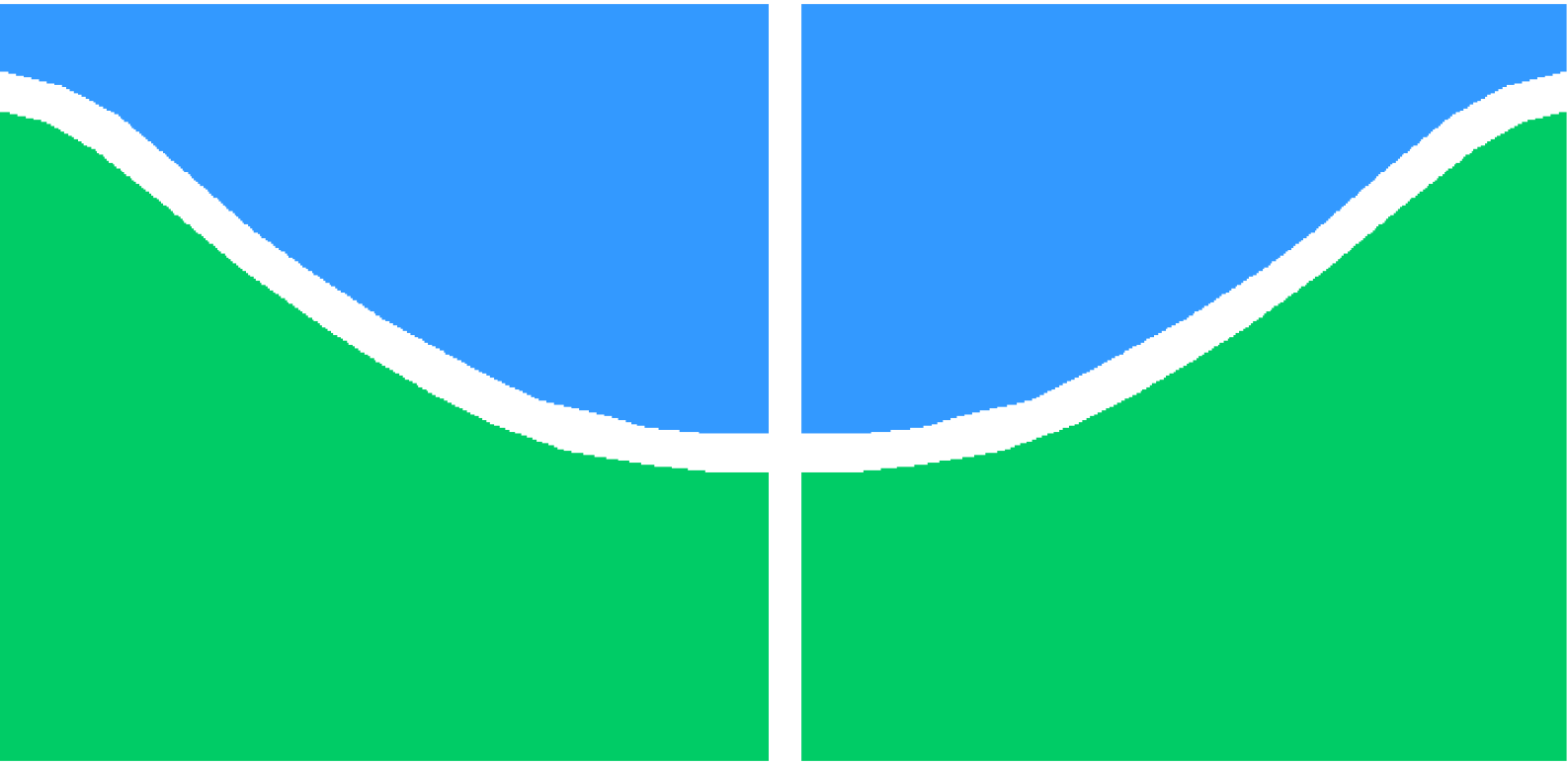


Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Implementação de uma aplicação *mobile* de troca de mensagens com ênfase em segurança digital

Autor: Renata Soares dos Santos

Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF

2018

Renata Soares dos Santos

Implementação de uma aplicação *mobile* de troca de mensagens com ênfase em segurança digital

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF

2018

Renata Soares dos Santos

Implementação de uma aplicação *mobile* de troca de mensagens com ênfase em segurança digital/ Renata Soares dos Santos. – Brasília, DF, 2018-

126 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Tiago Alves da Fonseca

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. Palavra-chave01. 2. Palavra-chave02. I. Prof. Dr. Tiago Alves da Fonseca. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação de uma aplicação *mobile* de troca de mensagens com ênfase em segurança digital

CDU 02:141:005.6

Renata Soares dos Santos

Implementação de uma aplicação *mobile* de troca de mensagens com ênfase em segurança digital

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 9 de julho de 2018 – Data da aprovação do trabalho:

Prof. Dr. Tiago Alves da Fonseca
Orientador

Prof. Dr. Fernando William Cruz
Convidado 1

Prof. Dr. Renato Coral Sampaio
Convidado 2

Brasília, DF
2018

Resumo

Com o extenso número de usuários na Internet que realizam cada vez mais transações críticas, das quais deve ser mantido o acesso restrito dos dados ao autor e ao destinatário, a segurança digital se tornou uma preocupação em comum entre os usuários e os desenvolvedores, sendo considerada e priorizada durante todo o ciclo de desenvolvimento de um *software*. O aumento do número dos usuários da Internet também provocou a ascensão de invasores/adversários que buscam beneficiar-se dos dados de terceiros interceptados em transações de caráter sigiloso. Portanto, garantir a anonimidade e privacidade de um usuário na rede promove mais liberdade e confiabilidade do mesmo ao usar o *software*. Sendo assim, a proposta do trabalho é desenvolver uma aplicação de mensagens *mobile*, em Android, que utilize mecanismos de defesa oriundos da engenharia de segurança. Um dos principais mecanismos aplicado foi a rede de anonimização Tor para garantir que os usuários possam trocar mensagens e não serem identificados como os autores destas mensagens. Desta forma, a aplicação foi implementada para garantir que o usuário possa utilizá-la e ser protegido de possíveis ataques.

Palavras-chave: Segurança Digital. Segurança. Engenharia de Segurança. Aplicação de Mensagens. Android. Mobile. Redes de Anonimização.

Abstract

With the vast number of Internet users who are increasingly engaged in critical transactions, from which restricted access of data to the author and recipient must be maintained, the importance of digital security has become a common concern among users and developers. So it should be considered and prioritized throughout the development cycle of software. The rise of Internet users has also sparked a rise in intruders seeking to benefit from third-party data intercepted in stealthy transactions. Therefore, ensuring the anonymity and privacy of a user on the network promotes more freedom and reliability when using a software. The proposal of this work is to develop a mobile application of messages, in Android, that uses defense mechanisms from security engineering. One of the main mechanisms applied was the Tor anonymization network to ensure that users can exchange messages and not be identified as the authors of these messages. In that way, the application has been implemented to ensure that the user can use it and be protected from possible attacks.

Key-words: Digital Security. Security. Security Engineering. Message Application. Android. Mobile. Anonymization Networks.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AES	*Advanced Encryption Standard*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
BPMN	*Business Process Model and Notation*
BaaS	*Backend-as-a-Service*
DAX	*Amazon DynamoDB Accelerator*
DL	*Discrete Logarithm*
DNS	*Domain Name System*
E2EE	*End-to-end Encryption*
EC	*Elliptic Curve*
EMS	Segredo Mestre Cifrado ou *Encrypted Master Secret*
GCM	*Galois/Counter Mode*
HTTP	*HyperText Transfer Protocol*
IP	Internet *Protocol*
JWT	*Json Web Token*
KDC	*Key Distribution Center*
MAC	*Media Access Control*
MAC	*Medium Acess Control Protocol*
MSF	*Microsoft Solutions Framework*
MS	Segredo Mestre ou *Master Secret*
OSI	*Open System Interconnection*
P2P	*peer-to-peer*
PFS	*Perfect Foward Secrecy*

SSL	*Secure Socktes Layer*
TCP	*Transmition Control Protocol*
TLD	Domínio de Alto Nível ou *Top Level Domain*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*
VPN	*Virtual Private Network*

Sumário

1 Introdução

A sociedade, ao longo de sua história, já se utilizou de vários meios de comunicação a fim de difundir informação e possibilitar uma comunicação remota nas relações interpessoais, e, com isso, vem a cada dia investindo em inovação nas infraestruturas que proveem essa comunicação. “O que tem acontecido ao longo de todos os anos é a evolução dos meios de comunicação juntamente com a tecnologia, mudando sempre a maneira de como o receptor da mensagem vai recebê-la” (CARRIÇO et al., 2012).

Atualmente, os principais meios de comunicação estão hospedados na Internet. Algumas vantagens desta nova maneira de comunicar é a rapidez, o alcance global e o fácil acesso às ferramentas que conduzem essas informações. “Um dos aspectos mais salientes da comunicação no mundo moderno é que ela acontece numa escala cada vez mais global. Mensagens são transmitidas através de grandes distâncias com bastante facilidade, de tal maneira que indivíduos têm acesso à informação e comunicação provenientes de fontes distantes” (THOMPSON, 2011).

Uma preocupação associada às ferramentas modernas de comunicação é a segurança dos dados transmitidos de forma privada entre usuários. Essas informações podem ser alvo de fraudes e serem acessadas por pessoas não autorizadas e com más intenções. Os golpes de ataques digitais estão cada vez mais frequentes. Um caso conhecido mundialmente e relatado pela EFF (2016) foi a espionagem digital realizada pela Agência de Segurança Nacional dos Estados Unidos (NSA) que coletou, ilegalmente, várias informações privadas de diferentes alvos em vários países.

Para garantir a segurança dos conteúdos enviados via Internet, Kurose (2005) define que se deve aferir a integridade, o sigilo e a intelecção dos dados enviados, certificar as identidades de remetentes e destinatários e detectar e prevenir acessos indevidos a informações privadas. Essas propriedades impedem que invasores consigam alterar, inserir, observar ou apagar qualquer elemento disponível na rede.

1.1 Justificativa

Atualmente, os usuários de plataformas móveis as utilizam para a realização de várias transações sigilosas pela Internet como acessar um extrato bancário, utilizar carteiras virtuais para pagamentos, trocar mensagens com outras pessoas, entre outras. Se, porventura, algumas dessas transações for prejudicada, trará consequências severas ao usuário tais como prejuízo financeiro, danos na vida social e assim por diante. Portanto, é importante garantir que nenhuma dessas transações serão violadas por terceiros.

Entretanto, vários desenvolvedores não possuem a preocupação de aplicar mecanismos de segurança em seus produtos até que o sistema seja prejudicado por possuir vulnerabilidades. A segurança da aplicação deve ser considerada no início e durante o seu desenvolvimento para evitar que intrusos prejudiquem os usuários do *software* o quanto antes (KINOSHITA; MCNAB, 2017). Portanto, é importante considerar a segurança das aplicações em plataformas móveis constantemente. Principalmente em aplicações em que os usuários fornecem dados sigilosos como as aplicações de mensagens.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo principal da pesquisa é desenvolver um aplicativo *mobile* de troca de mensagens que garanta a privacidade dos dados fornecidos pelos usuários prevenindo ataques externos e oferecendo-lhes a confidencialidade necessária.

1.2.2 Objetivos Específicos

- Estudar vulnerabilidades de aplicações de mensagens na rede.
- Selecionar soluções de segurança digital a serem incorporadas na implementação.
- Desenvolver uma aplicação *mobile* segura de mensagens.
- Testar as soluções de segurança da aplicação.

1.3 Estrutura do Documento

O documento possui sete capítulos:

Introdução: Breve descrição do trabalho.

Fundamentação Teórica: Contém os conceitos teóricos necessários para a assimilação da solução proposta.

Proposta de Trabalho: Contém o detalhamento da proposta de desenvolvimento da aplicação.

Solução: Contém a descrição dos principais módulos da aplicação e da arquitetura da solução.

Metodologia: Contém a descrição da metodologia utilizada no desenvolvimento do trabalho.

Resultados e Discussões: Contém os resultados gerados e discussões acerca dos mesmos.

Conclusão: Contém uma análise crítica do trabalho.

2 Fundamentação Teórica

2.1 Sistemas Computacionais Distribuídos

Um computador é um dispositivo que recebe uma entrada, executa uma ação e retorna uma saída. De acordo com Tanenbaum (2016), um computador digital é um dispositivo que recebe instruções e as processa para a resolução de problemas. Um sistema computacional consiste em todos os componentes de um computador. Segundo Stallings (2000), os principais componentes são processadores, memórias e elementos responsáveis pela gerência de entrada e saída de dados na máquina.

O uso dos computadores para a resolução de problemas cresceu significativamente nos últimos anos. Atualmente, várias atividades da sociedade dependem de sistemas computacionais como o gerenciamento de recursos de uma empresa, o relacionamento interpessoal estabelecido em redes sociais, a resolução de problemas matemáticos complexos, o acesso instantâneo à informações de fontes distantes e dispersas no mundo, o gerenciamento de sistemas bancários, entre outras.

À medida que o uso dos sistemas computacionais expandiu, as atividades requeridas sobrecarregavam as máquinas, surgiu-se então a necessidade de fracionar a execução dessas tarefas em vários computadores interligados. O computador único foi substituído por vários computadores diferentes. Sendo assim, emergiu uma rede de computadores que permite a comunicação de sistemas computacionais distantes geograficamente, possibilitando a troca de informações entre eles e a distribuição de recursos computacionais.

Uma rede de computadores consiste em computadores autossuficientes interligados que conseguem compartilhar dados entre si. Segundo Tanenbaum (2003), essa conexão pode ser intermediada por fios de cobre, fibras ópticas, microondas e ondas de infravermelho. Um exemplo de infraestrutura de redes de computadores é a Internet que provê uma rede de redes de computadores, fornece serviços a aplicações e é a maior existente até então. “A Internet é uma rede de computadores que interconecta milhares de dispositivos computacionais ao redor do mundo” (KUROSE, 2005).

A Internet possibilita a troca de informações instantâneas entre pontos distantes da rede. Atualmente, é possível realizar serviços simultâneos em tempo real e em diferentes computadores, como a edição de um arquivo, a realização de vídeos conferências, a troca de mensagens, entre outras. “Organizações com centenas de escritórios dispersos por uma extensa área geográfica normalmente esperam, com um simples pressionar de um botão, poder examinar o *status* atual até mesmo de suas filiais mais remotas” (TANENBAUM, 2003).

A coleta e fornecimento de dados na rede podem ser realizados pelas aplicações conforme um dos modelos de arquitetura a seguir: o cliente-servidor e o *peer-to-peer*.

2.1.1 Modelo Cliente-Servidor

O modelo cliente-servidor estabelece uma relação hierárquica entre servidores e clientes. Servidores são os pontos da rede responsáveis por receber requisições de clientes e fornecer respostas a elas. Sendo assim, os clientes são os pontos da rede que enviam requisições aos servidores. Os servidores detém a lógica de processamento central da aplicação e são capazes de receber requisições de um ou de vários clientes.

Basicamente, os servidores são responsáveis por prover serviços aos clientes. Esses clientes e servidores são processos, ou seja, programas em execução em sistemas computacionais distintos. Um programa cliente é o solicitante da requisição já o programa servidor é quem receber essa solicitação e provê o serviço ao cliente. O fluxo deste modelo está ilustrado na Figura 1.

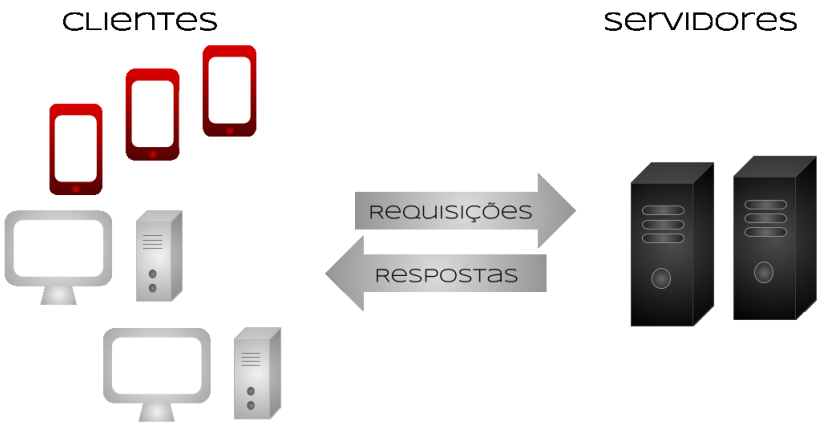


Figura 1 – Modelo cliente-servidor.

Este modelo funciona em pequenas e grandes distâncias geográficas entre os servidores e os clientes que estão requisitando-os. De acordo com Tanenbaum (2003), o modelo cliente-servidor é amplamente usado e constitui a base da grande utilização da rede. O *Google* é um exemplo de aplicação que segue o modelo cliente-servidor. Os clientes solicitam pesquisas e o servidor retorna os resultados baseados nos caracteres inseridos na requisição.

2.1.2 Modelo *peer-to-peer*

O modelo *peer-to-peer* é um modelo não hierárquico e não depende de um servidor disponível para seu funcionamento. Todo ponto da rede pode funcionar como cliente ou como servidor, ou seja, pode solicitar ou responder requisições. Este ponto da rede é

denominado como par nesta arquitetura. Utilizando como exemplo o compartilhamento de um arquivo em uma aplicação *peer-to-peer*, o fluxo funciona da seguinte forma:

- Um par inicialmente conterá o arquivo e o denominaremos de x.
- Um outro par, que denominaremos de y, solicita este arquivo de x.
- Portanto, após a transmissão do arquivo, ele estará contido em x e em y.
- Um outro par z pode solicitar o arquivo tanto de x quanto de y. E posteriormente, outros pares podem solicitar este mesmo arquivo de z, x ou y e assim sucessivamente.

Dessa forma, a responsabilidade de prover serviços aos clientes não é centralizada e isso reduz os custos referentes a infraestrutura necessária para manter servidores dedicados. Kurose (2005) afirma que a arquitetura *peer-to-peer* possui autoescalabilidade inerente a ela, consequência do papel duplo dos nós da rede. A estrutura da rede formada por um aplicação *peer-to-peer* forma um grafo bidirecionado, portanto um par pode socilitar informações contidas em um outro par conectado indiretamente a ele. A Figura 2 ilustra esta estrutura, onde os vértices representam as requisições entre os pares.

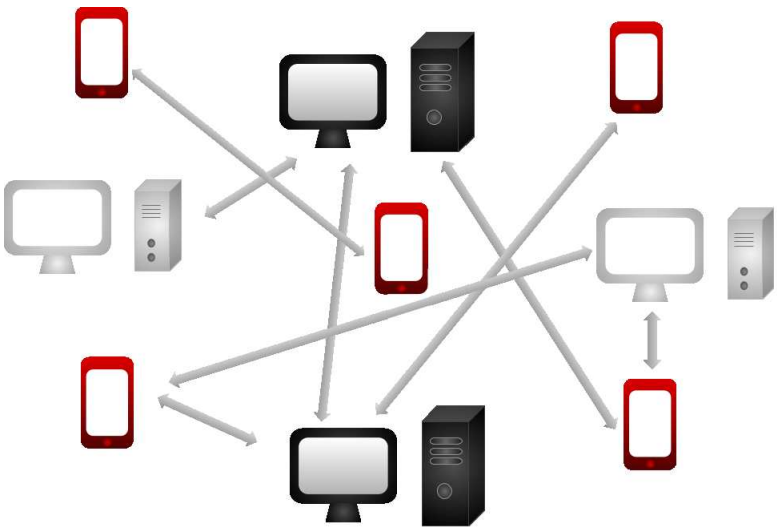


Figura 2 – Modelo *peer-to-peer*.

O *BitTorrent* é um exemplo de aplicação *peer-to-peer* para *download* de arquivos.

2.2 Redes de Computadores

Para que um pacote[1] trafegue na rede de um computador a outro, é necessário que ele seja gerenciado por vários protocolos contidos na rede de computadores. Esses protocolos definem a convenção da comunicação entre os nós da rede, ou seja, determinam quais mensagens devem ser recebidas ou emitidas para que alguma ação desejada ocorra. Por exemplo, quando um cliente deseja acessar uma página *web*, ele deve emitir uma mensagem para o servidor com que deseja estabelecer uma conexão; o servidor irá receber essa mensagem e deferir a conexão. Caso a conexão esteja estabelecida, o cliente manda uma mensagem requisitando a página e então o servidor retorna o conteúdo da página. Toda comunicação que ocorre na rede é controlada por protocolos que definem regras para as características das mensagens trocadas entre dois pontos da rede como as ações que devem ser realizadas, os campos que devem ser preenchidos e com o que devem ser preenchidos, o tamanho, entre outras.

A rede é composta por camadas de protocolos, cada protocolo é dedicado a um tipo de serviço. A arquitetura da rede de computadores é uma arquitetura de camadas, isto é, todos os serviços são modularizados. Cada camada possui os protocolos referentes aos serviços que oferecem. Dessarte, a arquitetura é composta por camadas de protocolos empilhadas. Uma camada só se comunica com as suas camadas adjacentes. Essa abordagem permite que a refatoração de um serviço não interfira na implementação dos outros, porém podem ocorrer funcionalidades repetidas e dependências entre as camadas que não permita a modularização por completo. No modelo TCP/IP utilizado pela Internet, existem cinco camadas: a de aplicação, de transporte, de rede, de *enlace* e a física.

2.2.1 Camada de Aplicação

As aplicações consistem nos *softwares* que oferecem serviços aos usuários nos sistemas finais. O *Google*, o *Facebook* e o *Linkedin* são exemplos de aplicações, cada uma delas oferece um tipo de serviço diferente e necessita da premissa de que será executado em sistemas finais que devem comunicar entre si. Através das linguagens de programação, é possível implementar essas aplicações. “Uma linguagem de programação é um método padronizado que usamos para expressar as instruções de um programa a um computador programável” (GOTARDO, 2015). C#, *Ruby*, *Python*, *Clojure*, *Java*, *Haskell*, C, C++ são exemplos de linguagens de programação.

