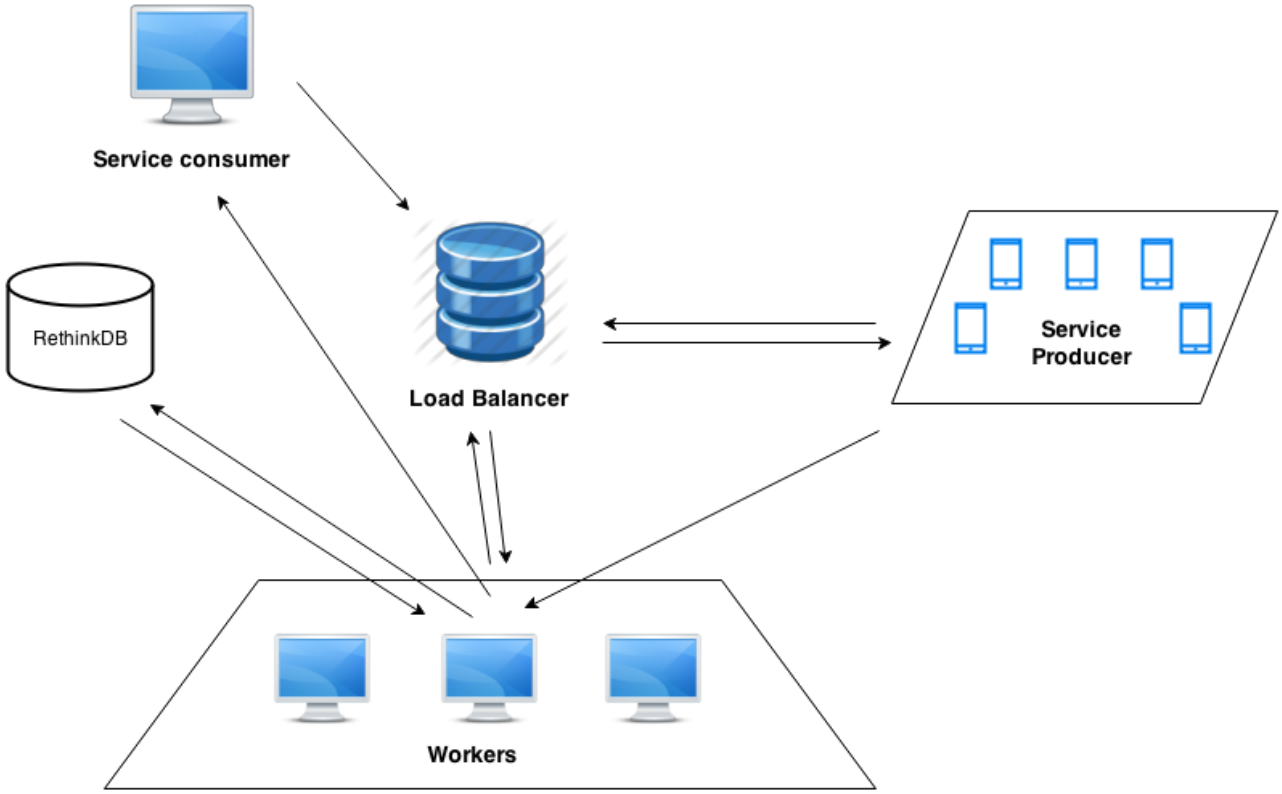
CS3301 Service-oriented middleware for mobile sensing

Architectural design

The architectural design consists of five main elements; service producers (i.e, mobile phones), a load balancer (broker), worker servers, a database to store needed data, and a service consumer (client), which requests mobile data. The design is illustrated in picture 01.



