Component Technology - Middleware

110011264
School of Computer Science
University of St Andrews

110002255
School of Computer Science
University of St Andrews

ABSTRACT

Abstract

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

Instructions

See “README.md” in the project directory for directions on how to run the servers, producers, and consumers.

Design

System components

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

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

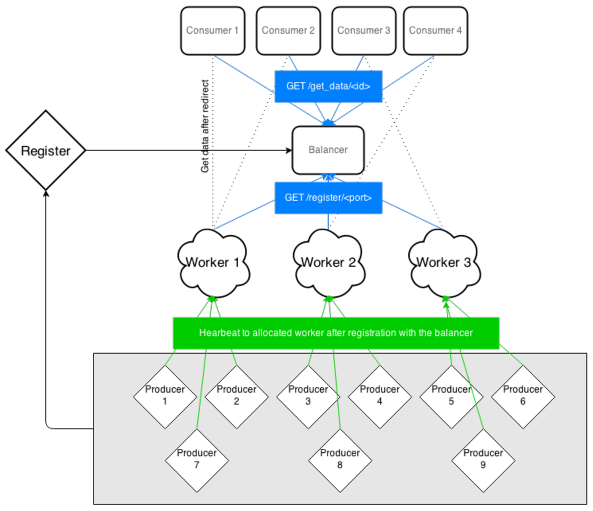

Figure 1. The final system architecture

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.

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

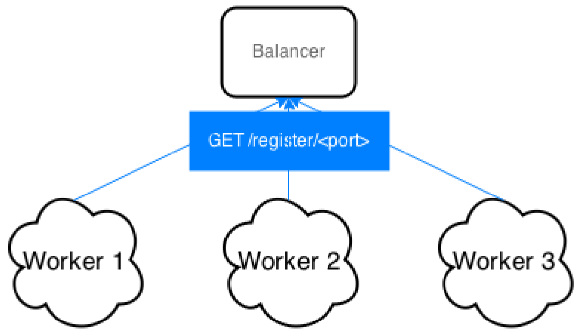

Figure 2. Workers need to register with the balancer before they can start accepting work.

do they speak to the load balancer again. To register, the producers send the

An */register/int:id* API call is made by the producer to the balancer with its ID. The balancer keeps track of the relationship between producers and workers which is used when the consumer requests are handled. When a worker is found, the producer gets redirected to it and the communication between the producer and balancer is over.

A typical use case: A producer is started by providing a balancer address and producer ID. The finds a worker for the producer or returns a message that no workers are available. If a worker is available the corresponding address is returned. All errors regarding connection loss or interruption are handled.

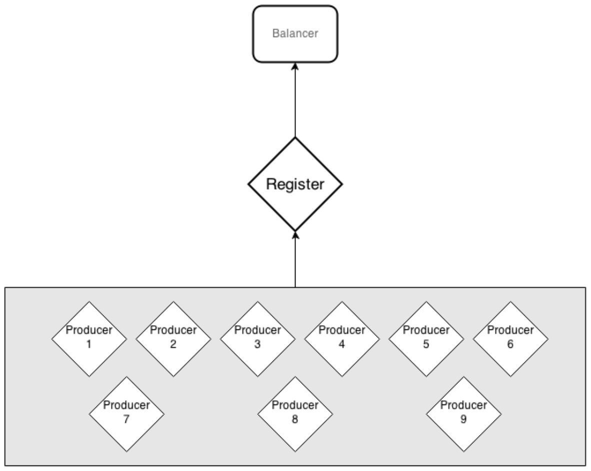

Figure 3. A producer first registers with the balancer and is assigned a worker

Workers and producers: heartbeats

After a producer receives its assigned worker address it starts sending heartbeats (periodic messages). They contain 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

DATABSE STUFF HERE If the worker goes offline / crashes the producer can try to reconnect a specified number of times. It does that through the balancer so if more workers are available it immediately gets given a worker or the moment a worker becomes available. The number of retries can be specified by the user and it can be indefinite, meaning the producer can always be looking for a connection.

A typical use case: A producer receives a worker address from the balancer and uses it to start making hearbeats to the worker, providing data and information. If the worker happens to go down, a retry mechanism triggers and the producers tries to re-register with the balancer. If workes are available the producer gets a new worker and continues heartbeats. All errors regarding connection loss or interruption are handeled.