Na implementação de uma aplicação, não é necessário preocupar-se em desenvolver funcionalidades da arquitetura de rede (a arquitetura de camadas descrita na Seção 2.2),

[1] “Quando um sistema final possui dados para enviar a outro sistema final, o sistema emissor segmenta esses dados e adiciona *bytes* de cabeçalho a cada segmento. Os pacotes de informações resultantes, conhecidos como pacotes no jargão de redes de computadores, são enviados através da rede ao sistema final de destino, onde são reagregados aos dados originais” (KUROSE, 2005).

pois essas são estáticas. É necessário somente projetar a arquitetura da aplicação que determina como a aplicação irá proceder nos sistemas finais. Os principais modelos de arquitetura de aplicações são o modelo cliente-servidor (Seção 2.1.1) e o modelo *peer-to-peer* (Seção 2.1.2). Existem também aplicações híbridas que implementam os dois modelos.

A camada de aplicação possui os protocolos responsáveis por gerenciar a comunicação de diferentes processos da aplicação em diferentes computadores, especificamente como um processo irá receber ou enviar uma mensagem. Essa comunicação é realizada através de interfaces denominadas *sockets*, que consistem em primitivas de comunicações entre processos que, através de identificadores dos processos, entregam e recebem mensagens entre eles. Processos são programas em execução que podem ser classificados como clientes ou servidores, clientes são os que enviam as requisições primeiramente e servidores são os que recebem essas requisições. Na arquitetura P2P, os processos também são classificados como clientes ou servidores dependendo do seu comportamento atual, ou seja, se mandou a requisição primeiro ou se a recebeu.

As principais funções dos protocolos da camada de aplicação são:

- Definir um modelo para identificação de mensagens de requisição e de resposta.
- Definir o padrão dos campos da mensagens de cada tipo.
- Definir o significado dos valores contidos nos campos das mensagens.
- Definir regras para o comportamento de um processo, ou seja, se ele deve mandar ou receber mensagens.

Um exemplo de protocolo da camada de aplicação é o *HyperText Transfer Protocol* (HTTP). O HTTP define como devem ocorrer as requisições de solicitação e resposta das páginas *web*, portanto, atua tanto nos processos clientes quanto nos processos servidores.

2.2.1.1 *Proxy*

Um *proxy* é um servidor que intermedia a conexão entre uma rede interna de usuários e a Internet. Todas as requisições de um usuário que pertence a rede interna serão repassadas ao *proxy* antes de completar o seu trajeto ao destinatário da requisição. O *proxy* é responsável por administrar o que entra e o que sai da rede interna. Por essa razão, um *proxy* pode: funcionar como um filtro para as requisições, prover anonimato dos dados das requisições, se comportar como um servidor de dados compartilhados entre os clientes da rede interna, entre outros. “Os *proxies* foram inventados para adicionar estrutura e encapsulamento aos sistemas distribuídos” (PROXY, 2017).

2.2.2 Camada de Transporte

A camada de transporte possui os protocolos responsáveis por prover o serviço de movimentação de dados de um processo de um sistema final para a rede e da rede para um processo de outro sistema final.

Resumidamente, os serviços da camada de transporte são:

Demultiplexação: Receber os segmentos[2] da camada de rede e os entregar para um determinado processo identificado por uma porta.

Multiplexação: Receber dados de processos e os encapsular[3] em segmentos e entregar os segmentos para a camada de rede.

Portanto, a camada de transporte provê a comunicação lógica entre os processos de diferentes sistemas finais. A comunicação lógica permite que esses sistemas finais entendam que estão conectados diretamente entre si.

Os principais protolocos da camada de transporte na Internet são o *Transmission Control Protocol* (TCP) e o *User Datagram Protocol* (UDP). O TCP e o UDP fornecem serviços de multiplexação e demultiplexação na camada de transporte. De acordo com Kurose (2005), o TCP fornece um serviço de entrega confiável garantindo a entrega, a integridade e a ordem dos pacotes, controla possíveis congestionamentos na rede (tráfegos de pacotes excessivos) e é orientado para conexão. Já o UDP provê um serviço não confíavel e não orientado para conexão, ou seja, não é necessário que os processos se "cumprimetem" antes de trocar dados.

2.2.3 Camada de Rede

A transferência de mensagens entre os processos de diferentes hospedeiros são entregues na rede pelos protocolos da camada de transporte, esses protocolos não regem o comportamento do pacote no núcleo da rede. A movimentação no núcleo da rede é de responsabilidade do serviço dos protocolos da camada de rede. A camada de rede recebe os segmentos da camada de transporte e os encapsula em datagramas[4], além de receber datagramas e os encapsular em segmentos para entregar para a camada de transporte.

As duas principais funções da camada de rede é o repasse e o roteamento. Segundo Kurose (2005), o repasse consiste em transferir um dado de um hospedeiro remetente para um roteador mais próximo x, de x para um outro roteador y mais próximo do outro hospedeiro e de y para o hospedeiro destinatário. E o roteamento é responsável por traçar

[2] Pacotes da camada de transporte.
[3] Adicionar campos da respectiva camada no cabeçalho do pacote.
[4] Pacotes da camada de rede.

a melhor rota, através de algoritmos de roteamento, que um datagrama deve percorrer entre os hospedeiros, pois devido à extensão da rede existem várias possibilidades de rotas a serem percorridas entre dois pontos da rede.

O Internet *Protocol* (IP) é o principal protocolo da camada de rede. "Ao contrário da maioria dos protocolos da camada de rede mais antigos, o IP foi projetado desde o início tendo como objetivo a interligação de redes" (TANENBAUM, 2003). O protocolo IP empenha-se em fornecer um transporte de pacote bem-sucedido entre quaisquer pontos da rede, porém não há garantias de entrega.

2.2.4 Camada de *Enlace*

Para apresentar esta camada é necessário definir o que é um *enlace*. Um *enlace* é o canal de comunicação entre dois nós próximos da rede de computadores. Portanto, o serviço dessa camada é a transferência de quadros[5] de um nó da rede diretamente a outro. O fluxo deste serviço funciona da seguinte forma: um nó remetente deve apreender um datagrama, o encapsular em um quadro e transmitir a outro nó que irá extrair o datagrama do quadro recebido. Um protocolo da camada de *enlace* só se responsabiliza pelo transporte de um para outro em um único *enlace*. Sendo assim, um datagrama pode percorrer a rede por diferentes protocolos da camada de *enlace*.

Segundo Kurose (2005), além de transmitir quadros de um nó para outro, os protocolos dessa camada oferecem outros serviços, como:

Enquadramento de Dados: Receber um datagrama e adicionar campos no cabeçalho para que se torne um quadro.

Acesso ao *enlace*: Definir as regras e o fluxo que a transmissão de dados entre os *enlaces* deve cumprir. Um exemplo de protocolo que implementa este serviço é o *Medium Acess Control Protocol* (MAC).

Entrega confiável: Garantir a entrega de um quadro de um *enlace* a outro.

Controle de Fluxo: Controlar o tráfego de quadros entre os *enlaces*.

Detecção de Erros: Detectar se durante a transmissão houve algum tipo de erro como a violação da integridade do quadro transmitido.

Correção de Erros: Detectar onde o erro ocorreu e o corrigir se possível.

***Half-duplex* e *Full-duplex*:** Determinar a capacidade de transmissão de um nó. Se for *half-duplex*, um nó poderá somente receber ou enviar um pacote. Se for *Full-duplex*, poderá receber e enviar simultâneamente.

[5] Pacotes da camada de *enlace*.

Um dos principais protocolos da camada de *enlace* é a Ethernet. “Desde sua invenção, até meados da década de 1970, a Ethernet continuou a se desenvolver e crescer e conservou sua posição dominante no mercado” (KUROSE, 2005). A Ethernet oferece um serviço não confiável e não orientado para conexões.

2.2.5 Camada Física

Toda a infraestrutura física dos canais da rede como cabos coaxiais, fibras-óticas, satélites, espectro eletromagnético compõem a camada física. “A camada física é o alicerce sobre o qual a rede é construída” (TANENBAUM, 2003). O serviço que a camada física oferece é a transferência de cada *bit* de um quadro através de um *enlace*. A transferência de *bits* ocorrem de formas distintas e os protocolos devem ser específicos a uma estrutura. De acordo com Kurose (2005), a Ethernet possui vários protocolos da camada física como um para fios de cobre trançado, um para cabos coaxiais, entre outros.

O fluxo de uma requisição e de uma resposta na rede pela Internet está ilustrado na Figura 3. Todos os dados são tratados e conduzidos pelos serviços de todas as camadas para tornar possível a comunicação entre diferentes sistemas computacionais.

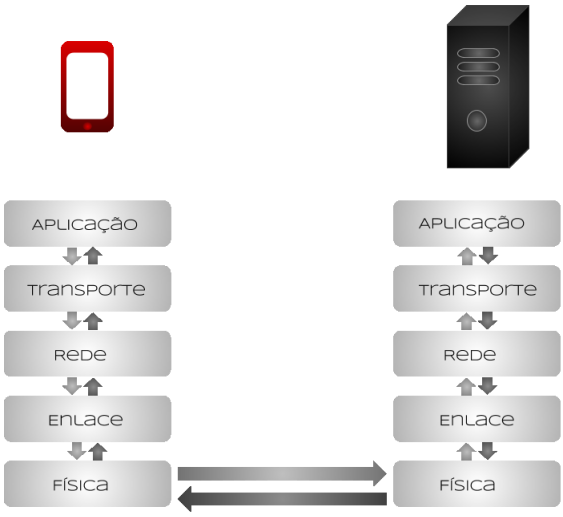


Figura 3 – Arquitetura de camadas TCP/IP.

2.3 Segurança na Rede

A segurança é uma preocupação geral dos sistemas e pessoas. Dispositivos como cadeados e cercas elétricas comprovam que mecanismos para proteção da privacidade e da propriedade são desenvolvidos há tempos. A segurança é um assunto que engloba várias áreas desde de tecnologias aplicadas à *softwares* e *hardwares*, às leis que asseguram uma proteção legal, a processos organizacionais e à psicologia. É imediato concluir que

segurança é um aspecto sistêmico crítico, pois pode afetar fatores como a saúde humana, a economia, em conformidade com o sistema a qual está inserida.

Inicialmente, a segurança na rede de computadores não era considerada, devido ao seu uso restrito na época. Geralmente, era utilizada para conectar os poucos computadores com dispositivos externos como impressoras e para troca de mensagens internas em uma organização. Com a Internet, o uso da rede de computadores se expandiu exorbitantemente e consequentemente os serviços realizados através dela também. Esses serviços, utilizados por milhões de pessoas, começaram a requerer o sigilo dos dados por representarem transações críticas que envolviam informações sensíveis como os dados de uma transferência bancária. Além disso, assim como o número de usuários aumentou, o número de invasores buscando beneficiar-se com o conteúdo dos dados também. Portanto, a preocupação em desenvolver soluções para proporcionar segurança aos usuários da rede se tornou presente.

Ademais, percebe-se que estamos cercados de sistemas eletrônicos conectados à Internet em nossas atividades diárias. Fechadura digital, aplicativos para transações bancárias e geladeiras são exemplos desses sistemas. Nota-se que vários sistemas que antigamente não ofereciam a interação com o usuário através de *softwares* pela Internet, como geladeiras e fechaduras, estão sendo modernizados para oferecerem seus serviços típicos e interagirem com a Internet. Portanto, são necessárias tecnologias que proporcionem segurança para esses sistemas. Com o extenso número de usuários da rede, serviços como transações bancárias, compras em lojas virtuais, consultas de resultados de exames médicos, matrículas em escolas e outros que exigem dados pessoais sigilosos, tornou a segurança da rede uma característica essencial.

Ataques à sistemas conectados a Internet são muito comuns, conclui-se então que a engenharia de segurança adotada em vários sistemas atualmente não é suficiente. Vários sistemas são vulnerabilizados com frequência e os erros cometidos por um, são repetidos por outros constantemente. Para resolver esses problemas, surgiu o campo de estudo da engenharia de segurança virtual. De acordo com Anderson (2010), a essência da engenharia de segurança virtual é detectar todas as potenciais ameaças do seu sistema e projetar mecanismos de defesa para essas ameaças identificadas. Estudar o que já deu errado e o que já deu certo em outros sistemas é um bom guia para detectar possíveis ameaças. Doravante, restrinjamos as discussões a sistemas de comunicações eletrônicas que usam a Internet como meio de comunicação.

Segundo Tanenbaum (2003), para que uma rede seja segura, os principais fatores que ela deve garantir durante as transações de dados são sigilo, autenticação, não-repúdio e controle de integridade. O sigilo é relacionado à privacidade dos dados contidos na requisição, somente pessoas autorizadas devem obter acesso aos dados. A autenticação é a garantia da identidade do remetente e do destinário; a rede deve certificar que o dado

realmente chegou ao nó da rede especificado na requisição. O não-repúdio é a garantia da realização de uma transação por um usuário e está relacionado a assinaturas. Por fim, o controle da integridade previne e certifica que o dado enviado não foi adulterado durante seu trajeto ao destinatário.

Alguns protocolos, como o TCP, já oferecem serviços como o controle de integridade, porém não proveem autenticação, sigilo e não-repúdio. Portanto, soluções foram estruturadas para prover a segurança providenciando os serviços que contemplam os fatores citados acima.

Algumas dessas soluções são a criptografia (Seção 2.3.1), o *Transport Layer Security* (Seção 2.3.2) e o *Virtual Private Network* (Seção 2.3.3).

2.3.1 Criptografia

A criptografia tem como principal objetivo evitar que um intruso consiga capturar o conteúdo de um pacote e o compreender. De forma resumida, a criptografia funciona da seguinte forma: um texto em claro P é alterado através de algum algoritmo de criptografia $E_K(x)$ para se tornar um texto cifrado C. Geralmente, esse algoritmo de criptografia envolve uma chave K que só o remetente e o destinatário da mensagem devem conhecer de modo a cifrar e decifrar o conteúdo. Tanenbaum (2003) demonstra esse fluxo através das Equações 2.1, 2.2 e 2.3.

$$C = E_K(P) \tag{2.1}$$

A Eq. 2.1 representa a operação da cifração. O texto em claro P é cifrado por uma função E com a chave k. O resultado da cifração é o texto cifrado C.

$$P = D_K(C) \tag{2.2}$$

A Eq. 2.2 representa a operação de decifração. O texto cifrado C é processado pela função $D_K(x)$ com a chave k, o que gera o texto claro P.

$$D_k(E_k(P)) = P \tag{2.3}$$

Em linhas gerais, uma função E que recebe o texto em claro P e o cifra com a chave K deve possuir uma função de decifração D que, com a mesma chave K, gerará o texto em claro P.

Tanenbaum (2003) afirma que a criptografia tem dois princípios fundamentais: a redundância e a atualização. A redundância é necessária para combater que intrusos possam alterar o conteúdo e isto passar despercebido pelo destinatário. Já a atualização

combate a duplicação de conteúdos, ou seja, garante que aquela mensagem recebida é atual e não uma cópia enviada por um intruso tempos depois.

Um questionamento válido acerca da criptografia é se o algoritmo pode ser conhecido por terceiros e continuar garantindo o sigilo dos textos cifrados. De acordo com Kerckhoffs (1883), o algoritmo pode ser partilhado mas a chave deve ser confidencial de forma que não seja possível através da criptoanálise[6] desvendar um conteúdo cifrado sem a chave. Essa premissa é conhecida como o princípio de *Kerckhoff*. O fator que deve ser levado em conta é o tamanho da chave, pois quanto maior a chave, menor é a possibilidade de um texto cifrado ser descriptografado devido as inúmeras possibilidades que deverão ser reproduzidas. A possibilidade de decodificação da chave é exponecial em relação ao seu tamanho.

A criptografia que utiliza uma chave única compartilhada entre os dois pontos comunicantes é considerada segura, pois não possui nenhuma informação da mensagem atrelada à ela. Os algoritmos de chave simétrica consistem na utilização de uma mesma chave para a codificação e decodificação de uma mensagem. Apesar de ser segura, essa abordagem oferece algumas desvantagens como a possibilidade de captura da chave por terceiros durante o compartilhamento da mesma, dificuldade de armazenamento do conteúdo, pois a extensão do conteúdo aumenta em consequência do tamanho da chave e a possibilidade de falta de sincronismo entre o remetente e o destinatário que pode danificar o conteúdo original e o tornar ilegível. Um exemplo de algoritmo de chave simétrica é o Rijndael que, de acordo com Tanenbaum (2003), utiliza permutações e substituição.

Para sanar os problemas citados acima, os algoritmos de chave pública utilizam dois tipos de chaves: as chaves públicas e as chaves privadas. As chaves públicas podem ser acessadas por todos, já as chaves privadas são exclusivas dos seus donos. Portanto, não há um compartilhamento de uma chave única na rede que deve ser sigilosa. A chave pública é utilizada para cifrar a mensagem e a chave privada para decifra-la. A chave privada não é derivada da chave pública ou vice-versa tornando impossível a geração de uma chave privada através de uma chave pública.

Supondo que X quer mandar uma mensagem para Y, X pega a chave pública de Y $\mathbf{E_y}$ e envia $\mathbf{E_y}(\mathbf{mensagem})$ à Y, Y recebe a mensagem cifrada e, com sua chave privada $\mathbf{D_y}$, calcula $\mathbf{D_y}(\mathbf{E_y}(\mathbf{mensagem}))$ resultando no conteúdo decifrado enviado por X. O RSA é um exemplo de algoritmo de chave pública que "sobreviveu a todas as tentativas de rompimento por mais de um quarto de século e é considerado um algoritmo muito forte" (TANENBAUM, 2003).

[6] "A arte de solucionar mensagens cifradas" (TANENBAUM, 2003).

2.3.1.1 Curvas Elípticas em Criptografia

De acordo com Hankerson (2006), há três características principais que devem ser avaliadas durante a escolha de um esquema de chaves públicas: a funcionalidade, a segurança e o desempenho. Portanto, deve ser avaliado se o esquema oferece à aplicação o que é necessário em termos de funcionalidades e se a segurança provida é ideal para a aplicação assim como o desempenho.

O desempenho de um algoritmo deve ser medido em relação aos recursos que gasta durante a sua execução. Exemplos de recursos são o tempo, a memória e o processamento. O desempenho dos algoritmos podem ser classificadas em lineares ou exponenciais. As exponenciais podem ser sub-exponenciais ou totalmente exponenciais que possui um desempenho pior do que a sub-exponencial. Essa classificação é obtida através da medição de quantas operações o algoritmo faz no pior caso e de que forma elas se propagam em relação ao recurso (pode ser o tempo, memória, entre outros). Nota-se que as exponenciais se propagam de uma forma bem mais rápida do que as lineares, por isso são piores em relação o desempenho. Por exemplo, um algoritmo com desempenho exponencial com um *input* de tamanho n demanda muito mais recurso do que um algoritmo com desempenho linear com um *input* do mesmo tamanho n.

Existem três tipos principais de esquemas de chaves públicas: a fatoração de inteiros em sistemas RSA, o logaritmo discreto para sistemas DL (*Discrete Logarithm*) e o logaritmo discreto com o uso de curvas elípticas para sistemas EC (*Elliptic Curve*). Para analisar o desempenho desses esquemas em relação à memória, é necessário conhecer o desempenho do melhor algoritmo daquele sistema para um *input* de tamanho satisfatório de *bits* que garanta a segurança necessária ao sistema.

O algoritmo para fatoração de inteiros possui como paramêtro um número N que é um produto de dois números primos de tamanho $\frac{\mathbf{L}}{\mathbf{2}}$ *bits*. Portanto, a entrada desse algoritmo possui um tamanho de L *bits*. O algoritmo mais eficiente para solucionar a fatoração é o *Number Field Sieve* (NFS) que é um algoritmo subexponencial. De acordo com Hankerson (2006), desde de 2003 o maior número fatorado com o NFS possuía um tamanho de 530 *bits*.

O algoritmo para o logaritmo discreto possui como parâmetros P e Q onde Q é um divisor primo de P-1. P possui um tamanho de L *bits* e Q de T *bits*. O tamanho da entrada do algoritmo é de l *bits*. O algoritmo mais eficiente para o logaritmo discreto também é o NFS que se comporta da mesma forma. De acordo com Hankerson (2006), desde de 2003 o maior número solucionado com o NFS para logaritmo discreto possuía um tamanho de 397 *bits*.

Por fim, o algoritmos para o logaritmo discreto com curvas elípticas possui como paramêtro um inteiro $\mathbf{D} \in$ [1, n-1] de tal modo que $\mathbf{Q} = \mathbf{dP}$ onde n é um primo com

tamanho de T *bits* e P é um ponto de ordem n de uma curva elíptica definida sobre um campo finito $\mathbb{F}_{\mathbf{p}}$ e $\mathbf{Q} \in \langle \mathbf{P} \rangle$. O tamanho da entrada é de T *bits*. O algoritmo com melhor desempenho é o *Pollard's rho algorithm* que pode ser paralelizado para otimização do uso dos recursos durante a execução otimizando o processamento. Segundo Hankerson (2006), desde de 2003 o maior número solucionado com *Pollard's rho algorithm* possuía um tamanho de 109 *bits*.

Conclui-se então que o tamanho necessário de entrada no algoritmo é menor no esquema que utiliza as curvas eliptícas. Conforme Hankerson (2006), as vantagens são de que as chaves e os certificados são menores e os algoritmos com menores paramêtros são mais rápidos ou seja possuem melhor desempenho em relação aos recursos de memória, tempo e processamento.

2.3.1.2 *End-to-end Encryption* (E2EE)

Nos sistemas do modelo cliente-servidor (Seção 2.1.1), a segurança da aplicação é garantida somente do caminho entre o cliente ao servidor. Portanto, é possível que terceiros que estejam executando estes servidores possuam acesso a chave para decifrar o conteúdo trocado entre os usuários. A *End-to-end Encryption* consiste em estabelecer chaves para a criptografia que somente os pontos comunicantes possuam acesso, assegurando que terceiros não obtenham o conteúdo original. Esta tática também dificulta o ataque de *man in the middle*, pois a chave só é conhecida pelos dois pontos comunicantes originais.