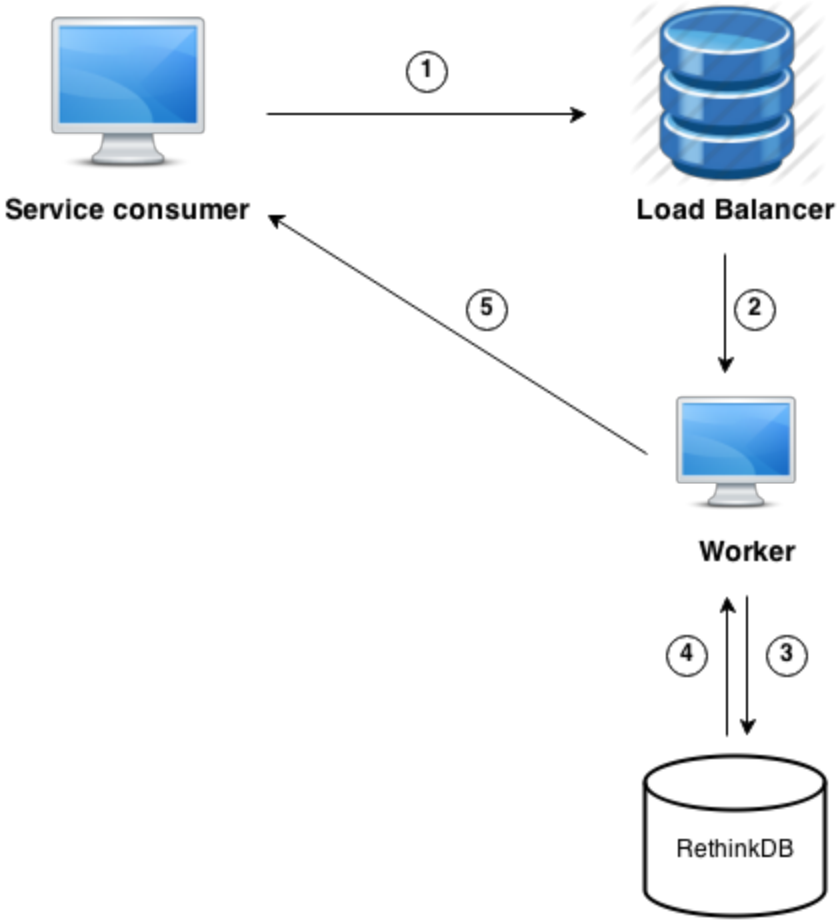
Picture 01: Architectural design of the system

Load Balancer

The load balancer acts as a broker trying to balance the load (incoming requests) across different workers, which can simply be different processes on the same machine. It is responsible for tracking the workers and finding the one with the minimum load for the new incoming requests. This is done by maintaining a description of how much of a load each worker has had to bear.

Service consumer

Whenever the client wants to retrieve mobile sensing data, it sends a GET request to the load balancer. The GET request includes a longitude, latitude, and some distance radius in kilometers. It is then the balancer responsibility to find an available worker, which will retrieve the data from the database and send it back to the client. The data that is returned from the request is a list of sensed data from mobile devices that are within that radius of the given latitude and longitude. The process is illustrated in the picture below.