On each heartbeat POST request, the worker extracts the “id”, “location”, and “data”. 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 “heartbeat” object (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.

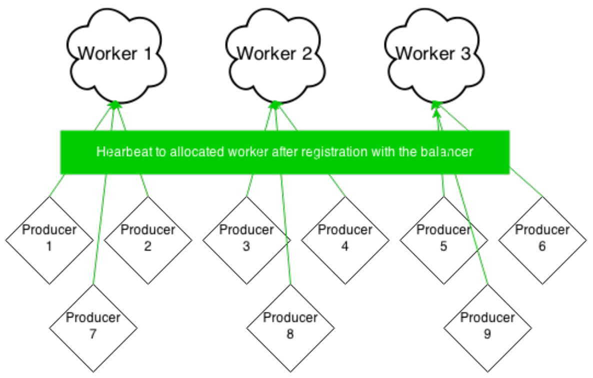

Figure 4. Producers send heartbeats to their assigned workers.

Load balancer and consumers: redirection
When a consumer requests data it does it through the balancer. As mentioned above the balancer keeps track of the relationship between producers and workers. It redirects the consumer to the appropriate worker to where the data is being stored / received in real time.

A typical use case: A consumer makes a GET request to the balancer. The balancer checks if the ID of the specified producer is being handled by some of the workers. If so the consumer get redirected to the appropriate worker which will handle the request. All connection errors that might occur that might occur are handeled and do not cause crashes.

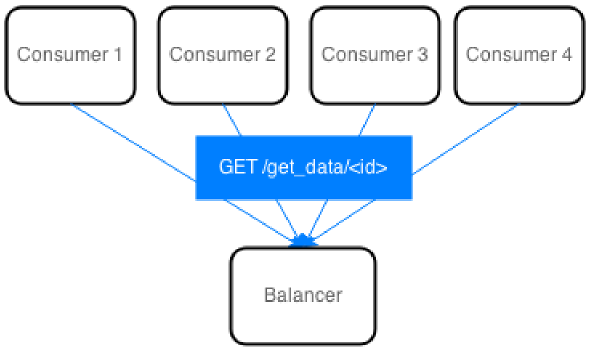

Figure 5. Consumers request data from the load balancer

Workers and consumers
The worker does several things after being contacted by a redirected consumer. First it checks if the requested producer is online and sending that. If yes, then it gets the data from the cache without making queries to the database. In case that the producer is unreachable and there is no current data in the cache a call to the database is made. This greatly reduces the number of database queries while imroving response times.

A typical use case: After the consumer is redirected to a worker, the worker checks if live data for the ID is available in the cache. If not it makes a query to the database. The data is returned to the consumer.

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. It is very convenient to use on mobile devices as well. The language that was chosen was Python and bash for the shell scripts used to simulate and test the system.

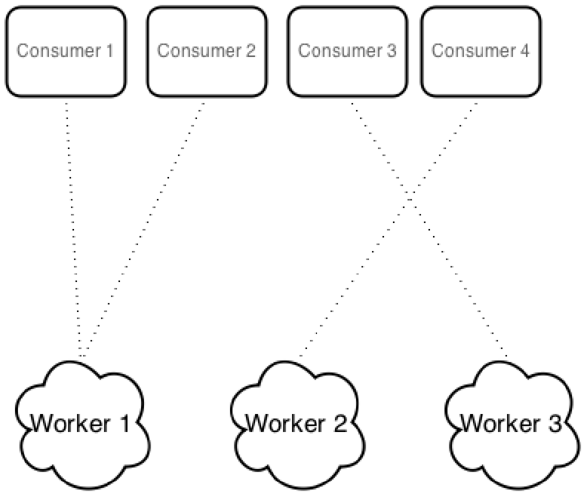

Figure 6. Workers and consumers relationships

Evaluation

Testing

The team has included

Timing Test 1: Load balancer
Setup:

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

```
On one machine:
   python run_balancer 5000
   python run_worker 6000 localhost 5000
   python run_worker 6001 localhost 5000
   python run_worker 6002 localhost 5000

On a different machine:
   ./run_producers.sh balancer_addr
       balancer_port 3000 20
   ./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:

```
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.