Para que isso seja possível, a troca de chaves entre dois pontos da rede deve proceder para que as chaves privadas nunca sejam compartilhadas, como no protocolo *Diffie-Hellman*. Este é um protocolo que define um método seguro para a troca de chaves em um canal público e inseguro de comunicação permitindo que dois pontos comunicantes na rede que não obtenham conhecimento prévio mútuo estabeleçam uma chave secreta de forma que cada parte possua uma chave que só pode ser acessada pelo seu proprietário.

2.3.1.2.1 *Diffie-Hellman key exchange*

O protocolo *Diffie-Hellman* utiliza um número primo P e um número G, onde G é uma raiz primitiva módulo P, ou seja, é um número A relativamente primo a P[7] que é congruente a uma potência de G módulo P, ou seja, para qualquer inteiro A relativamente primo a P, existe um inteiro k que satisfaz a Eq. 2.4 gerando a raiz primitiva G.

$$G^k = A \ mod \ P \tag{2.4}$$

[7] Um número A e P são relativamente primos se o maior divisor comum (MDC) entre eles é 1.

O primeiro passo na troca de chaves é acordar entre as partes os valores de P e G. A partir desses valores, serão gerados novas chaves através da Eq. 2.5, onde s é a chave privada de cada nó.

$$X = G^s \ mod \ P \tag{2.5}$$

Cada nó possuirá uma chave pública X gerada pela Eq. 2.5 e que será trocada entre as partes. Então, será calculado um novo segredo em comum a partir da Eq. 2.6, sendo X a chave recebida do outro nó.

$$S = X^s \ mod \ P \tag{2.6}$$

A Eq. 2.7 e a Eq. 2.8 demonstram que as chaves geradas na Eq. 2.6 são iguais, onde a e b são as chaves privadas secretas de cada nó e $\mathbf{X_a}$ e $\mathbf{X_b}$ são as chaves geradas pela Eq. 2.5.

$$X_a^b \ mod \ P = G^{ab} \ mod \ P = G^{ba} \ mod \ P = X_b^a \ mod \ P \tag{2.7}$$

$$(G^a \ mod \ P)^b \ mod \ P = (G^b \ mod \ P)^a \ mod \ P \tag{2.8}$$

A chave gerada pela Eq. 2.6 poderá ser utilizada para cifrar mensagens enviadas entre os nós. Nota-se que, durante todo o procedimento da troca de chaves, a chave privada nunca é compartilhada sendo permanecida em segredo.

2.3.2 *Transport Layer Security* (TLS)

De acordo com Kurose (2005), o TLS, denominado *Secure Socktes Layer* (SSL) em sua primeira versão, é um aperfeiçoamento do TCP, pois adiciona aos seus serviços a integridade dos dados e a autenticação do servidor e do cliente. Esses serviços impedem que os dados sejam adulterados por algum interceptor mal-intencionado e garante a autenticidade da identidade do remetente e do destinatário, ou seja, certifica que a transação não será redirecionada para nenhuma "cópia" do servidor ou do cliente.

Segundo Tanenbaum (2003), o TLS é uma nova camada posicionada entre a camada de aplicação e a camada de transporte como desmonstra a Figura 4.

Para exemplificar o fluxo do TLS, ilustrado pela Figura 5, o seguinte cenário será considerado: um nó da rede x envia uma requisição para outro nó y utilizando o TLS.

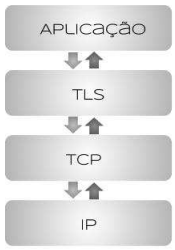


Figura 4 – Localização do TLS na arquitetura de camadas.

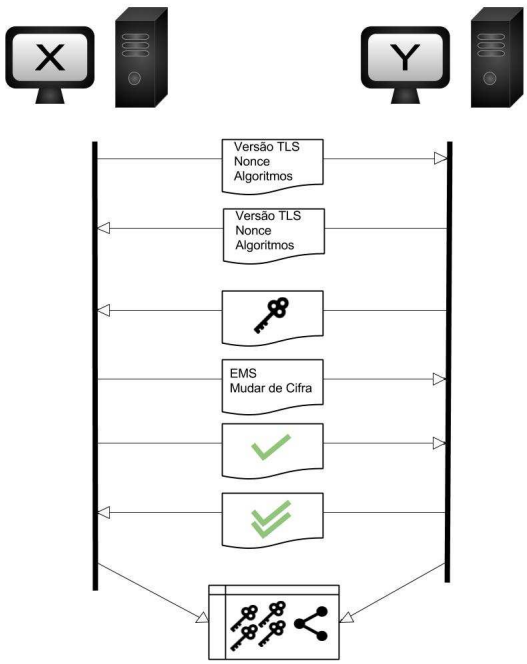


Figura 5 – Fluxo do TLS.

1. x deve se apresentar para y enviando a versão do TLS, um *nonce*[8] e uma lista de algoritmos criptográficos que suporta.

2. y irá responder x também com a versão do TLS, um *nonce* e os algoritmos da lista recebida que ele suporta.

3. y deve enviar um certificado[9] com sua chave pública.

4. x irá receber o certificado e enviar o Segredo Mestre Cifrado (EMS) que é uma chave cifrada através da chave pública de y e de um Segredo Mestre (MS) elaborado aleatoriamente por x.

5. x enviará uma requisição de mudança de cifra para y e o notificará que terminou o estabelecimento da conexão segura.

[8] "Os números aleatórios, usados apenas uma vez, são chamados *nonces*, que é uma contração de *"number used once"* (número usado uma vez)" (TANENBAUM, 2003).

[9] Certificado de chave pública emitido por uma *Key Distribution Center* (KDC).

6. y enviará para x duas confirmações: de término da mudança de cifra e do estabelecimento da conexão.

7. O EMS será utilizado para a geração de quatro chaves: duas chaves de criptografia e duas chaves MAC.

8. O MAC é um *hash* gerado por um algoritmo de *hash* combinado que depende de uma chave. O valor do MAC que é anexado a um número de sequência para evitar a duplicação de pacotes e depois é criptografado com o algoritmo de criptografia simétrica combinado entre x e y.

9. x e y enviam um MAC de todas as mensagens enviadas e recebidas anteriormente para evitar qualquer alteração das informações públicas compartilhadas entre x e y como a lista de algoritmos de criptografia.

10. A partir do passo 8, o MAC irá autenticar e cifrar todas as mensagens trocadas entre x e y.

11. Por fim, os pacotes serão transferidos pela conexão TCP.

Portanto, o fluxo passa por três fases: a de apresentação, a de derivação de dados e a de transferência de dados como Kurose (2005) confirma. Para o encerramento da conexão, o TLS deve indiciar a solicitação de término da conexão com um campo de tipo anexado ao pacote antes de ocorrer o encerramento padrão do TCP.

2.3.3 *Virtual Private Network* (VPN)

Uma rede privada pode oferecer o benefício do acesso exclusivo a usuários pré-autorizados, porém exige uma demanda de recursos dispendiosos. Uma forma de obter uma rede privada sob a Internet pública é pela implementação de uma Virtual Private Network VPN. As VPNs reduzem os custos de uma rede privada exclusiva e provê principalmente a autenticação e a integridades dos dados para os seus usuários evitando que um interceptor não-autorizado altere ou tenha acesso aos datagramas enviados.

De acordo com Tanenbaum (2003), uma VPN é uma rede *overlay*. A rede *overlay* é resultante de uma técnica de interligação de diferentes redes denominada tunelamento. O tunelamento possibilita a comunicação entre *hosts* de diferentes protocolos. Por exemplo, uma rede que utiliza um protocolo IPvx deseja enviar pacotes na Internet IPv4 para outra rede IPvx, para que isso seja possível, um roteador multiprotocolos[10] irá "encapar" o pacote IPvx com umcabeçalho IPv4 e o enviar como um pacote IPv4 até que chegue ao seu destino, onde o cabeçalho IPv4 será eliminado voltando a ser um pacote IPvx.

[10] "Um roteador que pode lidar com vários protocolos de rede é chamado de roteador multiprotocolos" (TANENBAUM, 2003).

Uma característica de segurança importante de uma rede *overlay* é que os pontos finais da rede que estão estabelecendo a comunicação não possuem acesso ao pacote durante o seu tunelamento.

A criação de uma VPN pode ser facilitada através de alguns *softwares* como o OpenVPN e o NeoRouter. Esses produtos facilitam a concepção, a configuração e o gerenciamento de VPNs.

2.4 Serviços de Anonimização

Os mecanismos de segurança na rede citados acima garantem o sigilo, a autenticação, o não repúdio e o controle de integridade mas não proporcionam o anonimato na rede ao usuário. É possível conhecer a identidade dos nós comunicantes da rede. Portanto, se alguém emitir um conteúdo sigiloso na rede, essa pessoa é identificada como autora da transação impedindo-a de obter uma privacidade íntegra durante o seu uso. Para sanar este problema foram desenvolvidos os serviços de anonimização.

Os serviços de anonimização são soluções em *software* que proveem o anonimato na rede. Segundo Pfitzmann (2001), o anonimato consiste na existência de um conjunto de objetos que são potencialmente semelhantes ao original dificultando a identificação do mesmo. Pfitzmann (2001) afirma que o anonimato tem dois pilares principais: a desvinculação e a inobservabilidade. A desvinculação consiste em dificultar que um interceptor infira qual a relação entre mensagens ou outros eventos que ocorram entre dois pontos da rede. E a inobservabilidade consiste em dificultar que um interceptor distingua qualquer objeto do mesmo tipo na rede.

Segundo Aranha (2006), as técnicas modernas utilizadas pelos serviços de anonimização são:

Descentralização: Subdivisão das responsabilidades do serviço de anonimização para evitar ataques a um ponto da rede centralizado. Um exemplo de aplicação da descentralização é a arquitetura *peer-to-peer* (Seção 2.1.2) nos serviços de anonimização.

Indireção: Todo ponto da rede atua como um roteador, recebendo e repassando pacotes. Dessa forma, uma requisição não percorrerá o trajeto direto entre o ponto de origem e o ponto de destino dificultando que interceptores saibam quais pacotes foram enviados ou recebidos naquele ponto.

Igualdade entre as mensagens: Igualdade dos tamanhos dos pacotes para evitar análises de correspondência do tamanho de pacotes que saem ou chegam em um ponto da rede. Como exemplo, caso um pacote de um determinado tamanho seja recebido em um ponto da rede e nenhum do mesmo tamanho foi retornado, infere-se que

este pacote é provindo de uma aplicação que está em execução naquele ponto da rede. Essa análise permite o ataque denominado correlação temporal. “A análise de tráfego utilizada para se tentar estabelecer relações entre os pacotes enviados e recebidos, com a finalidade de separar o tráfego originado e consumido localmente, caracteriza um ataque de correlação temporal” (ARANHA, 2006).

Injeção de ruído: Injeção de pacotes aleatórios na rede para que sejam confundidos com pacotes de requisições verdadeiras dificultando o discernimento do interceptor de qual pacote não é um ruído. Essa técnica auxilia na anonimização mas traz desvatagens como o possível congestionamento na rede.

Atraso no envio: Um pacote deve ser mantido por um determinado tempo antes de ser repassado. Essa técnica também evita a correlação temporal mas prejudica o desempenho da requisição e utiliza mais espaço na memória dos roteadores.

Sigilo: Aplicação da criptografia (Seção 2.3.1) nas requisições entre os nós para garantir a eficácia de todas as técnicas explicitadas anteriormente. A vantagem desta técnica é a dificuldade do interceptor em compreender o conteúdo da requisição que facilitaria o ataque, tendo como exemplo, o descarte de possíveis pacotes de ruídos que seguem um determinado padrão.

Nota-se que as técnicas de descentralização, indireção e atraso no envio estão relacionadas ao princípio de desvinculação e as técnicas de igualdade entre as mensagens, injeção de ruídos e sigilo ao princípio de inobservabilidade.

De acordo com Aranha (2006), as principais redes de anonimização são o *Freenet*, o *GNUnet* e o Tor (Seção 2.4.1). O *Freenet* é uma plataforma que utiliza a arquitetura *peer-to-peer* (Seção 2.1.2) para combater as censuras das requisições na rede e prover privacidade para os usuários. O *GNUnet* é outra plataforma que provê privacidade em aplicações distribuídas desenvolvidas utilizando-o como base.

2.4.1 Tor

O Tor é um *software* livre[11] cujo principal objetivo é garantir a privacidade de seus usuários na rede através do uso de servidores voluntários denominados *relays*. Uma requisição realizada através do Tor não vai direto do remetente ao destinatário, ela passa por vários *relays* por uma rota aleatória antes de chegar ao seu destino. Dessa forma, garante que nenhum interceptor possa acessar os dados contidos no cabeçalho dos pacotes que indicam algumas informações da requisição como a sua origem, o seu destino, o tamanho do conteúdo, entre outras. Além de permitir o acesso a conteúdo bloqueados.

[11] “Por *software* livre devemos entender aquele software que respeita a liberdade e senso de comunidade dos usuários. Grosso modo, isso significa que os usuários possuem a liberdade de executar, copiar, distribuir, estudar, mudar e melhorar o software” (FOUNDATION, 2017).

O acesso aos dados de um cabeçalho por um interceptor permite uma análise denonimada análise de tráfego. A análise de tráfego possibilita que os interceptores saibam, por exemplo, quais foram todos os nós da rede para qual um nó mandou uma requisição e qual destino é o mais frequente e assim consegue deduzir o comportamento e as preferências de um nó na rede. A título de exemplo, esses dados podem ser utilizados por empresas que desejam oferecer anúncios sem autorização prévia porque mesmo que o conteúdo esteja criptografado, as informações contidas no cabeçalho são públicas.

2.4.2 Como o Tor funciona?

Quando um cliente deseja acessar um determinado *site* através da rede Tor, ele consulta todos os *relays* disponíveis em um *directory server*. A partir disso, ele escolhe uma rota que passa por vários *relays* randômicos até chegar ao seu destino. Em cada passagem do pacote de um ponto a outro, exceto a passagem do último *relay* ao destino final, são negociadas chaves criptográficas entre o cliente e os *relays* para impedir o rastreamento da conexão pelos própios *relays* garantindo a anonimidade da transação. A Figura 6 ilustra o fluxo descrito acima.

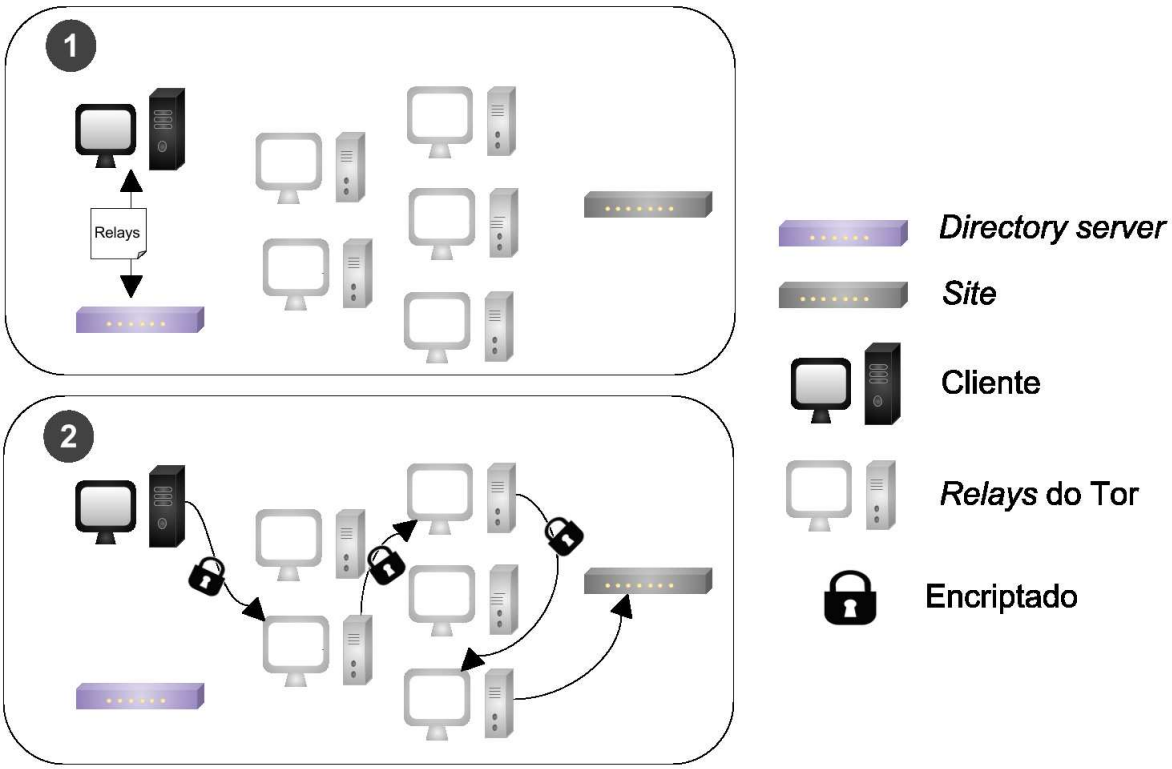


Figura 6 – Funcionamento do Tor.

2.4.3 Tor *Bridges*

Os *relays* do Tor são conhecidos publicamente, portanto, os provedores de Internet podem bloquear as conexões com os *relays*. Para isso, existem os *bridges* que são *relays* não contidos no diretório principal do Tor e que podem ser utilizados no caso deste bloqueio. Alguns *bridges* também podem ser localizados e bloqueados, neste caso é necessário localizar outros *bridges* que estão acessíveis.

2.4.4 Tor *Entry Guards*

Supondo que Alice deseja acessar um site X utilizando o Tor como serviço de anonimização e que Trudy consiga observar o tráfego da rede de Alice e do site X. Trudy pode correlacionar o volume e o tempo das informações que chegam e saem nos dois pontos. De acordo com Tor (2018), a probabilidade de um intruso que está observando C *relays* de um total de N *relays* existentes correlacionar os pacotes enviados por alguém que escolhe pontos randômicos como entrada é de $(\mathbf{C}/\mathbf{N})^{\mathbf{2}}$.

A solução provida para este possível ataque são os *entry guards*. Os *entry guards* são *relays* do Tor que funcionam como pontos de entrada para todas as requisições realizadas, ou seja, qualquer requisição do Tor passará primeiramente por ele antes de percorrer a rota randômica. Cada cliente seleciona alguns *relays* randômicos como *entry guards* que serão fixos durante um certo período de tempo, geralmente entre 2 a 3 meses e depois serão substituídos novamente.

Se um atacante não conseguir observar o *entry guard*, o usuário está seguro. Caso contrário, o atacante poderá observar o fluxo intenso do tráfego do usuário mas as chances de correlacionar as informações são bem menores com a probabilidade na ordem de $(\mathbf{N} - \mathbf{C})/\mathbf{N}$.

2.4.5 *Hidden Service*

O protocolo *hidden service* permite que serviços como *sites* e aplicações sejam publicados omitindo a sua localização.

De acordo com Tor (2017) e conforme ilustrado na Figura 7, o protocolo *hidden service* funciona da seguinte forma:

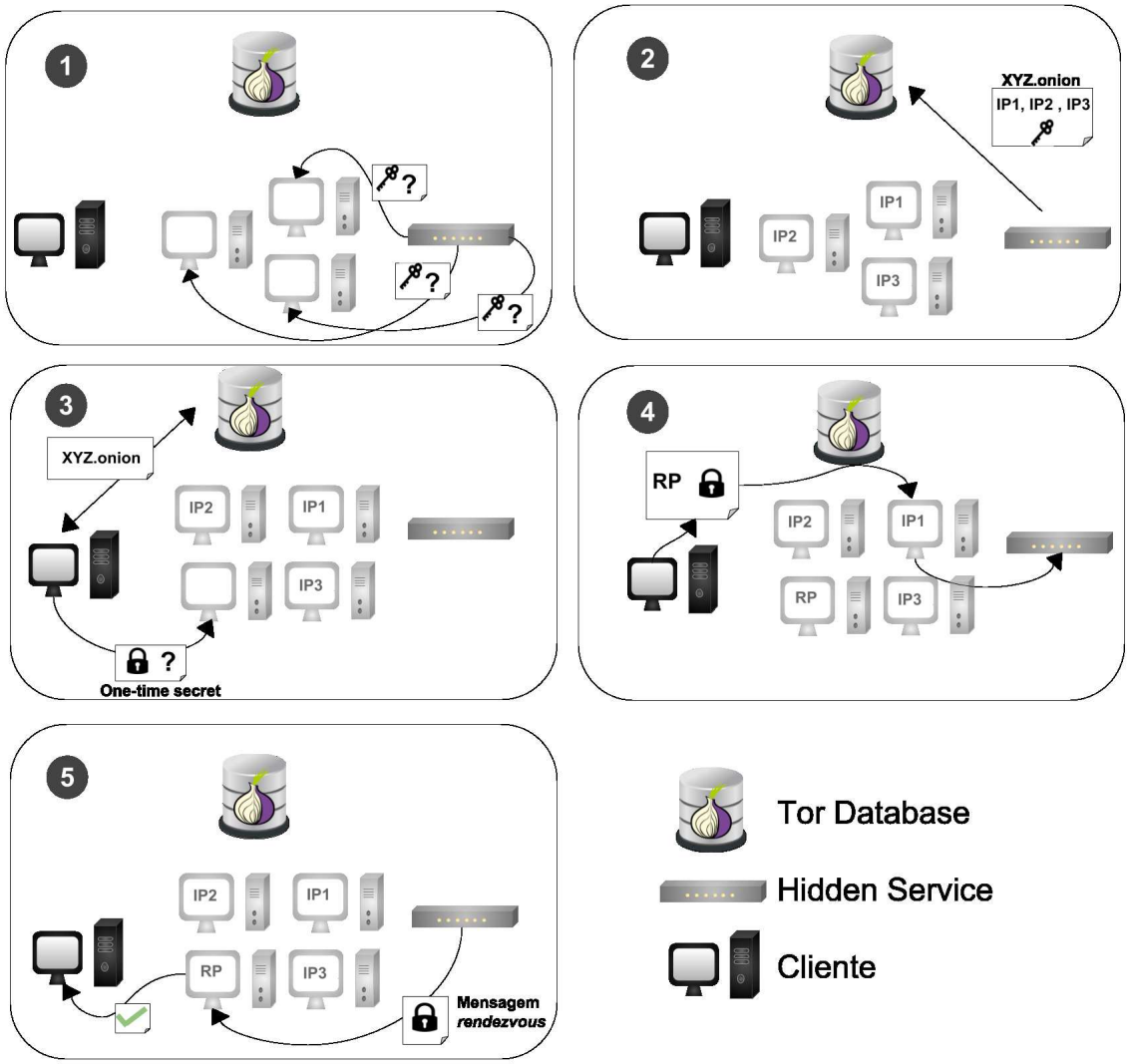


Figura 7 – Protocolo *Hidden Service*.