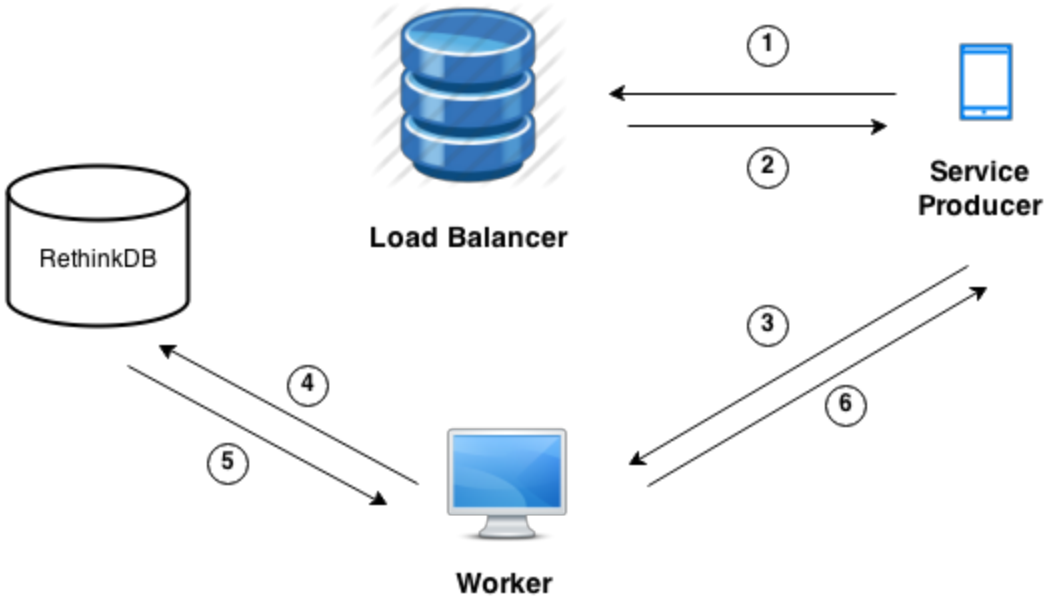


Picture 02: Sequence diagram for a service consumer requesting data

Service producer

The service producers act in a similar manner, as shown in picture 03. When a service producer needs to send data, they first send request to the load balancer and ask for a URL which corresponds to a worker. The load balancer sends back the url of a worker with minimum load. The producers ask for a new worker URL after every 5 requests in order to

make sure that the load is continuously balanced. Once the producer has the URL, it sends a POST requests with the sensor data and associated metadata to the worker. The worker then writes the data into the database.



Picture 03: Sequence diagram for a service producer sending data

Workers

Apart from handling requests from the service consumer and producers, the workers also send heartbeat messages to the load balancer. These messages are being sent at some time interval to update the load information about the worker and to advertise the worker's services. The message itself contains the address of the worker and the number of requests the worker has had to deal with. This structure is shown below.

Database

The database holds the mobile data collected from the service providers. We decided that we need the database so to have a persistent system that handles concurrency. We also figured that if there already exists a service that is thread safe, and can be run on multiple cores, that there would be no need to reinvent the wheel for data storage. The database is a crucial part of this system because it allows us to inherently deal with producer and consumer failure. Since all of the data is stored persistently, there is no need for a one to one connection between the workers and the producers and therefore producers and workers can fail without any major disruption to the system. Also, because we are using a

database, the clients are simply querying the database through the use of our load balancer and workers.

Implementation

Load balancer

The load balancer implementation can be found in sense/src/sense/broker . It has been implemented using the Go programming language. It provides a RESTful API used by the producers, consumers and workers, as well as methods for finding the worker with the lowest amount of load, and routing the client and mobile devices to a worker. The load balancer is implemented by tracking the frequency of requests to each of the workers and routing new requests to less utilized workers. The number of requests is sent with the heartbeat messages from the workers to the load balancer. This ensures that the true number of requests to each worker is known by the load balancer.

Service consumer

The consumer implementation is written in Python and can be found in sense/sensipy/client.py . The client sends a GET request with the appropriate URL parameters to the server to receive the mobile data. Since our system uses a database, the service consumer basically calls a GET request to the load balancer that forwards the request to a worker which queries the database.

Service producer

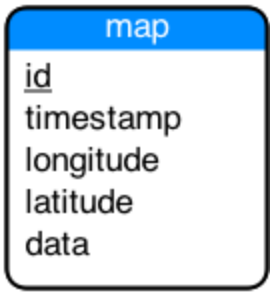
There are two different implementations of the service producer. The first one is done using JavaScript and html and can be found in sense/static. This version has a GUI (found in producer.html), which is used only for testing purposes, but it also allows the user to see the data being sent and to choose a sending rate. The data that is being sent is the IP address, which is used as an identifier (id), a timestamp, longitude, latitude and some arbitrary sensor data. The geographical data is retrieved using a REST API. The sensor data that is gathered is some device orientation data, but in case the device does not support this functionality, we are sending some random values, because using fake sensor data is not critical for our testing needs.

In addition to that solution there is a Python version implemented in sense/sensipy/producer.py . It sends the same type of data, but in this case we are faking

all of the information. This version has been additionally added in order to easily create a big number of producers and test the scalability of the system. Therefore, it does not allow any user interaction.

