Licenciatura em Engenharia Informática e de Computadores

Aplicação Android 5G QoS

Afonso Nobre, nº 44777

Ricardo Silva, nº 44837

Orientadores

José Simão

Nuno Cota

Relatório beta realizado no âmbito de Projeto e Seminário,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2020/2021

Maio de 2021

Resumo

Os recentes desenvolvimentos na rede 5G oferecem não só uma interface de rádio sofisticada como também uma eficiente arquitetura de sistemas de rede. Estes avanços disponibilizam um leque de oportunidades e novas aplicações, como por exemplo a tecnologia na construção de veículos autónomos. Para determinar a influência das condições da rede em aplicações que dela dependem, é usado um sistema já existente capaz de gerar tráfego em diferentes níveis e protocolos permitindo a recolha de informação para análise da qualidade do serviço.

A solução apresentada integra um sistema de monitorização de redes 5G designado IQ-NPE, o qual é composto essencialmente por três tipos de componentes, nomeadamente:

- *On Board Units (OBU)* - Peça de *hardware* e *software* móvel instalada em veículos de forma a gerar tráfego e recolher os parâmetros de medida de performance em diferentes pontos de controlo e observação.
- *Fixed Side Units (FSU)* - Agente de *software* fixo instalado em localizações estrategicamente pensadas tanto em Portugal como em Espanha. Estes têm a responsabilidade de gerar tráfego e comunicar com as sondas móveis para recolher parâmetros de rede tais como o *downlink* e o *uplink.*
- *Management System* - Plataforma de *software* centralizada responsável pela gestão e configuração de planos de testes. Também é responsável por recolher e processar todos os resultados obtidos pelas sondas durante a execução dos planos de testes.

O desenvolvimento do projeto é motivado pela oportunidade da criação de uma aplicação móvel para simular uma sonda móvel mais simplificada tendo como alvo smartphones comuns, de forma a complementar o sistema já existente oferecendo mais portabilidade e versatilidade.

Palavras-chave: 5G; smartphone; aplicação móvel; Management System

Abstract

The recent developments in 5G technology brings not only a sophisticated radio interface, but also a performant network system architecture. These technical achievements may bring new opportunities and new applications, for example, autonomous vehicles. To determine the influence of radio network conditions on the applications performance, it will be used a system that generates synthetic traffic at different levels and different protocols to collect information, allowing the system to analyze the service quality.

The current solution presents a 5G network monitorization system named IQ-NPE. The system architecture is mainly composed by three components:

- On Board Unit (OBU) – Hardware and software probe to be installed on vehicles to generate traffic and collect performance measurements at different points of control and observation.

- Fixed Side Units (FSU) - Software agent to be installed on both Portugal and Spain. The fixed side unit is used to generate traffic and collect performance measurements on the network side, on both downlink and uplink traffic.

- Management System - Centralized software platform used to manage test plan configuration. It will also be responsible for collecting and processing all performance assessment results obtained during test trials.

The development of this project is motivated by the opportunity of developing a mobile application to simulate a simplified OBU in an ordinary mobile phone to complement the Management System by offering more portability and versatility.

Keywords: 5G; Mobile; Application; Management System

Índice

Lista de Figuras

1. Introdução

Neste capítulo é introduzida a iniciativa para este projeto, a motivação, os objetivos e por último uma breve descrição da estrutura do relatório.

1.1 Motivação

O surgimento de avanços no desenvolvimento da tecnologia 5G abre um novo leque de oportunidades. Surgiu então uma motivação de criar uma aplicação que seja capaz de se servir desta nova tecnologia.

O âmbito é criar uma aplicação que permita oferecer uma nova frente ao projeto *Quality of Service* (*QoS*), para que este possa criar um plano exaustivo de análise à cobertura à nova tecnologia.

Uma vez que o sistema já desenvolvido tem uma componente portátil bastante limitada surgiu a necessidade de desenvolver uma aplicação móvel de forma a tentar minimizar este problema.

O *smartphone* é um dispositivo cada vez mais comum no quotidiano da sociedade, pelo que se torna no candidato perfeito para a complementação do sistema. Com um equilíbrio entre a portabilidade e as necessidades de *hardware*, é criada uma janela de oportunidade para desenvolver uma aplicação que tire partido da flexibilidade e usabilidade do *smartphone* para executar os planos de testes criados pelo sistema de gestão.

Figura 1 - Representação de uma sonda *OBU* num veículo

1.2 Objetivo

O objetivo deste projeto é desenvolver uma aplicação móvel para complementar o sistema já existente simulando de forma simplificada uma *On Board Unit* oferecendo mais portabilidade e simplicidade na interação local com a sonda. Esta tem que ser capaz de recolher parâmetros de rede de forma passiva, sem criar tráfego ou influência na rede, e realizar planos de testes propostos pelo sistema de gestão.

1.3 Contextualização e Especificações do Projeto

Para garantir que a aplicação seja realmente capaz de executar estas tarefas deverá cumprir alguns requisitos.

- Fazer a realização de testes passivos, que consistem na recolha de um conjunto de parâmetros de forma a não interferir no tráfego da rede do dispositivo. Estes parâmetros variam entre dados de débito binário, dados de georreferenciação e dados de qualificação do sinal de rede;
- Posteriormente à realização dos testes passivos, a aplicação tem de ser capaz de fazer a disponibilização ilustrativa dos resultados, por exemplo sob forma de gráficos ou tabelas. Tanto a recolha dos parâmetros como a sua ilustração são feitas localmente no dispositivo;
- Fazer a realização de testes ativos que, ao contrário dos testes passivos, vão causar tráfego intencional na rede, de forma a conseguir uma avaliação da qualidade e cobertura da mesma. A execução de uma sequência de *Pings* para um domínio explícito é um exemplo de um teste ativo que a aplicação realiza;
- Fazer a recolha de planos de testes. Estes planos são fornecidos pelo sistema central via *Web API* e consistem numa lista de diferentes tipos de testes que são executados pela aplicação de forma autónoma e sem interação do utilizador. Os testes provenientes de cada plano podem ser de natureza passiva ou ativa;
- Todos os resultados de testes realizados fora do contexto de um plano de testes são passíveis de serem guardados na base de dados local do dispositivo;
- Reportar os resultados da execução testes provenientes dos planos de testes para o sistema de gestão central.

