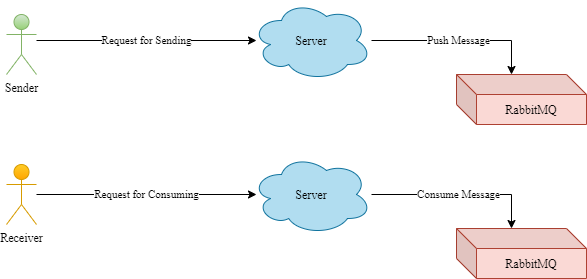
## RabbitMQ

To provide the common messaging behavior for our application, we considered using queuing system by exploiting RabbitMQ. Each user must have a queue for receiving the messages. Explicitly, each user is a consumer, and consumes messages from the queue corresponding to his/her username. In this way, we can also manage the synchronization issue related receive a message. The first message comes into the queue will be consumed and delivered to the user first, so we have FIFO queues. Since all the communication pass through the server in our application, we need to send the request for consuming to the server when a user gets online. Accordingly, when a user sends a message to another one, the message should be sent to the server and then the server passes it to the receiver queue. Finally, the queues must be persistent since we do not want to the not consumed messages when the connection between server and RabbitMQ goes down.

In the following example, user A (Sender) sends a message to user B (Receiver). User A does not care if the B is online or not. Request for Consuming will be sent to the server automatically after the B gets online.



### Architecture

We used direct exchange of RabbitMQ, since our messaging system only contains one to one communication, so we do not consider the possibility of creating channels or groups.

We run a process to handle each user request (sending or consuming). This process adopts gen\_server behavior to provide several functionalities to the application. After the initialization, the process would react to one of the following commands:

1. Reset: The process will be restarted and re-initialized.
2. Stop: The process will be killed, so no more request will be handled. We use this command when stopping the whole server application.
3. Push: The input message will be converted to dictionary of binary entries and encoded. Then it will be sent to the receiver queue. If the push was successful, *pushed* atom will be returned.
4. Delete User: The queue of the user will be deleted.
5. Request Consuming: When a user sends this request, we must dedicate a channel and consuming process to the user. The channel is required for communication between the consuming process and RabbitMQ, because it connects to the specified queue. The consuming process receives any message from the channel. Then the incoming message will be sent to the user (Receiver) process id.
6. Terminate Consuming: When the user logouts, we must close its channel. Otherwise, the opening channels may cause the overhead in our server.

In addition, after a certain amount of time (in our case it is thirty minutes), we refreshed the channels by deleting and creating another one. In practice, the heartbeat connection between RabbitMQ and the server will determine to close the connection after not receiving any data for a certain period. Therefore, we must also check if the connection is dead, recreate the connection and channel again.

### Rebar3

Since our server is implemented in erlang, we realized that we can use Rebar3 to manage our application in proper way. Rebar3 is a build tool and package management for creating and deploying erlang applications [1 – 2]. To use amqp (RabbitMQ) and jsx (JSON) libraries, we added the dependencies in Rebar3 config file, so all of them work under a unified project. To compile and run our server application we simply run the following command:

*rebar3 shell --name unisup\_server@localhost --setcookie unisup --script src/run\_listener.escript*

# References

[1] - <https://github.com/erlang/rebar3>

[2] - <https://rebar3.org/docs/getting-started/>