1. Para que os clientes possam dialogar com o *hidden service*, ele necessita notificar a sua presença na rede. Para isso, o *hidden service* monta uma rota de *relays* randômicos e solicita com a sua chave pública que eles ajam como *introductions points*. Percebe-se que os clientes não podem conhecer a localização do serviço somente a sua chave pública.

2. Uma descrição do serviço é criada contendo a sua chave pública e um sumário dos *introductions points*.

3. A descrição é assinada através da chave privada do serviço e enviada para uma *hash table* distribuída.

4. Essa descrição será encontrada por clientes que requisitam um XYZ.onion, onde o XYZ é um conjunto de 16 caracteres derivados provientes da chave pública do serviço. Dessa forma, o cliente pode identificar se está contactando o serviço correto.

5. Primeiramente, o cliente deve saber o endereço *onion* do serviço para que possa estabelecer uma conexão baixando a descrição contida na *hash table* distribuída.

6. Caso o serviço ainda esteja funcionando, ou seja, há uma descrição para o XYZ.onion, o cliente terá acesso aos *introductions points* e à chave pública do serviço.

7. O cliente cria uma rota randômica para um *relay* qualquer e solicita com um *one-time secret*[12] que ele haja como um *rendezvous point*.

8. O cliente monta uma mensagem de introdução encriptada pela chave pública do serviço que contém informações como o endereço do *rendezvous point* e o *one-time secret*.

9. O cliente manda essa mensagem para um *introduction point* requisitando que ela seja entregue ao *hidden service*. Importante ressaltar que não é possível relacionar o endereço do cliente à sua mensagem de introdução respeitando a sua anonimidade.

10. O serviço descriptografa a mensagem de introdução e cria uma rota ao *rendezvous point* do cliente e manda uma mensagem *rendezvous* que contém o one-time secret.

11. Caso o passo anterior seja bem-sucedido, o *rendezvous point* e o cliente estão efetivamente conectados. Portanto, eles podem se comunicar entre si através das rotas randômicas escolhidas por eles até o *rendezvous point* que retransmite as mensagens entre o cliente e o serviço do qual não conhece a identidade.

12. Basicamente, a conexão utiliza seis *relays*: três escolhidos pelo cliente (incluindo o *rendezvous point*) e três escolhidos pelo serviço.

2.4.6 Tor no *Android*

Em busca de proporcionar segurança em aplicativos *mobile*, o Guardian Project desenvolve algumas soluções baseadas na rede de anonimização Tor como o Orfox, o Orbot e o NetCipher.

2.4.6.1 Orfox

O Orfox é um Tor *browser* para Android, portanto, permite a navegação *web* através do celular de forma segura. É uma adaptação do Tor *Browser* para proporcionar a compatibilidade com o Firefox para Android.

2.4.6.2 Orbot

O Orbot é um aplicativo de *proxy* que permite a utilização do Tor no Android proporcionando a anonimização no uso de outros aplicativos como o *Twitter*.

[12] Uma chave utilizada como uma senha na conexão.

2.4.6.3 NetCipher

O NetCipher é uma biblioteca para aplicações Android que busca o Orbot no celular do usuário e o habilita como *proxy* da aplicação, caso o Orbot não esteja instalado, solicita a instalação pela Google Play. Pode ser utilizado também para o uso do protocolo TLS e para a configuração de um *proxy* sem utilizar a rede Tor.

3 Proposta de Trabalho

A proposta de trabalho consiste no desenvolvimento de uma aplicação *mobile* de mensagens, em Android nativo, que proporciona o máximo de privacidade aos seus usuários. É importante ressaltar que o objetivo do trabalho é fornecer a segurança de sua aplicação de ponto-a-ponto, pois há mecanismos de segurança que devem ser adotados de forma específica às camadas (Seção 2.2) nas quais estão inseridos na rede. O Apêndice H possui um detalhamento das características de plataformas móveis e do sistema operacional Android utilizados no desenvolvimento da solução.

Primeiramente, é discutido a importância da anonimização na rede para garantir a privacidade dos usuários, depois são apresentados conceitos sobre aplicações de mensagens e exemplos de aplicações com propostas semelhantes.

3.1 Importância da Anonimização

Atualmente, os sistemas de *software* acessados na Internet tendem a solicitar ou captar informações pessoais de um usuário para estratégias com interesses econômicos e/ou políticos. Sendo essa captação por voluntariedade do usuário ou por observação comportamental através de contéudos explícitos do tráfego na rede. Isso implica que as informações privadas dos usuários estão cada vez mais difundidas e em muitos casos, essas informações são detidas por entidades que não capazes de protegê-las. Como exemplo, há o caso recente de vazamento de informações privadas pela empresa *Cambridge Analytica* que captou dados dos usuários do *Facebook* como noticiado em Facebook notifica usuários que tiveram dados vazados; 443 mil são no Brasil. Ademais, essas entidades violam os princípios de privacidade e liberdade dos usuários.

Com as informações explícitas de um pacote é possível traçar um padrão comportamental como os interesses do usuário, o seu posicionamento político, a sua universidade e/ou trabalho, o seu banco, as suas amizades, entre outros. Apenas analisando as páginas que esse usuário acessa na Internet. Sendo a privacidade deste usuário cassada retirando o poder de controle de suas próprias informações. Além disto, há ainda em muitos países a censura na Internet sendo o conteúdo acessado pelos cidadãos direcionado pelo governo. Ou seja, o governo controla os conteúdos acessados pelos usuários do seu país provocando a alienação da população e infringindo a liberdade dos mesmos. Aranha (2007) enfatiza que quando um entidade de poder controla a informação que a sociedade recebe e consequentemente obtém a capacidade de manipular as opiniões se estabelece como um regime de mentiras ou meias verdades, pois qualquer contradição aos ideais da entidade é banida.

É importante também desvincular a ação ao autor para que as pessoas sejam motivadas a compartilhar informações sem se preocuparem que sua identidade esteja vinculada à elas. De forma à proporcionar a liberdade de expressão dos usuários garantindo que possam compartilhar os conteúdos de seu interesse sem serem rastreados. De acordo com Marx (1999), essa importância se aplica, por exemplo, em caso de discussões em grupos de apoio para ajudar pessoas com vícios, com doenças/traumas psicológicos, doenças graves marginalizadas pela sociedade como a AIDS, em caso de perseguições, entre outros.

Os serviços de anonimização limitam as informações concedidas na rede e as desvincula ao autor, portanto, são responsáveis por garantir a privacidade e a liberdade do usuário na Internet. Dessa forma, os usuários podem controlar o acesso às suas informações e acessar conteúdos censurados por entidades de poder.

3.2 Aplicações de Mensagens

Aplicações de mensagens ou *chats* são *softwares* que possuem como principal funcionalidade o diálogo virtual entre os usuários em tempo real. Os primeiros *chats* da Internet funcionavam como salas de bate-papo que podiam ser filtradas por assuntos, por idade ou por região, os usuários entravam na sala de bate-papo e iniciavam a troca de mensagens com os integrantes da mesma sala. Em 1995, surgiu o mIRC que oferecia além dos canais públicos também filtrados por tema, a troca direta de mensagens entre dois usuários sem que nenhum terceiro usuário obtivesse acesso.

Ao longo do tempo, surgiram outros *chats* como o ICQ, o msn, o Yahoo *Messenger* que aprimoraram as funcionalidades deste tipo de *software*. E em 2009 surgiu o *WhatsApp*, um aplicativo *mobile*[1] de troca de mensagens amplamente utilizado até os dias de hoje. De acordo com *WhatsApp* (2017), o aplicativo conecta um bilhão de pessoas, envia 55 bilhões de mensagens, 4.5 bilhões de fotos e 1 bilhão de vídeos diariamente além de possuir suporte para 60 idiomas diferentes.

Com o grande volume de dados trocados e com o crescimento de atacantes mal-intencionados na rede, os usuários das aplicações de mensagens exigem a privacidade do seu conteúdo. Portanto, essas aplicações aprimoraram as suas funcionalidades de segurança a fim de oferecer confiabilidade para os seus usuários.

3.2.1 Aplicações seguras de mensagens

As aplicações seguras de troca de mensagens visam garantir a total privacidade para seus usuários de forma que os mesmos possam obter a liberdade de compartilhar

[1] Também possui uma versão *desktop* integrada com o aplicativo *mobile*.

qualquer conteúdo à sua escolha.

A solução proposta é uma aplicação de mensagens que utiliza serviços de segurança para proteger a privacidade e a liberdade de seus usuários. Esses serviços são, principalmente, o Tor como recurso de anonimização, as técnicas de criptografia e criptografia ponta-a-ponta e a troca de chaves presencial. O sistema garante que uma pessoa possa trocar mensagens com outra sem ser rastreada como usuária da aplicação na rede e portanto, sendo desvinculada com as mensagens enviadas. Ademais, o conteúdo das mensagens são inacessíveis a terceiros e é garantido a autenticidade do destinatário das mensagens pelo esquema de trocas de chaves presencial. A troca de mensagens pode ocorrer independente de bloqueios aplicados por entidades externas. Portanto, a aplicação garante que o usuário possa usufruir dos recursos de mensageria de forma anônima e segura.

Algumas das aplicações de mensagen focadas na privacidade dos usuários estão descritas abaixo pelos atributos Descrição, Documentação, Linguagem de Programação, Repositório e Portabilidade.

3.2.1.1 Bit Chat

Descrição: É uma aplicação de mensagens livre baseada na arquitetura *peer-to-peer*.

Documentação: Disponível em <https://bitchat.im/>.

Linguagem de Programação: C#.

Repositório: <https://github.com/TechnitiumSoftware/BitChatClient>.

Portabilidade: *Windows* 10/8.1/8/7/Vista e distribuições *Linux*.

3.2.1.2 FireFloo *Communication*

Descrição: É uma aplicação de mensagens baseada em várias tecnologias de encriptação de dados.

Documentação: Disponível em <http://firefloo.sourceforge.net/>.

Linguagem de Programação: C++.

Repositório: <https://svn.code.sf.net/p/firefloo/code/firefloo-code>.

Portabilidade: *Android, Jolla Sailfish, Ubuntu Mobile Devices, Mac, Windows* e *Linux*.

3.2.1.3 Riot

Descrição: É uma aplicação de mensagens com foco na criptografia dos dados.

Documentação: Disponível em <https://about.riot.im/>.

Linguagem de Programação: *JavaScript, Objective-C* e *Java*.

Repositório: <https://github.com/vector-im>.

Portabilidade: IOS, *Android, Browser, Windows*, Mac e *Linux*.

3.2.1.4 Tor *Messenger*

Descrição: É uma aplicação de mensagens que funciona na rede Tor e utiliza o protocolo *Instantbird* de mensagens desenvolvida pelo *Mozilla*.

Documentação: Disponível em <https://blog.torproject.org/tor-messenger-beta-chat-over-tor-easily>.

Linguagem de Programação: *Javascript*.

Repositório: <https://gitweb.torproject.org/tor-messenger-build.git>.

Portabilidade: *Linux, Windows* e Mac. As mensagens podem ser transportadas por aplicações de mensagens como Jabber (XMPP), IRC, *Google Talk, Facebook Chat, Twitter* e *Yahoo*.

3.2.1.5 Ricochet

Descrição: É uma aplicação de mensagens baseada na arquitetura *peer-to-peer* e funciona como um *hidden service* do Tor.

Documentação: Disponível em <https://ricochet.im/>.

Linguagem de Programação: C++.

Repositório: <https://github.com/ricochet-im/ricochet>.

Portabilidade: *Linux, Windows* e Mac.

3.2.1.6 Comparação entre as aplicações

A fim de relacionar com a aplicação desenvolvida, abaixo das descrições há uma comparação avaliando se as aplicações fornecem as mesmas características de segurança e se sim, como fornecem. As comparações da Tabela 1 analisam se as aplicações fazem o uso de serviços de anonimização, se implementam criptografia e criptografia ponta-a-ponta.

Tabela 1 – Comparação das aplicações.

	Messaging Application	Bit Chat	FireFloo Communication	Riot	Tor Messenger	Ricochet
Serviços de Anonimização	Utiliza o Orbot para prover os serviços de anonimização no Android.	Não utiliza serviços de anonimização. A abordagem da aplicação é a comunicação peer-to-peer.	Não utiliza serviços de anonimização.	Não utiliza serviços de anonimização.	Utiliza o Tor para prover os serviços de anonimização.	Utiliza o Tor para prover os serviços de anonimização.
Criptografia	Criptografa o conteúdo das mensagens utilizando o algoritmo de criptografia AES no modo Galois/Counter Mode (AES-GCM).	Criptografa o contéudo das mensagens com o algoritmo de criptografia AES.	Criptografa o contéudo das mensagens com o algoritmo de criptografia AES.	Criptografa o contéudo das mensagens com o algoritmo de criptografia AES.	Criptografa o contéudo das mensagens com os algoritmos SHA3, ed25519 e Curve15519.	Criptografa o contéudo das mensagens.
Criptografia ponta-a-ponta	Provê a criptografia ponta-a-ponta com o protocolo de troca de chaves Diffie-Hellman utilizando a curva elíptica Curve25519.	Provê a criptografia ponta-a-ponta com o protocolo de troca de chaves Perfect Forward Secrecy (PFS).	Provê a criptografia ponta-a-ponta.	Provê a criptografia ponta-a-ponta utilizando o protocolo Signal.	Não provê criptografia ponta-a-ponta.	Provê criptografia ponta-a-ponta com protocolo próprio.

4 Metodologia

4.1 Metodologia de Desenvolvimento

A metodologia de desenvolvimento adotada foi baseada em metodologias àgeis (Apêndice J) e adaptada ao contexto do projeto.

As principais práticas adotadas que devem ser ressaltadas são:

- O planejamento incremental.
- Os requisitos da aplicação em formato de estórias.
- O desenvolvimento do produto em *sprints* com duração de uma semana cada.
- O processo cíclico de desenvolvimento.

4.1.1 Processo de Desenvolvimento

O processo de desenvolvimento da aplicação demonstrada na Figura 8 no formato de *Business Process Model and Notation* (BPMN) representa as seis atividades realizadas em todas as *sprints*. Este processo é cíclico, portanto é realizado várias vezes conforme o número de *sprints* planejadas. Todas as atividades estão descritas abaixo de acordo com os seus respectivos atributos: artefatos de entrada, descrição e artefatos de saída. Os artefatos de entrada são os artefatos necessários para a execução da atividade, a descrição apresenta as ações que devem ser realizadas e os artefatos de saída são os artefatos produzidos após o término da atividade.

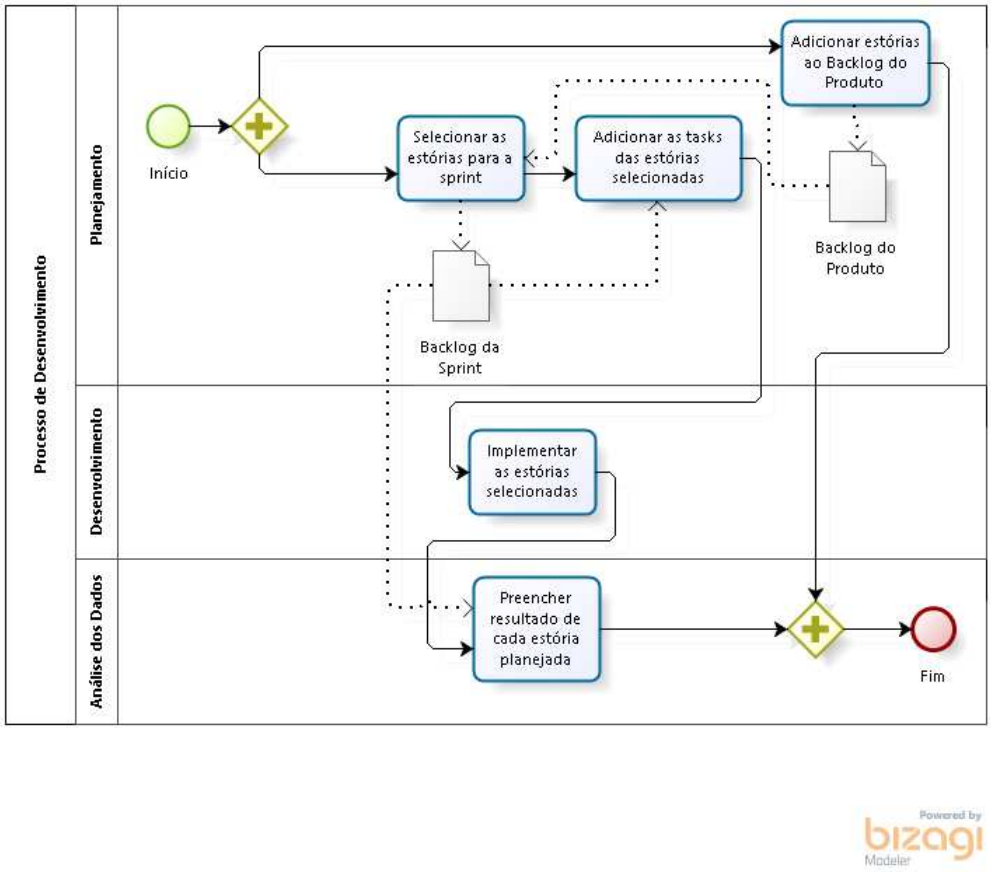


Figura 8 – Processo de Desenvolvimento.

4.1.1.1 Selecionar as estórias para a *sprint*

Artefatos de Entrada: *Backlog* do Produto.

Descrição: Seletar as estórias para que sejam desenvolvidas na *sprint* vigente.

Artefatos de Saída: *Backlog* da *Sprint*.

4.1.1.2 Adicionar as *tasks* das estórias selecionadas

Artefatos de Entrada: *Backlog* da *Sprint*.

Descrição: Adicionar tarefas detalhadas que devem ser realizadas para que a implementação da estória seja concluída.

Artefatos de Saída: *Backlog* da *Sprint*.

4.1.1.3 Implementar as estórias selecionadas

Artefatos de Entrada: *Backlog* da *Sprint*.

Descrição: Desenvolver as *tasks* descritas na atividade anterior e concluir a estória.

Artefatos de Saída: Não possui.

4.1.1.4 Preencher resultado de cada estória planejada

Artefatos de Entrada: Não possui.

Descrição: Analisar se todas as *tasks* foram concluídas e classificá-las como concluídas, parcialmente concluídas ou não concluídas.

Artefatos de Saída: *Backlog* da *Sprint*.

4.1.1.5 Adicionar estórias ao *Backlog* do Produto

Artefatos de Entrada: *Backlog* do Produto.

Descrição: Adicionar estórias ao *Backlog* do Produto. É importante ressaltar que esta atividade ocorre em paralelo ao processo de desenvolvimento, de forma contínua.

Artefatos de Saída: *Backlog* do Produto.

4.1.2 Artefatos de Desenvolvimento

Os artefatos produzidos no projeto estão nos Apêndices A e B.

4.1.2.1 *Backlog* do Produto

O *Backlog* do Produto contém todas as estórias do projeto classificadas em estórias de usuário e estórias técnicas. As estórias de usuários são funcionalidades da aplicação requisitadas a partir do ponto de vista de um usuário. A estória de usuário está no seguinte formato:

Eu, como usuário, desejo <ação> para <finalidade>.

Já as estórias técnicas são requisições do desenvolvedor para a aplicação. A estória técnica está no seguinte formato:

Eu, como desenvolvedor, desejo <ação> para <finalidade>.

Cada estória possui um identificador único, os identificadores das estórias de usuário são precedidos pelo prefixo US e os identificadores das estórias técnicas são precedidos pelo prefixo TS. Ambos, possuem dois dígitos após os respectivos prefixos.

4.1.2.2 *Backlog* da *Sprint*

O *Backlog* da *Sprint* possui as estórias planejadas de cada *sprint*. Em cada *sprint*, são preenchidos três campos: o identificador das estórias selecionadas para a *sprint*, as *tasks* e o resultado de cada estória.

O resultado de uma estória pode ser classificado como:

Concluída: Todas as *tasks* foram implementadas.

Parcialmente concluída: Parte das *tasks* foram implementadas.

Não concluída: Nenhuma das *tasks* foi implementada.

Cada categoria acima é identificada no documento, de acordo com a legenda da Figura 9.

Figura 9 – Legenda dos possíveis resultado de uma estória.

4.1.2.3 Cronograma

O cronograma (Tabela 2) possui as atividades principais do projeto e em que mês foram executadas. As atividades são: a revisão bibliográfica sobre o tema, a estruturação da fundamentação teórica, o desenvolvimento da aplicação, a elaboração da descrição da metodologia e a discussão acerca dos resultados do projeto.

Tabela 2 – Cronograma

	Agosto	Setembro	Outubro	Novembro	Março	Abril	Maio	Junho
Revisão Bibliográfica	X							
Fundamentação Teórica		X	X	X	X			
Desenvolvimento				X	X	X	X	X
Metodologia				X	X	X	X	X
Resultados e Discussões								X

5 Solução

Neste Capítulo, são discutidas as principais características da aplicação em duas Seções: Módulos da Solução e Diagramas. Na Seção 5.1, é descrito como foi desenvolvido os principais módulos do sistema. E na Seção 5.2, a arquitetura do sistema é detalhada em três tipos de diagramas: o diagrama de pacotes, os diagramas de sequência e o diagrama de interação dos pacotes com os serviços. O diagrama de pacotes apresenta todos os pacotes do sistema e a interação entre eles. Os diagramas de sequência demonstram o fluxo das principais interações da aplicação. O Apêndice K possui os diagramas de classes da aplicação que ilustram todas as classes do sistema, divididas por pacotes, com os seus respectivos atributos e metódos e a relação entre elas.