1.4 Estrutura do relatório

Ao longo do relatório vai ser feita uma descrição geral e detalhada dos aspetos mais relevantes da aplicação implementada. No próximo capítulo vai ser feita uma análise ao sistema já existente e como é que foi feita a integração da nova aplicação no que já existia. Depois é feita uma descrição da arquitetura desenhada para a implementação da aplicação, desenvolvendo na premissa dos desafios que foram propostos e qual a abordagem tomada para os tentar solucionar. O capítulo 4 entra em detalhe sobre cada ponto da implementação da aplicação. O capítulo 5 representa uma avaliação com uma comparação da aplicação implementada com uma aplicação já existente na qual esta foi inspirada. Por fim, no capítulo 6 é feita uma conclusão sobre o trabalho feito e disponibilizada uma *overview* de como a aplicação pode ainda crescer no sentido de novas *features* que possam ser implementadas.

2. Integração com o sistema

2.1 Sistema já existente

O sistema QoS (figura 1) é uma complexa rede de sondas que, ao comunicar entre si, conseguem criar condições de tráfego virtual no âmbito de executar testes e análises à cobertura da rede na posição em que se encontram.

As sondas *on-board* (*OBU*) encontram-se instaladas em veículos, e contém vários *modems* para fazer os testes. Contêm uma interface *web* local, que permite que a sua manutenção seja feita diretamente no dispositivo. Estas conseguem também ligar-se ao *CAN BUS* (1) do veículo, de forma a aumentar a eficiência e precisão das análises feitas. A maior parte da análise é feita *offlline*, e só apenas depois de todas as medidas estarem completas é que é feita a transmissão dos dados recolhidos para o sistema central.

As sondas fixas (*FSU*) encontram-se posicionadas em certas localizações ao longo da Península Ibérica e correm numa máquina virtual dedicada, que vai ser instala na *Multi-access Edge Computing* (*MEC)* (2) da tecnologia *5G* e/ou no centro *Cooperative Intelligence Transport Systems (*C-*ITS*) (3). A nível de *hardware*, estas sondas são semelhantes às sondas *on-board*, com a exceção que estas não têm *modems* de comunicação móvel, nem a interface *web* local, porque visto que estas se encontram ligadas à rede central, podem ser geridas pelo sistema de gestão.

O sistema de gestão central é a ponte entre todos estes dispositivos. Este sistema está ligado a todas as sondas e faz a gestão da execução de testes, para evitar interferências no tráfego da rede *5G*. É este sistema que constrói e delega os diferentes planos de testes que as sondas vão executar, bem como recebe os resultados e os processa, para depois os entregar a ferramentas especializadas na sua análise.

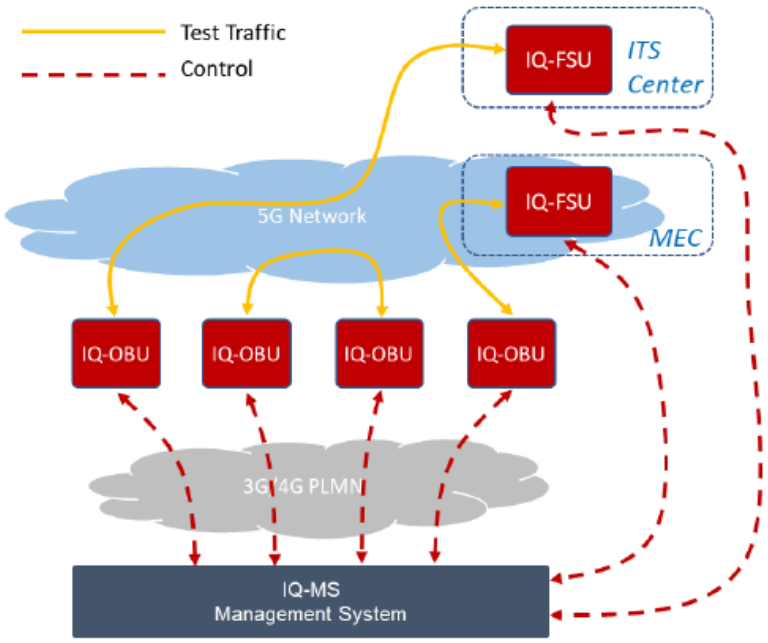

Figura 2 - Arquitetura do sistema *IQ-NPE*.

2.2 Integração no sistema

A aplicação móvel foi desenhada à imagem de uma sonda *on-board*. Mas, visto que o *smartphone* não é tão preciso e tão capaz como uma dessas sondas, pequenas adaptações foram feitas na modelação da aplicação. Apesar dos testes passivos à cobertura serem semelhantes (excetuando a interface de ligação ao veículo), os planos foram simplificados para que o telemóvel se comporte com o desempenho desejado. Na face desta condição, tiveram de ser criadas novas rotas na *API*, desenvolvidas exclusivamente para a nova sonda móvel (*Mobile Unit – MU*). Assim sendo, a interação entre sondas e *MUs* é feita de forma semelhante a sondas com outras sondas, e dessa forma é possível manter coerência na nova entrada no sistema (figura 2). A comunicação com o sistema é realizada através de uma *Web API* usando como autenticação *Basic Authentication*.