Database

The information the database has to hold is stored in just one table as shown in picture 04. The tables stores the identifier (id) of the producer (which is also the primary key of the entity), the timestamp, longitude and latitude describing the location of the producer, and some additional sensor data which is stored as a JSON string. There is no need to define the structure of the additional data because the mobile devices should be able to sensor arbitrarily structured sensor data. We decided to use RethinkDB as our database because it is simple to use, it has language bindings, and it is something we are familiar with.



Picture 04: Database design

Evaluation of the system

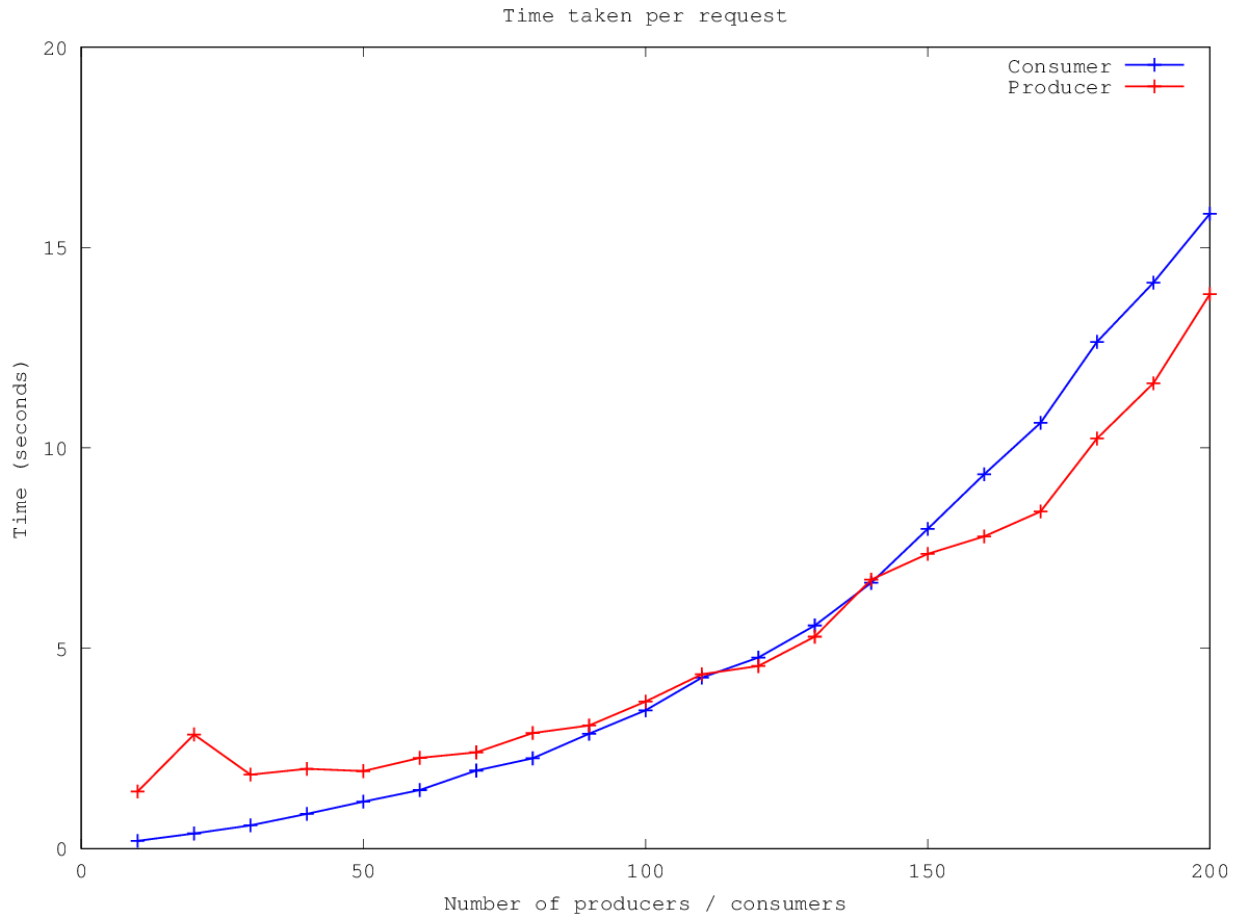
Method

The tests were run by creating a number of consumer and producer processes which automatically retrieve and send data from the server respectively. For each testing interval, the same number of consumer processes and producer processes were spawned. Inside the producer and consumer processes, the time taken for each request was recorded and written to a file. This data is then averaged and is used to represent the average time needed per producer / consumer per request. This metric represents how responsive the system is for consumers and for producers. The produced data can be found in sense/data . Plotting the metric against the number of consumers / producers used for testing will show how scalable the system is.