5.1 Módulos da Solução

5.1.1 Autenticação

A autenticação é realizada através do *Facebook*, da conta *Google* e de qualquer *e-mail* e é gerenciada pelo *Firebase Authentication* (Seção I.1.4).

5.1.2 Integração com o Orbot

Para que o usuário acesse a aplicação, é necessário checar se a integração com o Orbot (Seção 2.4.6.2) está completa. A checagem é realizada através de uma API do Tor disponível pela *url* <https://check.torproject.org/api/ip>. É realizada uma requisição HTTP para a API de forma assíncrona para obter o campo *isTor*, caso o conteúdo deste campo contenha *true*, o acesso ao aplicativo é liberado.

5.1.3 Registro de Chaves

O registro das chaves pública e privada do usuário é acionado assim que o usuário acessa a tela principal do aplicativo.

A chave privada do usuário é armazenada no próprio aparelho em uma pasta de acesso restrito criada pela aplicação de forma que nenhuma outra aplicação do celular possa acessar essa chave privada. Já a chave pública deve ser publicada no *Virgil Cards Services* com uma identidade opcional associada de forma que os usuários possam encontrar as chaves públicas uns dos outros.

Para publicar as chaves no *Virgil Cards Services*, é necessário obter um *token* de acesso à sua aplicação no *Authorization Header* na chamada HTTP. Esse *token* é um *Json*

Web Token (JWT)[1] gerado através de uma chave privada da sua aplicação registrada no serviço do *Virgil Security*. Conforme *Virgil Security* (2018b), a estrutura de um cartão é composta por uma chave pública, o escopo e os campos opcionais de identificação como o tipo de identificação, a própria identificação, chaves customizáveis e as informações do aparelho. O cartão é assinado pela chave privada do usuário antes de ser publicado.

Há também outro tipo de chave associada ao usuário: o QRCode. O QRCode é um código de barras bidimensional no qual pode-se extrair o código em texto. O QRCode do usuário é registrado assim que a tela de *Profile* é acessada no *DynamoDB* (Seção I.2.1). O registro contém o QRCode, um identificador do usuário e um TTL (Time-to-Live) que possui a data de 60 segundos após a data do registro em que o mesmo deve ser deletado do banco dados.

5.1.4 Troca de chaves

O primeiro passo para a troca de chaves é a leitura do QRCode, de forma que os usuários se reconheçam. Depois em que uma sala de mensagens (Seção 5.1.7) for criada, toda mensagem enviada deve ser cifrada pelas chave públicas dos participantes da sala que são recuperadas no *Virgil Security*. Dessa forma, o sigilo é viabilizado na medida em que as mensagens só podem ser decifradas com as chaves privadas dos participantes.

5.1.5 Criptografia

O esquema de cifração de mensagens, conforme ilustrado na Figura 10, funciona da seguinte forma:

- Assina-se a mensagem com a chave privada do remetente.
- Cifra-se para as chaves públicas dos destinatários das mensagens captadas no *Virgil Cards Services*.
- Envia-se a mensagem cifrada para o servidor da aplicação.

Já o esquema de decifração de mensagens, conforme ilustrado na Figura 11, funciona da seguinte forma:

- Recebe-se a mensagem cifrada do servidor da aplicação.
- Verifica-se a assinatura da mensagem através da chave pública do remetente.
- Decifra-se a mensagem através da chave privada do destinatário.

Figura 10 – Esquema completo de encriptação.
Fonte: (SECURITY, 2018a)

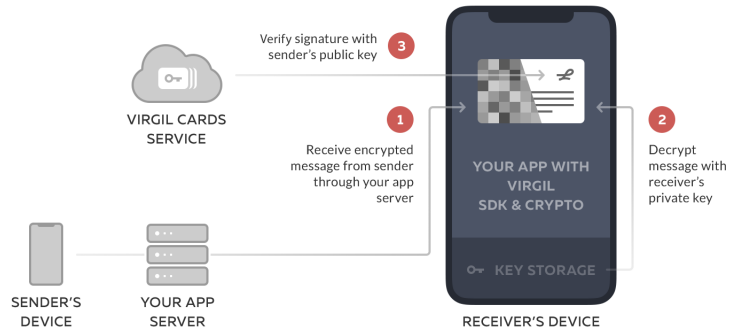


Figura 11 – Esquema completa de decriptação.
Fonte: (SECURITY, 2018a)

A geração de chaves do serviço é realizada através da função PBKDF2 (2018), que é uma função de derivação de chaves *password-based*, ou seja, a chave é derivada de uma senha passada como parâmetro da função. A troca de chaves no *Virgil Security* segue o protocolo *Diffie-Hellman* (Seção 2.3.1.2.1) utilizando a curva elíptica *Curve25519* (Seção 2.3.1.1) que oferece 128 *bits* de segurança.

O algoritmo de cifra simétrica utilizado é o AES no modo *Galois/Counter Mode* (GCM) que provê criptografia autenticada e é baseado na cifra de bloco de chave simétrica com tamanho de 128 *bits*. De acordo com Dworkin (2007), o GCM garante a confidencialidade dos dados utilizando a variação do *counter mode* nas operações de criptografia, a autenticidade e autenticação.

O *counter mode* transforma uma cifra de bloco, que consiste em um algoritmo aplicado em blocos de tamanho fixo transformado por uma chave simétrica, em uma cifra de fluxo, onde os blocos do texto em claro são combinados com um fluxo de dígitos cifrados pseudo-aleatórios. Além disso, o texto cifrado é combinado com um código de autenticação para fornecer um selo que serve para garantir a integridade dos dados.

[1] "Os JSON Web Tokens são um método RFC 7519 padrão da indústria aberto para representar declarações com segurança entre duas partes"(JWT.IO, 2018).

5.1.6 Criptografia ponta-a-ponta

Como descrito anteriormente, a chave privada é gravada no dispositivo móvel do usuário de forma que somente a aplicação cliente deste usuário possui acesso à mesma e a troca de chaves no serviço utiliza o protocolo *Diffie-Hellman*. Essas funcionalidades garantem a criptografia ponta-a-ponta (Seção 2.3.1.2) à aplicação.

5.1.7 Sala de Mensagens

A criação de uma sala de mensagens é realizada através da tela de *Profile*. Após a leitura do QRCode, o usuário destinatário receberá uma solicitação de criação de sala de mensagens que contém o nome da sala. Caso aceite a solicitação, a sala de mensagens é criada e enviada para o *Firebase* com as seguintes características: os identificadores dos usuários e o nome da sala. O acesso à sala é restrito aos usuários possuidores dos identificadores. Caso recuse a solicitação, nenhuma sala de mensagens é criada e os usuários não poderão trocar mensagens.

5.1.8 Troca de Mensagens

As mensagens trocadas são gerenciadas pelo Firebase Realtime Database (Seção I.1.1), pois é um banco de dados que sincroniza os dados em tempo real e mantém-se funcionando mesmo quando o cliente não está utilizando a aplicação. O conteúdo de uma mensagem contém um texto ou uma imagem. Toda mensagem possui uma sala em que está contida podendo ser lida somente nesta sala.

5.2 Diagramas

5.2.1 Diagrama de Pacotes

A Figura 12 contém o diagrama de pacotes que ilustra as dependências entre os pacotes da aplicação.

A aplicação é composta por quatros pacotes: a *Activity*, a *Model*, a *Controller* e a *Service*.

***Activity*:** Esse pacote contém todas as *activities* da aplicação que são as telas exibidas ao usuário.

***Model*:** Esse pacote possui todas as classes de modelo de dados da aplicação.

***Controller*:** Esse pacote possui as controladoras da aplicação que dependem das *activities* e das *models*, sendo o intermediário entre os dados e a interação com o usuário.

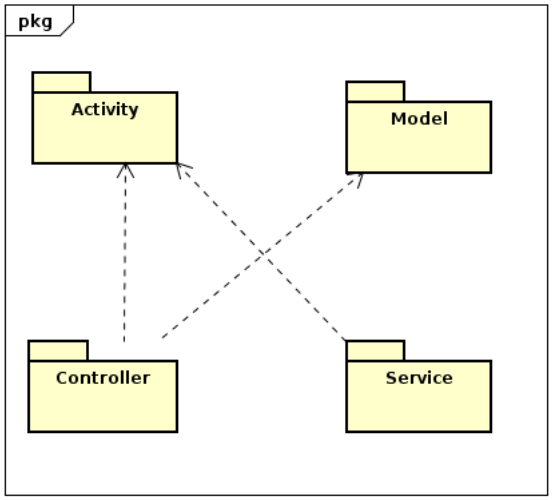


Figura 12 – Diagrama de Pacotes.

***Service*:** Esse pacote possui as classes que executam os serviços em segundo plano da aplicação e dependendo de eventos interage com as telas da aplicação. Portanto, possui dependência com as *activities*.

5.2.2 Interação dos pacotes da aplicação com os serviços

Aplicação utiliza serviços na nuvem para prover as funcionalidades de segurança. Os serviços consumidos na aplicação são: o *Virgil Security*, o *Firebase* e o *DynamoDB*. Esses serviços estão descritos no Apêndice I. O diagrama da Figura 13 abaixo demonstra o relacionamento dos pacotes da aplicação com estes serviços.

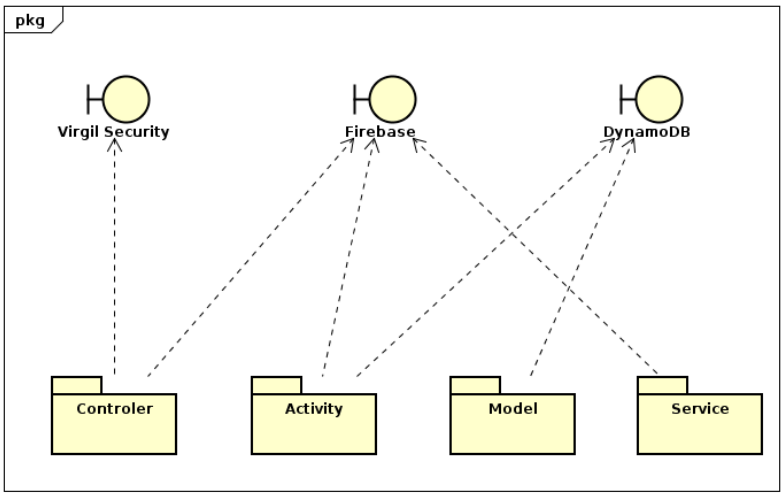


Figura 13 – Interação dos Pacotes da Aplicação com os Serviços.

O pacote *Controller* consome os serviços do *Virgil Security* e do *Firebase* para realizar o processo de geração, registro e troca de chaves, a criptografia, a autenticação, o registro de dados e o regaste de dados no banco de dados.

O pacote *Activity* consome os serviços do *DynamoDB* e do *Firebase* para realizar as operações CRUD (*Create, Read, Update e Delete.*) nos dados dos respectivos banco de dados.

O pacote *Model* consome os serviços do *DynamoDB* para mapear as tabelas do banco de dados com as classes modelo. Vale ressaltar que esse mapeamento não é necessário no *Firebase*.

O pacote *Service* consome os serviços do *Firebase* para gerenciar as notificações recebidas na aplicação.

5.2.3 Diagramas de Sequência

Abaixo estão os diagramas de sequência que representam o fluxo das principais interações do sistema: a integração com o *Orbot*, o registro de chaves dos usuários, a troca de chaves e a troca de mensagens entre os usuários.

5.2.3.1 Integração com o *Orbot*

O fluxo de mensagens durante a integração com o *Orbot*, representado pelo diagrama de sequência da Figura 14, funciona da seguinte forma:

Quando o aplicativo é iniciado, é verificado se o *Orbot* está instalado. Caso não esteja, o usuário é redirecionado para a página de *download* do *Orbot* na *Play Store*. Caso esteja, o aplicativo verifica se o *Orbot* está configurado para as mensagens da aplicação. Senão, redireciona o usuário para o aplicativo do *Orbot*. Por fim, se estiver instalado e configurado, redireciona para alguma das telas da *ChatActivity*.

As entidades representadas no diagrama são: a *ChatActivity* K.1, a *OrbotActivity* K.1, o *Orbot* e a *Play Store*.

5.2.3.2 Registro de chaves

O fluxo do registro de chaves da aplicação, representado pelo diagrama de sequência da Figura 15, funciona da seguinte forma:

Quando o usuário acessa a tela principal da aplicação (*ChatActivity* K.1), as chaves pública e privada do usuário são geradas. A chave privada é registrada no sistema de arquivos do dispositivo do usuário. A chave pública é publicada no *Virgil Security* associado ao identificador do usuário.

Quando o usuário acessa a tela de *profile* da aplicação (*ProfileActivity* K.1), um código de QRCode é gerado e registrado no banco de dados *DynamoDB*. A cada 60 segundos, o antigo código é deletado e um novo é criado.

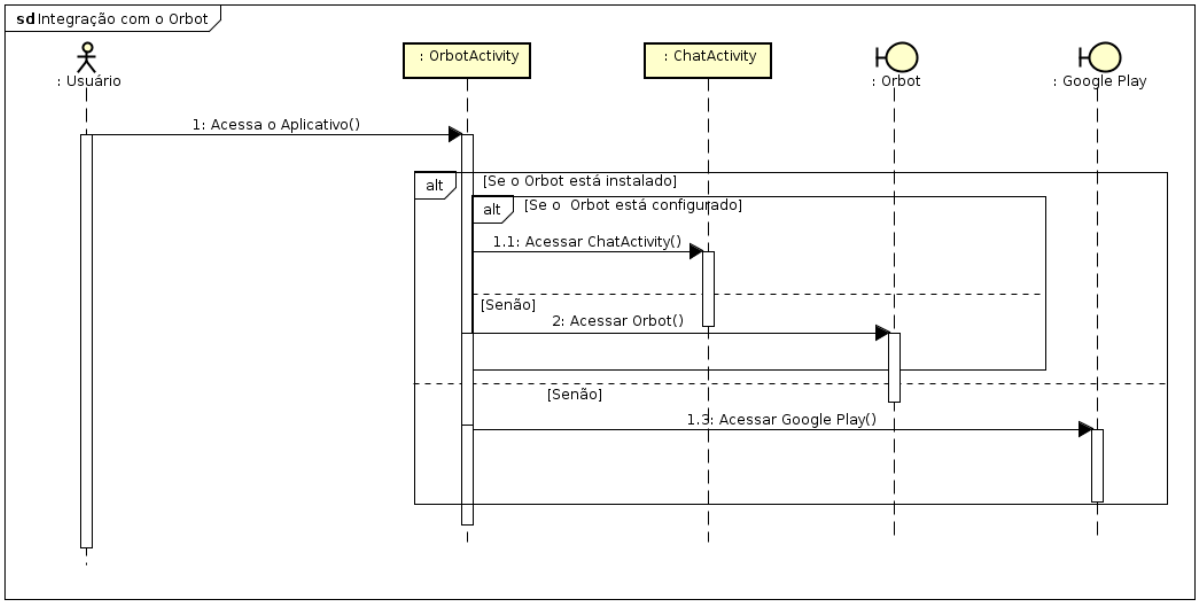


Figura 14 – Diagrama de Sequência da Integração com o *Orbot*.

As entidades representadas no diagrama são: o Usuário, a *ChatActivity* K.1, o dispositivo móvel do Usuário, o *Virgil Security*, *ProfileActivity* K.1 e o *DynamoDB*.

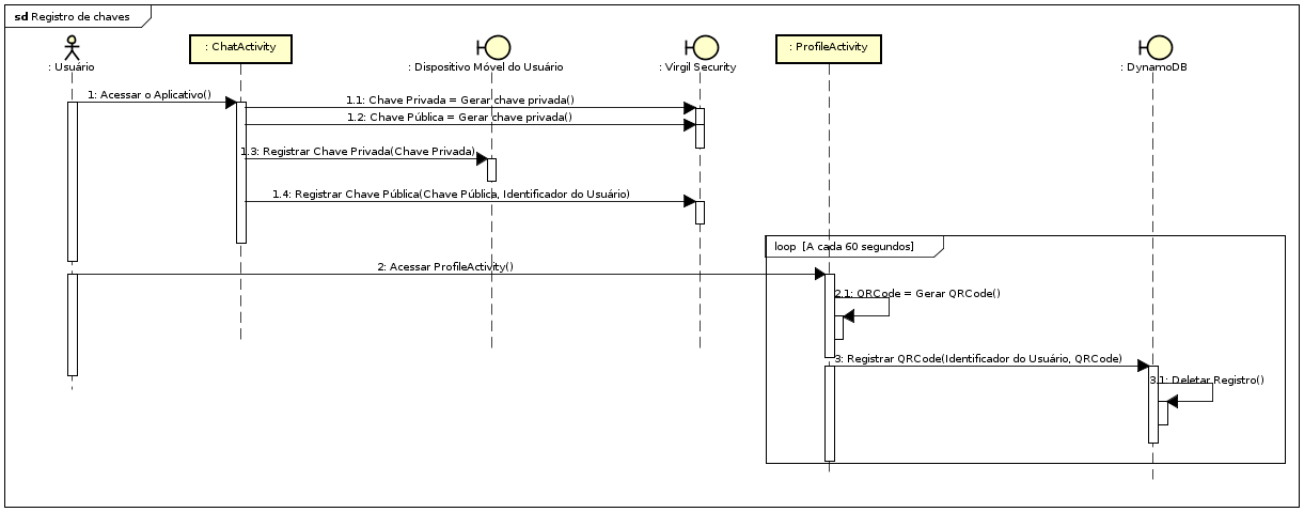


Figura 15 – Diagrama de Sequência do Registro de Chaves.

5.2.3.3 Troca de chaves

O fluxo da troca de chaves entre dois usuários (Usuário 1 e Usuário 2), representado pelo diagrama de sequência da Figura 16, funciona da seguinte forma:

Primeiramente, o usuário (Usuário 1) que deseja estebelecer a comunicação deve ler o QRCode do usuário destinatário (Usuário 2). Subsequentemente, o Usuário 1 irá consultar o QRCode obtido no *DynamoDB* e atestar se o código é válido. Se for válido, ou seja, se não estiver expirado e estiver associado ao identificador do Usuário 2, obtém o

identificador. Com o identificador é capaz de procurar as chaves públicas do Usuário 2 no serviço do *Virgil Security* utilizando-os para criptografia de mensagens. Se não for válido, o usuário recebe a mensagem de Üsuário não encontrado:

As entidades representadas no diagrama são: o Usuário 1, o Usuário 2, o *Realtime Database* e o *Virgil Security*.

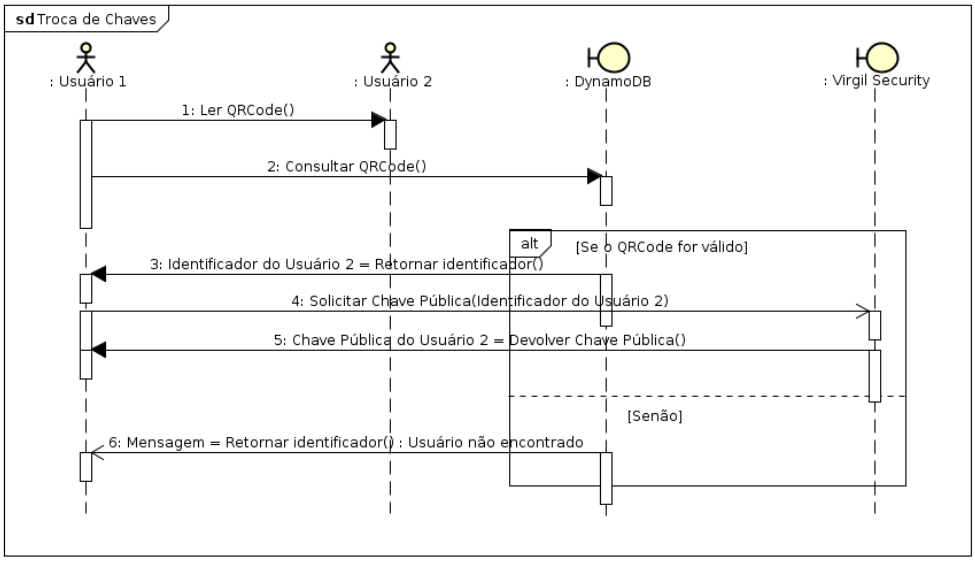


Figura 16 – Diagrama de Sequência da Integração da Troca de Chaves.

5.2.3.4 Troca de mensagens

O fluxo completo de uma troca de mensagens na aplicação, representado pelo diagrama de sequência da Figura 17, funciona da seguinte forma:

Um usuário (Usuário 1) deve enviar para outro (Usuário 2) uma solicitação de criação de uma nova sala de mensagens caso não exista. Se o Usuário 2 aceitar a solicitação, uma nova sala de mensagens é criada onde poderá ser trocadas as mensagens. Ao enviar uma mensagem, a mesma é criptografa pelo serviço do *Virgil Security* e então registrada no *Realtime Database*. Para ter acesso ao conteúdo das mensagens resgatadas do *Realtime Database* pertencentes à atual sala de mensagens, os usuários devem descriptografar as mensagens utilizando o serviço do *Virgil Security*. Se o Usuário 2 não aceitar a solicitação, o campo de resposta da solicitação é alterado no banco de dados e a solicitação é deletada, tornando impossível a troca de mensagens entre os usuários.

As entidades representadas no diagrama são: o Usuário 1, o Usuário 2, o *Realtime Database* e o *Virgil Security*.

