Centro Federal de Educação Tecnológica de Minas Gerais
Engenharia de Computação
Sistemas Distribuídos
1/2017

TRABALHO PRÁTICO 2: DETECÇÃO DE FALHAS

Professora: Anolan Barrientos
Alunos: Pedro Felipe Froes
Saulo Antunes

Trabalho prático realizado para a disciplina Sistemas Distribuídos do curso de Engenharia de Computação do CEFET-MG, lecionada pela Prof. D.Sc. Anolan Barrientos.

I. Introdução

Uma aplicação distribuída para o sistema operacional Android será projetada, com o desenvolvimento em Java sendo realizado por meio da IDE Android Studio e o desenvolvimento web por meio de HTML, CSS e JavaScript. Na aplicação, variáveis obtidas por meio de sensores do dispositivos Android são compartilhadas entre os componentes do sistema distribuído fazendo uso de protocolos de troca de mensagens e detecção de falhas.

Variáveis correspondentes à localização e à luz ambiente dos dispositivos participantes podem ser compartilhadas por meio de uma integração *machine-to-machine* (M2M). O protocolo Message Queuing Telemetry Transport (MQTT) pode ser utilizado para implementar um sistema de *publishers* e *subscribers*, onde os dispositivos Android publicam suas variáveis, e essas podem ser acessadas por usuários inscritos no *feed* do dispositivo por meio de um cliente web.

II. Desenvolvimento

II.i. *Setup* inicial e coleta de variáveis

Um novo projeto pode ser criado no Android Studio selecionando a opção Empty project ao iniciá-lo. O projeto criado possui as pastas `manifest` e `java` , que contem arquivos de metadados e de código do projeto, respectivamente, além de uma seção de *scripts* do Gradle que auxiliam na compilação e criação do projeto.

Para acessar variáveis do dispositivo como localização e luz ambiente, é necessário importar a API do Google Play para o projeto, o que pode ser feito através do Android Studio na seção de Tools/Android/SDK Manager, incluindo a API no Gradle posteriormente. Foram importadas especificamente a API

relacionada à localização e luz ambiente. Posteriormente, é necessário atualizar as permissões da aplicação para acessar a localização do dispositivo.

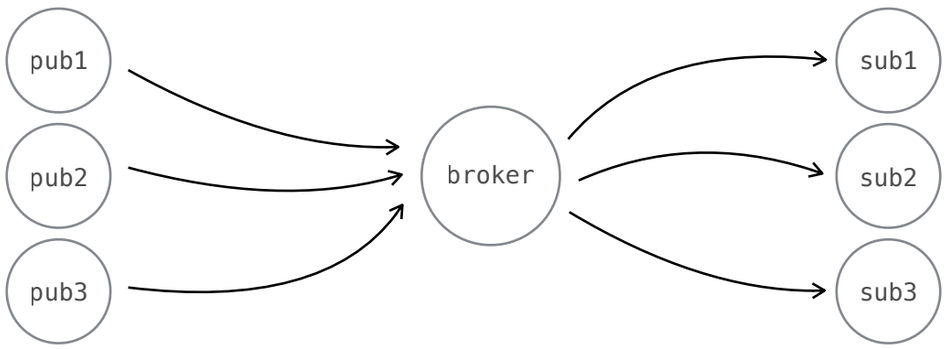
A localização do dispositivo pode ser obtida implementando os métodos da interface `LocationListener` na classe principal. É necessário inicializar um objeto do tipo `LocationManager`, que será responsável por chamar os métodos `getSystemService` e `requestLocationUpdates`. Esse último método possui quatro parâmetros principais:

- `provider`, que recebe o provedor da localização do dispositivo – nesse caso, o GPS do mesmo;
  `minTime`, o intervalo mínimo entre a atualização da localização;
- `minDistance`, a distância mínima entre a atualização da localização;
- `listener`, o método que implementa a interface `LocationListener` – nesse caso, a classe `MainActivity` da aplicação.

Já a luz ambiente pode ser obtida por meio dos objetos `Sensor` e `SensorManager`, sendo especificada no método `getDefaultSensor` com o parâmetro `TYPE_TEMPERATURE` sendo passado. Finalmente, tanto a localização e quanto a luz ambiente são atualizadas em tempo real por meio do métodos `onLocationChanged` e `onSensorChanged`.

