem 5).** List all types of event notices that will be recorded in the FEL.

**Hint:** An event notice is (type of event, time of event) and you will have as many event notices as type of events in the system). You need to define all types of events: e.g. arrival in queue 1 is event of type 1. The answer should be a list of the form (type 1, time), (type 2, time),…, with the event types specified separately.

|  |  |
| --- | --- |
| Time (seconds) | Type of event |
| 0 | 1, 2, 3, 4, 5, 6 |
| 0.01 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 5, 6 |
| 0.02 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 19, 20 |
| 0.03 | 17, 18 19, 20 |
| 0.5 | 1, 2, 3, 4, 5, 6 |
| 0.51 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 5, 6 |
| 0.52 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 19, 20 |
| 0.53 | 17, 18 19, 20 |
| ... | events above keep occurring every 0.5 seconds (inter-arrival time) |
| 2.0 | 21, 1, 2, 3, 4, 5, 6 |
| 2.01 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 5, 6 |
| 2.02 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 19, 20 |
| 2.03 | 17, 18 19, 20 |
| 2.1 | 22 |
| ... | events above keep occurring every 2 seconds (due to delay) |
| 20.0 | 23, 21, 1, 2, 3, 4, 5, 6 |
| 20.01 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 5, 6 |
| 20.02 | 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 19, 20 |
| 20.03 | 17, 18 19, 20 |
| 22.1 | 24, 22 |
| 22.11 | 25 |
| ... | Events above repeat every 20 seconds |

|  |  |
| --- | --- |
| Type of event | description |
| 1 | Client 1 adds update to api 1 queue |
| 2 | Client 2 adds update to api 1 queue |
| 3 | Client 3 adds update to api 2 queue |
| 4 | Client 4 adds update to api 2 queue |
| 5 | Update enters api 1 queue |
| 6 | Update enters api 2 queue |
| 7 | Api 1 done processing update |
| 8 | Api 2 done processing update |
| 9 | Api 1 add to update local cache queue |
| 10 | Api 2 add to update local cache queue |
| 11 | Api 1 add to update job queue |
| 12 | Api 2 add to update job queue |
| 13 | Api 1 send to pub sub queue (with delay 2s) |
| 14 | Api 2 send to pub sub queue (with delay 2s) |
| 15 | Api 1 send to central server queue (with delay 20s) |
| 16 | Api 2 send to central server queue (with delay 20s) |
| 17 | Api 1 done updating local cache |
| 18 | Api 2 done updating local cache |
| 19 | Api 1 done updating job manager |
| 20 | Api 2 done updating job manager |
| 21 | Pub-sub delay ends – send to pub sub orchestrator |
| 22 | Pub-sub done – send updates to all peers |
| 23 | Central server update delay ends – send to central |
| 24 | Central server – send to database queue |
| 25 | Database – saved update |