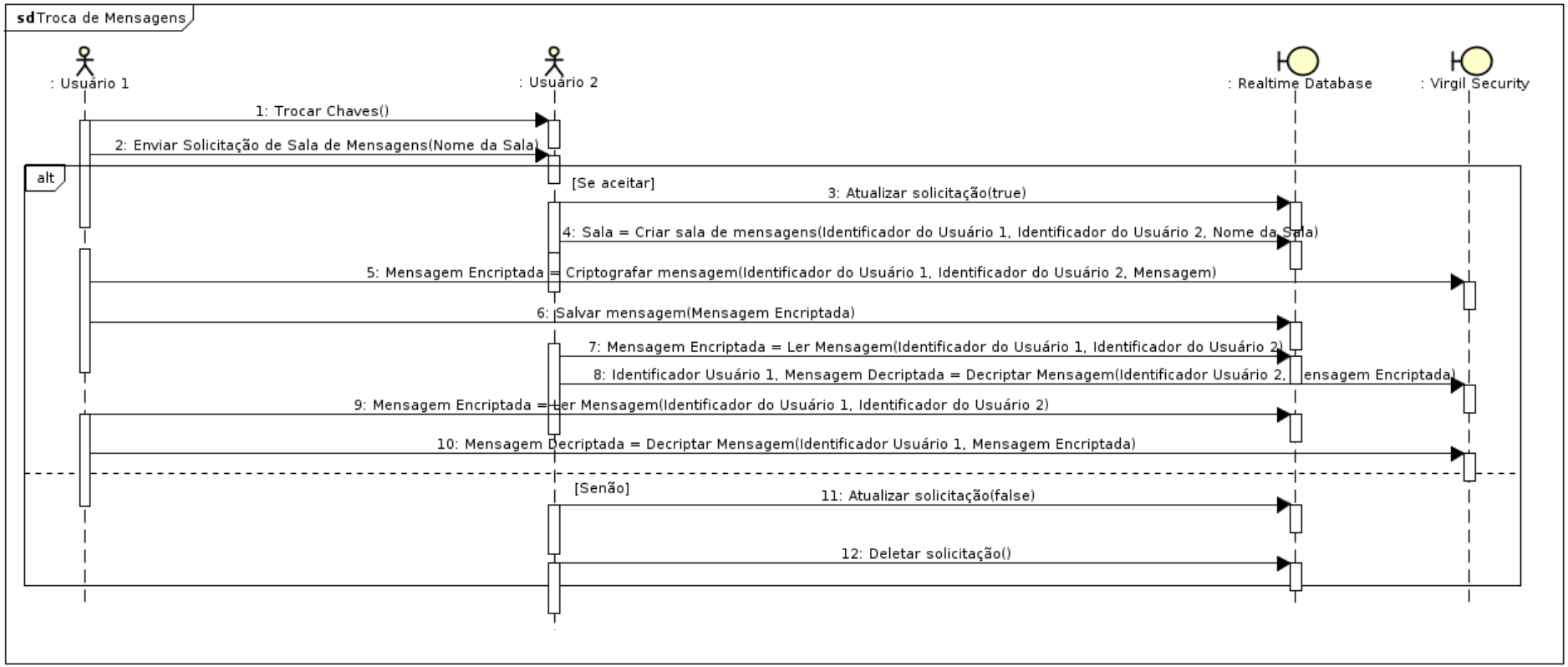


Figura 17 – Diagrama de Sequência da Troca de Mensagens.

6 Resultados e Discussões

6.1 Anonimização

Para observar o comportamento da rede de um usuário do aplicativo foi utilizado a ferramenta *WireShark*. O *WireShark* é uma ferramenta que analisa o tráfego da rede e possui uma interface gráfica com as informações dos pacotes capturados. Os pacotes são descritos por número, tempo, fonte, destino, protocolo, tamanho e conteúdo. A instalação do *WireShark* no *Ubuntu* 14.04 LTS (versão do sistema operacional utilizado para os testes) está descrita no Apêndice D.

A fim de limitar a análise à uma rede utilizada somente pelo aplicativo, foi configurado um *HotSpot* (um ponto de acesso *Wi-Fi*) no computador. A configuração de um *HotSpot* no *Ubuntu* 14.04 LTS está descrita no Apêndice C.

Após a preparação do ambiente, um dispositivo móvel com o aplicativo instalado foi conectado ao *HotSpot* para a realização dos testes. Para filtrar ainda mais a análise da rede, foi adicionado um filtro com o IP do dispositivo móvel na ferramenta.

Os resultados presentes na Figura 18 foram obtidos durante a observação de pacotes enquanto era enviada uma mensagem na aplicação.

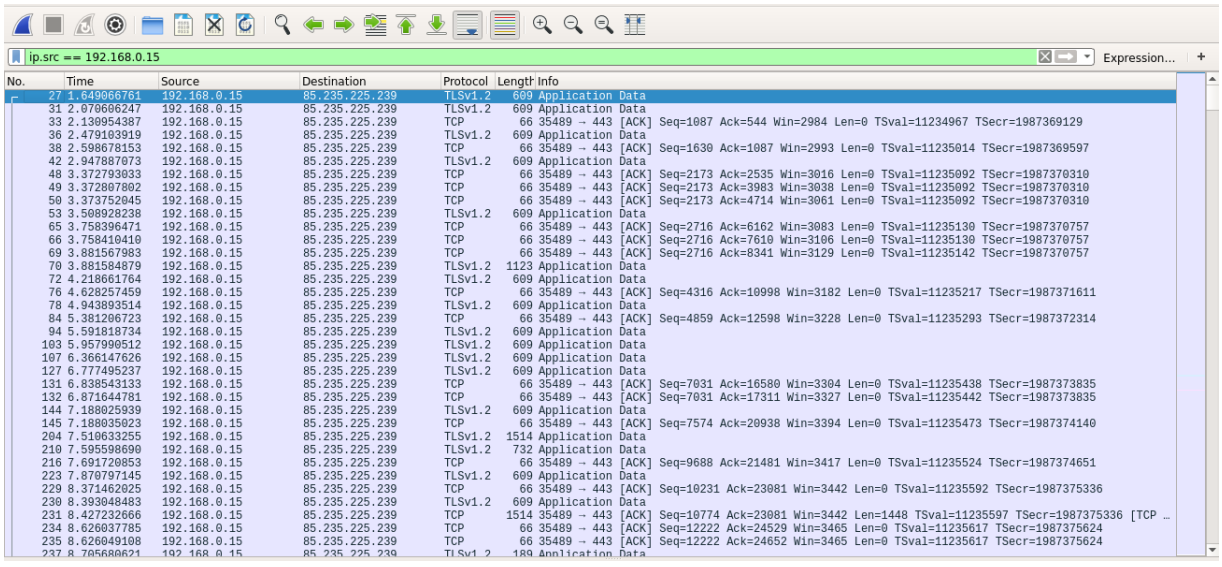


Figura 18 – Captura dos pacotes durante um envio de mensagem.

Observa-se que o IP de destino é sempre o mesmo. Este IP é o *Tor Entry Guard* (Seção 2.4.4) do usuário da aplicação. Para a conferência de que esse IP de destino é um *relay* do Tor, o mesmo foi consultado no ExoneraTor disponível no endereço <https://exonerator.torproject.org/>. O ExoneraTor é um *site* do Tor Project que possui em sua

base de dados todos os IP's que são ou que já foram *relays* do Tor. A consulta é realizada através do IP e da data de vigência desejada. Como mostrado na Figura 19, o IP de destino na data em que a requisição foi realizada é um *relay* do Tor válido apelidado como *TykRelay03*.

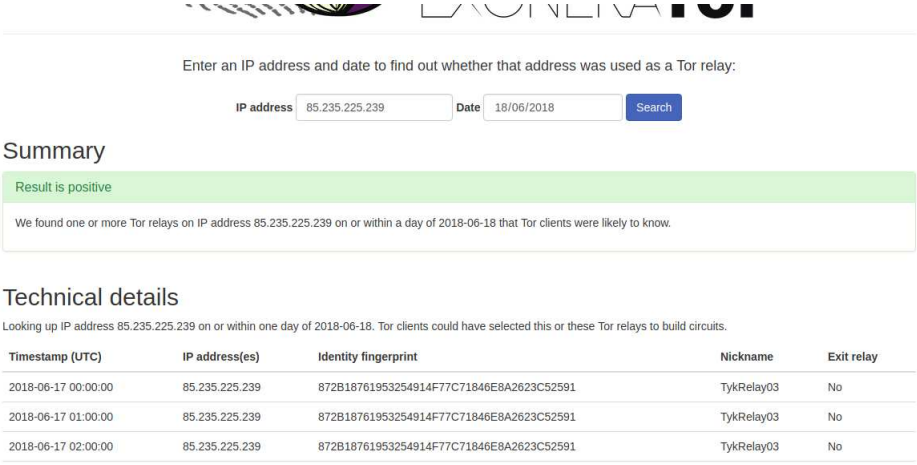


Figura 19 – Consulta do Entry Guard no ExoneraTor.

A Figura 20 possui o *log* fornecido pelo Orbot durante o envio da mensagem. Verifica-se que as rotas traçadas pelo Tor que possuem como primeiro ponto sempre o *Entry Guard TykRelay03* identificado anteriormente.

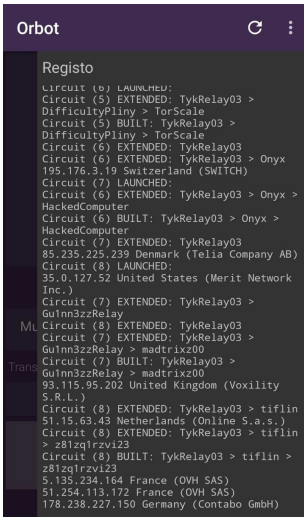


Figura 20 – *Log* capturado no Orbot durante o envio da mensagem.

O Tor também provê uma API hospedada em <https://check.torproject.org/> para checagem de seu funcionamento. Para que o serviço possa ser checado e continuar anonimizado, a API fornece uma lista de todos os nós contidos na lista do Tor em vez de coletar e conferir se o endereço IP específico do serviço é um nó do Tor. Portanto, pode-se averiguar se o endereço IP do serviço corresponde a algum endereço IP da lista fornecida. Há garantias de que o endereço IP que consulta o serviço não seja armanezado de pela API.

Essa API é consultada antes do usuário acessar as funcionalidades da aplicação. Conclui-se, então, que os serviços de anonimização são providos pela aplicação garantindo a privacidade de seus usuários.

6.2 Criptografia

As mensagens enviadas na aplicação são criptografadas e portanto ilegíveis à terceiros. A Figura 21 possui um exemplo de duas mensagens enviadas com o mesmo remetente e o mesmo destinatário na aplicação. Na Figura 22, pode-se observar que as duas mensagens enviadas estão armazenadas no *Realtime Database*. Observa-se também que o campo *bodyMessage* que representa o conteúdo da mensagem possui dois códigos em *Base64* diferentes. O *Base64* é um método de codificação de dados binários para que sejam transformados em texto.

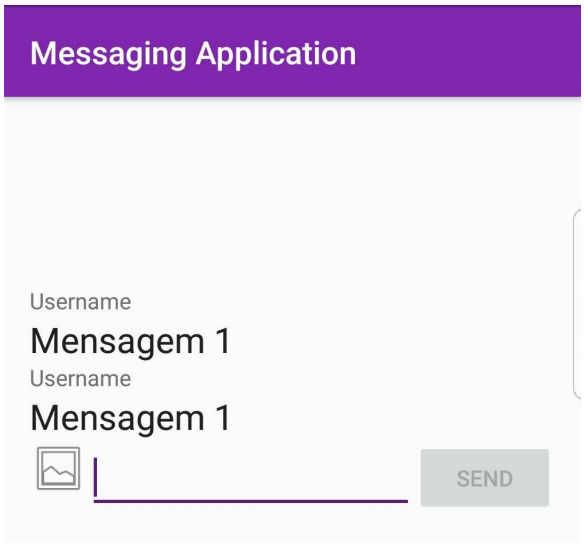


Figura 21 – Mensagens enviadas na aplicação.

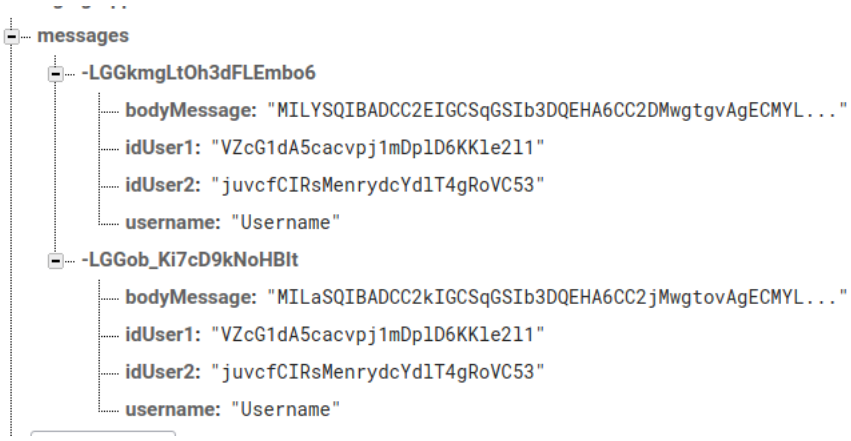


Figura 22 – Mensagens criptografas no banco de dados.

Abaixo há dois histogramas que representam as frequências dos *bytes* extraídos a partir da decodificação dos códigos em *Base64*. A Figura 23 possui o histograma da mensagem em claro e a Figura 24 possui o histograma da mensagem encriptada.

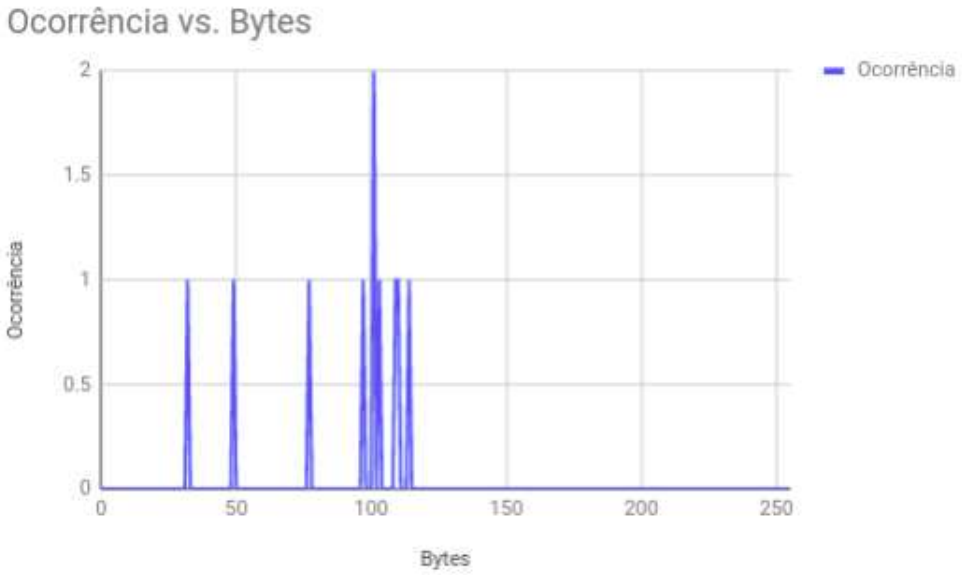


Figura 23 – Histograma da mensagem em claro.

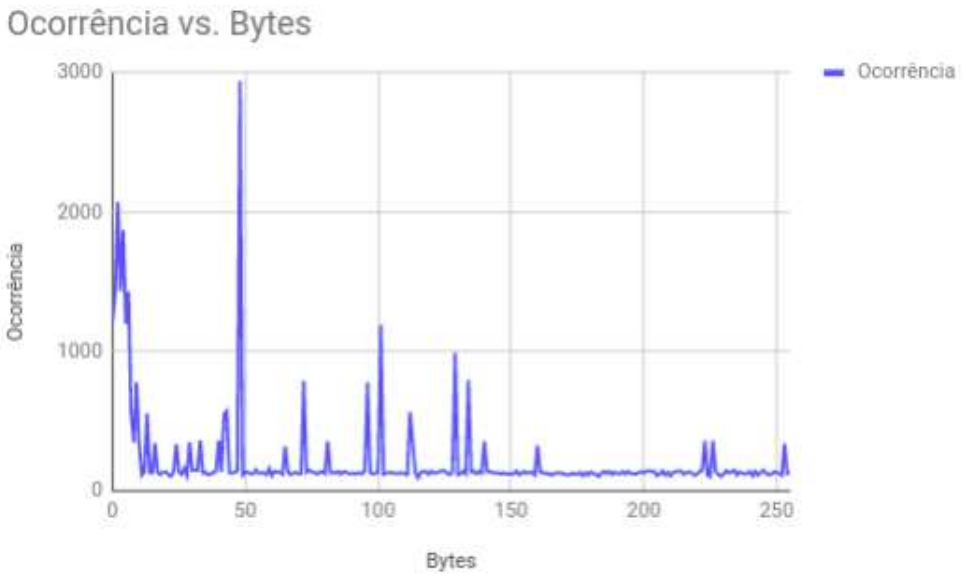


Figura 24 – Histograma da mensagem encriptada.

Nota-se que a mensagem encriptada possui um número muito maior de *bytes* além de conter ocorrências de todos os *bytes* possíveis (0 a 255).

6.3 Chaves

6.3.1 QRCode

O QRCode associado ao identificador de um usuário é renovado a cada 60 segundos. No *DynamoDB*, é inserido uma coluna nomeada TTL para cada registro que contém uma data no formato *Long* de quando o registro deve ser deletado. A Figura 25 possui um exemplo de um registro de QRCode, este registro foi destruído no dia 20 de junho de 2018 às 8 horas e 21 minutos e 51 segundos e foi criado exatamente 60 segundos antes desta data.

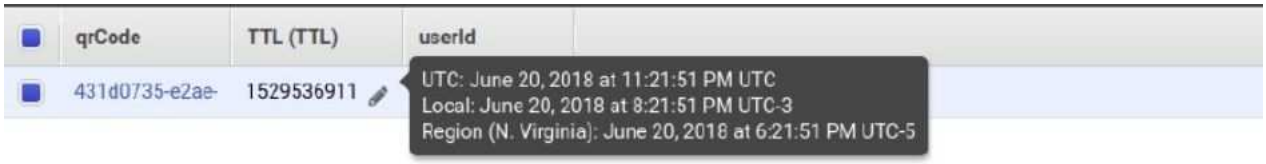


Figura 25 – TTL de um registro de QRCode.

O registro completo do QRCode pode ser visualizado na Figura 26. Observa-se que os campos são: o QRCode gerado randomicamente pela aplicação, o TTL e o identificador do usuário.

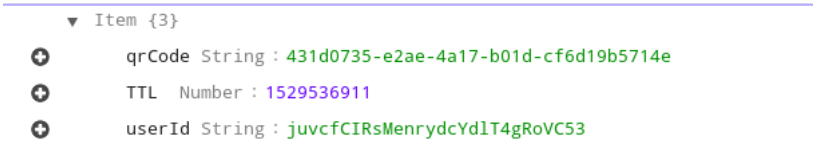


Figura 26 – Registro de um QRCode.

6.3.2 Chaves pública e privada

As chaves privadas são armazenadas nos dispositivos dos usuários e não são acessíveis. Já as chaves públicas são armazenadas em cartões no serviço do *Virgil Security*. A Figura 27 mostra as informações disponíveis no *dashboard* do *Virgil Security* sobre a aplicação.

Naquele momento, haviam 165 cartões publicados e 346 requisições realizadas. É importante ressaltar que as informações contidas nos cartões só são acessíveis pela aplicação e somente se o usuário possuir um *token* de autenticação válido para realizar a requisição.

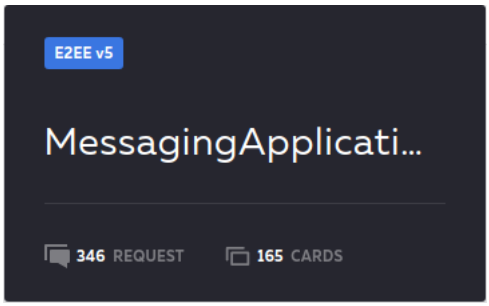


Figura 27 – Informações da aplicação no *Virgil Security*.

6.4 Aplicação

A aplicação atendeu os requisitos de anonimicidade utilizando o Tor como demonstrado na Seção 6.1. O Tor garante que o pacote percorra um circuto randômico, exceto o *entry guard*, para desassociar o remetente do destinatário. Previne que seus usuários possam ser rastreados por terceiros que estejam observando o tráfego da rede. Porém, se alguém conseguir observar o tráfego da rede da origem e do destino, pode relacionar o tempo de recebimento ou envio dos pacotes e consequentemente associar a origem ao destino. Isso pode acontecer em casos de suspeita de comunicação entre dois pontos da rede.

O esquema de criptografia utilizado na aplicação garante que ninguém tenha acesso ao conteúdo original exceto os dois participantes da sala de mensagens. Como a chave privada é armazenada no dispositivo, o usuário que acessar a aplicação em outro dispositivo não acessará o conteúdo original das mensagens enviadas e recebidas no dispositivo anterior.

Outra restrição da aplicação é a exigência do encontro físico para o estabelecimento de uma conversa devido à exigência da leitura dos QRCodes associados aos usuários.

Abaixo estão as principais telas do aplicativo.

6.4.1 *OrbotActivity*

As Figuras 28 e 29 apresentam capturas das telas geradas pela *OrbotActivity*.

Figura 28 – Tela apresentada ao usuário que ainda não instalou o Orbot.

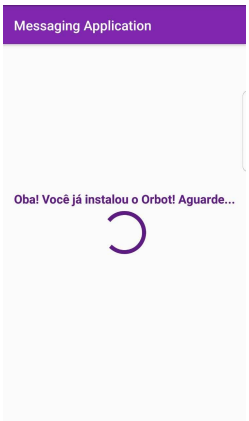


Figura 29 – Tela de espera para a verificação da integração com o Orbot.

6.4.2 *ChatActivity*

As Figuras 30, 31 e 32 apresentam capturas das telas geradas pela *ChatActivity*.

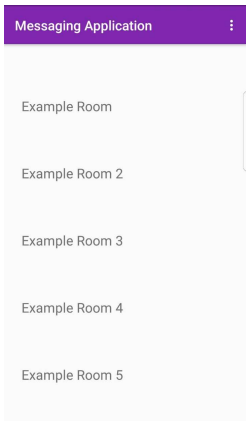


Figura 30 – Lista de sala de mensagens que o usuário participa.

