UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Barbosa Barreto

Uma biblioteca Android para análise de redes de computadores

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Barbosa Barreto

Uma biblioteca Android para análise de redes de computadores

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Rodrigo Sanches Miani

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2023

Gustavo Barbosa Barreto

Uma biblioteca Android para análise de redes de computadores

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 22 de junho de 2023

Rodrigo Sanches Miani
Orientador

Paulo Henrique Ribeiro Gabriel

Renan Gonçalves Cattelan

Uberlândia, Brasil
2023

Dedico este trabalho à memória da minha querida avó, Maria de Resende Barbosa, cujo amor, apoio e inspiração foram fundamentais em minha jornada acadêmica.

Agradecimentos

Agradeço primeiramente a Deus, pela oportunidade de viver e poder experienciar momentos tão incríveis.

Aos familiares, em especial meus avós, que se dedicaram ao máximo para que eu tivesse uma boa educação e sempre estiveram ao meu lado.

A minha esposa, por estar sempre me apoiando, compartilhando comigo os momentos bons e os mais difíceis.

Ao meu filho, José Felipe Barbosa, que mesmo tão jovem, trouxe alegria e inspiração constante durante o fim da minha graduação.

Ao meu orientador, Professor Rodrigo Sanches Miani pela paciência, empatia e amor pelo ensino, sempre me apoiando e confiando no potencial do meu trabalho.

A todos os professores, pelo profissionalismo, paciência e dedicação ao ensino, possibilitando que ótimos profissionais sejam formados através desta universidade.

Resumo

O crescimento de usuários de dispositivos móveis com o sistema operacional Android traz consigo desafios significativos no que se refere a segurança. Isso se deve ao fato de que há constantes esforços dos atacantes para encontrar maneiras de obter informações pessoais, senhas bancárias e outros dados importantes. Diante desse cenário, é necessário incentivar o avanço de estudos relacionados à segurança da informação em redes de computadores. Dessa forma, o objetivo deste trabalho é disponibilizar uma biblioteca Android denominada NetScan, que ofereça um conjunto de ferramentas de rede para facilitar a criação de projetos e pesquisas para explorar aspectos relacionados a redes de computadores, contribuindo para o avanço do conhecimento nessa área. Além disso, são descritos os benefícios do uso da biblioteca NetScan por meio da implementação de uma aplicação que exemplifica o uso das suas funcionalidades. Nesta aplicação, foi desenvolvida uma funcionalidade para identificar dispositivos vulneráveis ao *malware* Mirai com o auxílio das funções providas pelo NetScan.

Palavras-chave: Android, Desenvolvimento móvel, Segurança da Informação, Monitoramento de redes.

Lista de ilustrações

Lista de abreviaturas e siglas

ANR	*Application Not Responding*
API	*Application Programming Interface*
BSSID	*Basic Service Set Identifier*
ICMP	*Internet Control Message Protocol*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
Kbps	*Kilobits* por segundo
MAC	*Media Access Control*
MVC	*Model View Controller*
MVP	*Model View Presenter*
MVVM	*Model View View Model*
POO	Programação Orientada a Objetos
SO	Sistema Operacional
SOLID	*Single-responsibility, Open-closed, Liskov Substitution, Interface Segregation, Dependency Inversion*
SSID	*Service Set Identifier*
TCP	*Transmission Control Protocol*
UML	*Unified Modeling Language*
UDP	*User Datagram Protocol*
XML	*Extensible Markup Language*

Sumário

1 Introdução

A crescente presença da tecnologia da informação leva a sociedade a transformações profundas em diversos setores. Estas transformações podem ser observadas desde a forma como as pessoas se comunicam, até a maneira como são realizadas atividades profissionais, entretenimento e interação social. Tudo isso evidencia o papel fundamental desempenhado pela tecnologia.

Segundo estudos recentes, a expectativa é que, em 2025, o número de dispositivos móveis no mundo chegue a 18 milhões (STATISTA, 2021). Ainda nessa linha, um levantamento feito pela Fundação Getúlio Vargas, mostra que no Brasil há cerca de 424 milhões de dispositivos digitais em uso (FGVCIA, 2021).

Tais números estão diretamente ligados ao crescimento do sistema operacional Android, que, segundo dados da Statista (2023), é utilizado por cerca de 71% dos usuários de dispositivos móveis no mundo. Um fator relevante para que o Android tenha toda essa base de usuários é o número de fabricantes que adotam o sistema, o que torna os aparelhos acessíveis a todos os públicos. Toda essa popularidade traz consigo uma série de benefícios, como facilidade de uso, a diversidade de aplicativos e a conectividade constante. No entanto, surgem também novos desafios, especialmente no que se diz respeito a aspectos relacionados a segurança dos usuários e das redes em que estes dispositivos estão conectados.

Dados obtidos no relatório de ameaças a dispositivos móveis de 2023 da McAfee revelam que criminosos empregam uma variedade de métodos para disseminar *malwares* entre os usuários. Isso acontece através da execução de tarefas comuns, como na edição de fotos e leitura de arquivos. Tais atividades fazem com que o usuário seja exposto a *malwares* capazes de capturar informações, corromper arquivos, comprometer o funcionamento do sistema, entre outras ações maliciosas (MCAFEE, 2023).

O Mirai é um destes *malwares* que tem o comportamento de se espalhar infectando dispositivos e formando uma rede de máquinas que são usadas para ataques de Negação de Serviço, em inglês *Distributed Denial of Service* (DDoS) (ANTONAKAKIS et al., 2017). Diante disso, estudos como os conduzidos por Costa (2018) e Silva-Junior (2019), se dedicaram em identificar dispositivos suscetíveis ao ataque do Mirai em ambientes domésticos, com o objetivo de conscientizar os usuários sobre essas vulnerabilidades. Isso foi feito através do desenvolvimento de um aplicativo Android chamado Mirai Scanner.

Nos trabalhos mencionados anteriormente, é notável a quantidade de código exigido para o desenvolvimento do aplicativo Mirai Scanner. A escrita de todo esse código consome tempo, que é gasto desde o entendimento de como construir a solução, quanto

na resolução de problemas na implementação.

Nesse contexto, surge uma necessidade de otimização, possibilitando aos pesquisadores direcionarem seus esforços para a análise dos resultados. Uma abordagem promissora para essa otimização seria a utilização de bibliotecas que oferecessem os recursos necessários para a criação do projeto, reduzindo assim a carga de trabalho relacionada à implementação dos aspectos técnicos.

Atualmente, a biblioteca desenvolvida por Rollings (2023) possui este intuito. Nela, o autor disponibilizou uma gama de ferramentas práticas para desenvolvedores, de modo a facilitar o acesso a recursos comuns utilizados em redes de computadores. Uma questão de grande relevância encontrada é a ausência de atualizações e a eventual descontinuidade do trabalho, ocasionando a obsolescência de determinados recursos e a não correção de falhas na implementação. Portanto, a ideia deste trabalho é desenvolver uma biblioteca para manipulação de funções de para Android que seja mantida ao longo do tempo e devidamente atualizada para as novas versões do referido sistema operacional. A escolha do sistema Android foi feita devido ao número de usuários e a abrangência do sistema operacional.

1.1 Objetivos

1.1.1 Objetivo geral

O objetivo geral deste trabalho é a criação de um conjunto abrangente de ferramentas de rede que sejam regularmente atualizadas, incorporando os mais recentes padrões e recursos do Android. Dentro desta biblioteca, intitulada NetScan, encontra-se um conjunto de ferramentas que pode efetuar varredura de portas, pesquisa por dispositivo na rede, varredura de redes domésticas, verificação de conexão com a Internet, pesquisa por redes sem fio e fornecer uma visão da velocidade da conexão com a Internet.

1.1.2 Objetivos específicos

Os objetivos específicos envolvem:

- Manter o repositório da NetScan aberto para contribuições.
- Disponibilização da NetScan para download pela comunidade.
- Criação de uma aplicação Android demonstrando o uso das ferramentas da NetScan.
- Usar a NetScan para replicar parte do aplicativo criado por Costa (2018) para encontrar dispositivos vulneráveis ao *malware* Mirai.

1.2 Organização do Trabalho

A estrutura adotada para organização desta monografia é feita em capítulos e segue o seguinte formato:

- No Capítulo 2, é feita uma apresentação da fundamentação teórica, a fim de que se entenda alguns dos padrões, ferramentas, técnicas e trabalhos relacionados.
- No Capítulo 3, é descrito detalhes sobre o desenvolvimento da biblioteca, diagrama de classes e informações sobre a implementação de cada uma das ferramentas.
- No Capítulo 4, é mostrado um exemplo real de aplicação utilizando a biblioteca, de modo a exemplificar o uso das ferramentas.
- Por fim, no Capítulo 5, é feita a conclusão, tendo as considerações finais e uma perspectiva das limitações da biblioteca.

2 Fundamentação teórica

Neste capítulo é feita a revisão da bibliografia acerca de tópicos importantes relacionados a redes de computadores e ferramentas de varredura de redes em dispositivos Android. Além disso, é mostrado o funcionamento dos protocolos utilizados no desenvolvimento das ferramentas, as particularidades do *framework* Android, os padrões de projeto utilizados e trabalhos relacionados.

2.1 Modelo de referência OSI

O modelo de referência OSI (*Open Systems Interconnection*, em inglês) foi proposto pela ISO (*International Standards Organization*, em inglês) e trata da interconexão de sistemas abertos, descrevendo como as redes de computadores podem ser organizadas em camadas para permitir a comunicação entre diferentes sistemas. A Figura 1 representa as sete camadas criadas.

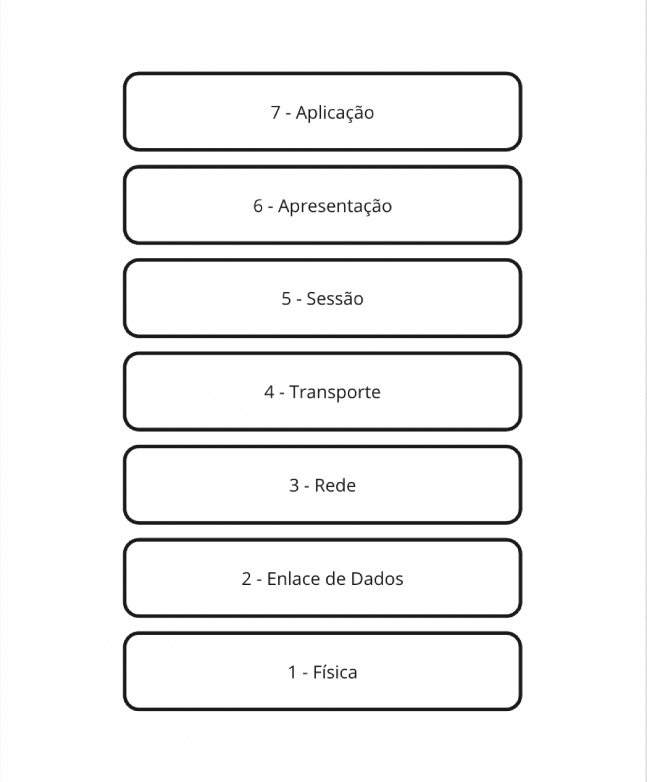

Figura 1 – Modelo de referência OSI. Fonte: Adaptado de (TANENBAUM, 2003)

Cada uma das camadas foi desenvolvida tendo como ideia a criação de abstrações e uma separação de responsabilidades bem definida (TANENBAUM, 2003). Sendo assim, serão descritas as responsabilidades de cada uma delas.

- 1 - Física: Responsável pela transmissão física dos dados através dos cabos, fibras ópticas, ondas de rádio e outros meios.
- 2 - Enlace de dados: Realiza o controle de acesso ao meio físico, garantindo a transmissão confiável dos dados entre os nós de uma rede.
- 3 - Rede: Faz o gerenciamento de rota dos pacotes na rede, fragmentação, remontagem, controle de fluxo e congestionamentos.
- 4 - Transporte: Provê meios de conexão confiáveis através do estabelecimento de conexão, segmentação de dados e confirmação de entrega das informações.
- 5 - Sessão: Responsável por gerenciar e finalizar o diálogo entre a origem e o destino da comunicação.
- 6 - Apresentação: Camada responsável por lidar com criptografia, compressão e tradução de dados para que eles sejam entendidos pelas aplicações.
- 7 - Aplicação: É a camada mais próxima do usuário, nela são abrigados os protocolos, HTTP, FTP, DNS e outros. Tem como responsabilidade fornecer uma ligação entre as aplicações e os serviços de rede.

2.2 Protocolo IP

Para que os pacotes de dados sejam trafegados através das redes de computadores, faz-se necessário um protocolo para orquestrar tal ação. Esse protocolo é chamado Protocolo IP (*Internet Protocol*, em inglês), através dele se garante que os pacotes cheguem aos seus destinos, independente dos equipamentos disponíveis.

O Protocolo IP é pertencente a camada de rede e adota um modelo de protocolo sem conexão, não estabelecendo uma conexão prévia entre os dispositivos da rede. É importante considerar que tal abordagem pode ocasionar problemas, uma vez que, pacotes de dados podem ser perdidos ou duplicados em decorrência de falhas no processo de comunicação.

Para garantir a integridade dos dados trafegados na rede, o Protocolo IP utiliza *checksums*, que são que códigos de verificação de integridade, adicionados ao cabeçalho do pacote de dados. Para cálculo desse código são considerados os dados do pacote que posteriormente podem ser validados pelo destinatário de modo a garantir que as informações não foram corrompidas.

Além disso, é adotado pelo Protocolo IP o uso de um endereço numérico para atribuir uma identificação única a cada dispositivo na rede, sendo denominado endereço IP, podendo ser classificado em dois tipos, IPv4 e IPv6 (KUROSE; ROSS, 2014).

O IPv4 é um padrão largamente adotado e o mais antigo, criado em meados de 1981, utiliza endereços de 32 bits para identificar os dispositivos na rede, o que possibilita a identificação de cerca de 4 bilhões de dispositivos. Já o IPv6 é mais recente, criado em meados de 2012, utiliza um endereço de 128 bits, o que possibilita ter um número maior de endereços disponíveis na rede.

2.3 Protocolos TCP, UDP e ICMP

A camada de transporte possui protocolos muito importantes para a comunicação de dispositivos em redes de computadores. Nesta seção são abordados três que estão diretamente ligados as ferramentas desenvolvidas neste trabalho, sendo eles: TCP (Transmission Control Protocol), UDP (User Datagram Protocol) e o ICMP (Internet Control Message Protocol).