For the tests below, we were using maximum of 16 worker processes. When there are too few requests, not all of the workers get utilized. As the number of requests increases, the load balancer utilizes the workers and more requests are sent to each of the processes.

Scalability

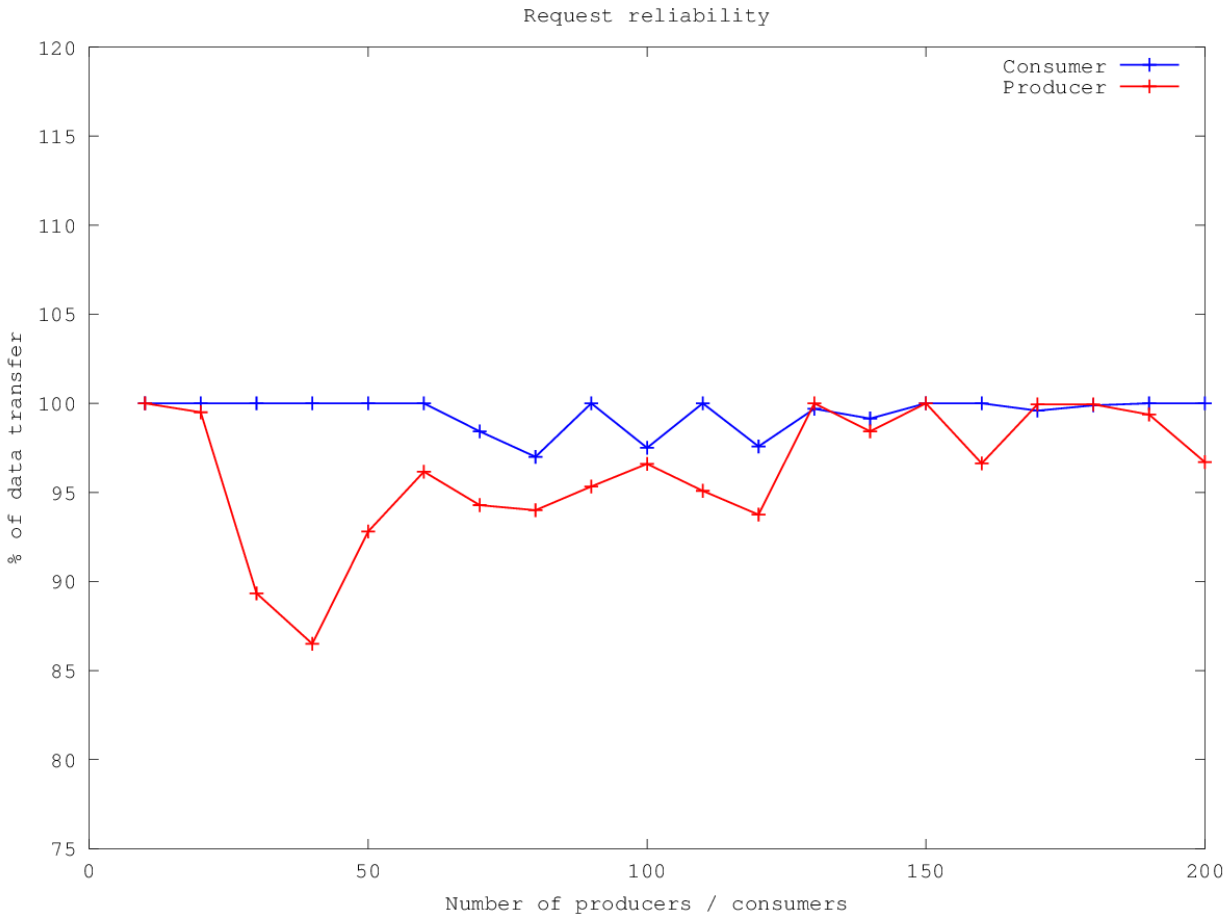
The diagram below shows the results from the tests. It can be also found in sense/figs . From the diagram we can see that the system scales gracefully as the number of producers and consumer increase. As the number of consumer and producer increases, there is a major absolute time increase needed per request. This is most likely due to the number of processes that were running on the same machine, using the same resources. If these consumer and producer processes were run from different machines, we are likely to see the same trend but the actual time used per request would be much less. Also, if the database was to be hosted on an external server rather than on the same machine as the load balancer, we would see a major performance increase. The database was what was using most of the CPU and memory.