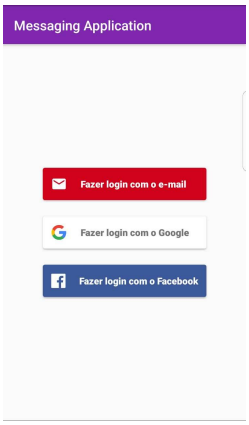


Figura 31 – Tela de autenticação da aplicação.

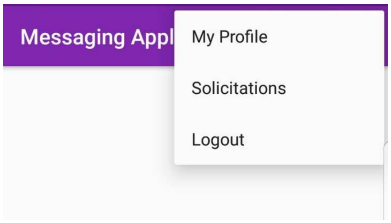


Figura 32 – Menu selecionável no canto superior direito da tela.

6.4.3 MainActivity

A Figura 33 apresenta uma captura da tela gerada pela *MainActivity*.

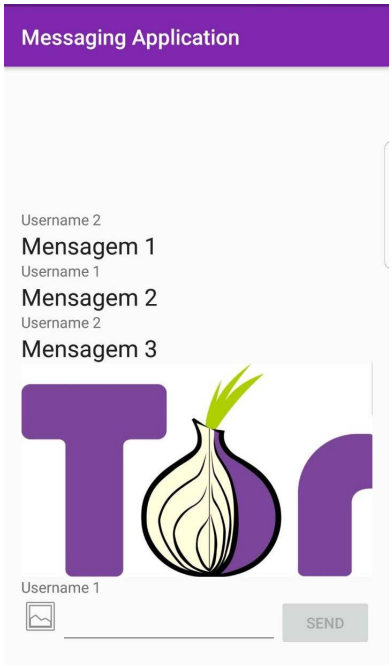


Figura 33 – Mensagens enviadas entre usuários em uma sala de mensagens.

6.4.4 *AcceptionActivity*

A Figura 34 apresenta uma captura da tela gerada pela *AcceptionActivity*.

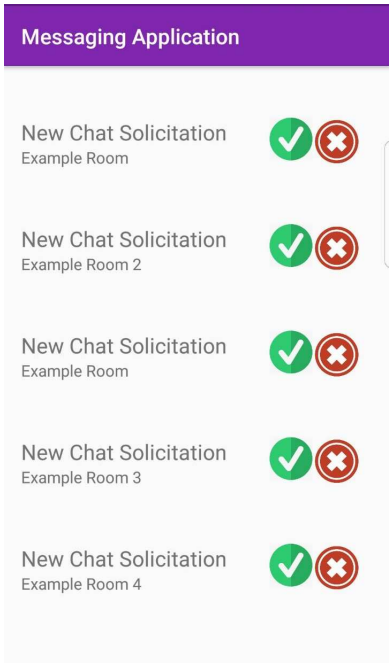


Figura 34 – Lista de solicitações enviadas para o usuário.

6.4.5 *ProfileActivity*

A Figura 35 apresenta uma captura da tela gerada pela *ProfileActivity*. Essa tela contém o botão para criação de novas salas de mensagens, um campo para preenchimento do nome da sala de mensagens e uma imagem de um QRCode renovado a cada 60 segundos conforme o *timer* da tela indica.

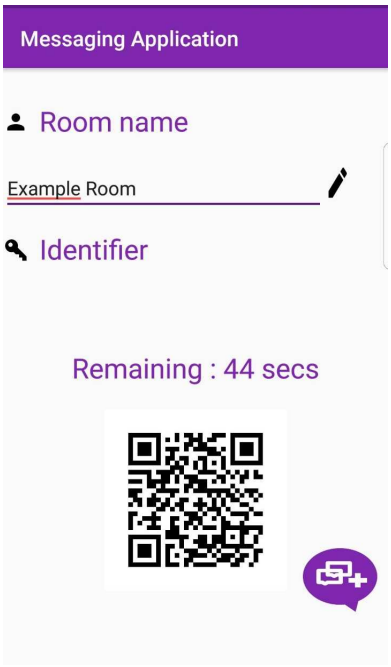


Figura 35 – Tela da *ProfileActivity*.

6.4.6 *QrCodeReaderActivity*

A Figura 36 apresenta uma captura da tela gerada pela *QrCodeReaderActivity*.

Figura 36 – Tela para leitura de QRCode.

6.4.7 *UsernameActivity*

A Figura 37 apresenta uma captura da tela gerada pela *UsernameActivity*.

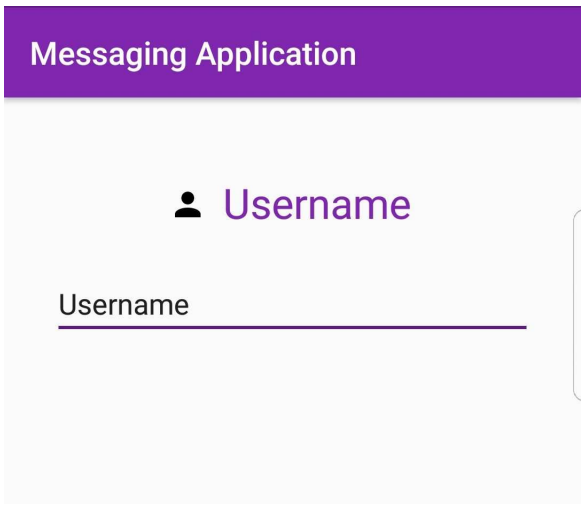


Figura 37 – Tela para edição do nome do usuário.

7 Conclusão

Ao utilizar sistemas na Internet, os usuários se expõem na rede podendo se tornar alvos de intrusos maliciosos. Para garantir a segurança na rede, foram implementados mecanismos para fornecer o sigilo, a autenticidade, o não repúdio e o controle de integridade. Ou seja, garante-se que as mensagens são enviadas e recebidas pelos destinatários e remetentes autênticos com o contéudo íntegro e inacessível à terceiros. Porém, os pacotes trocados na rede possuem informações explícitas que podem tornar viável a um intruso que esteja o observando saber quais páginas e com que frequência o usuário está acessando, permitindo-o traçar um perfil comportamental infringindo a privacidade dos usuários. Além disto, é possível utilizar esses conhecimentos para bloquear o acesso do usuário à certas aplicações.

A aplicação desenvolvida concede o direito de privacidade e de liberdade de seus usuários. Isto é, visa proporcionar todas as características de segurança citadas acima e, principalmente, a anonimidade na rede. Para garantir a anonimidade, é utilizado o serviço de anonimização Tor. O Tor basicamente direciona o pacote de um remetente à uma rota randômica (exceto o primeiro ponto da rota) até chegar ao destinatário, proporcionando a desvinculação do remetente ao destinatário. Além disto, também são utilizadas outras técnicas de segurança na aplicação como a criptografia e o registro e troca de chaves que garante que esta criptografia seja ponta-a-ponta.

A aplicação foi desenvolvida na linguagem de programação *Java* e possui portabilidade para sistemas operacionais *Android*. O fornecimento das funcionalidades de segurança em termos arquiteturais deve-se ao fato de que o sistema consome serviços em nuvem para aplicações *mobile* que proporcionam as aplicabilidades necessárias. E consequentemente, a arquitetura desta aplicação pode ser utilizada em outras com diferentes propósitos que visam proporcionar a anonimicidade para os seus usuários.

7.1 Trabalhos Futuros

Algumas melhorias a serem implementadas no sistema são:

- A utilização do *Perfect Foward Secrecy* (PFS) que permite que as chaves de sessão utilizadas na criptografia não sejam comprometidas caso a chave privada também seja.
- A deleção das mensagens depois de um tempo, assim como ocorre com o QRCode. Para prevenir possíveis ataques à criptografia das mensagens.

- O desenvolvimento de um sistema de autenticação próprio. Para evitar o fornecimento dos dados pessoais em aplicações como o *Facebook* e *Google* que utilizam-os para fins econômicos.
- O desenvolvimento de uma solução remota de troca dos QRCodes considerando a segurança dessa comunicação.

Referências

AMAZON. *Amazon DynamoDB*. 2018. <https://aws.amazon.com/pt/dynamodb/>. Citado na página 113.

AMAZON. *Infraestrutura global da AWS*. 2018. <https://aws.amazon.com/pt/about-aws/global-infrastructure/>. Citado na página 113.

ANDERSON, R. J. *Security engineering: a guide to building dependable distributed systems*. [S.l.]: John Wiley & Sons, 2010. Citado na página 33.

ANDROID. *Arquitetura da Plataforma*. 2017. <https://developer.android.com/guide/platform/index.html?hl=pt-br>. Acesso em: 26 de setembro de 2017. Citado na página 109.

ANDROID. *Componentes de aplicativo*. 2017. <https://developer.android.com/guide/components/index.html?hl=pt-br>. Acesso em: 26 de setembro de 2017. Citado na página 110.

ARANHA, D. d. F. et al. Serviço de nomes e roteamento para redes de anonimização de trafego. [sn], 2007. Citado na página 49.

ARANHA, D. de F. Redes de anonimizaçao de tráfego: aplicações e implicações, teoria e prática. 2006. Citado 2 vezes nas páginas 41 e 42.

CARRIÇO, B. et al. Novos meios de comunicação. *Revista Eletrônica de Comunicação*, v. 6, n. 1, 2012. Citado na página 21.

DWORKIN, M. J. *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. [S.l.], 2007. Citado na página 63.

EFF, E. F. F. *NSA Spying*. 2016. <https://www.eff.org/nsa-spying>. [Online; accessed 10-September-2017]. Citado na página 21.

FIREBASE. *Firebase Realtime Database*. 2017. <https://firebase.google.com/docs/database/?hl=pt-br>. Acesso em: 5 de novembro de 2017. Citado na página 111.

FIREBASE. *Sobre as mensagens do FCM*. 2017. <https://firebase.google.com/docs/cloud-messaging/concept-options>. Acesso em: 6 de novembro de 2017. Citado na página 111.

FORCE, I. E. T. *PKCS #5: Password-Based Cryptography Specification*: Version 2.0. 2018. <https://www.ietf.org/rfc/rfc2898>. Citado na página 63.

FOUNDATION, F. S. *O que é o software livre?* 2017. <http://www.gnu.org/philosophy/free-sw.html>. Acesso em: 23 de setembro de 2017. Citado na página 42.

GOTARDO, R. A. Linguagem de programação. SESES, 2015. Citado na página 28.

HANKERSON, D.; MENEZES, A. J.; VANSTONE, S. *Guide to elliptic curve cryptography*. [S.l.]: Springer Science & Business Media, 2006. Citado 2 vezes nas páginas 36 e 37.

JWT.IO. *Json Web Token*. 2018. <https://jwt.io/>. Citado na página 63.

KERCKHOFFS, A. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef*. [S.l.]: Librairie militaire de L. Baudoin, 1883. Citado na página 35.

KINOSHITA, L.; MCNAB, C. *Avaliação de segurança de redes: Conheça a sua rede*. NOVATEC, 2017. ISBN 9788575225653. Disponível em: <https://books.google.com.br/books?id=aEfCDgAAQBAJ>. Citado na página 22.

KUROSE, J. F. *Computer networking: A top-down approach featuring the internet, 3/E*. [S.l.]: Pearson Education India, 2005. Citado 9 vezes nas páginas 21, 25, 27, 28, 30, 31, 32, 38 e 40.

MARX, G. T. What's in a name? some reflections on the sociology of anonymity. *The Information Society*, Taylor & Francis, v. 15, n. 2, p. 99–112, 1999. Citado na página 50.

NETMARKETSHARE. *Market Share Report*. 2017. <http://www.netmarketshare.com/>. Acesso em: 26 de setembro de 2017. Citado 2 vezes nas páginas 107 e 108.

PFITZMANN, A.; KÖHNTOPP, M. Anonymity, unobservability, and pseudonymity—a proposal for terminology. In: SPRINGER. *Designing privacy enhancing technologies*. [S.l.], 2001. p. 1–9. Citado na página 41.

PROXY. In: WIKIPÉDIA: a enciclopédia livre. Wikimedia, 2017. cap. 1, p. 1–1. Disponível em: <https://pt.wikipedia.org/wiki/Proxy>. Acesso em: 20 de setembro de 2017. Citado na página 29.

SECURITY, V. *Encrypted Communication*. 2018. <https://developer.virgilsecurity.com/docs/java/use-cases/v4/encrypted-communication>. Citado na página 63.

SECURITY, V. *Virgil Cards Service*. 2018. <https://developer.virgilsecurity.com/docs/api-reference/card-service/v4>. Citado na página 62.

SOMMERVILLE, I. et al. *Engenharia de Software*. [S.l.]: Addison Wesley São Paulo, 2003. Citado na página 115.

STALLINGS, W. *Computer organization and architecture: designing for performance*. [S.l.]: Pearson Education India, 2000. Citado na página 25.

TANENBAUM, A. S. *Redes de computadoras*. [S.l.]: Pearson Educación, 2003. Citado 10 vezes nas páginas 25, 26, 31, 32, 33, 34, 35, 38, 39 e 40.

TANENBAUM, A. S. *Structured computer organization*. [S.l.]: Pearson Education India, 2016. Citado na página 25.

TELES, V. M. Extreme programming. *São Paulo: Novatec*, 2004. Citado na página 115.

THOMPSON, J. *A mídia e a modernidade: Uma teoria social da mídia*. Editora Vozes, 2011. ISBN 9788532642172. Disponível em: <https://books.google.com.br/books?id=EQAtBgAAQBAJ>. Citado na página 21.

TOR. *Tor: Hidden Service Protocol*. 2017. <https://www.torproject.org/docs/hidden-services.html.en>. Acesso em: 23 de setembro de 2017. Citado na página 44.

TOR. *What are Entry Guards?* 2018. <https://www.torproject.org/docs/faq.html.en#EntryGuards>. Citado na página 44.

WHATSAPP. *Connecting One Billion Users Every Day.* 2017. <https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day>. Acesso em: 26 de setembro de 2017. Citado na página 50.

Apêndices

APÊNDICE A – *Backlog*

A.1 Backlog - Estórias de usuários

Identificador	Eu, como usuário, desejo	Para
US01	enviar mensagens de texto	estabelecer uma comunicação com os outros usuários.
US02	receber mensagens de texto em tempo real	obter acesso ao conteúdo das mensagens enviadas por outros usuários.
US03	enviar mensagens com imagem	compartilhar um arquivo de foto com os outros usuários.
US04	receber mensagens com imagem em tempo real	obter acesso ao arquivo enviado para mim.
US05	me autenticar na aplicação através do meu e-mail ou do meu facebook	garantir o acesso restrito às minhas mensagens.
US06	encerrar minha sessão no aplicativo	não acessar o conteúdo restrito à minha conta.
US07	que minhas mensagens não possam ser rastreadas	garantir a segurança das minhas ações na aplicação.
US08	possuir um identificador único	poder ser reconhecido por outros usuários.
US09	ler um QRCode	identificar outros usuários.
US10	validar um QRCode	me certificar de que o QRCode é válido.
US11	deletar o QRCode associado ao meu identificador	garantir que terceiros não me identifiquem sem a minha autorização.
US12	receber solicitações de novas salas de mensagens	estabelecer uma conversa com outros usuários.
US13	enviar solicitações de novas salas de mensagens	estabelecer uma conversa com outros usuários.
US14	responder solicitações de novas salas de mensagens	estabelecer uma conversa com outros usuários.
US15	receber uma notificação sempre que uma solicitação chegar	estabelecer uma conversa com outros usuários.
US16	participar de uma sala de mensagens	estabelecer uma conversa com outros usuários.
US17	deletar uma sala de mensagens	que os participantes não tenham acesso ao contéudo da sala de mensagens.
US18	criptografar as mensagens ao serem enviadas	terceiros não obtenham acesso ao conteúdo original da mensagem.
US19	descriptografar as mensagens	obter acesso ao contéudo original da mensagem.

Figura 38 – Estórias de usuário do *Backlog* do Produto.

A.2 Backlog - Estórias técnicas

Estórias Técnicas (*Technical Stories*)		
Identificador	**Eu, como desenvolvedor, desejo**	**Para**
TS01	criar um repositório	versionar o código da aplicação.
TS02	configurar uma integração contínua ao projeto	rodar os testes sempre que uma mudança for integrada ao sistema.
TS03	verificar se o Orbot está instalado no dispositivo do usuário	para realizar a integração com o Tor.
TS04	não permitir o acesso do usuário ao aplicativo caso o Orbot não esteja instalado	garantir a integração com o Tor.
TS05	criar um proxy da aplicação no Orbot	para integrar o aplicativo com o Tor.
TS06	soliticar a instalação do Orbot caso não esteja instalado	prosseguir com o fluxo de integração com o Tor.
TS07	configurar a integração da aplicação com o DynamoDB	armazenar dados na nuvem.
TS08	configurar as permissões de acesso aos dados	garantir a segurança dos mesmos.
TS09	registrar a chave privada do usuário no dispositivo	End-To-End encryption.
TS10	registrar a chave pública do usuário	que os usuários tenham acesso às chaves públicas uns dos outros.
TS11	organizar as classes em pacotes	melhor visualização do código.
TS12	que as telas sejam responsivas	a aplicação seja acessível à todos os tamanhos de dispositivos.
TS13	melhorar a navegabilidade da aplicação	a experência do usuário seja a melhor possível.
TS14	testar a anonimicidade	ceritificar o funcionamento da aplicação.
TS15	testar a criptografia	ceritificar o funcionamento da aplicação.

Figura 39 – Estórias técnicas do *Backlog* do Produto.

APÊNDICE B – *Sprints*

As Seções abaixo possuem o *Backlog* de cada *sprint*.

B.1 *Sprint* 1

B.1.1 *Backlog* da *Sprint* 1

As estórias de usuário selecionadas para a *sprint* 1, listadas no *Backlog* da *sprint* (Figura 40), possuem ênfase na troca de mensagens e na gerência de usuários da aplicação. Além de uma estória técnica para preparação do ambiente de desenvolvimento.

Backlog da *Sprint*

Estória	*Tasks*	*Resultados*
TS01	1. Criar um repositório no GitHub. 2. Criar um READme descrevendo a aplicação.	✓
US01	1. Integrar aplicação com o Realtime Database do Firebase 2. Criar uma Activity principal com os recursos visuais do chat. 3. Criar uma classe para representar as mensagens da aplicação e seus atributos. 4. Criar um Adapter para receber as mensagens e encadeá-las em uma List View. 5. Acionar um push no banco quando o botão de enviar a mensagem seja acionado.	✓
US02	1. Extrair o conteúdo das mensagens sempre que o banco receber um novo dado.	✓
US03	1. Integrar a aplicação com o Firebase Storage 2. Abrir um Intent da galeria do celular 3. Fornecer o caminho da imagem 4. Enviar a imagem para o Storage	✓
US04	1. Receber o caminho da imagem no Storage 2. Baixar a imagem do Storage	✓
US05	1. Integrar o FirebaseUI na aplicação. 2. Inserir os provedores de e-mail, gmail e facebook para login. 3. Cadastrar a aplicação no Facebook Developers. 4. Colocar o ID da aplicação gerado pelo Facebook no Firebase Console e nas strings. 5. Pegar a URI de redirecionamento para o Facebook gerado pelo Firebase e colocá-lo na lista de URLs da aplicação nas configurações de Login do Facebook Developer. 6. Coletar as duas chaves hash do Android. 6. Tornar a aplicação do Facebook pública.	✓
US06	1. Colocar uma opção de Logout em um menu. 2. Realizar o Logout através do Firebase UI.	✓

Figura 40 – *Backlog* da *Sprint* 1.

B.2 *Sprint* 2

B.2.1 *Backlog* da *Sprint* 2

A estória desta *sprint*, listada no *Backlog* da *sprint* (Figura 41), foi planejada para que fosse estudado as possíveis soluções de integração da aplicação com o Tor, decisão arquitetural crítica do projeto, para prosseguir no planejamento do *backlog* das *sprints* seguintes.

Backlog da *Sprint*		
Estória	**Tasks**	**Resultados**
US07	1. Estudar as possíveis soluções 2. Implementar pequenas provas de conceito	

Figura 41 – *Backlog* da *Sprint* 2.

B.3 *Sprint* 3

B.3.0.1 *Backlog* da *Sprint* 3

A *sprint* 3 foi composta somente por estórias técnicas, listada no *Backlog* da *sprint* (Figura 45), que abragem a integração da aplicação com o Tor.

Backlog da *Sprint*		
Estória	***Tasks***	***Resultados***
TS03	1. Enviar uma requisição para a API do Tor. 2. Verificar se foi retornado 'true' no campo IsTor.	
TS04	1. Criar uma Activity para verificar a utilização do Orbot. 2. Se estiver utilizando o Orbot, redirecionar para a tela de autenticação. 3. Se não estiver utilizando o Orbot, redicionar para a página inicial do Orbot (se estiver instalado) ou para a Google Play.	
TS05	1. Autorizar a utilização da aplicação somente se estiver integrado com o Orbot.	
TS06	1. Procurar nos pacotes do dispositivo se o Orbot está instalado. 2. Se estiver instalado, verificar se está integrado com a aplicação. 3. Se não estiver instalado, redirecionar para a Google Play.	

Figura 42 – *Backlog* da *Sprint* 3.

B.4 *Sprint* 4

B.4.0.2 *Backlog* da *Sprint* 4

A *sprint* 4 foi composta por estórias técnicas e estórias de usuários relacionadas à implementação da lógica do QRCode na aplicação r estão listadas no *Backlog* da *sprint* (Figura 43).

Backlog da *Sprint*		
Estória	***Tasks***	***Resultados***
US08	1. Criar um QRCode randomicamente. 2. Construir a imagem do QRCode gerado. 3. Criar uma Activity para exibir a imagem.	
US09	1. Integrar aplicação com o ZXingScannerView. 2. Verificar se a aplicação possui acesso à câmera. 3. Garantir o acesso à câmera pela aplicação.	
TS07	1. Integrar a aplicação com o DynamoDB. 2. Importar os arquivos de configuração gerados pelo DynamoDB. 3. Importar as classes modelo geradas as partir das tabelas do DynamoDB.	
US10	1. Consultar no DynamoDB se o QRCode existe. 2. Verificar se o identificador associado é um usuário da aplicação.	
US11	1. Adicionar uma coluna de TTL (Time-To-Live) na tabela dos QRCodes. 2. Acionar a trigger do DynamDB para excluir o registro assim que a data do TTL chegar.	