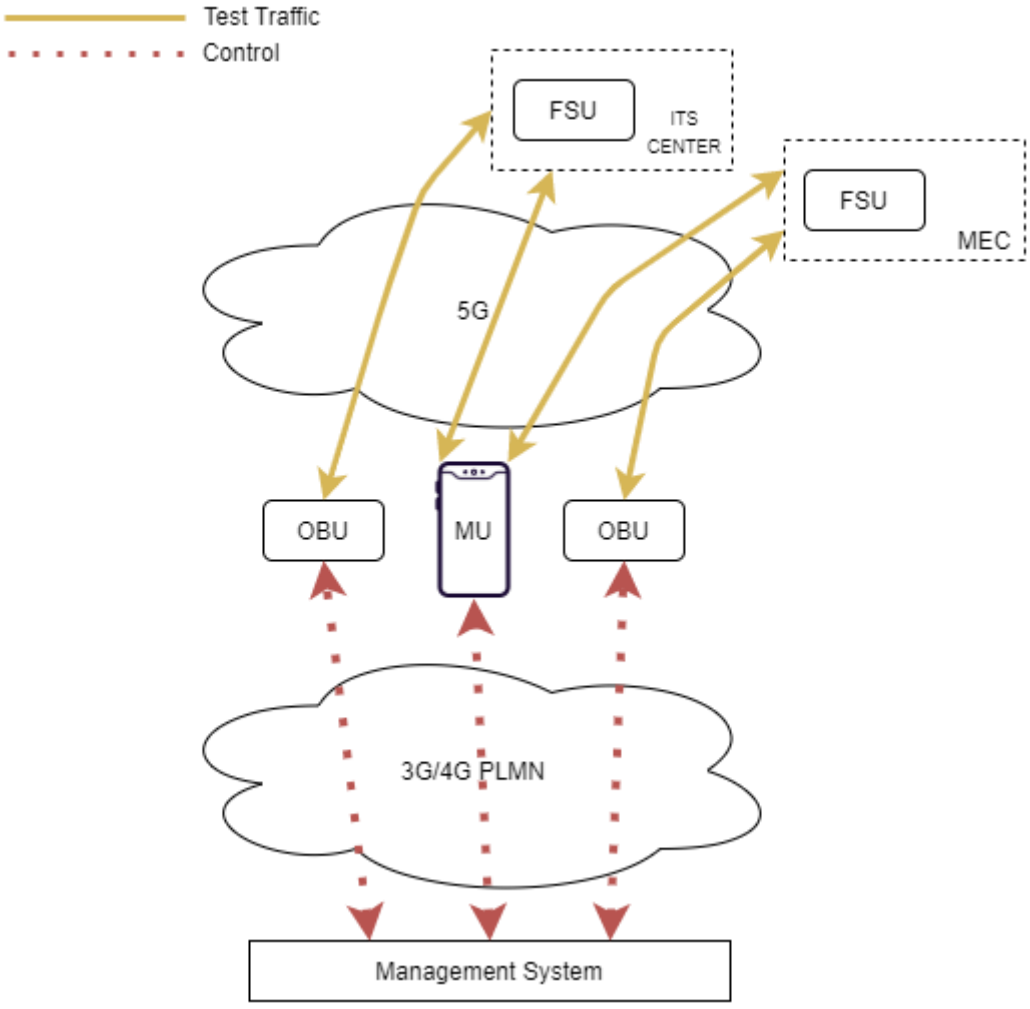

Figura 3 - Arquitetura do novo Sistema com a *MU*.

3. Arquitetura

Neste capítulo é realizada uma descrição detalhada da arquitetura adotada para a construção da aplicação android e a sua integração no sistema previamente desenvolvido.

3.1 Desafios Propostos

Desenvolver uma aplicação de testes à cobertura da rede não é tão linear quanto soa.

O desafio inicial foi fazer a execução de testes passivos na aplicação. Estes testes consistem na recolha de dados de naturezas diferentes, pelo que são retirados sobre contextos diferentes.

Além da simultaneidade na execução de testes, é necessário garantir que a análise de rede seja feita em segundo plano, bem como garantir que a análise seja feita quando a aplicação se encontra tanto em *foreground* como no *backstack* do *Android*. Quando a aplicação se encontra em *foreground*, as recolhas feitas em segundo plano são lidas e apresentadas ao utilizador, sob forma de gráfico, tabela, ou mesmo um conjunto de campos nomeados.

Existe um conceito de sessões de teste controladas no contexto da aplicação. Estas são sessões cujo propósito é fazer a realização de testes passivos com o intuito de armazenar os seus resultados no dispositivo de forma a possibilitar ao utilizador a visualização e a extração, para ficheiro *.xls*, de todos os resultados recolhidos. O objetivo desta funcionalidade é permitir ao utilizador fazer uma consulta posterior dos resultados que foram obtidos no momento da recolha.

Além do conceito de sessões controladas, existe também um conceito de planos de testes. Estes planos contêm listas de testes ativos ou passivos, que permitem à aplicação executar uma bateria de avaliações à rede em determinado momento. Os testes são disponibilizados pelo sistema de gestão e a estes vem acoplada uma data de início do plano. O objetivo é que seja possível programar a execução de vários testes para um futuro breve. Com este objetivo surgem dois problemas. O primeiro é a possível dificuldade de comunicação com o sistema de gestão, uma vez que o *token* de autenticação necessário poderá ficar inválido, e o segundo é a natureza dos diferentes testes. Alguns dos testes do plano podem ser de natureza assíncrona, o que derrota o princípio da execução de testes isolada de forma a minimizar as interferências na rede.

3.2 Solução para os objetivos propostos

Para o primeiro desafio foi implementado um padrão que contenha diferentes implementações de recolha para cada conjunto de parâmetros de rede, e um algoritmo que invoque em simultâneo cada implementação. Assim, é possível recolher todos os tipos de parâmetros de rede que os testes passivos consistem e ilustrar os seus resultados ao utilizador em tempo real.