Picture 05: Diagram for time taken per request

The trend does show favorable results. Using one machine for the consumers, workers, producers, and load balancer, we were able to achieve a quadratic trend with a small derivative. Between 120 and 200 producers and consumers, the trends seem to be turning linear. This is most likely because there are enough requests to properly balance the load. Once the number of producers and consumers reach a threshold, the work can be delegated more appropriately. This is because the workers assigned to the producers and consumers is continuously being switched when there are not enough requests to satisfy the workers. When there are too little requests, the workers are not being utilized and the balancing does not really give any advantage.

Since the lines above are not exponential, our system is scalable in terms of time per request, and as mentioned before, if the system was to be run in a distributed manner, we would see a more linear trend because there would be less shared physical resources between the consumers, producers, load balancer, and workers.



Picture 06: Percentage of requests satisfied on the first try

The figure above shows the percentage of requests that are satisfied by the server on the first try. As we can see, the system is reliable. For consumers, it is almost 100% reliable all of the time on the first try. For producers, it is a less than 100% on the first try, but this is because there is data being sent in the post form from the producer. Also, the producer has to make more requests because it has to check which worker to use and does not get redirected automatically. Since the lines above are linear and flat around 100%, our system scales in reliability.

Robustness

The system handles crashing service consumers, because they are just sending GET requests and therefore their crashing does not affect the system at all. The consumers do not need to check in with the server and therefore if a consumer stops requesting data from the server, nothing on the server is changed.

The system is also able to withstand failing producers. Producers check in with the server by sending heartbeat messages with their server host name and all of their sensor data. If a producer stops sending data to the server, it will not affect any other component of the system because the last data sent from the producer is stored in the database. When a consumer receives that data, it is their responsibility to check the timestamp to see whether the data is too old to use.

Moreover, the system can withstand failing workers. Workers interact with the server by sending heartbeat messages advertising their services. The server then routes consumers and producers to the appropriate workers based on the load. If a worker dies, the consumers can no longer be routed through that worker and the consumer is rerouted by the load balancer. Since the consumer's only point of entry to the system is the load balancer, the load balancer will know when a worker dies because no heartbeat messages are being sent and the load balancer will no longer route consumers to the dead worker. Producers interact with the server a bit differently. A producer first requests a URL to send data to, and then deals solely with the server it has been assigned. After a certain number of requests, the producer requests a new URL from the load balancer to make sure that the load continues to be balanced. When a worker dies, it can no longer reach the desired worker URL and an error is raised. Upon catching this error, a new URL is requested (which will not be of the dead worker due to the heartbeat advertisement to the server) and the producer can continue sending data to the system.

Also, please note that even though we do not have a graph here, we ran the tests again and whilst they were running, we randomly killed worker processes. This did not crash the system. The server was still able to delegate work to the remaining workers.

Appendix

To install:
- Install Go
- Install RethinkDB
- Install RethinkDB bindings for go (https://github.com/christopherhesse/rethinkgo)

To run the server:

```
export GOPATH=$(pwd)
rethinkdb &
bash run.sh [number of workers]
```

To run extra workers:

```
export GOPATH=$(pwd)
bash run_workers.sh [number of workers]
```

To run producers:

```
bash run_producers.sh [number of producers]
```

To run consumers:

```
bash run_clients.sh [number of clients]
```