Figura 43 – *Backlog* da *Sprint* 4.

B.5 *Sprint* 5

B.5.0.3 *Backlog* da *Sprint* 5

As estórias de usuário selecionadas para a *sprint* 5, listadas no *Backlog* da *sprint* (Figura 44), possuem ênfase nas solicitações de criação de salas de mensagens e nas salas de mensagens. Além de estórias técnicas para organização do código em pacotes e de configuração da aplicação com o serviço de notificação.

Backlog da *Sprint*		
Estória	***Tasks***	***Resultados***
US12	1. Criar uma Activity para solicitações. 2. Criar uma classe para representar as solicitações da aplicação e seus atributos. 3. Criar um Adapter para receber as solicitações e encadeá-las em uma List View.	
US13	1. Se o QRCode for válido, enviar informações para a AcceptionActivity. 2. Criar uma nova solicitação.	
US14	1. Criar botões para o layout da solicitação. 2. Adicionar listeners aos botões. 3. Quando obtiver resposta, atualiza o atributo response da solicitação.	
US15	1. Criar a classe que registra o token da aplicação. 2. Integrar com o Firebase Cloud Messaging. 3. Fazer um POST para o token do receptor da solicitação sempre que ela for criada. 4. Criar classe para reagir o evento de receber uma solicitação.	
US16	1. Criar uma Activity para as salas de mensagens. 2. Criar uma classe para representar as salas de mensagens da aplicação e seus atributos. 3. Criar um Adapter para receber as salas de mensagens e encadeá-las em uma List View. 4. Enviar informações para a criação de uma nova sala de mensagens caso o usuário aceite. 5. Restringir a permissão de acesso das mensagens somente aos participantes da sala. 6. Adicionar identificador de sala para as mensagens. 7. Restringir a permissão de acesso às salas de mensagens.	
TS11	1. Criar os pacotes. 2. Inserir as respectivas classes. 3. Refatorar as classes.	

Figura 44 – *Backlog* da *Sprint* 5.

B.6 *Sprint* 6

B.6.0.4 *Backlog* da *Sprint* 6

A *sprint* 6 foi composta por estórias técnicas e de usuário, listada no *Backlog* da *sprint* (Figura 45), que abragem a criptografia de mensagens da aplicação.

Backlog da *Sprint*		
Estória	***Tasks***	***Resultados***
US18	1. Integrar aplicação com o Virgil Security. 2. Criar um CardManager. 3. Criar um KeyStorage. 4. Criar um VirgilCrypto. 5. Criptografar as mensagens com as chaves públicas dos usuários.	✓
US19	1. Criptografar as mensagens com as chaves privadas dos usuários.	
TS09	1. Gerar a chave privada do usuário com o método KeyPair do Virgil Security.. 2. Salvar a chave privada no dispositivo.	✓
TS10	1. Gerar a chave pública do usuário com o método KeyPair do Virgil Security. 2. Gravar um cartão com a chave pública do usuário.	✓

Figura 45 – *Backlog* da *Sprint* 6.

B.7 *Sprint* 7

B.7.0.5 *Backlog* da *Sprint* 7

A *sprint* 7 foi composta somente por estórias técnicas, listada no *Backlog* da *sprint* (Figura 45), com ênfase na melhoria da aplicação.

Backlog da *Sprint*		
Estória	***Tasks***	***Resultados***
TS08	1. Criar regras de acesso ao Firebase. 2. Utilizar os dados de autenticação do Firebase.	✓
TS12	1. Atualizar os elementos do XML da ProfileActivity.	✓
TS13	1. Criar um atalho para acesso a ChatActivity em todas as telas.	✓

Figura 46 – *Backlog* da *Sprint* 7.

B.8 *Sprint* 8

B.8.0.6 *Backlog* da *Sprint* 8

A *sprint* 8 foi composta somente por estórias técnicas, listada no *Backlog* da *sprint* (Figura 45), com ênfase na melhoria da aplicação e na verificação dos resultados.

Backlog da *Sprint*		
Estória	***Tasks***	***Resultados***
TS12	1. Refatorar layouts da aplicação.	
TS13	1. Adicionar botão para retonar ao home da aplicação.	
TS14	1. Fazer análise de tráfego da rede durante uma troca de mensagens. 2. Identificar o Entry Guard. 3. Verificar se o Entry Guard é um relay do Tor. 4. Analisar log do Orbot.	
TS15	1. Decodificar base64 das mensagens. 2. Montar histograma das mensagens. 3.Verificar se a encriptação de uma mesma mensagem com as mesmas chaves resultam em base64 diferentes.	

Figura 47 – *Backlog* da *Sprint* 8.

APÊNDICE C – Configuração de um *HotSpot* no *Ubuntu* 14.04 LTS

O passo-a-passo para configuração de um *HotSpot* está descrito a seguir:

1. Instale os pacotes *hostapd* e *bridge-utils* com o comando abaixo:

```
sudo apt-get install hostapd bridge-utils
```

2. Conecte o seu computador com um cabo *ethernet* para que a interface *eth0* seja a fonte consumida pelo *HotSpot*.

3. Abra ou crie, caso não exista, o arquivo ***/etc/hostapd/hostapd.conf***.

4. Adicione o conteúdo abaixo no arquivo ***/etc/hostapd/hostapd.conf***, substituindo os valores de *<SSID>* e *<PASSWORD>* pelo nome e senha que desejar para o *HotSpot* respectivamente.

```
interface=wlan0
bridge=br0
ssid=<SSID>
hw_mode=g
channel=6
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=<PASSWORD>
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
macaddr_acl=0
```

5. Para ativar o *HotSpot*, abra o arquivo ***/etc/default/hostapd*** e altere o valor de *DAEMON_CONF* com o caminho do arquivo *hostapd.conf*, conforme exemplo abaixo.

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

6. Para criar a ponte entre as redes, altere o arquivo ***/etc/network/interfaces*** com o conteúdo abaixo.

```
auto lo
iface lo inet loopback

iface eth0 inet dhcp

auto br0
iface br0 inet dhcp
bridge_ports eth0 wlan0
```

7. Configure o *HotSpot* para que seja habilitado sempre que o computador for iniciado com o comando abaixo.

```
sudo update-rc.d hostapd enable
```

8. Reinicie o computador.

9. Se desejar interromper o *HotSpot*, utilize o comando abaixo:

```
sudo service hostapd stop
```

10. Se desejar iniciar o *HotSpot*, utilize o comando abaixo:

```
sudo service hostapd start
```

APÊNDICE D – Instalação do *WireShark* no *Ubuntu* 14.04 LTS

O passo-a-passo para a instalação do textitWireShark está descrito a seguir:

1. Adicione o repositório do *WireShark* com o comando abaixo.

```
sudo add-apt-repository ppa:wireshark-dev/stable
```

2. Execute o *update* para que os repositórios configurados no *apt* sejam atualizados.

```
sudo apt-get update
```

3. Instale o *WireShark*

```
sudo apt-get install wireshark
```

APÊNDICE E – Configuração do *Firebase* em uma aplicação *Android*

O passo-a-passo para a configuração do *Firebase* em uma aplicação *Android* está descrito a seguir:

1. Crie uma conta e adicione um projeto no *Console do Firebase*.

2. Acesse as configurações do projeto, vá na Seção **Geral** e depois em **Seus aplicativos**, selecione a opção **Adicionar o Firebase ao seu app para Android** e siga os passos descritos pelo *Firebase*.

3. Seguindo os passos acima, ocorrerá o *download* de um arquivo nomeado ***google-services.json***.

4. Insira o ***google-services.json*** na pasta **app/**.

5. Acesse o ***build.gradle*** nível raíz do projeto e adicione o *google-services plugin* e o *Google's Maven repository* como indicado abaixo.

```
buildscript {
    dependencies {
        classpath 'com.google.gms:google-services:<version>'
    }
}

allprojects {
    repositories {
        maven {
            url "https://maven.google.com"
        }
    }
}
```

6. Acesse o nível módulo do projeto e adicione o *Google Services* como indicado abaixo. É necessário que seja adicionado na última linha do arquivo.

```
apply plugin: 'com.google.gms.google-services'
```

7. Agora os serviços do *Firebase* estão disponíveis para serem utilizados no aplicativo configurado.

APÊNDICE F – Configuração do *DynamoDB* em uma aplicação *Android*

O passo-a-passo para a configuração do *DynamoDB* em uma aplicação *Android* está descrito a seguir:

1. Crie uma conta no AWS.
2. Crie um projeto *Mobile* no *Mobile Hub Projects*.
3. Ao criar um projeto, selecione a plataforma *Android* e faça o *download* do arquivo de configuração ***awsconfiguration.json***.
4. Coloque esse arquivo na pasta ***<nome_do_projeto>/app/src/main/res/raw*** do seu aplicativo. Se a pasta não existir, crie-a.
5. Adicione a dependência do AWS no ***build.gradle*** nível módulo da aplicação incluindo o conteúdo abaixo em *dependencies*.

```
compile('com.amazonaws:aws-android-sdk-mobile-client:
        <version>') { transitive = true; }
```

6. Vá no projeto criado no e adicione o *NoSQL Database* no projeto na Seção *Add More Backend Features*.
7. Clique em *Integrate* e atualize os arquivos de configuração do AWS no projeto.
8. Por fim, os serviços do *DynamoDB* estão disponíveis para serem utilizados na aplicação.

APÊNDICE G – Configuração do *Virgil Security* em uma aplicação *Android*

O passo-a-passo para a configuração do *Virgil Security* em uma aplicação *Android* está descrito a seguir:

1. Crie uma conta no *dashboard* do *Virgil Security*.

2. Adicione as dependências do *Virgil Security* no projeto incluindo o conteúdo abaixo em *dependencies* do ***build.gradle***.

```
compile 'com.virgilsecurity.sdk:crypto-android:<version>'
compile 'com.virgilsecurity.sdk:sdk:<version>'
```

3. Crie um método, como no exemplo abaixo, para gerar um *token* de acesso ao *Virgil Security*. A chave privada da aplicação pode ser coletada durante a criação do projeto no *Virgil Security*, depois não é acessível em nenhum local. O APP ID e o API KEY ID podem ser coletados no *dashboard* do *Virgil Security*.

```
private static Jwt generateToken(String identity) throws

    String apiKeyBase64 = <chave_privada>;

    byte[] apiKeyData =
        ConvertionUtils.base64ToBytes(apiKeyBase64);

    VirgilCrypto crypto = new VirgilCrypto();
    PrivateKey apiKey =
        crypto.importPrivateKey(apiKeyData);

    AccessTokenSigner accessTokenSigner =
        new VirgilAccessTokenSigner();

    String appId = <APP_ID>;
    String apiKeyId = <API_KEY_ID>;
    TimeSpan ttl = TimeSpan.fromTime(1, TimeUnit.HOURS);

    JwtGenerator jwtGenerator =
```

```
        new JwtGenerator(appId, apiKey, apiKeyId, ttl,


    Jwt tokenUser = jwtGenerator.generateToken(identity);

    return tokenUser;
}
```

4. Crie um método, como no exemplo abaixo, que chame o método de gerar o *token* e o valide.

```
public static AccessTokenProvider setupAccessTokenProvider
                    (final String userIdentifier) {

  CallbackJwtProvider.GetTokenCallback getTokenCallback =
        new CallbackJwtProvider.GetTokenCallback() {

      @Override
      public String onGetToken(TokenContext tokenContext){
          String token = \"\";
          try {
              token = generateToken(userIdentifier)
                .stringRepresentation();
          } catch (CryptoException e) {
              e.printStackTrace();
          }
          return token;
      }
  };
  AccessTokenProvider accessTokenProvider =
        new CallbackJwtProvider(getTokenCallback);

  return accessTokenProvider;
}
```

5. Os serviços do *Virgil Security* estarão disponíveis para uso utilizando os métodos anteriores para o gerenciamento de acesso às chamadas HTTP.

APÊNDICE H – Plataformas Móveis

As plataformas móveis consistem em dispositivos eletrônicos portáteis como o celular, o *tablet*, o relógio digital, entre outros. Os *softwares* destas plataformas permitem a interação do usuário com o dispositivo para usufruto dos recursos disponíveis. O gerenciamento dos recursos e de aplicações em uma plataforma móvel é de responsabilidade de *softwares* denominados sistemas operacionais. O Android, o IOS, o Windows Phone, o CooperheadOS são exemplos de sistemas operacionais para plataformas móveis.

H.1 Android

O Android é um *software* livre mantido pela Google utilizado em celulares, *tablets*, TV, relógios digitais, entre outros. De acordo com os dados do *Market Share* (2017) apresentados na Figura 48, em agosto de 2017, o Android correspondia ao sistema operacional utilizado por 64.76% das plataformas. No intervalo de um ano, o Android se manteve como o sistema operacional mais utilizado superando o IOS com quase o dobro de usuários.

H.1.1 Arquitetura

A arquitetura do Android corresponde a pilha de *softwares* ilustrada na Figura 49. Cada software está contido em um componente da arquitetura. Os componentes da arquitetura são: Linux Kernel, *Hardware Abstration Layer* (HAL), Android Runtime, bibliotecas nativas do C/C++, Java API *Framework* e aplicativos do sistema.

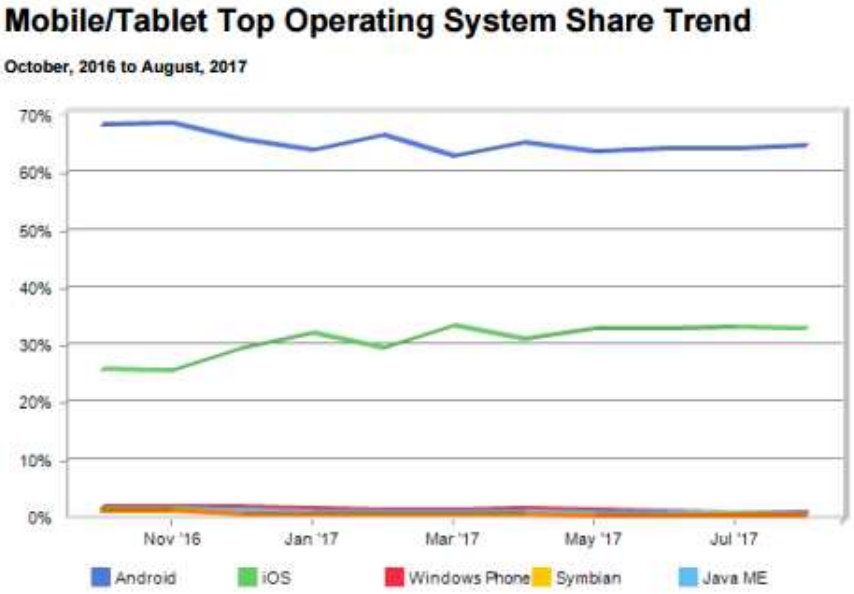
H.1.1.1 Linux Kernel

O Kernel é responsável por abstrair a parte mais baixo nível do sistema operacional. A gerência de memória, de comunicação entre processos, de permissão de usuário[1], do uso da bateria são exemplos de responsabilidades do Kernel. Também é onde se encontra os *drivers* que são *softwares* que gerenciam os dipositivos de entrada e saída como o teclado, o USB, a câmera, entre outros dispositivos.

H.1.1.2 *Hardware Abstration Layer* (HAL)

O *Hardware Abstration Layer* (HAL) é responsável por abstrair o acesso aos componentes do *hardware* para a API do Java (Seção H.1.1.5). Para cada componente de *hardware* como a câmera, os sensores e o bluetooth, o HAL possui uma biblioteca oferecendo à API uma manipulação desses componentes.

[1] Restringe o acesso direto aos recursos computacionais.



Month	Android	iOS	Windows Phone	Symbian	Java ME	Other
October, 2016	68.54%	25.78%	1.95%	1.01%	1.54%	1.18%
November, 2016	68.67%	25.71%	1.75%	0.95%	1.57%	1.34%
December, 2016	65.87%	29.52%	1.76%	0.85%	1.44%	0.57%
January, 2017	63.99%	32.03%	1.48%	0.84%	1.14%	0.51%
February, 2017	66.71%	29.55%	1.41%	0.85%	1.09%	0.40%
March, 2017	62.94%	33.39%	1.33%	0.90%	0.99%	0.44%
April, 2017	65.19%	31.06%	1.59%	0.75%	0.95%	0.46%
May, 2017	63.66%	33.03%	1.21%	0.82%	0.90%	0.38%
June, 2017	64.20%	32.92%	0.99%	0.74%	0.82%	0.33%
July, 2017	64.38%	33.09%	0.89%	0.70%	0.64%	0.30%
August, 2017	64.76%	32.93%	0.81%	0.64%	0.59%	0.26%

Figura 48 – Estástica do uso dos sistemas operacionais de plataformas móveis. Fonte: (NETMARKETSHARE, 2017).

H.1.1.3 Android Runtime

O Android Runtime é responsável por alocar e desalocar memória (*Garbage Collection*), pela compilação através de um *bytecode*[2] contido em arquivo da extensão DEX e por gerar os relatórios de erros das aplicações. Para cada aplicação, o Android Runtime executa uma máquina virtual para interpretar os *bytecodes*, sendo assim, também é responsável pela gerência da execução de várias máquinas virtuais paralelas.

H.1.1.4 Bibliotecas nativas do C/C++

Vários componentes da arquitetura são desenvolvidos em C/C++. Portanto, o sistema operacional possui algumas bibliotecas nativas dessas linguagens que podem ser acessadas pela API do Java e serem utilizadas no desenvolvimento de aplicativos. Algumas exemplos dessas bibliotecas são: o OpenGL (biblioteca para computação gráfica), Libc

[2] Código em *bytes* derivado do código-fonte para execução em uma máquina virtual.

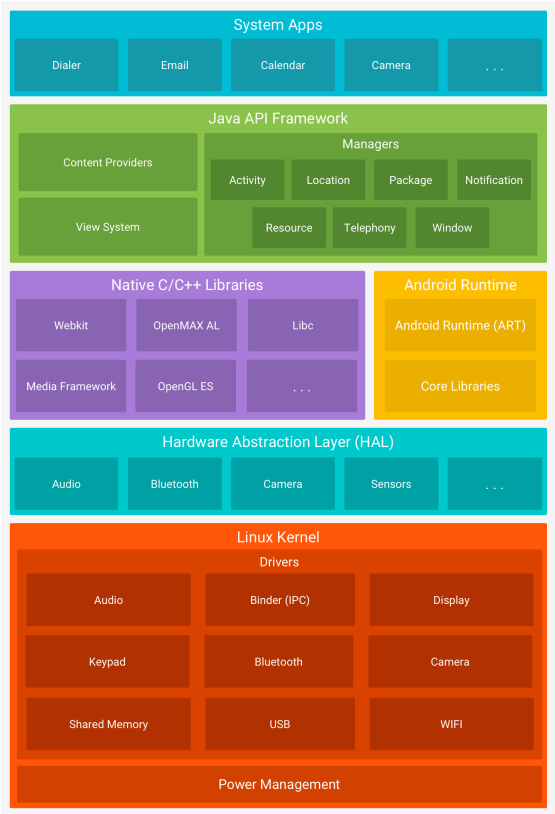


Figura 49 – Componentes da arquitetura do Android.
Fonte: (ANDROID, 2017a).

(biblioteca padrão do C), entre outras.

H.1.1.5 Java API *Framework*

O Java API *Framework* abstrai funcionalidades genéricas como o controle de componentes de um aplicativo Android e possibilita o desenvolvimento de diferentes aplicativos através da linguagem de programação Java. Os principais módulos da API são: um sistema de visualização, um gerenciador de recursos, um gerenciador de notificação, um gerenciador de atividade e provedores de conteúdo.

De acordo com o Android (2017a), a definição de componente é:

Sistema de visualização: Provê os elementos programáveis da interface do usuário.

Gerenciador de recursos: Fornece acesso a recursos como gráficos, arquivos de *layout*, entre outros.

Gerenciador de notificação: Possibilita que uma aplicação notifique o usuário através da barra de *status* do dispositivo.

Gerenciador de atividade: Controla o ciclo de vida dos componentes dos aplicativos.

Provedores de conteúdo: Disponibiliza o acesso à dados de outros aplicativos.

H.1.1.6 Aplicativos do sistema

São aplicativos padrões do sistema operacional como: os Contatos, a Câmera, o Calendário, a Calculadora, dentre outros.

H.1.2 Componentes

"A estrutura de aplicativo do Android permite criar aplicativos avançados e inovadores usando um conjunto de componentes reutilizáveis" (ANDROID, 2017b). Os principais componentes de aplicativos Android são:

***Activity*:** São as telas exibidas no dispositivo que possibilitam a interação do usuário com a aplicação.

***Service*:** Executa serviços da aplicação em segundo plano, ou seja, controla as funcionalidades do sistema.

Intent Objeto de mensagem responsável por iniciar um *Service* ou uma *Activity*, realizar transmissões do aplicativo e auxiliar na comunicação entre componentes do sistema.

***Widgets*:** Miniaturas da aplicação que podem ser encoporadas em outras aplicações e serem atualizadas peoriodicamente.

***Content Providers*:** Provedores de conteúdo que permitem o acesso aos dados de outra aplicação.

APÊNDICE I – *Backend-as-a-Service* (BaaS)

Um BaaS é um *middleware*[1] que fornece à aplicações de *software* serviços de *back-end* em nuvem. Serviços como gerenciamento de dados, gerenciamento de usuários, integração com redes sociais, entre outros. As principais vantagens do uso de BaaS em aplicações é a facilidade da implantação dos serviços que possui a economia do tempo do desenvolvimento como consequência, a desnecessidade de investir em recursos físicos e a menor responsabilidade sobre o serviço, pois qualquer ataque, queda de servidores, entre outros eventos críticos é de responsabilidade da provedora do BaaS. Porém, são serviços que não são customizáveis e depende que a aplicação se adapte a eles.

I.1 *Firebase*

O *Firebase* é um *Backend-as-a-Service* (BaaS) desenvolvido pela *Google* que provê serviços em nuvem para aplicações *web* e *mobile* facilitando o desenvolvimento das mesmas com alta escalabilidade e desempenho. