Component Technology - A Middleware for Sensor Data

110011264
School of Computer Science
University of St Andrews

110002255
School of Computer Science
University of St Andrews

ABSTRACT

The report describes the design, implementation, and evaluation of a message oriented middleware that handles the transmission of mobile sensor data. The concepts of producers and consumers are used. The design of a server load balancer is also discussed.

REPORT

Planning

Initially, the team implemented a simple server. The server receives heartbeats, in the form of POST request, from the producers. Each heartbeat consisted of an unique identifier of the mobile phone device and its current location. The sender advertises itself as a producer to the server. When a consumer asks the server for sensor data from that particular producer, the server will send a request to the producer, and then pass on the results back to the producer. However, this approach requires the producers listening on some ports to handle data requests from the server. As a result, the consumer will spend significant time waiting for the server to retrieve data from the producer. The solution will create too much overhead on the server side, and it is unlikely that the server can deal with increasing consumer requests.

Later on, the team arrived to a more elegant solution. On each heartbeat, the producer sends sensor data to the server. The server caches the data, and flushes them to a persistent database on occasion. Now, when a consumer requests sensor data from a particular producer, the server can respond quickly, since the sensor data are cached. A more detailed description of the overall setup is explained in the next section.

Instructions

See "README.md" in the project directory for directions on how to run the servers, producers, and consumers. The file describes the arguments needed to run each system component.

Design

System components

The final system architecture at the time of our submission consists of:

- 1 Load balancer
- N Workers
- X Producers
- Y Consumers

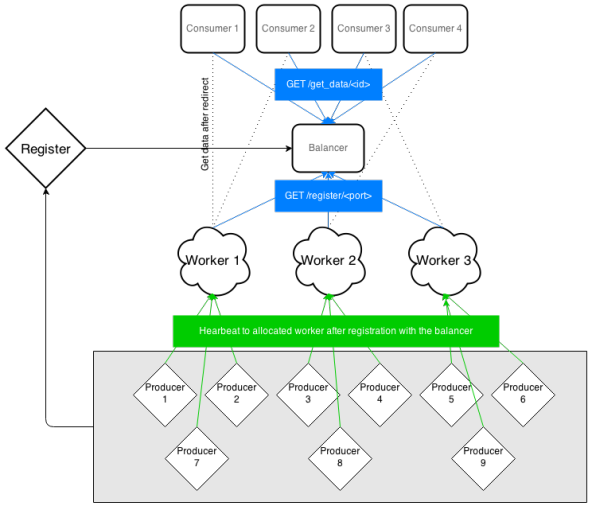

Figure 1. The final system architecture

A load balancer is the primary component of the system. It manages every other system component and balances the server load. A worker is a separate unit that is responsible for data retrieval and stroage. A producer generates sensor data, and a consumer requests those data. An example of a producer would be a mobile phone, and an example of a consumer would be a web service that uses location data to recommend music to its users.

Load balancer and workers

The load balancer is the entry point of the entire system. It is initialised to be listening on a specified port. It supports a number of RESTful API calls that allows other entities to interact with its system componenets, such as a worker. The worker is initialised with the IP address and the port number of the load balancer, so that it can register itself as a worker for the load balancer via with the */balancer/int:port* API call. The load balancer processes the registration request, and stores the worker's IP address in memory. The load balancer keeps track of every worker, and will distribute the sever load evenly, using the round robin method. Every worker must go through the above registration process in order to let itself known to the load balancer, whom will redirect producers and consumers to the worker. The worker also listens on a specified port, for handling heartbeats (described

below) and sensor data requests. It is worth noting that the worker will exit if it cannot connect to the load balancer.

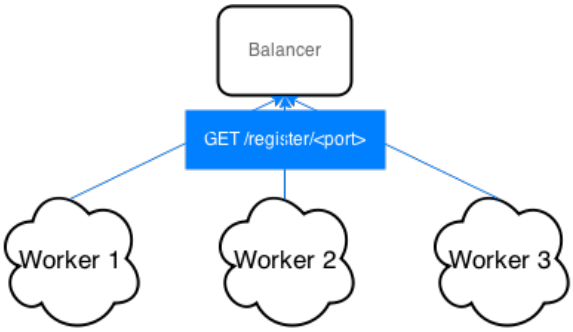

Figure 2. Workers need to register with the balancer before they can start accepting work.