II.ii. Sistema *publish-subscribe*

Um sistema *publish-subscribe* é um padrão de troca de mensagens onde componentes enviam mensagens (os *publishers*) de um determinado tópico, enquanto outros componentes (os *subscribers*)



recebem mensagens de um ou mais tópicos. Os *publishers* não precisam necessariamente saber quais *subscribers* assinam suas mensagens; analogamente, os *subscribers* não precisam saber quais *publishers* publicam em certo tópico. O funcionamento de um sistema *publish-subscribe* é ilustrado na Figura 1.

Figura 1 – Esquema de funcionamento de um sistema *publish-subscribe*.

Nesse trabalho, os *publishers* correspondem aos dispositivos móveis, que atualizam um cliente web com variáveis como localização e luz ambiente, por exemplo, em tempo real por meio de um Broker MQTT. O Broker é responsável por distribuir as variáveis para clientes interessados de acordo com o tópico da mensagem.

O Broker MQTT será implementado através de uma biblioteca *open-source* em JavaScript, a MQTT.js, cuja instalação dá-se por meio do Node Package Manager (NPM). A MQTT.js fará o papel do Broker, distribuindo as informações das variáveis para quem tiver assinado para receber.

O cliente web do sistema *public-subscribe* foi construído por meio de HTML, CSS e da biblioteca em JavaScript Mows, que é implementada em conjunto com a MQTT.js, criando um canal com o Broker através do protocolo Websocket na porta `8080`. Portanto, o cliente web utiliza o Websocket para se conectar no Broker, enquanto os dispositivos Android utilizam o protocolo MQTT. Esses dispositivos

podem mandar suas variáveis na forma de mensagem através da URL `ws://iot.eclipse.org:80/ws` para a porta 1883, enviando uma mensagem e o tópico a qual ela pertence, como mostrado no código abaixo:

```
public void sendMessage(String topic, String message) {
      try {
            MQTT mqtt = new MQTT();
            mqtt.setHost("iot.eclipse.org", 1883);
            BlockingConnection connection = mqtt.blockingConnection();
            connection.connect();
            connection.publish(topic, message.getBytes(), QoS.AT_LEAST_ONCE,
                  false);
      } catch (Exception e) {
            e.printStackTrace();
      }
}
```

Bloco de código 1 – Método na aplicação Android que conecta o dispositivo com o Broker por meio da biblioteca MQTT.js na porta 1883.

No cliente, primeiramente o usuário deve-se conectar ao endereço do Broker MQTT no Websocket. Após isso, ele pode escolher qual tópico deseja assinar, e as mensagens desse tópico chegarão em tempo real para o usuário. Também é possível que o usuário deixe de assinar um tópico se ele quiser. A Figura 2 mostra o cliente web com as opções de conectar e desconectar do Broker, e se inscrever ou não em determinado tópico especificado pelo usuário.

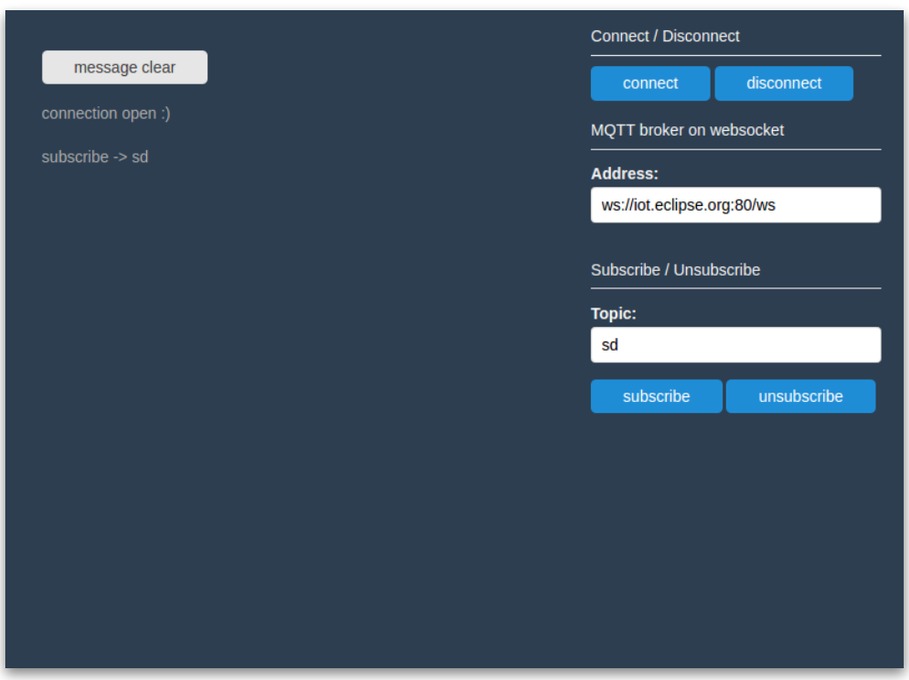


Figura 2 – Cliente web conectado ao Broker MQTT e inscrito no tópico `sd`, que não contém nenhuma mensagem.

II.iii. Detecção de falhas

A implementação da deteção de falhas em uma aplicação distribuída é necessária, uma vez que as máquinas participantes do sistema estão sujeitas a falhas e isso influencia o funcionamento da aplicação como um todo. O serviço de detecção de falhas atua a cada vez que um dispositivo entra, sai, ou sofre uma falha no grupo de dispositivos conectados por meio do sistema.

Para implementar o serviço de detecção de falhas, utilizou-se a biblioteca Atomix para a linguagem Java, dado que a mesma disponibiliza ferramentas para realizar o gerenciamento de grupos e algoritmos de eleição para lidar com a falha de nós da aplicação distribuída. Além disso, como é utilizada por meio de Java, sua utilização é possível em diferentes plataformas. Para ser utilizada, basta incluí-la como dependência no Maven em um projeto Java.

Com o Atomix instalado, deve-se primeiro realizar um *bootstrap* de um novo *cluster*, que será constituído pelo grupo de dispositivos participantes do sistema distribuído. Um *cluster* é formado por réplicas que gerenciam o estado dos recursos distribuídos e por clientes que permitem a criação remota de recursos. O *cluster* pode ser *bootstraped* através do comando:

```
AtomixReplica replica = AtomixReplica.builder(new Address("123.456.789.0",
     8700)).build();
replica.bootstrap().join();
```

Bloco de código 2 – Realizando o *bootstrap* de um novo *cluster* no endereço `123.456.789:8700`

Após a criação do *cluster*, as réplicas pertencentes a ele se coordenam para eleger um líder entre elas, enquanto as remanescentes se tornam seguidoras. Isso ocorre por meio do algoritmo de eleição, que é responsável por eleger o líder do *cluster* e lidar com a detecção de falhas através de *heartbeats* entre os participantes. Líderes são selecionados por meio de votação, e para que um líder seja eleito, é necessário que a maioria dos membros do *cluster* seja capaz de se comunicar entre si.

O Atomix permite configurar diferentes *timeouts* do seu algoritmo de consenso. A propriedade `electionTimeout` corresponde ao tempo em que uma nova eleição ocorrerá caso um seguidor não receba um *heartbeat* do líder. Já o `heartbeatInterval` corresponde ao tempo entre os *heartbeats* enviados do líder para seus seguidores, e deve ser necessariamente menor que o `electionTimeout`. Enfim, o `sessionTimeout` corresponde ao tempo em que um nó é removido do *cluster* após não enviar uma confirmação de recebimento do *heartbeat* (um *keep-alive*) para o líder.

III. Resultados

Pode-se utilizar a aplicação através de um emulador de Android ou no próprio dispositivo. Utilizando em um emulador, deve-se utilizar um simulador de sensores para captar alguma leitura. Para utilizar em um dispositivo, basta abrir a aplicação que tanto a localização quanto a luz ambiente serão atualizadas em tempo real, enviando seus valores para o cliente web do sistema *public-subscribe*.

No cliente web, o tópico escolhido tem suas mensagens exibidas em tempo real. A medida que a localização é atualizada, o cliente é atualizado com a localização de um *publisher* conforme mostrado na Figura 3.

No âmbito da detecção de falhas, os parâmetros `electionTimeout`, `heartbeatInterval`, `sessionTimeout` foram iniciados com os valores de `0.50`, `0.25` e `5` segundos, respectivamente. Além disso, foram realizados diferentes testes a fim de verificar o serviço de detecção de falhas. No primeiro teste, mediu-se a quantidade de mensagens trocadas por segundo entre 4 máquinas distintas, e no segundo mediu-se a mesma quantidade, porém considerando uma mudança no grupo.

Por meio da equação `2(n–1)/heartbeatInterval`, é possível calcular o número de mensagens trocadas no espaço de tempo de um *heartbeat*. Para o primeiro teste, obteve-se uma quantidade de 6 mensagens/intervalo de *heartbeat* entre 4 dispositivos do cluster, o que corresponde a 24 mensagens por segundo. Já no segundo teste, obteve-se uma quantidade de 8 mensagens por *heartbeat*, o que geram 32 mensagens por segundo. É esperado que a quantidade do segundo teste seja maior que a

primeira, dado que ao ocorrer uma mudança no grupo, ocorre também uma troca de mensagens crescente para lidar com a mudança.

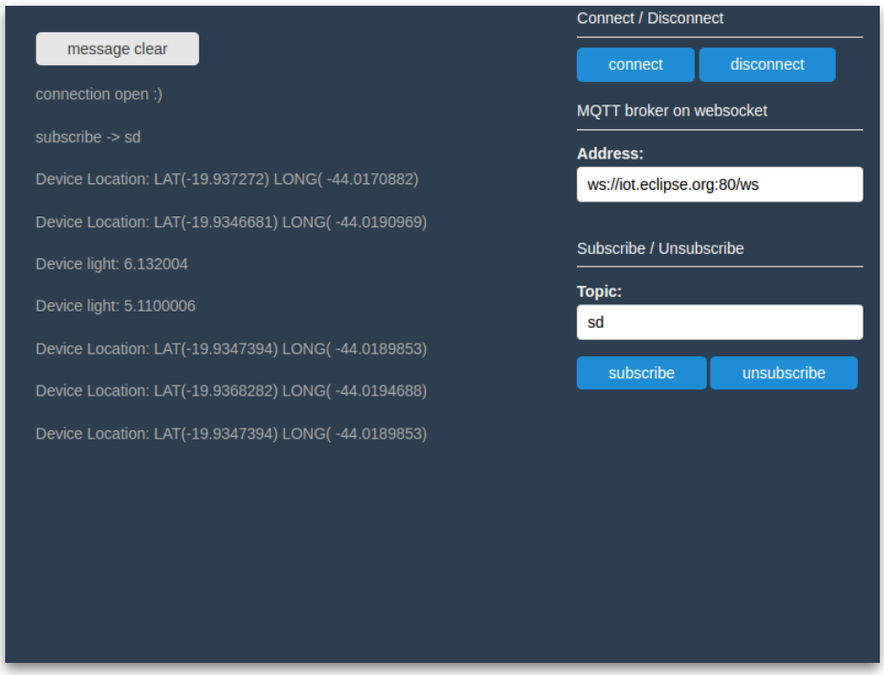


Figura 3 – Cliente web conectado ao Broker MQTT recebendo atualizações das variáveis de localização e luz no tópico `sd`.

Entre algumas dificuldades encontradas no decorrer do desenvolvimento do projeto, está a disponibilidade de sensores no hardware de dispositivos Android. Por exemplo, alguns dispositivos não possuíam sensores de temperatura, pressão, etc. Não existe uma lista de quais hardwares possuem quais sensores, de modo que essa dificuldade foi superada através de experimentação (verificou-se se o sensor estava presente testando sua leitura na aplicação).

IV. Referências

Android Developers: *Getting the Last Known Location*. Disponível em: <https://developer.android.com/training/location/retrieve-current.html>. Acesso em: 20, Mar, 2017.

Google API for Android: *Sensors Overview*. Disponível em: <https://developer.android.com/guide/topics/sensors/sensors_overview.html>. Acesso em: 15, Abr, 2017.

Google API for Android: *Set Up Google Play Services*. Disponível em: <https://developers.google.com/android/guides/setup>. Acesso em: 20, Mar, 2017.

TechLoveJump: *Android Tutorials. Android GPS – Location Manager Tutorial*. Disponível em: <http://techlovejump.com/android-gps-location-manager-tutorial/>. Acesso em: 20, Mar, 2017.

Atomix IO. Disponível em: < http://atomix.io/>. Acesso em: 15, Mai, 2017.