2.3.1 Protocolo TCP

O protocolo TCP oferece uma comunicação confiável garantindo que os dados chegarão sem erros, duplicação ou perdas ao destinatário. É um protocolo orientado a conexão, ou seja, é necessário que haja uma conexão estabelecida entre o remetente e o destinatário antes de iniciar a transmissão dos dados (KUROSE; ROSS, 2014).

Para que as informações trafeguem pela rede, o protocolo divide a informação em partes menores denominadas pacotes e com isso consegue fazer todo o controle e envio das mensagens, possibilitando com isso que os pacotes sejam reenviados caso sejam corrompidos ou perdidos.

2.3.2 Protocolo UDP

O protocolo UDP é utilizado em cenários em que a garantia de entrega e integridade dos dados não são importantes. Isso ocorre, pois, o protocolo UDP é definido como sem conexão, ou seja, não se faz necessário o estabelecimento de uma conexão entre às duas pontas de troca da informação.

Dessa forma, é criada uma conexão não confiável, onde não se tem garantia da entrega das informações, ordem dos dados, duplicações, além de outros problemas possíveis (TANENBAUM, 2003). Apesar dos aspectos negativos, trata-se de um protocolo mais simples que o TCP, portanto mais rápido, sendo muito utilizado em aplicações que funcionam em tempo real, como videoconferência, jogos, transmissão de áudio, entre outros.

2.3.3 Protocolo ICMP

Analisando agora o protocolo ICMP, ele é utilizado por roteadores e dispositivos para obterem informações sobre problemas de roteamento, erros de comunicação, congestionamento, entre outros problemas em uma transmissão de pacotes na rede (TANENBAUM, 2003).

Outra aplicação bastante conhecida do protocolo é a realização do comando Ping (ou eco), onde se consegue determinar se um endereço é acessível em uma rede. Para isso, é enviado uma mensagem a um dispositivo na rede e ele retorna com uma mensagem de resposta sinalizando estar disponível e pronto para o estabelecimento de uma comunicação.

2.4 Padrões de projeto e boas práticas de desenvolvimento

Padrões de projeto são comumente utilizados durante o desenvolvimento de aplicações móveis. Segundo Lou (2016) um dos mais clássicos é o Model-View-Controller (MVC), adotado em muitos projetos, porém substituído pela comunidade de desenvolvedores por padrões mais recentes, como Model-View-Presenter (MVP) e Model-View-ViewModel (MVVM). Além destes, outros padrões também são adotados na Programação Orientada a Objetos (POO) e podem ser divididos em três classes principais, sendo elas padrões criacionais, comportamentais e estruturais (FREEMAN, 2009). Sendo assim, esta seção tem como objetivo apresentar a explicação de parte destes padrões de projeto e os princípios de *design* de *software* abordados por Martin (2008) através do acrônimo SOLID.

2.4.1 Padrão MVP

No padrão MVP são encontrados três componentes, *Model*, *View* e *Presenter*. A *View* possui a responsabilidade de receber e notificar o usuário quando há interações com a tela. Por outro lado, o *Model* é a classe responsável pelos dados usados para apresentação das informações. Adicionalmente, o *Presenter* faz o intermédio da comunicação entre a interação do usuário através da *View* e o armazenamento e processamento dos dados no *Model* (LOU, 2016). Dessa forma, é apresentado na Figura 2 o diagrama UML (Unified Modeling Language, em inglês) do padrão MVP e como são as dependências entre as classes.

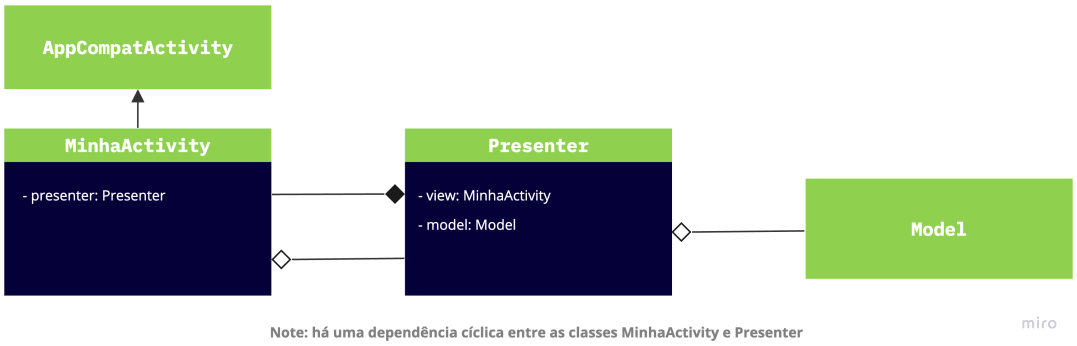

Figura 2 – Diagrama de classes do padrão MVP. Fonte: Adaptado de (LOU, 2016)

Na Figura 2, que ilustra o diagrama UML, é possível notar que existe uma dependência da classe **MinhaActivity** com a classe **Presenter**, esta dependência faz-se necessária para que haja a comunicação das interações do usuário com as demais camadas do padrão. Nessa linha, o **Presenter** possui uma referência à classe **MinhaActivity** e à classe **Model**, estas referências são utilizadas para que a tela seja notificada sobre possíveis atualizações e os dados sejam armazenados e processados na camada de modelo. Além dos relacionamentos apresentados é importante observar que para este padrão existe uma dependência cíclica entre a *Activity* e o *Presenter*.

Algumas derivações do diagrama acima podem existir, como por exemplo o uso de interfaces denominadas 'Contratos', que visam diminuir o acoplamento com classes concretas, seguindo bons padrões de códificação (MARTIN, 2008).

2.4.2 Padrão MVVM

O padrão MVVM possui como componentes o *Model*, a *View* e o *ViewModel*. Tanto o *Model* quanto a *View* possuem as mesmas responsabilidades apresentadas no padrão MVP, a diferença desse padrão se dá no *ViewModel* (LOU, 2016). Enquanto o *Presenter* possui uma referência direta da *View* e usa esta referência para notificar as mudanças que devem acontecer na interface do usuário, do inglês *User Interface* (UI), o *ViewModel* funciona de forma reativa. Dessa forma, o *ViewModel* mantém propriedades observáveis que representarão os dados do modelo, possibilitando assim, que em caso de mudanças no *Model*, a *View* seja notificada e consequentemente a UI atualizada. Na Figura 3 é apresentado o diagrama UML do padrão MVVM.

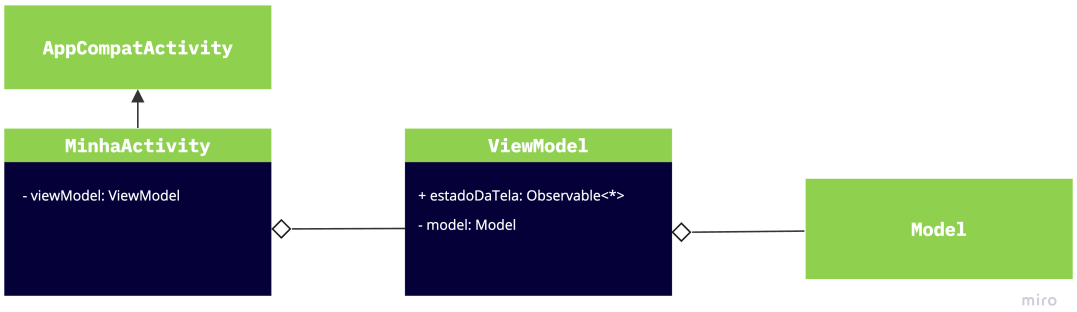

Figura 3 – Diagrama de classes do padrão MVVM. Fonte: Adaptado de (LOU, 2016)

Na Figura 3, é possível notar que a classe **MinhaActivity** possui uma referência da classe **ViewModel**, porém o **ViewModel** não referencia a Activity, removendo a dependência cíclica existente no padrão MVP. Como já mencionado, toda a notificação do **ViewModel** para a classe **MinhaActivity** é feita por meio de objetos observáveis. Adicionalmente, o **ViewModel** possui a referência da classe **Model**, seguindo o mesmo comportamento do MVP.

Atualmente, o Google incentiva os desenvolvedores a adotarem o MVVM como padrão oficial para o desenvolvimento de aplicações Android. É possível identificar tal incentivo através da biblioteca de *Lifecycle* presente no JetPack[1], nela está presente o *ViewModel* (GOOGLE, 2023c), classe que já está pronta para uso nas aplicações e que possui diversos benefícios.

Na documentação oficial, é possível notar que um dos principais benefícios do uso do *ViewModel* é o desenvolvedor delegar ao *ViewModel* a tarefa de conhecimento do ciclo de vida das *Activities* e *Fragments*. O salvamento dos dados também é feito pelo *ViewModel*, possibilitando armazenar o estado da interface do usuário (GOOGLE, 2023h). Tudo isso pode ser feito, pois o *ViewModel* tem o conhecimento do ciclo de vida das *Activities* e *Fragments*, conseguindo então, identificar quando os componentes estão sendo destruídos pelo Android, auxiliando no desenvolvimento e minimizando a possibilidade de erros por parte do desenvolvedor (GOOGLE, 2023h).

2.4.3 Padrão Observer

O *Observer* é um padrão de projetos pertencente ao conjunto de padrões comportamentais (FREEMAN, 2009). É muito útil quando existe dependência entre objetos e há a necessidade de que alguns objetos conheçam as mudanças de estado de um outro objeto, de modo que, atualizem seus comportamentos. Este padrão é utilizado com frequência em

[1] JetPack é um conjunto de bibliotecas desenvolvido e mantido pelo Google. Estas bibliotecas tem como objetivo ajudar os desenvolvedores a seguirem boas práticas de programação, reduzir a quantidade de código necessário para desenvolvimento das funcionalidades, oferecer suporte às diversas versões do Android, entre outros.

interfaces gráficas de usuário, onde a mudança de estado de um objeto requer uma atualização de informações visuais na tela. Um exemplo de uso do padrão *Observer* se dá no padrão MVVM citado anteriormente.(LOU, 2016). A Figura 4, ilustra o diagrama de classes que descreve o funcionamento do padrão.

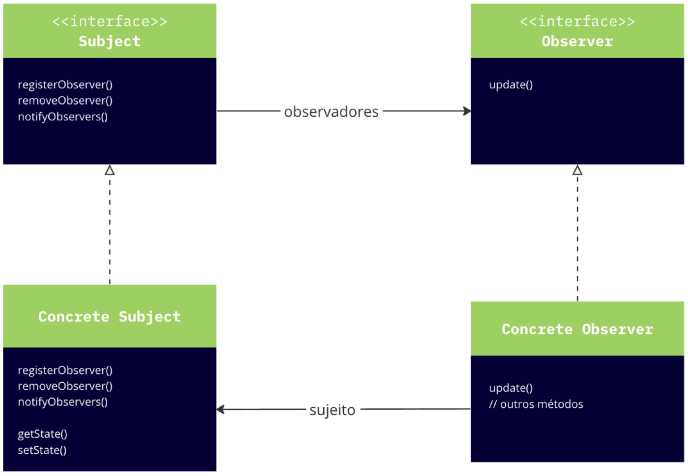

Figura 4 – Padrão Observer. Fonte: Adaptado de (FREEMAN, 2009)

É possível notar na Figura 4, que a interface **Subject** é o sujeito a ser observado e a interface **Observer** representa o observador. Dessa forma, sempre que houver uma atualização do **Subject**, ele notificará os observadores.

2.4.4 Padrão Strategy

O padrão *Strategy* também é considerado um padrão comportamental, e busca uma forma de encapsular uma família de algoritmos em classes distintas que implementarão uma interface comum (GAMMA et al., 1995). Nesta interface serão definidos os métodos que serão implementados pelas classes concretas, possibilitando assim que algoritmos diferentes sejam executados e com isso comportamentos diferentes aconteçam. Sendo assim, as implementações poderão ser facilmente intercambiáveis, o que facilita a adição de novos algoritmos e deixa o código flexível às mudanças (GAMMA et al., 1995). Na Figura 5, é apresentado o diagrama que contém detalhes da implementação desse padrão, tendo como exemplo um visualizador de texto.

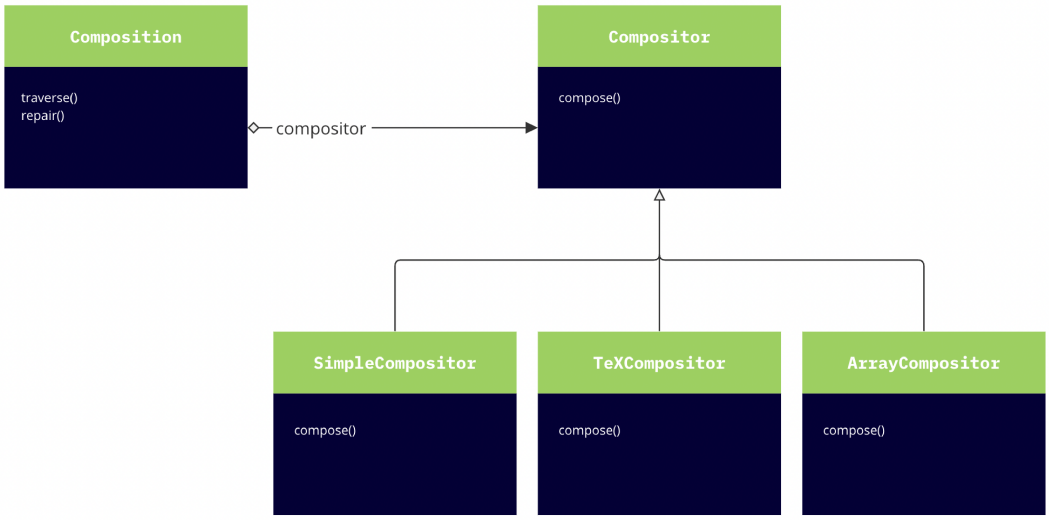

Figura 5 – Padrão Strategy. Fonte: Adaptado de (GAMMA et al., 1995)

Neste exemplo, é possível notar que há diversas formas para se quebrar linha em uma ferramenta de visualização de texto, para isso, foi criada uma interface denominada **Compositor** e por meio dela se tem diversas implementações, cada uma com sua responsabilidade.

- **SimpleCompositor**: implementa uma estratégia simples que determina quebras de linha uma de cada vez.
- **TeXCompositor**: implementa o algoritmo TeX para encontrar quebras de linha. Esta estratégia tenta otimizar as quebras de linha globalmente, ou seja, um parágrafo de cada vez.
- **ArrayCompositor**: implementa uma estratégia que seleciona quebras, para que cada linha tenha um número fixo de itens. É útil para quebrar uma coleção de ícones em linhas.

2.4.5 Padrão Facade

Segundo Freeman (2009), o padrão *Facade* ou Fachada em português, é um padrão estrutural, que visa simplificar operações complexas de subsistemas, fornecendo uma interface unica para os clientes. É utilizada a nomenclatura fachada, pois ele esconde a complexidade dos subsistemas adjacentes, com isso reduzindo a complexidade de uso de determinadas funcionalidades, fazendo com que o cliente dependa somente dessa interface unificada. A Figura 6, demonstra um cenário de aplicação do padrão.

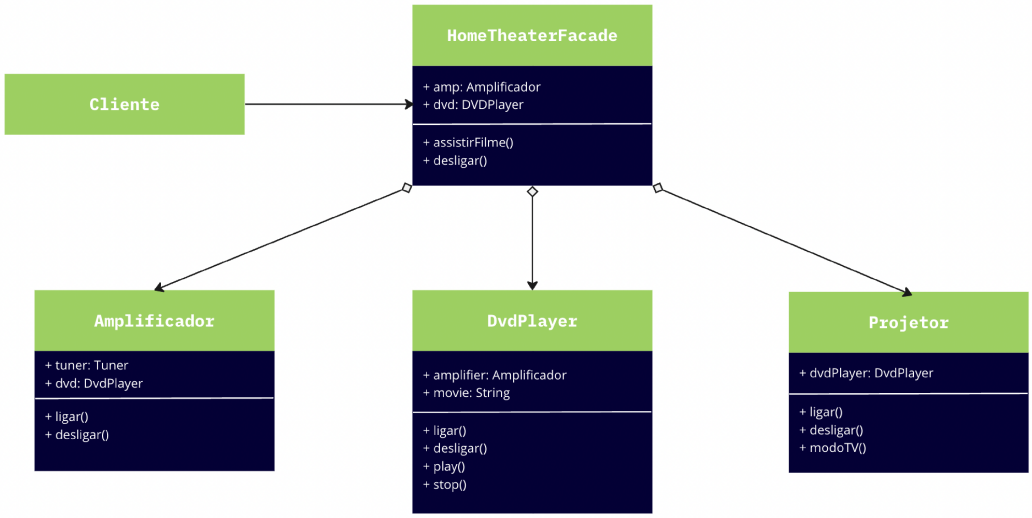

Figura 6 – Padrão Facade. Fonte: Adaptado de (FREEMAN, 2009)

A partir do exemplo, é possível notar que a classe **HomeTheaterFacade** foi projetada de modo a simular um aparelho de *Home Teather*. Nele, são feitas as comunicações com os subsistemas **Amplificador**, **DvdPlayer** e **Projetor**, facilitando assim que o **Cliente** consiga realizar a tarefa de assistir um filme.

2.4.6 Princípios do SOLID

SOLID é um acrônimo para alguns princípios de Programação Orientada a Objetos (POO), que visam criar um código mais limpo, flexível e de fácil manutenibilidade (MARTIN, 2008). Abaixo há uma breve explicação de cada um dos princípios.

- *Single Responsibility Principle* (Princípio da Responsabilidade Única): Uma classe deve possuir apenas uma razão para mudar, ou seja, apenas uma responsabilidade.
- *Open/Closed Principle* (Princípio do Aberto/Fechado): Uma classe deve ser aberta para extensão e fechada para modificações. Em outras palavras, você pode estender os comportamentos de uma classe, mas sem modificar seu código.
- *Liskov Substitution Principle* (Princípio da Substituição de Liskov): Subtipos devem ser substituíveis pelos seus tipos base. Isso quer dizer que você deve ser capaz de substituir uma instância de uma classe por outra instância da classe base, sem alterar o comportamento do *software*.
- *Interface Segregation Principle* (Princípio da Segregação de Interfaces): Interfaces devem ser segregadas em unidades menores e mais específicas. Em outras palavras, você não deve forçar uma classe a implementar métodos que não usa.

- *Dependency Inversion Principle* (Princípio da Inversão de Dependência): Dependa de abstrações, não de implementações concretas. Isso consiste em não depender de classes de baixo nível e sim de abstrações.

Todos esses princípios auxiliam os desenvolvedores a escrever códigos mais limpos, organizados e fáceis de manter. Outros grandes benefícios são a promoção do reuso de código e a facilidade na expansão do software (MARTIN, 2008).

2.5 Tecnologias

2.5.1 Android

O Android é um Sistema Operacional (SO) de código aberto baseado no SO Linux (GOOGLE, 2023b). Foi criado pela empresa Android Inc. em meados de 2003 e adquirido pela empresa Google Inc. em 2005, sendo hoje desenvolvido e mantido pelo Google. A arquitetura do Android é bem segmentada e conta com diversas camadas que compõem a pilha de software do SO. A Figura 7, apresenta a divisão destas camadas.

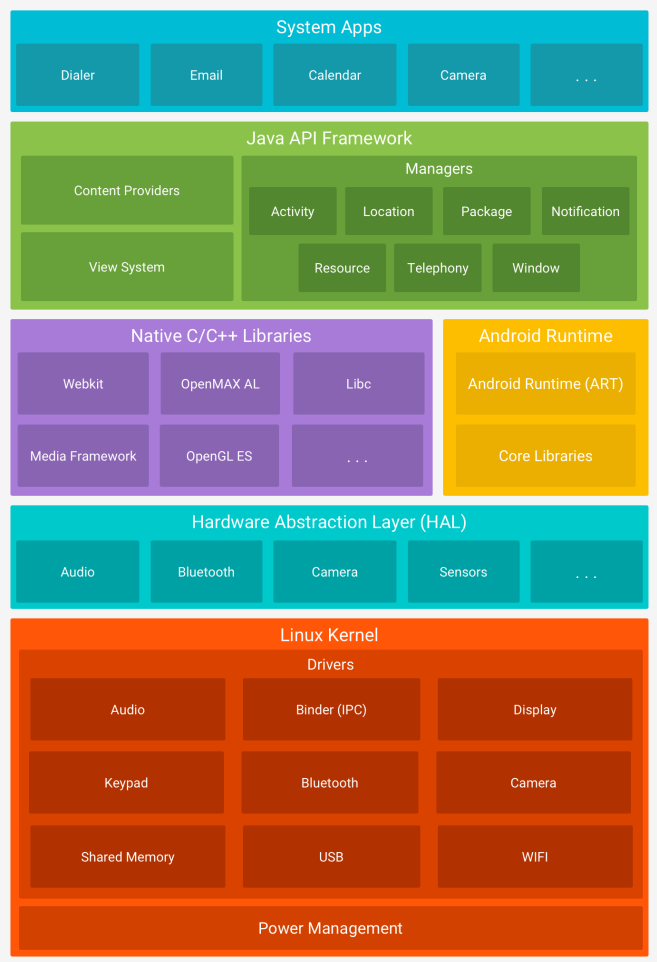

Figura 7 – Pilha de software Android. Fonte: Extraído de (GOOGLE, 2023d)

Na Figura 7, é possível notar que uma das camadas presentes na pilha de *software* é a 'Java API Framework'. Esta API (*Application Programming Interface*, em inglês), tem por objetivo prover uma abstração que facilite que os desenvolvedores possam criar suas aplicações de forma simplificada, não sendo necessário o conhecimento de componentes internos do *framework*.

Dentre as APIs que o Android provê, algumas estão relacionadas a interface do usuário, permissões, conectividade, armazenamento, mídia, notificações, entre outras. No *framework*, o SO e as APIs estão em constante atualização por parte do Google e da comunidade, uma vez que novos recursos surgem e vulnerabilidades são encontradas, necessitando assim da implementação de melhorias.

Atualmente o Android encontra-se na versão oficial 13, acompanhada pela API/SDK (Software Development Kit, em inglês) na versão 33, o que engloba uma variedade de melhorias, como desempenho, privacidade e segurança dos usuários. No entanto, a versão 14 do sistema operacional já está disponível para os desenvolvedores. Na Figura 8, é possível notar a distribuição do uso do Sistema Android por parte dos usuários, bem como os nomes e versões da SDK.

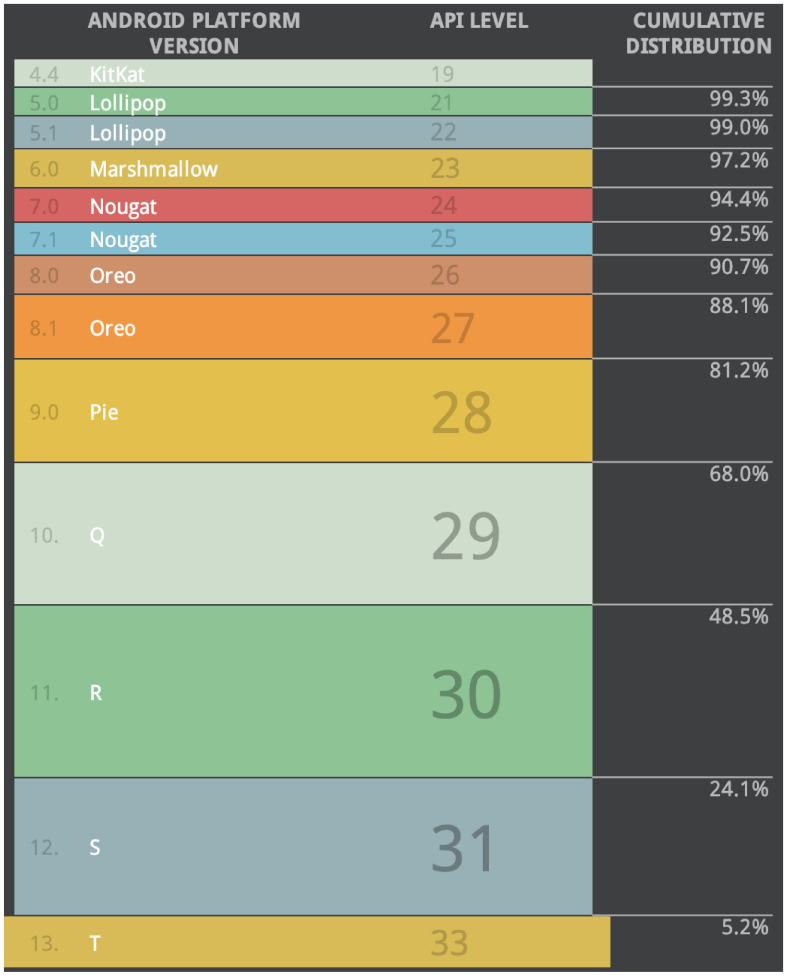

Figura 8 – Distribuição de versões Android Platform/API. Fonte: Extraído de (GOOGLE, 2023e)

Mesmo com as APIs apresentadas, ainda assim há uma certa complexidade para implementar tarefas simples. Um exemplo é o armazenamento de informações na base de dados do Android. Apesar de existirem abstrações que facilitem a implementação, permanece sendo uma tarefa complexa.

Tendo em vista tais dificuldades, o Google criou um conjunto de bibliotecas denominada Jetpack. Estas bibliotecas têm como objetivo facilitar o desenvolvimento de funcionalidades como a descrita anteriormente. Dentro do conjunto de bibliotecas do Jetpack é possível encontrar ferramentas que ajudam no controle da navegação entre telas (*Navigation*), paginação de dados (*Paging*), armazenamento de informações em banco de dados (*Room*), *LifeCycle* para conhecer o ciclo de vida dos componentes do Android, dentre outras que podem ser vistas visitando o site oficial[2].

[2] Disponível em: https://developer.android.com/jetpack

2.5.2 Ambiente de desenvolvimento Android Studio

O Android Studio é a ferramenta oficial de desenvolvimento para criação de aplicações Android. Esta ferramenta consiste em um ambiente de desenvolvimento integrado, também conhecida como IDE (*Integrated Development Environment*, em inglês). Por meio dela, é possível criar aplicativos de forma eficiente, contando com diversos recursos que auxiliam no processo de criação da aplicação (GOOGLE, 2023f). Algumas das ferramentas disponíveis nesse ambiente são:

- Depurador integrado.
- Editor visual para criação de interfaces.
- Análise de código para identificação de problemas de desempenho e erros.
- Plugins que facilitam o uso de padrões como MVP, MVVM, entre outros.
- Emulador de dispositivos Android.

Além das ferramentas citadas, o Android Studio possui suporte para as principais linguagens de programação utilizadas no desenvolvimento Android, sendo elas Java, Kotlin e C++.

2.5.3 Linguagem Kotlin

A linguagem de programação Kotlin foi introduzida pela JetBrains no ano de 2011 e, posteriormente, adotada pelo Google em 2017 como linguagem de programação oficial para o desenvolvimento de aplicativos Android. Foi projetada de modo a reduzir erros comuns de desenvolvimento e ser integrada de forma simples a aplicações já existentes. Entre os principais aprimoramentos trazidos pelo Kotlin está a correção de erros de segurança de tipos.

A introdução de recursos como atributos com tipos nulos e não nulos auxiliou os desenvolvedores a evitarem erros comuns que ocorrem em tempo de execução, como o **NullPointerException**[3]. Outro aspecto importante é a interoperabilidade do Kotlin com o código Java, o que permite a migração gradual de projetos para a nova linguagem (GOOGLE, 2023a).

2.6 Trabalhos correlatos

Nesta seção foram analisados os trabalhos correlatos a este projeto. Foi observado no trabalho de Rollings (2023) que o autor criou uma biblioteca contendo ferramentas de

[3] NullPointerException é uma exceção de tempo de execução que ocorre quando um programa tenta acessar um objeto não inicializado. É comum em linguagens orientadas a objetos como Java.

redes para o desenvolvimento de aplicações em dispositivos Android, de forma similar a NetScan. Em outros, como os de Silva-Junior (2019) e Costa (2018), os autores criaram aplicativos Android que fazem o uso de ferramentas de rede para encontrarem vulnerabilidades em dispositivos conectados em redes domésticas. Além desses trabalhos, são analisados projetos como os de Aubort (2023) e Wood (2023), que possuem ferramentas de rede em repositórios de código aberto, porém, não as exporta em formato de biblioteca.

Em Rollings (2023) é possível notar que o autor desenvolveu uma biblioteca contendo uma série de ferramentas, como: varredura de portas, descoberta de dispositivos na sub rede local, implementação do comando Ping, entre outras. Na data de acesso ao repositório hospedado no GitHub, a biblioteca contava com mais de mil e trezentas estrelas[4] e duzentos e setenta e quatro *forks*[5]. Dessa forma, é possível notar a aceitação e necessidade da comunidade por trabalhos como este. Apesar da biblioteca possuir muitas funcionalidades e uma boa adesão por parte da comunidade, o repositório não recebe atualizações desde 2021, e o responsável pelo projeto declarou na página principal do repositório e em resposta a alguns problemas abertos por usuários, que não irá mais atualizar a biblioteca.

Além da biblioteca citada, é possível encontrar projetos *Open Source*[6] como os de Wood (2023) e Aubort (2023), que também possuem ferramentas de rede capazes de fazer descoberta de IP público, varredura de portas, descoberta de dispositivos na sub rede local, busca por redes *Wi-Fi*, entre outras. Um aspecto importante a ser ressaltado, é que estes trabalhos não exportam as ferramentas em formato de biblioteca. Dessa forma, desenvolvedores se baseiam no código-fonte do projeto, porém precisam efetuar eventuais correções e evoluções, não compartilhando as melhorias com os demais.

Pesquisas também são feitas de modo a entender o comportamento de redes de computadores e até mesmo detectar anomalias e vulnerabilidades utilizando dispositivos Android. Exemplos disso, são os trabalhos de Silva-Junior (2019) e Costa (2018), onde foram desenvolvidos aplicativos para auxiliarem na descoberta de dispositivos vulneráveis ao Malware Mirai[7] em redes domésticas. Um fator que poderia ser melhorado nesses trabalhos é a diminuição do esforço necessário para desenvolver as ferramentas.

Sendo assim, o intuito deste trabalho é a criação de uma biblioteca *Open Source* que esteja atualizada com os novos requisitos do Google para as versões mais recentes do SO Android. Além disso, foram utilizadas boas práticas de desenvolvimento, de modo que

[4] No contexto dos repositórios do GitHub, o uso de estrelas se trata de uma funcionalidade que permite aos usuários favoritar repositórios. As estrelas servem como uma forma de recomendação, permitindo que os usuários identifiquem os projetos que consideram relevantes ou que desejam monitorar.

[5] Um 'fork' em um repositório Git, refere-se a uma cópia do repositório original, possibilitando o usuário a fazer modificações no código sem impacto no código original.

[6] Um projeto Open Source é aquele onde um grupo desenvolve um aplicativo ou sistema de forma gratuita e colaborativa, disponibilizando o código-fonte para que a comunidade possa visualizá-lo e contribuir com sua evolução.

[7] Mirai é um tipo de botnet de código aberto que infecta dispositivos da Internet das Coisas (IoT) através de credenciais padrão e vulnerabilidades conhecidas

a NetScan possa ser evoluída por outros desenvolvedores. Outro aspecto importante é conseguir prover formas simplificadas de implementação, sem a necessidade, por exemplo, do controle de *threads*[8], possibilitando com isso, que desenvolvedores com menos experiência consigam criar suas aplicações sem dificuldades.

Com a adoção dessas soluções, abre-se espaço para o desenvolvimento de projetos com mais agilidade, nos quais os autores poderão concentrar seus esforços na análise dos resultados obtidos. Nessa linha, menos ênfase será dada na implementação e na resolução de problemas de programação, permitindo uma abordagem mais focada e produtiva. De modo a demonstrar tais benefícios, foi desenvolvido um aplicativo que faz o uso das ferramentas presentes na NetScan. Um dos exemplos de aplicação das ferramentas foi feito por meio da implementação de uma funcionalidade que efetua uma varredura na rede doméstica e busca identificar um conjunto de portas abertas, de modo a detectar dispositivos vulneráveis ao *malware* Mirai, como nos trabalhos de Costa (2018) e Silva-Junior (2019).

[8] Uma *thread* é a menor unidade de processamento que pode ser programada em um sistema operacional (STALLINGS, 2018).

3 Desenvolvimento

Neste capítulo é abordado como a biblioteca foi desenvolvida. Na Seção 3.1 são apresentados os principais motivadores que impulsionaram a concepção deste trabalho. Em seguida, na Seção 3.2, é fornecida uma visão abrangente, acompanhada de um diagrama de classes que ilustra as interações entre os diversos componentes da biblioteca. Adicionalmente, na Seção 3.3, são discutidos os padrões adotados para a criação e estruturação do projeto, destacando sua importância na garantia da qualidade e manutenibilidade do código. Por fim, na Seção 3.4, são detalhadas as funcionalidades presentes no trabalho, bem como suas particularidades de implementação.

3.1 Biblioteca NetScan

A biblioteca NetScan oferece um conjunto de ferramentas de rede que permitirá aos desenvolvedores criarem aplicações de forma simplificada, eliminando a necessidade de se preocupar com detalhes complexos de implementação. Como descrito no Capítulo 2, foram conduzidas análises em algumas bibliotecas que apresentam características como as deste trabalho, com o intuito de compreender suas vantagens e áreas passíveis de aprimoramento.

Posteriormente à análise das bibliotecas, foi realizado um estudo de modo a identificar as melhores práticas de desenvolvimento e determinar quais ferramentas seriam pertinentes aos desenvolvedores. Nesse sentido, foram considerados os projetos mencionados e ferramentas comumente utilizadas no contexto de redes de computadores. Dessa forma, foram selecionadas as seguintes funcionalidades que são discutidas em detalhes na Seção 3.4.

- Varredura de portas
- Pesquisa por dispositivo na rede (Ping).
- Varredura em busca de dispositivos em redes domésticas.
- Verificação de conexão com a Internet.
- Lista de redes sem fio próximas.
- Velocidade da conexão com a Internet.

3.2 Visão geral

Nesta seção são detalhadas as interdependências entre as classes presentes na biblioteca, por meio da apresentação de um diagrama de classes. Esse diagrama proporcionará uma compreensão clara da estrutura do projeto e do funcionamento interno, permitindo então, uma análise aprofundada dos elementos envolvidos.

A representação do diagrama é direcionada às principais dependências presentes na NetScan, é possível obter mais informações acessando o código fonte, por meio do repositório do projeto hospedado no GitHub[1]. Além disso, algumas abordagens foram adotadas para representar funcionalidades específicas da linguagem Kotlin, as quais são expressas de maneiras distintas em outras linguagens, como o Java.

Um exemplo desse cenário, é a presença do objeto chamado *companion object* na linguagem Kotlin. Ao realizar a descompilação do *bytecode* gerado a partir do código em Kotlin para código Java, torna-se evidente que essa funcionalidade específica da linguagem Kotlin corresponde a uma classe estática na linguagem Java. Nas Figuras 9 e 10 estão representados os relacionamentos entre as classes da biblioteca, a imagem completa está disponível no repositório do GitHub[2].

Na porção superior da Figura 9, é perceptível a representação das dependências entre as principais classes da biblioteca, como a interface **NetScan**, sua implementação **NetScanImpl** e as classes que correspondem ao padrão *Factory*, adotado para prover os objetos utilizados na biblioteca. Na parte inferior da Figura 9, são apresentadas as interfaces associadas a cada uma das funcionalidades, suas implementações e as respectivas dependências. Informações mais abrangentes acerca dos padrões utilizados são exploradas na Seção 3.3.

Adicionalmente, na Figura 10, é possível notar na parte superior da imagem a representação do padrão *Observer*, utilizado na implementação dos métodos assíncronos. Do mesmo modo, na porção inferior, estão representadas as interfaces das demais funcionalidades e suas particularidades de relacionamento e implementação.

3.3 Padrões e boas práticas

Nesta seção são explorados em detalhes os padrões de projeto aplicados no desenvolvimento da biblioteca NetScan, visando garantir a facilidade de manutenção do código e a sua preparação para expansões futuras.

[1] Disponível em: https://github.com/gustavobarbosab/netscan
[2] Disponível em: https://github.com/gustavobarbosab/netscan/blob/main/docs/UML-V1.0.pdf

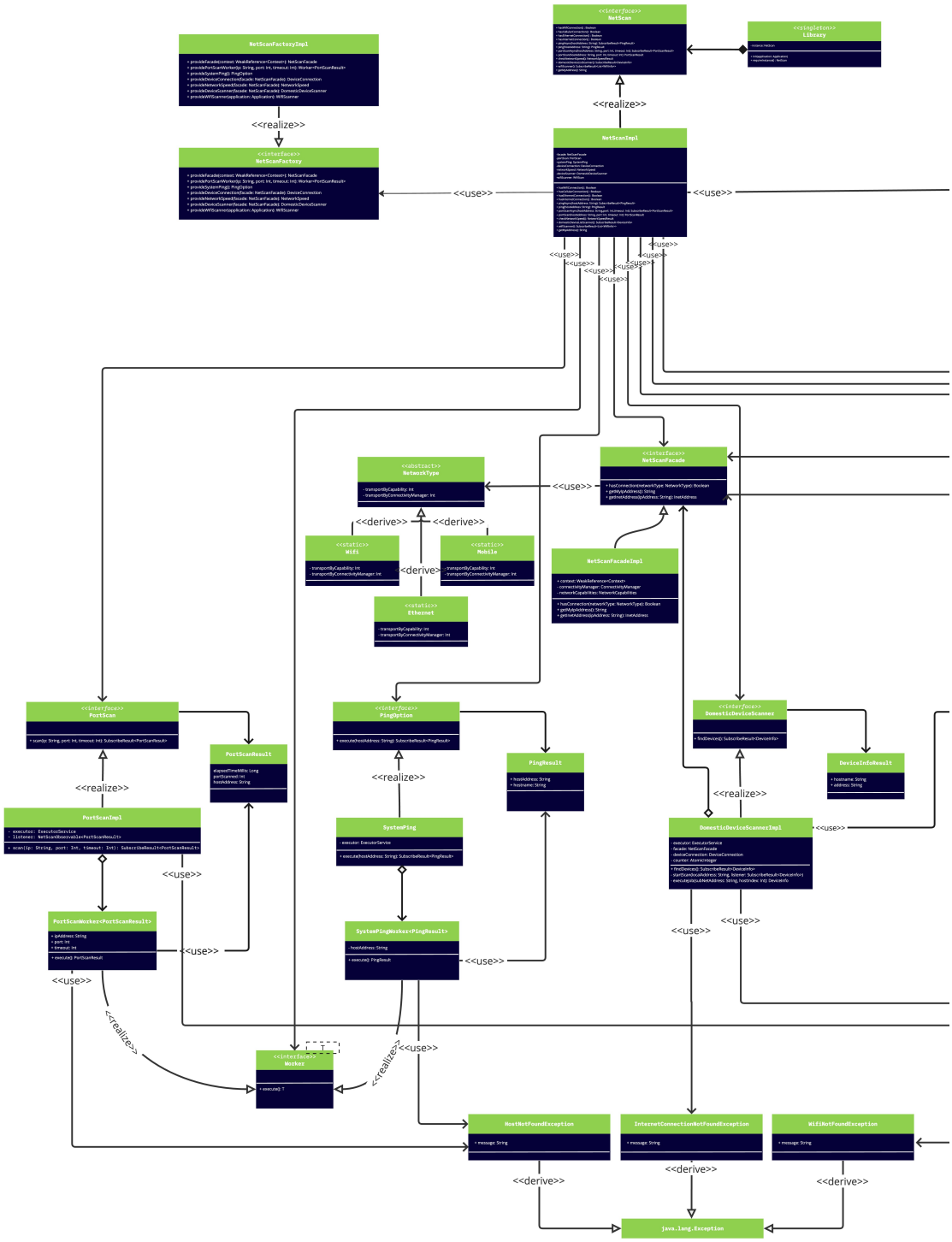

Figura 9 – Primeira parte do diagrama de classes da biblioteca

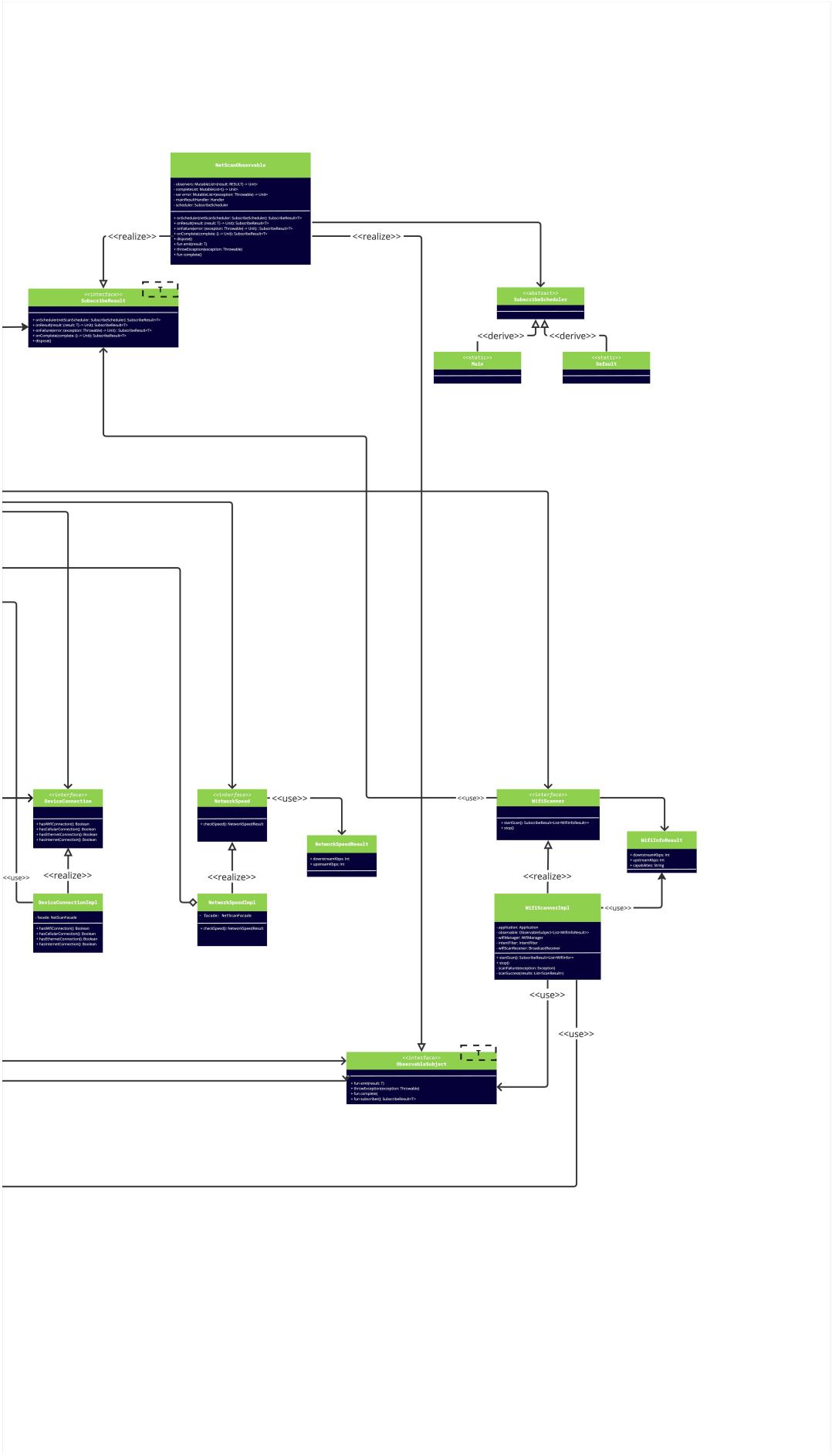

Figura 10 – Segunda parte do diagrama de classes da biblioteca

3.3.1 Singleton

Para se recuperar uma instância da biblioteca, foi adotado o padrão *Singleton*. Tal escolha foi feita, uma vez que, para ter acesso a informações globais da aplicação, faz-se necessário o conhecimento da classe **Context** do Android. Por meio desta classe, é possível acessar funcionalidades como os **Broadcasts**, **Services** e outros componentes do SO, necessários para o funcionamento das ferramentas.

É imprescindível exercer cautela ao utilizar a classe **Context** em um contexto específico. Isso se deve ao fato de que, ao associar um determinado contexto da aplicação, como uma *Activity* a uma tarefa prolongada, caso o Android tome a decisão de encerrar a *Activity* por algum motivo, a menos que a tarefa em execução seja interrompida e o contexto seja liberado, corre-se o risco de ocorrer um vazamento de memória (GOOGLE, 2021b).

Este tipo de problema acontece, pois, a atividade não será coletada pelo coletor de lixo (Garbage Collector, em inglês), uma vez que, é referenciada por uma tarefa em execução, mesmo depois de ter sido destruída pelo sistema operacional. Sendo assim, a biblioteca NetScan é inicializada através de um Singleton, recebendo como contexto a **Application**, desse modo o desenvolvedor não precisará se responsabilizar pelo controle de possíveis vazamentos durante o uso da biblioteca.

3.3.2 Factory

O uso do padrão *Factory* foi adotado de modo a centralizar a responsabilidade de criação das classes necessárias para o uso das ferramentas presentes da biblioteca. Com tal prática, é possível que a classe **NetScanImpl** tenha referência somente às interfaces presentes na biblioteca, fazendo com que sejam aplicados princípios como da Inversão de Dependência, Substituição de Liskov e Responsabilidade única (MARTIN, 2008).

3.3.3 Observable

O padrão *Observable* foi adotado de modo a facilitar o trabalho com atividades assíncronas, tornando a implementação mais simples por parte do desenvolvedor. Um dos grandes benefícios do uso deste padrão é delegar à biblioteca a criação e gerenciamento das *threads* usadas para execução das funcionalidades. Tal controle de *threads* é de extrema importância, pois pode gerar exceções do tipo *Application Not Responding (ANR)*[3], congelando a tela do usuário e gerando comportamentos indesejados na aplicação.

[3] Disponível em: https://developer.android.com/topic/performance/vitals/anr

3.3.4 Strategy

Outro padrão empregado é o *Strategy*, que permite que classes concretas realizem tarefas sem que as demais classes dependam diretamente delas. Nos diagramas representados nas Figuras 9 e 10 é possível observar as classes **SubscribeScheduler**, **PingOption**, **NetworkType** e **Worker**. Estas classes permitem a dependência com as abstrações, enquanto diferentes códigos podem ser implementados. Desse modo, o padrão está diretamente ligado a princípios como os de responsabilidade única e inversão de dependências, seguindo as boas práticas estabelecidas no SOLID (MARTIN, 2008).

3.3.5 Facade

O *Facade* foi utilizado com o objetivo de abstrair a complexidade de acesso a certos recursos do Android, simplificando essa ação para as classes que necessitam deles. Um exemplo dos benefícios desse padrão pode ser observado na classe **DeviceConnectionImpl**. Nesta classe, é necessário obter informações sobre a conectividade com a Internet no dispositivo. Nessa linha, o *Facade* é responsável por conhecer quais classes do sistema devem ser utilizadas para detectar esses comportamentos e encapsula toda a lógica de identificação das APIs do Android, permitindo determinar quais métodos devem ser invocados.

Esta abordagem garante a adesão a alguns princípios fundamentais do SOLID, como o de responsabilidade única. Isso ocorre, pois, ao delegar ao *Facade* o conhecimento dessas regras, evita-se a dispersão de responsabilidades em outras classes. Além disso, essa prática também se alinha ao princípio de inversão de dependência e ao princípio de substituição de Liskov, uma vez que a dependência é estabelecida em relação a uma interface, proporcionando maior flexibilidade e modularidade ao sistema (MARTIN, 2008).

3.4 Funcionalidades

Nesta seção são descritas as particularidades de implementação de cada uma das ferramentas presentes na biblioteca NetScan.

3.4.1 Varredura de portas

Na referida funcionalidade, foi utilizada a estratégia de criar uma conexão *socket* com o dispositivo de destino, de modo a entender se a porta está ou não aberta. Um *socket*, pode ser definido como uma interface entre um processo da camada de aplicação e o protocolo da camada de transporte (KUROSE; ROSS, 2014). Neste caso, o aplicativo criado é o processo presente na camada de aplicação e o protocolo TCP, existente na ca-

mada de transporte, é o responsável por garantir que as mensagens chegarão corretamente ao destinatário.

3.4.1.1 Método assíncrono

De modo a prevenir exceções ANR (GOOGLE, 2021a), é disponibilizado na biblioteca o método descrito abaixo, que encapsula todo o trabalho de criar uma *thread* e notificar as respostas na *thread* principal. Tal prática torna o processo de execução assíncrono. É importante ressaltar que, a *thread* principal, também conhecida como *Main Thread*, possui como uma das responsabilidades, a atualização dos recursos visuais da aplicação, sendo necessário atenção no seu uso.

Listing 3.1 – Definição do método de varredura de portas assíncrono

```
fun portScanAsync(
    hostAddress: String,
    port: Int,
    timeout: Int
): NetScanObservable<PortScanResult>
```

Este método possui como entrada três informações que são base para que a conexão seja estabelecida.

- *hostAddress* - Endereço IPv4 do dispositivo a ser encontrado.
- *port* - Porta do dispositivo a ser encontrado.
- *timeout* - Tempo limite para que haja uma resposta em milissegundos.

No Código 3.2 é exemplificado o uso do método **portScanAsync**, com o objetivo de buscar pela porta 80 no endereço '10.1.1.23'. Nota-se que, como retorno da chamada do método se terá um objeto observável, que é notificado através dos *listeners* de sucesso e falha, assim que houver uma resposta.

Listing 3.2 – Exemplo de uso do método de varredura de portas assíncrono

```
netScan.portScanAsync(
            hostAddress = "10.1.1.23",
            port = 80,
            timeout = 4000
        ).onResult { result ->
            Log.i("This is a value sent", result.toString())
        }.onFailure { exception ->
            Log.i("This is an exception", exception.toString())
```

```
        }
```

3.4.1.2 Método síncrono

Está disponível também um método síncrono, de modo a tornar a biblioteca adaptável a diversos fins. Dessa forma, fica sob responsabilidade do desenvolvedor a tarefa de criar uma *thread* para execução do trabalho e comunicar as respostas na *thread* principal. No Código 3.3 é possível ver a declaração do método.

Listing 3.3 – Exemplo de uso do método de varredura de portas síncrono

```
fun portScan(
    hostAddress: String,
    port: Int,
    timeout: Int
): PortScanResult
```

As entradas são as mesmas do método assíncrono e o retorno é um objeto do tipo **'PortScanResult'**. Este objeto possui os seguintes atributos:

- *hostAddress*: O endereço IPv4 do dispositivo encontrado na rede.
- *portScanned*: A porta do dispositivo encontrado.
- *elapsedTimeMillis*: O tempo gasto em milisegundos para encontrar o dispositivo.

De modo a padronizar as nomenclaturas dos métodos, foi convencionado que métodos assíncronos terão o sufixo *'Async'*. Caso contrário, o método executado é síncrono, como é o caso do método **'portScan'**, descrito no Código 3.3.

3.4.2 Ping

Durante o desenvolvimento do comando Ping foram adotadas estratégias específicas. Foi implementada uma abordagem que permite a execução do comando '/system/bin/ping' do Android em tempo de execução. Para isso, utilizou-se a classe **Runtime** do Java, que possibilita a execução do comando e a interpretação de sua saída pela biblioteca.

3.4.2.1 Método assíncrono

Assim como implementado na funcionalidade de varredura de portas, no comando Ping também foi feita a divisão entre método síncrono e assíncrono. O Código 3.4 representa o contrato do método assíncrono.

Listing 3.4 – Definição do método Ping assíncrono

```
fun pingAsync(
    hostAddress: String
): SubscribeResult<PingResult>
```

É possível notar que para execução do método, faz-se necessário informar o endereço IPv4 do dispositivo a ser encontrado. Dessa forma, uma vez informado o endereço, é possível então que sejam observados os retornos de sucesso e falha. O Código 3.5 demonstra como isso pode ser feito.

Listing 3.5 – Definição do método Ping assíncrono

```
netScan.pingAsync(hostAddress = "10.1.1.23")
        .onResult { result ->
            Log.i("This is a value sent", result.toString())
        }.onFailure { exception ->
            Log.i("This is an exception", exception.toString())
        }
```

3.4.2.2 Método síncrono

Da mesma forma, é apresentada a definição do método síncrono. Este método também espera como parâmetro o IPv4 do dispositivo a ser encontrado, tendo como diferença seu retorno. Isso pode ser notado no Código 3.6.

Listing 3.6 – Definição do método Ping síncrono

```
fun ping(hostAddress: String): PingResult
```

A classe **'PingResult'** é retornada por meio da chamada do método, e com ela, o desenvolvedor terá acesso aos seguintes atributos:

- *hostAddress*: O endereço IPv4 do dispositivo encontrado na rede
- *hostname* O nome do dispositivo encontrado

3.4.3 Varredura de redes domésticas

A funcionalidade de varredura de redes domésticas tem como base a ferramenta Ping. Para mapeamento dos dispositivos presentes na rede, é executada a varredura no intervalo de 1 a 254, de modo a identificar os dispositivos acessíveis. Foram ignorados os endereços finalizados em 0 e 255 por se tratarem de endereços reservados (KUROSE; ROSS, 2014). No Código 3.7 está representado o contrato definido para chamada do método.

Listing 3.7 – Definição do método de varredura de redes domésticas

```
fun domesticDeviceListScanner(): SubscribeResult<DeviceInfoResult>
```

Para utilização do método, deverá ser seguido o mesmo padrão de implementação das ferramentas com métodos assíncronos descritos nas seções anteriores. Sendo assim, por meio dos valores emitidos, é possível ter acesso à classe **'DeviceInfoResult'**, que contém os seguintes atributos:

- *address*: Endereço IPv4 do dispositivo encontrado na rede.
- *hostname* Nome do dispositivo (*hostname*) encontrado.

É importante destacar que, em algumas redes o Android pode retornar o próprio endereço IPv4 em vez do nome do dispositivo. Sendo assim, este é considerado como ponto de evolução para a NetScan.

3.4.4 Verificação de conexão com a Internet

Esta funcionalidade tem como objetivo fornecer ao desenvolvedor a capacidade de verificar se o dispositivo está conectado à Internet. Além disso, é possível realizar essa verificação considerando diferentes meios de conexão, como sem fio, por cabo ou dados móveis.

No desenvolvimento, foi utilizada a classe **NetworkCapabilities** para acessar os recursos necessários para obter informações sobre a conexão. No entanto, é importante destacar que alguns desses recursos foram marcados como obsoletos nas versões mais recentes das APIs do Android. Por exemplo, o método **getNetworkInfo**, que é usado para verificar se o dispositivo está conectado à Internet, foi depreciado.

De modo a evitar repetir os mesmos erros cometidos por outras bibliotecas que mantêm recursos marcados como depreciados, é necessário adotar uma abordagem alternativa. Para isso, foi possível utilizar o método **hasTransport**. Tal método fornece todas as informações necessárias para o retorno das informações sobre a conexão. Um ponto a ser considerado é que para uso do método o dispositivo deve possuir uma API superior a 23, também conhecida como Marshmallow. Isso traz uma limitação a biblioteca, fazendo com que a funcionalidade só possa ser utilizada em parte dos dispositivos.

Para resolver os problemas relacionados a recursos obsoletos e restrições de APIs, foi desenvolvida uma lógica interna na biblioteca. Essa lógica é responsável por identificar, em tempo de execução, a versão da API em que o dispositivo está sendo executado e chamar o método apropriado. Dessa forma, evita-se qualquer inconveniente causado pela obsolescência do método mencionado ou pelas restrições da API, viabilizando então o uso dos métodos criados em todos os dispositivos.

No Código 3.8, é possível visualizar o contrato definido para chamada dos métodos de verificação dos diferentes meios de conexão.

Listing 3.8 – Definição dos métodos para verificação de conexão

```
fun hasWifiConnection(): Boolean
fun hasMobileConnection(): Boolean
fun hasEthernetConnection(): Boolean
fun hasInternetConnection(): Boolean
```

Os métodos apresentados possuem os seguintes comportamentos:

- **hasWifiConnection** - Verifica de modo síncrono, se há conexão com a internet por meio de uma interface de rede sem fio.
- **hasMobileConnection** - Verifica de modo síncrono, se há conexão com internet utilizando dados móveis.
- **hasEthernetConnection** - Verifica de modo síncrono, se há conexão com a internet por meio de uma interface de rede cabeada.
- **hasInternetConnection** - Verifica de modo síncrono, se há conexão com a internet através de alguma das interfaces anteriores.

Os retornos dos métodos são feitos com valores booleanos, que irão indicar a presença ou não de conexão.

3.4.5 Varredura de redes *Wi-Fi*

A varredura de redes sem fio é uma ferramenta interessante, pois pode viabilizar por exemplo, a análise de segurança das redes próximas. Um dos motivadores para a criação desta ferramenta é a complexidade de sua implementação, uma vez que, requer que o desenvolvedor conheça e gerencie componentes do Android como o **'BroadcastReceiver'**. A ideia dessa ferramenta é encapsular todo esse trabalho dentro da biblioteca, de modo que, mais uma vez, o desenvolvedor não precise se preocupar com particularidades de implementação do Android. Para uso da ferramenta, basta que o desenvolvedor faça a chamada do método apresentado no Código 3.9.

Listing 3.9 – Definição do método de varredura de redes sem fio

```
@RequiresApi(value = Build.VERSION_CODES.M)
fun wifiScanner(): SubscribeResult<List<WifiInfoResult>>
```

O método ilustrado em 3.9 funciona de modo assíncrono, uma vez que, depende da resposta da notificação do SO para os resultados sejam acessados. Como retorno, o desenvolvedor terá uma lista de **'WifiInfoResult'**, que possui os seguintes parâmetros:

- *ssid* - SSID (Identificador de Conjunto de Serviço), é o nome atribuído a uma rede sem fio para identificá-la.
- *bssid* - BSSID (Identificador de Conjunto de Serviço Básico), é um endereço MAC (Media Access Control) de seis bytes que identifica de forma exclusiva um ponto de acesso sem fio.
- *capabilities* - É um conjunto de recursos que o dispositivo tem para suportar diferentes tipos de autenticação, criptografia e outras funcionalidades de rede sem fio.

Alguns pontos são importantes de serem ressaltados. Um deles é que para o uso da ferramenta, faz-se necessário uma API superior ou igual à 23, também conhecida como Marshmallow. Adicionalmente, é exigido que se tenha a permissão de acesso a informações de localização e redes sem fio. Isso poderá ser visto melhor no Código 4.1, onde é solicitada a permissão **Manifest.permission.ACCESS_FINE_LOCATION** [4] ao usuário.

Outro fator a ser considerado, é que durante a criação da funcionalidade houve a depreciação do método **'startScan'** da classe **WifiManager** na API 28 do Android. Esse método é utilizado internamente na biblioteca para iniciar o processo de varredura das redes sem fio. No entanto, foi notificado pelo Google na documentação oficial[5], que a função será removida em breve, o que poderá inviabilizar o uso da ferramenta, necessitando com isso de outra abordagem para efetuar a varredura.

3.4.6 Velocidade da conexão

Esta funcionalidade visa retornar ao desenvolvedor a largura de banda de download e upload, ou seja, a velocidade máxima teórica em kilobits por segundo (Kbps) suportada pela conexão de rede.

Esta informação é obtida utilizando a classe **'NetworkCapabilities'**, provida pelo Android. Através dela, é possível obter a largura de banda aproximada da conexão de rede disponível no dispositivo Android, ou seja, a velocidade máxima teórica da conexão com a rede local ou com a Internet.

[4] Quando é solicitada a permissão ACCESS_FINE_LOCATION, está sendo solicitado ao usuário acesso a informações precisas sobre localização do dispositivo, incluindo informações sobre redes Wi-Fi. O Android agrupa as informações de localização e as informações sobre redes Wi-Fi em uma única permissão.

[5] Disponível em: https://developer.android.com/reference/android/net/wifi/WifiManager

Um ponto importante a ser ressaltado é que a largura de banda é uma medida teórica, ou seja, representa a velocidade máxima que a conexão pode alcançar. Isso na prática, é apenas uma aproximação, uma vez que, a velocidade real pode variar por diversos fatores, como a qualidade da rede, a interferência de fatores externos na rede, distância do dispositivo em relação ao roteador, entre outros.

Para conseguir as informações de velocidade, basta que o desenvolvedor faça a chamada do método no Código 3.10.

Listing 3.10 – Definição do método de checagem de velocidade da conexão

```
@RequiresApi(value = Build.VERSION_CODES.M)
fun checkNetworkSpeed(): NetworkSpeedResult
```

O método descrito em 3.10 é um método síncrono, ele retornará para o desenvolvedor o objeto **'NetworkSpeedResult'** contendo os seguintes atributos:

- *downstreamKbps* - retorna a largura de banda de download em kilobits por segundo (Kbps) suportada pela conexão de rede.
- *upstreamKbps* - retorna a largura de banda de upload em kilobits por segundo (Kbps) suportada pela conexão de rede.

Assim como na varredura de redes sem fio, esta funcionalidade é acessível a dispositivos que possuam uma API superior ou igual à API 23.

3.5 Disponibilização da NetScan

Nesta seção, são fornecidos detalhes do processo de encapsulamento das funcionalidades em formato de uma biblioteca e demonstrado o processo de instalação da NetScan em projetos Android.

3.5.1 Encapsulamento das funcionalidades

Para desenvolvimento da biblioteca, foi escolhida a abordagem de manter os códigos relacionados à biblioteca e ao aplicativo de exemplo dentro do mesmo repositório. Esse direcionamento permitiu que as funcionalidades fossem encapsuladas em um formato de biblioteca, que foi criada como um módulo no projeto Android. Esta abordagem tem por objetivo isolar o código da biblioteca de eventuais códigos que não precisariam ser compartilhados com a comunidade.

Após feita a criação do módulo e o desenvolvimento do código correspondente as funcionalidades, foi feita a escolha do repositório de pacotes adotado para publicação da

biblioteca. Nesse contexto, foi adotado o repositório JitPack, devido a sua simplicidade de configuração em comparação com as outras opções disponíveis. Após feita a escolha do respositório para projetos Android, foi efetuada a configuração do arquivo Gradle[6] correspondente ao módulo da biblioteca. Neste arquivo, foram adicionadas as configurações conforme as orientações presentes na documentação oficial (JITPACK, 2023). Dessa forma, seguindo todo o processo descrito, a primeira versão da biblioteca foi publicada com sucesso.

3.5.2 Processo de instalação e inicialização

Para instalação da NetScan em projetos Android, é necessário que seja inclusa a URL (Uniform Resource Locator) correspondente ao JitPack no arquivo Gradle do projeto. No Código 3.11 é representada a inclusão da URL no arquivo 'settings.gradle'.

Listing 3.11 – Inclusão da URL do JitPack no arquivo setting.gradle

```
dependencyResolutionManagement {
repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
        maven { url 'https://jitpack.io' }
    }
}
```

Caso a configuração das dependências do projeto esteja seguindo outro padrão, é possível adotar uma abordagem alternativa. Com isso, o arquivo 'build.gradle' do projeto deverá ser alterado para inclusão da URL. No Código 3.12 é possível notar a inclusão da URL dentro do contexto **'repositories'**.

Listing 3.12 – Inclusão da URL do JitPack no arquivo build.gradle

```
allprojects {
    repositories {
        google()
        mavenCentral()
        maven { url 'https://jitpack.io' }
    }
}
```

[6] Gradle é uma ferramenta de automação de compilação e construção de projetos. Ele é usado principalmente no desenvolvimento de software para gerenciar as dependências, compilar o código-fonte, executar testes e criar os artefatos finais do projeto, como arquivos JAR ou APK.

Após adicionar a URL do repositório JitPack, é necessário incluir a dependência da biblioteca NetScan. No Código 3.13, é representada a inclusão da dependência.

Listing 3.13 – Inclusão da NetScan no arquivo build.gradle

```
dependencies {
    implementation 'com.github.gustavobarbosab:netscan:VERSAO'
}
```

No Código 3.13, é omitida a versão da biblioteca de modo que seja sempre utilizada a última versão disponível no repositório da NetScan no GitHub[7]. Concluída a inclusão da dependência, às próximas etapas são a inicialização e uso da biblioteca. Para isso, é necessário inicializar a biblioteca por meio da classe *Application* do projeto. No Código 3.14 é apresentado como efetuar a inicialização.

Listing 3.14 – Inicialização da NetScan

```
class MyApplication: Application() {
    override fun onCreate() {
        super.onCreate()
        NetScan.init(this)
    }
}
```

No Código 3.14 foram omitidos os *imports* de modo a simplificar a leitura. Nele, é possível notar que após a chamada do método **init**, a biblioteca estará pronta para ser utilizada. Nessa linha, no Código 3.15 é apresentado um exemplo de uso após a inicialização.

Listing 3.15 – Inicialização da NetScan

```
val netScan = NetScan.requireInstance()
        Log.d("POSSUI-CONEXAO?", netScan.hasInternetConnection()
            .toString())
```

No Código 3.15, é feita a recuperação da instância da biblioteca e é chamado o método **hasInternetConnection**. Dessa forma, é apresentado o uso de uma das funcionalidades.

[7] Disponível em: https://github.com/gustavobarbosab/netscan

4 Exemplo de uso da NetScan

De modo a exemplificar o uso das ferramentas descritas no Capítulo 3, foi desenvolvido um aplicativo Android que faz o uso da biblioteca NetScan para análises de redes de computadores.

4.1 Visão geral

Para o desenvolvimento da aplicação foi feito inicialmente o *design* da interface do usuário. Para isso, foram adotadas as técnicas de usabilidade descritas por Nielsen (2023), de modo a tornar o sistema amigável e de fácil uso. Na Figura 11, é possível visualizar a concepção da interface gráfica.

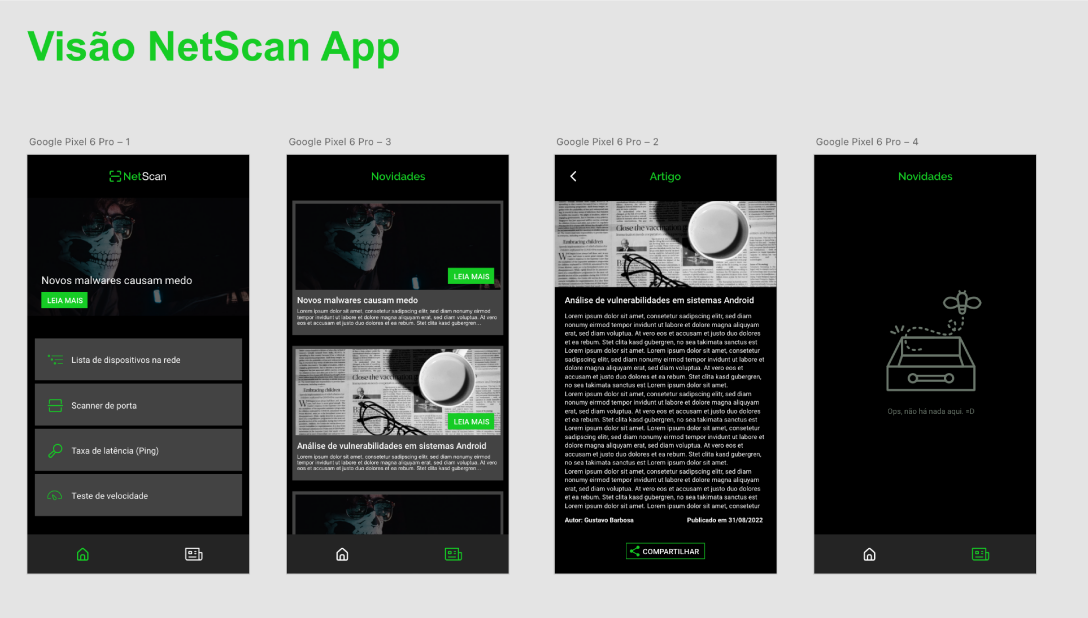

Figura 11 – Projeto da interface concebida para o app de exemplo.

Na concepção do projeto e criação da interface, foi idealizado o desenvolvimento de um portal de notícias, de modo a conscientizar os usuários e mantê-los informados sobre informações relacionadas à segurança da informação e redes de computadores. Esta funcionalidade não foi desenvolvida, uma vez que, neste momento o foco do trabalho é a exemplificação de uso das ferramentas da biblioteca. Sendo assim, este portal poderá ser criado futuramente.

É importante evidenciar que as funcionalidades presentes na aplicação foram testadas em um celular Motorola Moto One Hyper, com Android na versão 11 e, em um

Samsumg S21 FE, com Android 13. Além disso, para os testes foram utilizadas duas redes domésticas.

4.2 Interface gráfica do aplicativo

Na Figura 11, são demonstradas as telas iniciais. Além disso, é possível visualizar a lista de funcionalidades implementadas. Na Figura 13, são apresentadas as telas das funcionalidades, possibilitando assim o entendimento do funcionamento de cada uma.

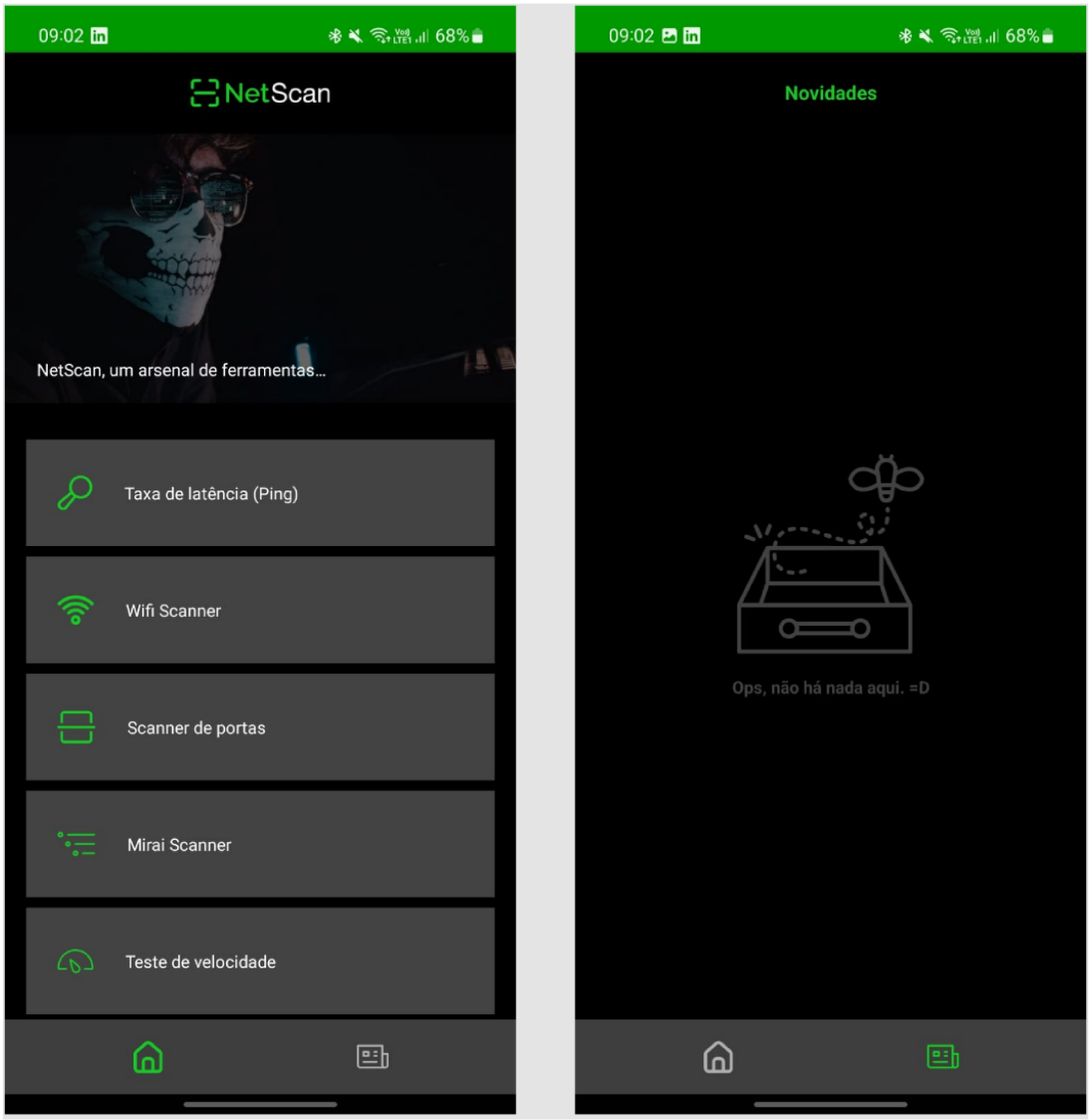

Figura 12 – Interface das telas iniciais desenvolvidas no app de exemplo

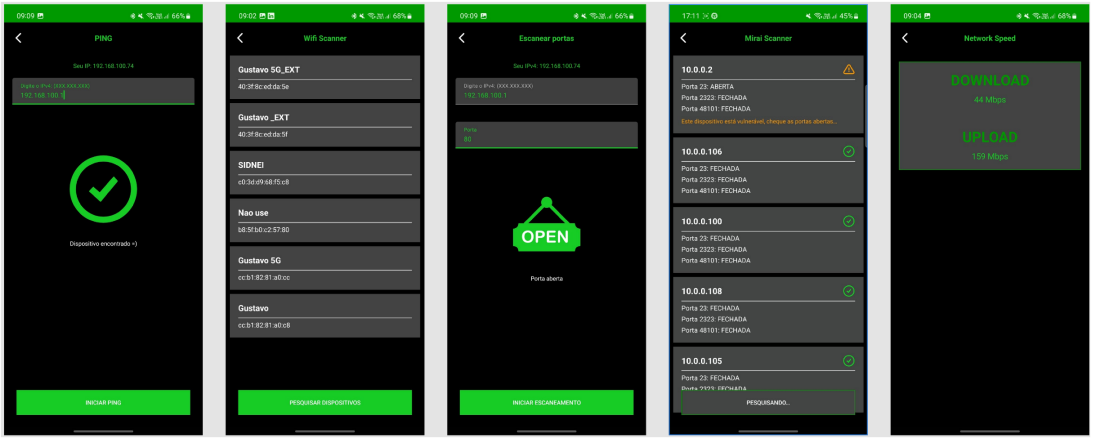

Figura 13 – Interface das funcionalidades desenvolvidas no App de exemplo

Nas capturas de tela, é possível notar os cenários de sucesso onde o usuário consegue encontrar um dispositivo via comando Ping, buscar a lista de redes sem fio próximas, encontrar a porta aberta em um dispositivo da rede, obter a visão da velocidade aproximada de conexão com a Internet e na Seção 4.4 é demonstrado em mais detalhes a funcionalidade de varredura dos dispositivos da rede, a fim de encontrar portas vulneráveis ao *malware* Mirai.

4.3 Organização do código do aplicativo

Para desenvolvimento da aplicação, foi utilizado o padrão MVVM descrito no Capítulo 2. Foi adotado também em algumas funcionalidades o uso de chamadas síncronas, para possibilitar o uso de ferramentas para programação assíncrona, como o conceito de Coroutines [1] introduzidas pelo Kotlin. Isso permite que o desenvolvedor consiga manter padrões adotados no projeto de modo a não necessitar de uma mudança de paradigma, o que tornaria a biblioteca menos flexível.

Na Figura 14, é possível visualizar a captura de tela feita na IDE Android Studio, de modo a entender um pouco mais da separação de módulos feita no projeto. No projeto foi adotado um padrão de modularização contendo a seguinte organização de módulos:

- *app*: é o módulo principal da aplicação.
- *common*: módulo que contém recursos comuns para as funcionalidades.
- *feature*: é a pasta que contém os módulos das funcionalidades presentes no projeto.

[1] Disponível em: https://developer.android.com/kotlin/coroutines

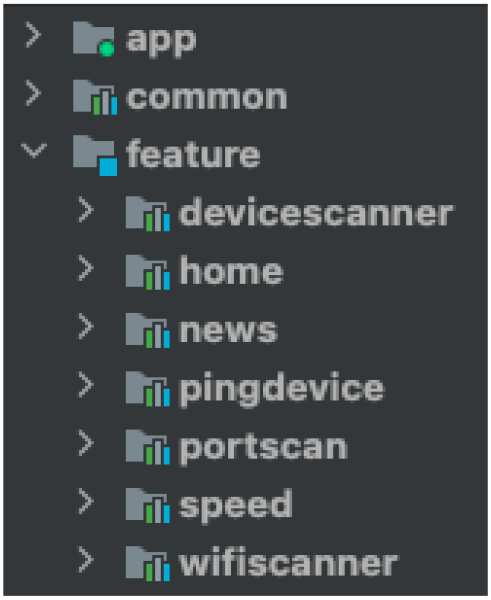

Figura 14 – Divisão de pacotes do App de exemplo

4.4 Réplica do Mirai Scanner

Anteriormente, foram citados os trabalhos de Silva-Junior (2019) e Costa (2018), que desenvolveram aplicações que utilizam de ferramentas de rede para identificar dispositivos vulneráveis a ataques do Malware Mirai em redes domésticas. Com o intuito de exemplificar um caso de uso da NetScan, é feito o uso das ferramentas de varredura de dispositivos e varredura de portas, para replicar parte do aplicativo Mirai Scanner. As demais ferramentas e seus respectivos códigos podem ser vistos no repositório no GitHub[2].

4.4.1 Mirai

De forma sucinta, como descrito por Antonakakis et al. (2017), Mirai é uma família de *malware* que se espalha como um *worm* e infecta dispositivos IoT, formando uma *botnet* de DDoS. Os *bots* do Mirai realizam uma varredura nos endereços IPv4 em busca de dispositivos que tenham Telnet ou SSH ativados. Eles tentam fazer login usando um conjunto fixo de credenciais pré-definidas para dispositivos IoT. Quando conseguem acesso, os bots enviam as informações do dispositivo infectado para um servidor de relatório. Esse servidor, por sua vez, inicia de forma assíncrona um processo de infecção no dispositivo. Os dispositivos infectados, por sua vez, continuam procurando por novas vítimas e obedecem a comandos de ataque DDoS provenientes de um servidor de controle (ANTONAKAKIS et al., 2017). Com auxílio do trabalho de Costa (2018), é possível observar que o Mirai busca dispositivos que possuam portas como 23, 2323 e 48101 abertas. Tais portas são correspondentes respectivamente a serviços de Telnet, SSH e da ferra-

[2] Disponível em: <https://github.com/gustavobarbosab/netscan.>

menta **ScanListen** do Mirai, que aguarda conexões TCP a fim de transmitir a listagem de dispositivos encontrados.

4.4.2 Interface gráfica da réplica

A Figura 15, apresenta a interface aplicada para mostrar os dispositivos seguros e os dispositivos vulneráveis ao Mirai. Além disso, é mostrado ao usuário quais portas estão abertas e quais estão fechadas, assim como feito no trabalho de Costa (2018).

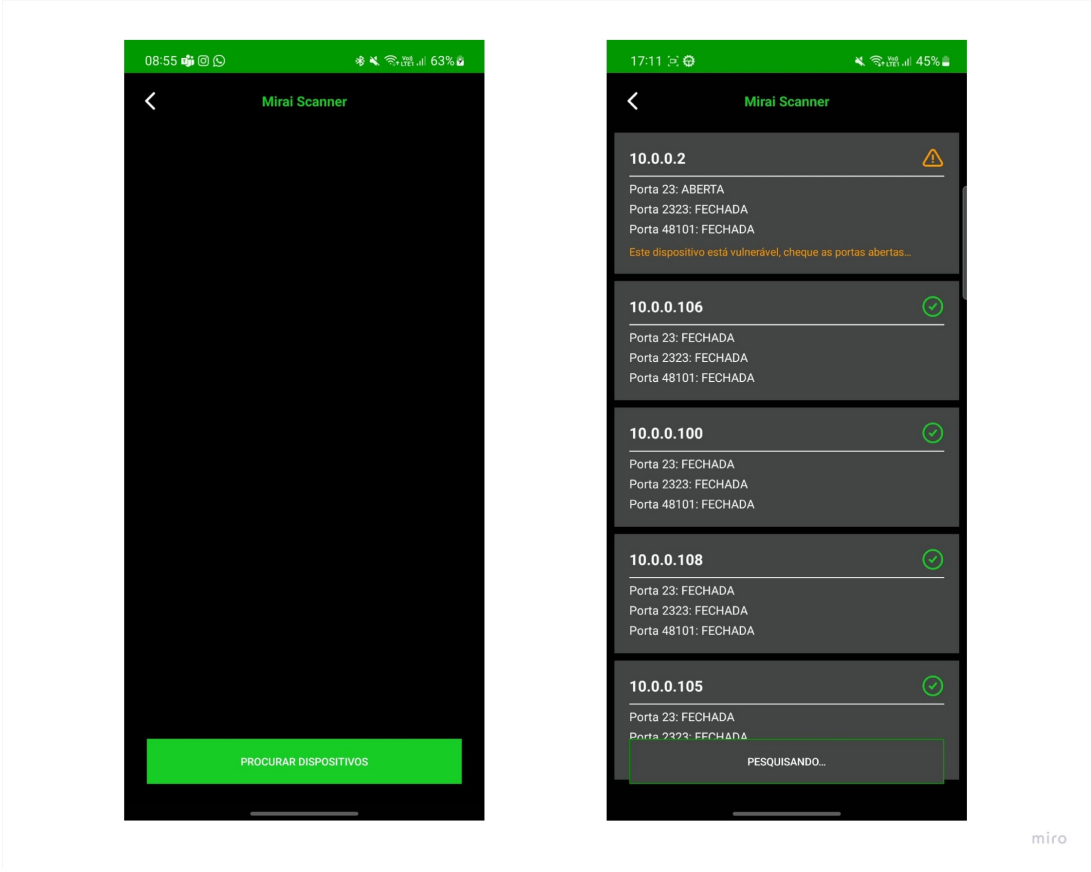

Figura 15 – Interface da réplica do Mirai Scanner

Ao acessar a funcionalidade o usuário verá a tela com o botão 'PROCURAR DISPOSITIVOS' e ao clicar o botão mudará sua forma notificando o usuário que está sendo feita uma pesquisa na rede. Sempre que um dispositivo é achado ele é incluso na lista, possibilitando com isso que o usuário veja se há ou não uma vulnerabilidade aparente.

4.4.3 Código do Fragment

Para que se entenda como foi implementada a solução, é importante que se veja os códigos do **Fragment** e do **ViewModel**. Sendo assim, é descrito agora o código *Fragment*,

responsável por toda a lógica de atualização e manipulação da interface gráfica mostrada ao usuário.

No Código 4.1 é possível observar que estão sendo utilizadas as bibliotecas *ViewBinding*[3] para vincular os elementos do código XML (*Extensible Markup Language*, em inglês), com o código Kotlin e o **ViewModel** provido pelo JetPack[4]. Outro fator importante, é que foram omitidos os *imports* das classes utilizadas neste e nos demais códigos, de modo a simplificar a visualização.

Listing 4.1 – Código do Fragment usado para réplica do Mirai

```
class DeviceScannerFragment : Fragment() {

    private val adapter = DeviceScannerAdapter()
    private var binding: FragmentDeviceScannerBinding? = null
    private val viewModel: DeviceScannerViewModel by viewModels
       { DeviceScannerViewModel.Factory }

    private val hasNotPermission
        get() = ContextCompat.checkSelfPermission(
            requireContext(),
            Manifest.permission.ACCESS_FINE_LOCATION
        ) != PackageManager.PERMISSION_GRANTED

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentDeviceScannerBinding.inflate(inflater,
            container, false)
        return binding!!.root
    }

    override fun onViewCreated(view: View, savedInstanceState:
       Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding?.apply {
            recycler.adapter = adapter
            findDevicesButton.setOnClickListener {
```

[3] Disponível em: https://developer.android.com/topic/libraries/view-binding?hl=pt-br
[4] Disponível em: https://developer.android.com/topic/libraries/architecture/viewmodel?hl=pt-br

```
            viewModel.scanDevices(hasNotPermission)
        }
    }

    netScanToolbar().apply {
        showBackButton { requireActivity().onBackPressed() }
        title(context.getString(R.string.
            device_scanner_button_title))
    }

    viewModel.screenState.observe(viewLifecycleOwner) {
        state ->
        when (state) {
            SearchingDeviceList -> {
                binding?.findDevicesButton?.apply {
                    isEnabled = false
                    text = context.getString(R.string.
                        device_scanner_button_searching)
                }
            }
            RequestPermission -> ActivityCompat.
                requestPermissions(
                requireActivity(),
                arrayOf(Manifest.permission.
                    ACCESS_FINE_LOCATION),
                12
            )
            is AddDevice -> adapter.addItem(state.device)
            DeviceScannerState.DeviceSearchFinished ->
                enableButton()
            DeviceScannerState.WifiDisconnected -> {
                Toast.makeText(
                    requireContext(),
                    getString(R.string.please_connect_wifi),
                    Toast.LENGTH_LONG
                ).show()
                enableButton()
            }
        }
```

```
        }
    }

    private fun enableButton() {
        binding?.findDevicesButton?.apply {
            isEnabled = true
            text = context.getString(R.string.
                device_scanner_button_search)
        }
    }
}
```

É importante notar que alguns estados foram mapeados a fim de atualizar a tela para o usuário. A seguir cada um destes estados é descrito:

- **SearchingDeviceList**: é o estado de carregamento da informação, usado quando há uma varredura em andamento.
- **DeviceSearchFinished**: estado usado para indicar que a pesquisa pelos dispositivos acabou.
- **WifiDisconnected**: este estado é usado para cenários onde não se há uma conexão com a rede sem fio.
- **AddDevice**: usado para adicionar um dispositivo na tela quando encontrado.
- **RequestPermission**: estado necessário para se requisitar permissão[5] ao usuário para que se tenha acesso a informações da rede sem fio.

Com os estados acima, o *Fragment* se inscreve para observar os estados providos pelo *ViewModel* e toma as ações necessários para atualização da interface gráfica quando notificado. A explicação dos demais métodos do *Fragment* pode ser vista com mais detalhes na documentação oficial (GOOGLE, 2023g).

4.4.4 Código do ViewModel

Aqui é demonstrado como foi desenvolvido o código do **ViewModel** (4.2), responsável por conter toda a lógica de uso da biblioteca NetScan, bem como a emissão dos estados responsáveis por notificar o **Fragment** sobre a necessidade de atualizar a interface do usuário.

A explicação do código é feita por métodos:

[5] https://developer.android.com/about/versions/13/features/nearby-wifi-devices-permission?hl=pt-br

- **fun scanDevices:** método chamado ao clicar no botão de iniciar varredura. Nele, inicialmente é feita a verificação se o usuário concedeu permissão de acesso às informações da rede sem fio, emitindo o estado correspondente caso não haja permissão. Logo em seguida, é emitido o estado de carregamento e solicitado a biblioteca NetScan que faça a varredura na rede, emitindo os estados correspondentes nos cenários de sucesso e falha.
- **fun findEnabledPorts:** este método privado é chamado quando é encontrado um dispositivo através da varredura feito pela NetScan. Nele, é feito um novo processamento para identificar quais portas estão abertas neste dispositivo e posteriormente é emitido um estado para adicionar o dispositivo na tela.
- **fun onCleared:** método próprio da classe ViewModel. Ele é chamado sempre que o ViewModel não está sendo mais usado e pode ser destruído. Nele, é importante parar a execução de todos os serviços assíncronos, como é o caso dos métodos observáveis que são chamados na NetScan.

Listing 4.2 – Código do ViewModel usado para réplica do Mirai

```
class DeviceScannerViewModel(private val netScan: NetScan) :
    ViewModel() {

    val screenState = SingleLiveEvent<DeviceScannerState>()
    private val netScanObservableUnbind =
        SubscribeResultDisposable()

    fun scanDevices(hasNotPermission: Boolean) {
        if (hasNotPermission) {
            screenState.value = DeviceScannerState.
                RequestPermission
            return
        }

        screenState.value = DeviceScannerState.
            SearchingDeviceList
        netScan.domesticDeviceListScanner()
            .onScheduler(SubscribeScheduler.Main)
            .onResult {
                findEnabledPorts(it)
            }.onFailure {
                if (it is InternetConnectionNotFoundException) {
```

```
                screenState.value = DeviceScannerState.
                    WifiDisconnected
            }
        }.onComplete {
            screenState.value = DeviceScannerState.
                DeviceSearchFinished
        }
        .also(netScanObservableUnbind::add)

}

private fun findEnabledPorts(device: DeviceInfoResult) {
    viewModelScope.launch {
        val hashMap = mutableMapOf<VulnerablePortsMirai,
            PortScanResult?>()
        withContext(Dispatchers.IO) {
            VulnerablePortsMirai.values().forEach {
                val response = runCatching {
                    return@runCatching netScan.portScan(
                        device.address,
                        it.portNumber,
                        TIMEOUT
                    )
                }
                hashMap[it] = response.getOrNull()
            }
        }

        screenState.value = DeviceScannerState.AddDevice(
            DeviceItem(
                device.address,
                hashMap[VulnerablePortsMirai.PORT_23] !=
                    null,
                hashMap[VulnerablePortsMirai.PORT_2323] !=
                    null,
                hashMap[VulnerablePortsMirai.PORT_48101] !=
                    null
            )
        )
```

```
        }
    }

    override fun onCleared() {
        super.onCleared()
        netScanObservableUnbind.dispose()
    }

    companion object {
        const val TIMEOUT = 5000

        val Factory: ViewModelProvider.Factory = object :
            ViewModelProvider.Factory {
            @Suppress("UNCHECKED_CAST")
            override fun <T : ViewModel> create(
                modelClass: Class<T>
            ): T {
                return DeviceScannerViewModel(NetScan.
                    requireInstance()) as T
            }
        }
    }
}
```

5 Conclusão

O presente trabalho detalhou o desenvolvimento da biblioteca NetScan. Primeiramente, foram estabelecidos os requisitos para o desenvolvimento da biblioteca: fornecer as principais funcionalidades de análise de rede em conformidade com os novos recursos do Android. Adicionalmente, foi feita a disponibilização da biblioteca para a comunidade e a criação de uma aplicação exemplificando o uso da NetScan. Durante o estudo, foi possível perceber a importância de fornecer aos pesquisadores e desenvolvedores um conjunto de recursos de fácil uso, que acompanhem as contantes evoluções do *framework*.

Os resultados esperados com o desenvolvimento desse projeto foram alcançados, isso pode ser notado através do conjunto de ferramentas desenvolvido e da aplicação delas no aplicativo de exemplo. Além disso, é possível observar que, em poucas linhas de código foi possível implementar a varredura de portas a fim de detectar dispositivos vulneráveis em redes domésticas, replicando então, os trabalhos feitos por Costa (2018) e Silva-Junior (2019). Isso evidencia o potencial da abordagem adotada e sua relevância para a comunidade de desenvolvedores.

É esperado que este trabalho contribua para a área de pesquisa em desenvolvimento de aplicativos móveis para o sistema Android. A disponibilização de um conjunto de ferramentas atualizadas e de fácil uso, proporciona um ambiente propício para a realização de estudos e experimentos, possibilitando o avanço do conhecimento nesse campo. Além disso, a simplicidade para implementação pode estimular novos desenvolvedores a explorarem o potencial do ambiente Android.

Embora se tenha alcançado bons resultados, é importante ressaltar a existência de limitações. Por exemplo, as ferramentas desenvolvidas foram testadas em um conjunto específico de cenários e dispositivos, podendo ser aprimoradas. Adicionalmente, há espaço para criação de novas ferramentas, ajuste dos recursos para serem compatíveis com IPv6 e expansão da biblioteca para outros sistemas como iOS. Além disso, na funcionalidade de varredura de redes sem fio, é utilizado um recurso que, segundo a documentação oficial, será removido em APIs futuras do Android, sendo necessário a implementação de uma nova abordagem para a varredura.

Em suma, este trabalho proporcionou a criação de um conjunto de ferramentas de rede atualizadas e acessíveis para o ambiente Android, visando facilitar o desenvolvimento de aplicações que necessitem de tais ferramentas. Os resultados alcançados evidenciam o potencial dessas ferramentas para impulsionar a pesquisa e contribuir para a melhoria da qualidade e segurança das redes e aplicações Android. Espera-se, que este trabalho seja um ponto de partida para novas investigações e contribuições no campo de desenvolvimento

móvel, segurança e redes de computadores.

Referências

ANTONAKAKIS, M. et al. Understanding the mirai botnet. In: *26th {USENIX} security symposium ({USENIX} Security 17)*. [S.l.: s.n.], 2017. p. 1093–1110. Citado 2 vezes nas páginas 10 e 46.

AUBORT, J.-B. *Android Network Discovery*. 2023. Repositório do GitHub. Disponível em: <https://github.com/rorist/android-network-discovery>. Acesso em: 13 jul. 2023. Citado na página 26.

COSTA, G. M. Implementação de um aplicativo para detecção de botnets iot em ambientes domésticos. Universidade Federal de Uberlândia, p. 60, 2018. Citado 7 vezes nas páginas 10, 11, 26, 27, 46, 47 e 54.

FGVCIA. *Brasil tem 424 milhões de dispositivos digitais em uso, revela 31ª Pesquisa Anual FGVcia*. 2021. Disponível em: <https://portal.fgv.br/noticias/brasil-tem-424-milhoes-dispositivos-digitais-uso-revela-31a-pesquisa-anual-fgvcia>. Acesso em: 13 jul. 2023. Citado na página 10.

FREEMAN, E. *Use a cabeça!: padrões de projetos*. Rio de Janeiro: Alta Books, 2009. 496 p. (Use a Cabeça!). ISBN 9788576081746. Citado 6 vezes nas páginas 6, 16, 18, 19, 20 e 21.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. India: Addison-Wesley, 1995. 395 p. ISBN 0201633612. Citado 3 vezes nas páginas 6, 19 e 20.

GOOGLE. *ANR (App Not Responding)*. 2021. Disponível em: <https://developer.android.com/topic/performance/vitals/anr?hl=pt-br>. Acesso em: 13 jul. 2023. Citado na página 34.

GOOGLE. *Memory Profiler*. 2021. Disponível em: <https://developer.android.com/studio/profile/memory-profiler?hl=pt-br>. Acesso em: 13 jul. 2023. Citado na página 32.

GOOGLE. *Abordagem Kotlin do Android*. 2023. Site Developer Android. Disponível em: <https://developer.android.com/kotlin/first?hl=pt-br>. Acesso em: 13 jul. 2023. Citado na página 25.

GOOGLE. *Android - Google Developers*. 2023. Disponível em: <https://developer.android.com>. Acesso em: 13 jul. 2023. Citado na página 22.

GOOGLE. *Android JetPack Lifecycle*. 2023. Disponível em: <https://developer.android.com/jetpack/androidx/releases/lifecycle>. Acesso em: 13 jul. 2023. Citado na página 18.

GOOGLE. *Android Platform*. 2023. Disponível em: <https://developer.android.com/guide/platform>. Acesso em: 13 jul. 2023. Citado 2 vezes nas páginas 6 e 22.

GOOGLE. *Android Studio*. 2023. Captura de tela. Disponível em: <https://developer.android.com/studio>. Acesso em: 13 jul. 2023. Citado 2 vezes nas páginas 6 e 24.

GOOGLE. *Android Studio Oficial Documentation*. 2023. Disponível em: <https://developer.android.com/studio/intro>. Acesso em: 13 jul. 2023. Citado na página 25.

GOOGLE. *Fragments*. 2023. Disponível em: <https://developer.android.com/guide/fragments?hl=pt-br>. Acesso em: 13 jul. 2023. Citado na página 50.

GOOGLE. *ViewModel Overview*. 2023. Disponível em: <https://developer.android.com/topic/libraries/architecture/viewmodel?hl=pt-br>. Acesso em: 13 jul. 2023. Citado na página 18.

JITPACK. *Publish an Android Library*. 2023. Disponível em: <https://docs.jitpack.io/android/>. Acesso em: 14 jul. 2023. Citado na página 41.

KUROSE, J.; ROSS, K. *Redes de computadores e a internet uma abordagem top-down*. 6th. ed. São Paulo: Pearson, 2014. 656 p. ISBN 9788581436777. Citado 4 vezes nas páginas 14, 15, 33 e 36.

LOU, T. A comparison of android native app architecture mvc, mvp and mvvm. Eindhoven University of Technology, p. 57, 2016. Citado 5 vezes nas páginas 6, 16, 17, 18 e 19.

MARTIN, R. C. *Código Limpo: Habilidades Práticas do Agile Software*. Rio de Janeiro: Alta Books, 2008. 464 p. ISBN 9780136083252. Citado 6 vezes nas páginas 16, 17, 21, 22, 32 e 33.

MCAFEE. *McAfee 2023 Consumer Mobile Threat Report*. 2023. Disponível em: <https://www.mcafee.com/blogs/internet-security/mcafee-2023-consumer-mobile-threat-report/>. Acesso em: 13 jul. 2023. Citado na página 10.

NIELSEN, J. *Ten Usability Heuristics*. 2023. Disponível em: <https://www.nngroup.com/articles/ten-usability-heuristics/>. Acesso em: 13 jul. 2023. Citado na página 43.

ROLLINGS, M. *AndroidNetworkTools*. 2023. Disponível em: <https://github.com/stealthcopter/AndroidNetworkTools>. Acesso em: 13 jul. 2023. Citado 3 vezes nas páginas 11, 25 e 26.

SILVA-JUNIOR, A. C. C. d. Aplicativo mirai scanner: modificaçõees e uma análise preliminar das varreduras. Universidade Federal de Uberlândia, p. 64, 2019. Citado 5 vezes nas páginas 10, 26, 27, 46 e 54.

STALLINGS, W. *Operating Systems: Internals and Design Principles, Global Edition*. 9th. ed. Londres: Pearson, 2018. 1128 p. ISBN 9781292214290. Citado na página 27.

STATISTA. *Number of Mobile Devices Owned Worldwide from 2010 to 2021, with a Forecast for 2022 (in billions)*. 2021. Disponível em: <https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/>. Acesso em: 13 jul. 2023. Citado na página 10.

STATISTA. *Global market share held by mobile operating systems from 2009 to 2023*. 2023. Disponível em: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>. Acesso em: 13 jul. 2023. Citado na página 10.

TANENBAUM, A. *Computer Networks*. 4th. ed. New Jersey: Person, 2003. 891 p. ISBN 9788177581652. Citado 4 vezes nas páginas 6, 13, 15 e 16.

WOOD, A. *PortAuthority*. 2023. Disponível em: <https://github.com/aaronjwood/PortAuthority>. Acesso em: 13 jul. 2023. Citado na página 26.