Service-oriented middleware for mobile sensing

110015230, 110006876

Goals

Modern mobile devices are equipped with various complex sensors – accelerometers, location, video, audio, temperature, altitude, etc. Not all of them provide scientific-grade precision or update frequency. However these disadvantage are often compensated by ubiquity and availability of such devices, allowing unprecedented deployment scale for scientific experiments.

Design

Key protocols

As described in the brief, the sensor data producer is deployed on a mobile device. This dictated a number of constraints that had to be considered while developing the system.

The technologies used to develop the producer had to be easy to deploy on a mobile device. In recent years, it has become increasingly popular to develop mobile web apps – applications that use device's browser as the execution environment. Such applications have an advantage of being easy to distribute and deploy on various platforms. HTML5 and Javascript, used to develop web apps, provide rich APIs for accessing sensor data – accelerometers, audio, video, etc. Thus, the project aimed to utilise technologies that are supported by browsers of modern mobile platforms.

WebSockets and WAMP

WebSockets is a protocol for full-duplex (two-way) communication through TCP. Additionally, compared to regular HTTP requests, WebSockets have a low header overhead (2 bytes compared to kilobytes in HTTP). Two-way communication and low overhead (thus, decreased latency and bandwidth usage) are advantageous for a data streaming platform.

WebSockets are supported by most of the modern desktop and mobile browsers (through Javascript APIs, without utilising any plugins), which makes it easy to use WebScokets-based communication in web-apps. Various libraries implement the WebSockets for other languages – Python, Java, Go, etc.

WAMP is a standard that builds upon WebSockets, providing some Pub-Sub and RPC functionality based through WebSockets. There are currently two versions of WAMP – V1 and V2. The more stable, V1 was selected due to abundance of libraries implementing the standard.

[http://wamp.ws/implementations/wamp1/]

REST

Representational state transfer (REST) is a set of principles that uses Web standards (such as HTTP and URL) to structure application-to-application communication. The system operates through a REST API, the registry offering various methods to clients (e.g. '/register_producer', '/request_brokers') which are called via GET requests.

Producer

It was decided that a producer must send the data only to one source. Multiple consumers might be interested in the data provided by the producer, which would cause scalability issues in the case of direct producer-to-consumer communication. Implementing data broadcasting through the producer would not be acceptable, since mobile devices are often limited in bandwidth.

Therefore, the producer should never directly communicate with the consumers, broadcasting its messages though a proxy – a broker.

Components

Registry

Broker

Consumer

Implementation

Components

Registry

Registry is used to: register and keep track of data providers, as well as providing addresses of relevant brokers to the inquiring consumers. A REST API is used for communicating with the registry. The popular Flask web framework is used to serve HTTP requests.

The registry maintains a list of brokers (and, thus, producers associated with them). For each producer-broker pair, we keep track of address of the broker, available sensors, current location, the time of last heartbeat and other information. At every change, the list is serialised and saved in a file (using Pickle). This allow to restore the state of the registry after it has crashed (on start, the contents of broker list are loaded from the serialised file).

The registry is responsible for spawning brokers for newly registered producers. Python's subprocess module was used to execute processes that would run brokers. The aim was to make the brokers independent of the registry. Generally, child processes are directly reliant upon the parent process. Thus, if the registry (parent) crashes, the brokers would be closed (children). Such behavior was suppressed by providing a number of special flags and commands:

- nohup – is a POSIX command that tells the process to ignore the Hangup signal
- preexec_fn=os.setpgrp – creates new process group and places the child process into it
- stdout=open('/dev/null', 'w'), stderr=open('logfile.log', 'a') – detaches the process from the console of the parent process
- close_fds = True – allows to prevent the child processes inheriting the parent's open ports (through file descriptors). This is important because otherwise the registry would not be able to access the default port after the restart, since the children are blocking it.

As a result, the brokers are running completely independently of the producer process. This allows the to restart the registry without disrupting the brokers (and vice versa).

Broker

Producer

Consumer

Testing

Reproducing tests

Future work

Conclusion