Load balancer and producers

Producers ask the load balancer to be assigned with a worker. Afterwards, the producers only communicate with their assigned workers, and only when the connection disconnects do they speak to the load balancer again. To register, the producers fire the */register/int:id* API call to the load balancer, with its ID. The producer ID is the unique identifier of the mobile device, also known as the International Mobile Station Equipment Identity, or IMEI. After registration, the load balancer returns the IP address of an available worker to the producer. The load balancer keeps track of the mapping between producers and workers. When a consumer requests data from a certain producer, the load balancer will redirect the request to the worker that has been in contact with that producer. HTTP errors such as disconnected connection and interruption are caught and handled.

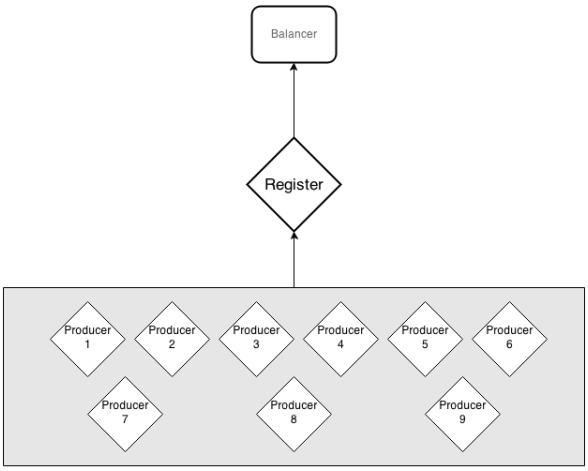

Figure 3. A producer first registers with the balancer and is assigned a worker

Workers and producers: heartbeats

After a producer receives its assigned worker and the corresponding IP address, it will continuously send heartbeats (periodic messages) to that worker. The message contains the following information:

```
received_heartbeat = {
  'id': id
  'location': location
  'data': data
}

new_heartbeat = {
  'id': int(request.json['id']),
  'location': request.json['location'],
  'ip_address': request.remote_addr,
  'timestamp': datetime.utcnow(),
  'data': request.json['data']
}
```

Code Fragment 1. Heartbeat format

On each heartbeat POST request, the worker extracts the "id", "location", and "data" from "received_heartbeat". In addition, the worker takes note of the remote IP address of the request, as well as the current time. The worker puts the information altogether into a new "heartbeat" object called "new_heartbeat" (a dictionary in $Python$). The team has designed a data structure called cache, defined in $worker/cache.py$. In essence, the cache stores a mapping (another dictionary) between the producer ID and the its latest "heartbeat". The worker will flush the "heartbeats" to the database every 20 seconds. The time interval is chosen for demonstration purposes, and more tests are needed to find out the optimal time. The worker will also flush the cache to the database when it reaches the size limit. The size is kept to 100, also for demonstration purposes. When the cache is full and the contained data are added to the database, half of the cache is cleared to allow more data to be added.

If the worker goes offline or crashes, the producer will ask for a new worker from the load balancer. The producer will do so for a specified number of times (10 in the current system), until it gets a new worker or uses all of the attempts. The number is chosen for no particular reason, and further experiments are required to find the optimal value. The number of retries can also be specified by the user and it can be indefinite, meaning that the producer will always be waiting for an available worker. When the producer gets a new worker, it will resume sending heartbeats. All errors regarding connection loss and interruption are handeled.

Load balancer and consumers: redirection

A consumer requests data through the load balancer. As mentioned above, the load balancer keeps track of the relationship between producers and workers. It redirects the consumer to the appropriate worker, where the data are received and stored in real time.

The following scenario describes a typical use case. A consumer makes the */get_data/int:id* API call to the load balancer. The load balancer checks whether the producer ID is being handled by anyone of the workers. If so, the consumer will be redirected to the appropriate worker. The worker will handle the request and return the data that the consumer asks for.

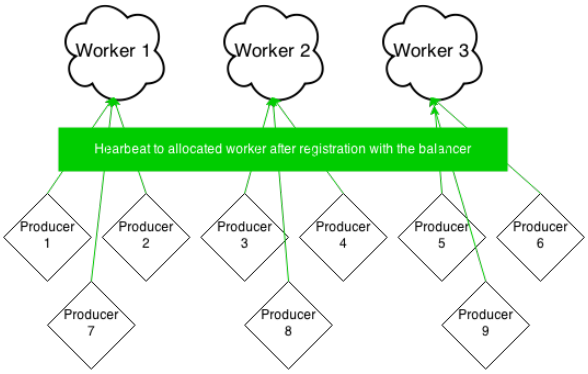

Figure 4. Producers send heartbeats to their assigned workers.