Na tarefa da execução de testes passivos a paralelização é conseguida através do uso de uma *thread* da *Thread Pool* do sistema para fazer a recolha e armazenamento, em base de dados, de cada tipo de parâmetro. Deste modo, atráves da base de dados é possível para a *UI* obter os resultados das recolhas.

para evitar influências indesejadas na rede, foi criado um conceito de sequencialização de tarefas. Dessa forma, a aplicação cria um *pipeline* nas tarefas que tem a executar dependendo da natureza da mesma.

Na paragem da aplicação, quando esta passa para o *backstack* do *Android*, a representação de resultados por parte da *UI* deixa de ser necessária, mas a recolha de parâmetros continua a ser executada pela *thread* de *background* a quem as tarefas foram delegadas, resolvendo assim o primeiro desafio.

Para resolver o problema do *token* de autenticação, é feita uma revalidação do *token* de 15 em 15 minutos, para que este seja válido no momento da execução dos testes.

Para resolver o problema do assincronismo dos testes, é necessário sequencializar a sua execução, garantindo que o teste seguinte apenas é executado quando o anterior terminar e devolver os resultados.

3.3 Visão geral das tecnologias utilizadas

À luz do facto de que a aplicação implementada não encaixar no estereótipo habitual de uma aplicação mobile, e fugir um pouco à modelação conceptual que é costume encontrar noutras aplicações, foi feita a escolha de implementar a mesma em nativo, usando a linguagem de *Kotlin*.

Foram usadas várias *Frameworks* para implementar a lógica da aplicação, designadamente:

- *Volley* (4) – Utilizado para fazer os pedidos *HTTP* à *API* disponibilizada. É possível redefinir o modo como os pedidos e respostas são processados para personalizar cada pedido à forma da aplicação.
- *MPAndroidChart* (5) – Esta é a *Framework* utilizada para o desenho dos vários gráficos de informação que se encontram ao longo da aplicação. Com uma estrutura clara, é possível definir o número e aspeto de varáveis presentes em cada gráfico e, através da afetação de *flags*, personalizar a navegação dentro de cada gráfico também.
- *Room* (6) – Implementado por base em *SQLite*, esta é a *Framework* que permite fazer a criação de uma base de dados local ao processo da aplicação. Através de um esquema de Entidades e classes *DAOs* (*Data Access Object*), é possível fazer pedidos de base de dados com facilidade.
- *Material UI* (7) – *Framework* de componentes visuais. É a fonte de vários componentes relativos à *UI* da aplicação, tanto estruturais como meramente visuais.

4. Detalhes de implementação

Nas próximas secções vai ser explicado a forma como a aplicação foi desenhada, partindo cada forma relevante de componente para uma maior facilidade de compreensão. Cada subsecção vai referir o seu próprio componente.

4.1 Autenticação

O primeiro componente da aplicação é o componente que faz a gestão da autenticação do utilizador. Para uma autenticação bem-sucedida é necessário que o utilizador faça o *login* no sistema, e que o dispositivo seja registado na rede, para poder guardar os resultados dos testes posteriormente.

Para o utilizador se autenticar no sistema é necessário que este tenha sido registado previamente. Essa responsabilidade faz parte dos administradores do sistema de gestão, pelo que apenas por pedido é possível fazer o registo. Após registado, um utilizador recebe o *username* e a *password*. Como apresentado na secção anterior, um diálogo inicial aparece a requerer ambos os campos preenchidos. A autenticação é feita através de *Base64,* e um *authentication token* é enviado para o sistema no *endpoint* de *login*.

Assim que o processo de *login* do utilizador estiver concluído, a aplicação vai passar a registar o dispositivo na rede. Sendo que cada dispositivo tem de atuar como uma sonda única, precisa de um código que sirva identificador. Inicialmente pensou-se em usar o *International Mobile Equipment Identity* (*IMEI*) para identificar cada telemóvel. O problema que surgiu, por questões de segurança e privacidade, é que este código único deixa de estar acessível programaticamente a partir da versão de *Android 10+*. Cada vez mais telemóveis vão estar equipados com essa versão ou superior, tornando a solução rapidamente obsoleta.

Para resolver o problema foi decidido que ao iniciar a aplicação pela primeira vez é necessário que o utilizador consulte o número de série do dispositivo e o insira quando for fazer o *login*, juntamente com o resto das credenciais. Após esse processo o identificador fica gravado na biblioteca de *settings* da aplicação. A vulnerabilidade da solução é que ao limpar os dados da aplicação ou a desinstalar a mesma, o identificador perde-se com os outros dados, e o utilizador terá de o inserir da próxima vez que fizer o *login.* Por outro lado, ao usar o número de série do dispositivo, pode-se garantir a identidade de cada *smartphone* no registo da rede do sistema, e torna-se possível *logins* de vários utilizadores num só dispositivo.

4.1.1 *Refresh* do *token*

O *token* tem formas de ficar inválido, sendo o *timeout* uma dessas formas. A aplicação tem como objetivo fazer sessões indeterminadas de testes, bem como recolher parâmetros a qualquer momento. Para isso é necessário que o utilizador tenha o *login* feito até que este decida o contrário. No entanto, deixar a responsabilidade de refazer o *login* sempre que o *token* expire parece impensável. Dessa forma, a partir do momento em que o *login* está feito, é lançado um *Worker* (9) que irá executar a tarefa de, ao fim de 45 minutos, fazer o *refresh* do *token*. Este *refresh* é feito através de um *endpoint* da *API*, para que a aplicação possa continuar a executar o seu propósito.

4.2 Recolha de parâmetros de rede

Uma vez dentro da aplicação, o utilizador é deparado com um ecrã com uma grande quantidade de informação. Neste momento, a aplicação já se encontra a recolher parâmetros de várias naturezas de forma autónoma. Esta subsecção explica como é feita cada uma dessas recolhas, e o porquê da abordagem implementada.

Em primeiro lugar é necessário destacar que são feitas duas recolhas em simultâneo: a recolha de parâmetros de rede móvel, de rádio e de localização (denominados daqui em diante de *radio parameters*) e a recolha dos valores instantâneos de débito binário, ou seja, a velocidade de *upload* e *download* (denominados *throughput*). Feita a distinção, assim que o utilizador passa do *login*, terão de ser lançadas duas tarefas de recolher esses parâmetros.

Inicialmente foi pensado em executar uma instância de *WorkManager* (10). Existindo duas formas de trabalho, único e periódico, e dado o facto de ser necessário uma análise de parâmetros contínua, parece final que a escolha fosse o trabalho periódico. O primeiro percalço foi que o tempo mínimo de periodicidade deste tipo de *Work* é de 15 minutos. Como o objetivo era fazer uma recolha com intervalos máximos de alguns segundos, a opção não se provou adequada. Então fez-se a decisão de criar um *UniqueWork* (11) cujo trabalho seria fazer a recolha num ciclo indefinido, sempre a executar as leituras dos parâmetros necessários. O dilema atingido de seguida foi a dificuldade no cancelamento deste trabalho, quando este já não fosse necessário, pois as funções de paragem disponibilizadas pela *framework* não são capazes de parar trabalho em execução. A solução do *WorkManager* foi dada então como inadequada e, portanto, encerrada.

Como nova solução foi feita uma implementação genérica de *JobService* (12). A maior diferença entre a implementação anterior é que esta *framework* permite fazer o cancelamento de um trabalho pelo seu *id*. A ideia é ter um delegado que executa as tarefas de recolha de parâmetros. Estas duas tarefas são executadas com uso do padrão de desenho *Strategy* (13). Foi definida uma abstração *IWorks* (figura 3) e, por cada recolha a ser feita, uma concretização dessa abstração que implementa a função *work*. Dessa forma é possível ter o delegado genérico que irá receber a injeção de dependências e executar cada instância da função *work*. Através de uma identificação por enumerado, a instância de *JobService* criada consegue identificar todas as concretizações desejadas e executar cada uma delas. Faz parte da responsabilidade deste *scheduler* saber quando executar cada um dos trabalhos de recolha. Dentro de cada concretização há duas outras funções chave que se revelam extremamente importantes para o correto funcionamento de cada trabalho: uma função que devolve o intervalo de frequência, em milissegundos, em que este trabalho deve ser executado, e uma que devolve uma lista de identificação de parâmetros que o trabalho precisa. O *scheduler* requer apenas de fazer a criação dos parâmetros necessários e chamar a instância de *work* sempre que o momento atual exceda o momento da última execução adicionado ao tempo de *timeout* (figura 4).

```
interface IJobs {

    fun job (params: Map<JobParametersEnum, Any?>)

    fun getJobTimeout () : Long

    fun getJobParameters():Array<JobParametersEnum>
}
```

Figura 4 - Implementação da abstração *IJobs*.

```
val workInstance = JobsMap.worksMap[jobType]

if (System.currentTimeMillis() > (lastRuns[index] +
workInstance!!.getJobTimeout())) {
    lastRuns[index] = System.currentTimeMillis()

    asyncTask({

        workInstance.job(createWorkerParams(workInstance))

    })

}
```

Figura 5 - Inicialização de um trabalho por parte do *Scheduler*.

De cada vez que o trabalho de cada uma das concretizações é executado, as informações recolhidas são gravadas na base de dados local do processo. Gravar estas informações na base de dados permite que a *UI* seja notificada sempre que houver uma alteração aos registos, ou seja, um novo bloco de informação relevante para mostrar ao utilizador.

Com esta implementação torna-se possível fazer facilmente a criação de novas concretizações de *IWorks* e o problema de cancelamento de trabalho fica resolvido.

4.3 Representação de dados

Uma vez feita a recolha de todos os parâmetros necessários, é altura de notificar a *UI* sobre as alterações que decorreram, para proporcionar ao utilizador um *insight* de como as alterações estão a fluir.

4.3.1 Recolha de dados

Para a *UI* poder ser notificada, tem de ser criada uma ponte entre a atividade e a base de dados. Essa ponte é feita através de uma observação em tempo real a uma *query* à base de dados. Por paradigma do *Room*, uma pesquisa na base de dados devolve uma instância de *LiveData* (14). Essa instância contém uma função denominada *observe()* que recebe dois parâmetros:

- O contexto da observação (que é a instância do *lifecycleOwner* (15));
- Uma instância de *Observer* (16);

Este segundo parâmetro de entrada atua como uma função *lazy* de *callback* que recebe como parâmetro os dados que se pretendem observar. Dessa forma, ao registar um *observe* a uma *query* à base de dados, é possível executar o *callback* sempre que o trabalho faça uma recolha com sucesso.

4.3.2 Representação de dados

Sempre que o *callback* de *Observer* executa, vai ser feita uma atualização dos gráficos que são apresentados no ecrã. Em cada execução, ao analisar o parâmetro de entrada do *callback*, é criada uma nova entrada na informação a apresentar. Esta nova informação vai ser apresentada sob várias formas, sejam elas uma tabela de detalhes sobre cada célula móvel que o dispositivo contém, ou num gráfico de linhas com um eixo de abcissas temporal, que revela a evolução dos valores de cada parâmetro de rede.

No primeiro caso, o utilizador depara-se com um separador que oferece uma tabela com dados e outros detalhes em relação às diferentes células de rede. Estes valores são atualizados sempre que o c*allback* dispara, ou seja, quando há uma alteração proveniente da camada de acesso a dados (figura 5).

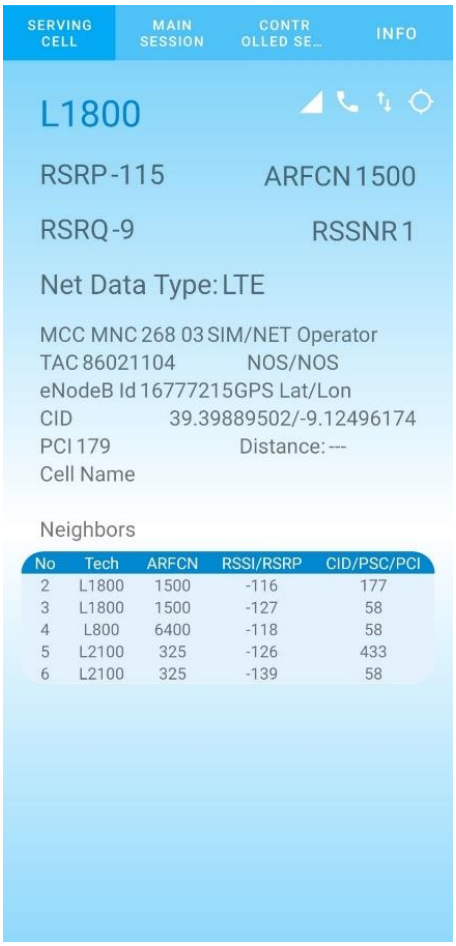

Figura 6 - Exemplo dos detalhes de cada uma das células de rede móvel.

No segundo caso, há um novo separador que oferece uma variedade de gráficos que apresentam partes de um todo a nível de parâmetros de rádio. Apresentam a evolução de todos os parâmetros de rede da célula de serviço atual do dispositivo. Em cada execução há uma atualização da informação por parte de cada gráfico, é criada uma nova entrada e cada instância de gráfico é notificada de que houve uma alteração. Posteriormente a aplicação avança o gráfico sobre o eixo temporal de forma que o utilizador esteja sempre a ver a informação atualizada (figura 6).

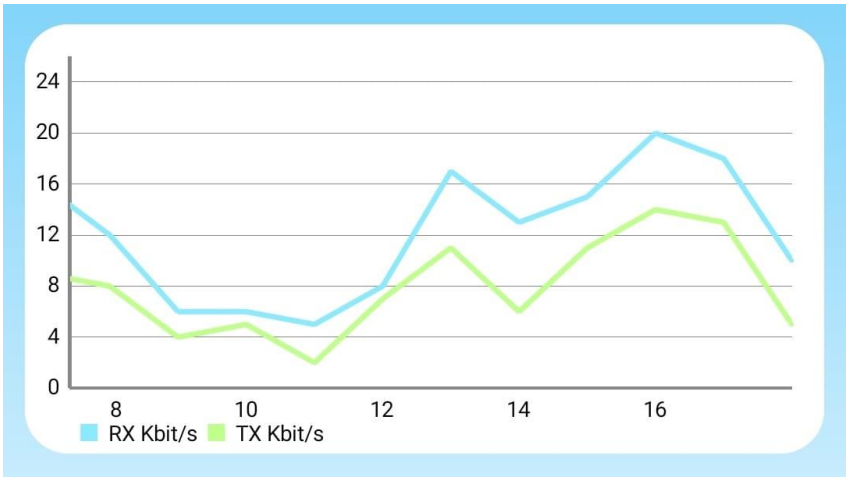

Figura 7 - Gráfico do *Throughput*.

Como foi referido anteriormente, existe um conceito de sessões de teste. Mas como a aplicação não passa 100% do tempo a executar sessões de teste, existe também uma sessão principal cujo objetivo é apenas disponibilizar uma interface gráfica para as alterações do momento, logo a sessão *default* não tem como propósito gravar dados para enviar para o sistema de gestão. Quando é para criar uma nova sessão de testes tem de se fazer a paragem da sessão *default* para que a aplicação se possa concentrar inteiramente na nova sessão, e para não haver *cluster* de informação. Nesta transição é necessário que todas as instâncias de *observe* registadas para a sessão anterior sejam canceladas, para se poderem registar novas instâncias que apontem para a nova sessão. Este processo é necessário no sentido em que se houver mais do que um *observe* para o mesmo gráfico, dá-se um conflito de informação retornada, pelo que o utilizador iria ver informação errada.

4.3.3 Comunicação entre fragmentos e criação de novas sessões de teste

//TODO

Tivemos um erro em que à distancia de 2 framentos, o fragmento que estava anteriormente em foreground passa a um estado de paragem e é chamado o onStop e a subscrição dos eventos do green robot era desfeita, então não dava para lançar o mambo para o outro lado para criar uma sessão nova, e deixava de atualizar os mambos, então passou para o ondestroy

4.4 Implementação de testes autónomos

A implementação dos testes autónomos foi algo atribulada, visto que o grupo se deparou com algumas dificuldades, dado à natureza dos requisitos destes testes.

Em primeiro lugar, foi estabelecido que os testes teriam de ser executados em qualquer momento. Isso implica que haja um protocolo que permita à aplicação uma monitorização constante da necessidade de execução dos mesmos. O que leva ao primeiro problema encontrado: ao fazer acesso a uma *API* segura, é necessário o uso de um *authentication token* passado nos *headers* de cada pedido *http*. Sendo que a monotorização tem de ser feita em segundo plano, verifica-se uma dificuldade de acesso ao *token* obtido previamente pelo processo de *login*. Para contornar esse problema, foi decidido que ao fazer o *refresh* automático do *token*, é também feita uma pesquisa sobre a eventual necessidade da execução de um plano de testes. Dessa forma, caso haja um plano para executar, o fio de execução já se encontra na posse do *authentication token*.

Estes testes têm a característica de terem um momento de início, que pode ser o momento da recolha do plano de testes, ou um momento mais à frente no tempo. De qualquer das formas é lançado um *Worker* (9) que irá fazer a realização do plano de testes. A dificuldade seguinte apresenta-se sob forma de dualidade. Um teste pode ser executado síncrona ou assincronamente, visto que alguns deles representam, por exemplo, repetições de sequências de *pings*. Visto que a ideia de fazer testes autónomos é avaliar o desempenho da rede, deseja-se um mínimo de interferência no fluxo da rede, para que os resultados da avaliação possam ser o mais claros possíveis. Foi então implementada uma função que irá fazer a execução de cada teste, e apenas posteriormente à sua conclusão, através de *callback* esta função é invocada uma próxima vez para executar o próximo teste. Após a conclusão de toda a bateria de testes, é enviada a informação recolhida de volta para o sistema de gestão central.

4.5 Modelo de Dados

A camada de acesso a dados é uma das partes mais importantes da aplicação móvel, na medida em que produz uma ponte para o armazenamento e consulta de dados com relativa facilidade. A *framework Room* foi desenhada sobre a linguagem *SQLite*.

Para aceder à base de dados e fazer comandos *SQL*, é necessário implementar várias classes que simulem as entidades que existiriam numa base de dados convencional. Implementados esses contentores de dados, o passo seguinte delimita-se por criar interfaces correspondentes por fazer cada tipo de comando para uma determinada entidade. Estas interfaces denominam-se *Data Access Object* (*DAO*). Vão conter várias funções, e cada função vai ser anotada com o seu comando. Estas anotações variam entre *Query*, Insert, *Update*, *Delete* e Patch. A anotação de *Query* é a mais versátil das 4, visto que esta recebe o comando *SQL* diretamente como parâmetro, tornando-se possível fazer qualquer tipo de comando (como por exemplo um *delete* por *id*).

A ponte entre a camada de negócio e a camada de acesso a dados denomina-se de *ViewModel* (17). Este é instanciado na criação de cada *View* que precisar de comunicação com a base de dados. No caso do *ViewModel* não precisar de parâmetros de entrada, uma simples chamada ao construtor é suficiente para o instanciar e fazer uso das suas funções. Em caso contrário é necessário fazer a criação de uma *Factory* (18), que permite passar os parâmetros desejados ao construtor do *ViewModel*.

Como foi mencionado anteriormente, *queries* retornam uma instância de *LiveData*. Este pedido assíncrono retorna esta instância para possibilitar à *main thread* fazer a observação do resultado de forma a ser notificada quando este chegar. À luz desse facto foi criado um *AbstractModel*, do qual todos os *models* da aplicação derivam. Este *AbstractModel* foi criado de forma que as funções utilitárias sob as instâncias de *LiveData* possam estar acessíveis por parte de todos os *models*. Foram criadas duas funções em concreto: *observe* e *observeOnce*. No primeiro caso, como o nome indica, é registada uma observação indefinida sob uma determinada *query* que notifica a *main thread* sempre que o resultado for diferente do anterior. No segundo caso, a observação é cancelada quando o primeiro resultado vier. Existe também uma outra função denominada *observeForever* que atua de forma semelhante ao *observe*. A diferença entre os dois é que o *observe* está restringido ao *lifecyle owner* (o tempo de vida da *view* que registou a observação), enquanto o outro não tem restrições quanto a isso, e tem o tempo de vida da aplicação.

5. Avaliação Experimental

Em traços gerais, a estrutura e apresentação, à exceção do esquema de cores, foi baseada e inspirada numa aplicação já existente de objetivo semelhante, denominada *NetMonitor* (19). Com esta base é possível averiguar a veracidade das recolhas efetuadas por parte da aplicação *QoS*. De outra forma seria complicado verificar se os parâmetros recolhidos apresentavam resultados com algum nível de verdade. Não é fácil fazer a verificação com toda a certeza, visto que no momento de trocar uma aplicação para plano de fundo e trazer a outra para primeiro plano algo pode mudar. Ainda assim com dois *screenshots* (figura 7), (figura 8) é possível compreender a coerência dos dados recolhidos.

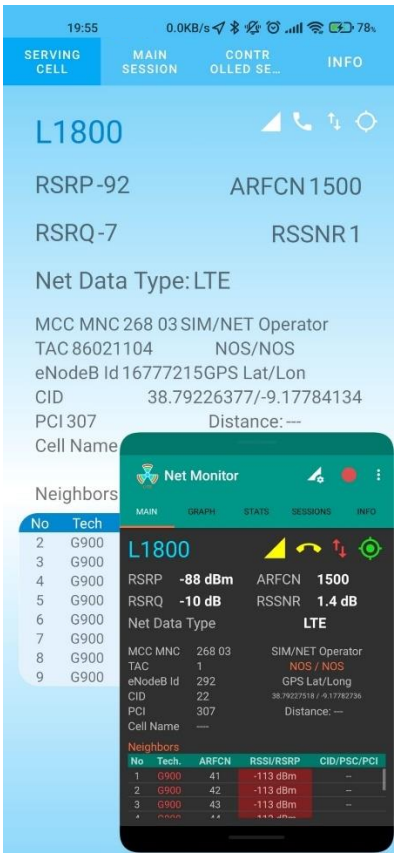

Figura 8 - Visualização da comparação de dados entre as duas aplicações.

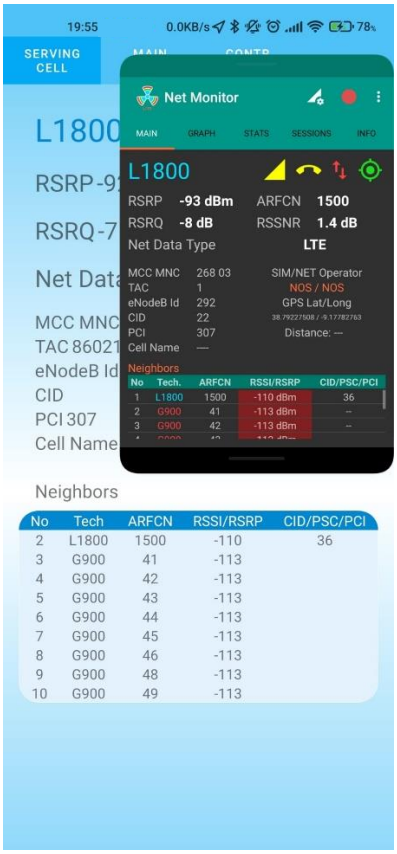

Figura 9 - Continuação da comparação.

É de notar que no curto espaço de tempo em foram tirados os *screenshots* que desapareceu a primeira linha da tabela, o que prova a dificuldade de fazer a avaliação com toda a certeza de correção.

6. Conclusão e trabalho futuro

6.1 Conclusão

O *smartphone* é cada vez mais um marco de extrema importância na vida da sociedade atual, pelo que desenvolver uma aplicação compatível com esse *gadget* se torna numa experiência inabalável e de grande relevância para os projetistas. De notar que a aplicação não está marcada ainda como terminada, nem no fim da sua linha evolutiva. Ainda há espaço para crescimento, e há certos pontos em que é certamente possível resolver alguns dos *bugs* que ou sejam produto da implementação ou produto das limitações do *Android* para a decisão de modelagem tomada. Todavia, foi possível que o grupo implementasse uma aplicação com potencial de ter bastante utilidade enquanto ferramenta de análise de rede móvel, tornando no geral este projeto uma experiência positiva.

6.2 Trabalho futuro

Face ao estado atual do projeto, é possível inferir que a aplicação ainda se encontra numa posição de possível evolução. Apesar da recolha de parâmetros estar otimizada a ser feita em paralelo com o resto das funcionalidades, e em *background*, existe espaço para aumentar o leque de abrangência de parâmetros recolhidos. É também perfeitamente possível que no futuro existam mais testes que a aplicação possa realizar, além das sequências de *ping*, visto que a aplicação se encontra preparada para fazer recolhas de novos tipos de dados, e realização de novos tipos de testes, precisando apenas da implementação da lógica desses mesmos testes, e da integração no resto da interface. Desse modo, há várias possibilidades de se poder contribuir para o melhor complemento do projeto *IQ-NPE*.

7. Referências

[1] Autoditex, "CAN BUS," [Online]. Available: https://autoditex.com/page/can-bus--controller-area-network-34-1.html. [Acedido em 5 4 2021].

[2] ETSI, "ETSI - Multi Access Edge Computing (MEC)," ETSI, [Online]. Available: https://www.etsi.org/technologies/multi-access-edge-computing. [Acedido em 5 4 2021].

[3] V. Trucks, "What are Cooperative Intelligent Transport Systems?," Volvo Trucks, [Online]. Available: https://knowledgehub.volvotrucks.com/technology-and-innovation/what-are-cooperative-intelligent-transport-systems. [Acedido em 5 4 2021].

[4] A. Developers, "Volley Overview," Android Developers, [Online]. Available: https://developer.android.com/training/volley. [Acedido em 13 4 2021].

[5] PhilJay, "PhilJay/MPAndroidChart," [Online]. Available: https://github.com/PhilJay/MPAndroidChart. [Acedido em 15 4 2021].

[6] A. Developers, "Room," Android Developers, [Online]. Available: https://developer.android.com/reference/androidx/room/package-summary. [Acedido em 17 4 2021].

[7] M. UI, "Develop - Android - Material Design," Material UI, [Online]. Available: https://material.io/develop/android. [Acedido em 19 4 2021].

[8] greenrobot, "EventBus - Events for Android," greenrobot, [Online]. Available: https://greenrobot.org/eventbus/. [Acedido em 10 5 2021].

[9] A. Developers, "Worker," Android Developers, [Online]. Available: https://developer.android.com/reference/androidx/work/Worker. [Acedido em 15 4 2021].

[10] A. Developers, "WorkManager," Android Developers, [Online]. Available: https://developer.android.com/reference/androidx/work/WorkManager. [Acedido em 15 4 2021].

[11] A. Developers, "Managing Work," Android Developers, [Online]. Available: https://developer.android.com/topic/libraries/architecture/workmanager/how-to/managing-work. [Acedido em 15 4 2021].

[12] A. Developers, "JobService," Android Developers, [Online]. Available: https://developer.android.com/reference/android/app/job/JobService. [Acedido em 2 5 2021].

[13] R. Guru, "Strategy," Refactoring Guru, [Online]. Available: https://refactoring.guru/design-patterns/strategy. [Acedido em 20 5 2021].

[14] A. Developers, “LiveData,” Android Deveolpers, [Online]. Available: https://developer.android.com/reference/android/arch/lifecycle/LiveData. [Acedido em 15 4 2021].

[15] A. Developers, “LifecycleOwner,” Android Developers, [Online]. Available: https://developer.android.com/reference/android/arch/lifecycle/LifecycleOwner. [Acedido em 15 4 2021].

[16] A. Developers, “Observer,” Android Developers, [Online]. Available: https://developer.android.com/reference/androidx/lifecycle/Observer. [Acedido em 15 4 2021].

[17] A. Developers, “ViewModel,” Android Developers, [Online]. Available: https://developer.android.com/reference/android/arch/lifecycle/ViewModel. [Acedido em 17 4 2021].

[18] A. Developers, “ViewModelProvider.Factory,” Android Developers, [Online]. Available: https://developer.android.com/reference/android/arch/lifecycle/ViewModelProvider.Factory. [Acedido em 21 4 2021].

[19] V. V, “NetMonitor,” Vitaly V, [Online]. Available: https://play.google.com/store/apps/details?id=ru.v_a_v.netmonitor. [Acedido em 4 4 2021].