All connection errors that might occur are handeled and do not cause crashes.

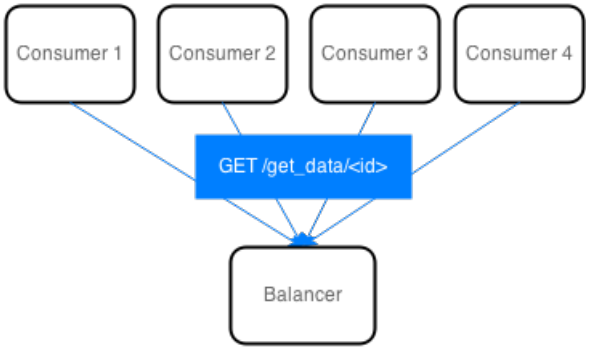

Figure 5. Consumers request data from the load balancer

Workers and consumers

The worker does several things after being contacted by a redirected consumer. Firstly, it checks whether it has the data in the cache. If so, the worker will return the data to the consumer. Otherwise, the worker will issue a database query to find such data from the database. The cache greatly reduces the number of database queries, while imroving response times.

Why Flask?

The team decides to use the python web framework Flask, alongside with Requests and SQLite. SQLAlchemy is used to mange and migrate the database with ease. There is no particular reason to use SQLite over other existing databases. A SQLite database is sufficient for the extent of this practical, and there has been no dicussion on the optimisation of the database. The python web framework Tornado is also used to help scale the system. It easily integrates with Flask and scales extremely well, which means it is ideal for real-time web services.

Why RESTful?

A decision was made to base the system on a RESTful API. RESTful API provide a simple standardized communication over HTTP, which is stateless. Also, the technology is very convenient to use on mobile devices. The language that was

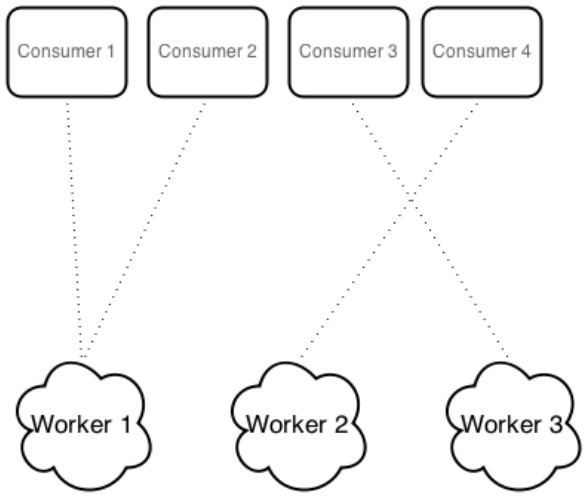

Figure 6. Workers and consumers relationships

chosen was Python and bash for the shell scripts used to simulate and test the system.

Testing

The team has included

Timing Test 1: Load balancer

Setup:

- 1 Load balancer
- 3 Workers
- 20 Producers
- 5 Consumers scripts

```
1 On one machine:
2   python run_balancer 5000
3   python run_worker 6000 localhost 5000
4   python run_worker 6001 localhost 5000
5   python run_worker 6002 localhost 5000
6
7 On a different machine:
8   ./run_producers.sh balancer_addr
        balancer_port 3000 20
9   ./run_consumers.sh balancer_addr
        balancer_port 3000 20
```

Code Fragment 2. Timing test 1: instructions

The time per request with all workers running is 0.02 seconds, whereas the time per request with only one worker running increases to 0.06 seconds. This test demonstrates that when the load balancer distributes the server load to a number of workers, it reduces the response time per request on the consumer side. It also reduces the resource load, for example memory, for each worker.

Timing Test 2: Concurrent connection

The setup is similar to that of the previous timing tests, with the addition of ab, an Apache HTTP server benchmarking

tool. The tool sends a number of concurrent requests to the server. We use this tool to examine whether the server displays any unexpected behaviour when the number of requests increases dramatically.

- 1 Load balancer
- 3 Workers
- 20 Producers
- 10 Consumers scripts
- Apache ab benchmark tool

Instructions:

```
1  ab -n 3000 -c 500 <balancer_addr>:<
       balancer_port>/get_data/<
       producer_id>
```

Code Fragment 3. Apache ab testing: instruction

The time per request is 3.86 ms across all concurrent requests. The Server handles the load well and does not crash when multiple concurrent requests are made.

Evaluation and Conclusion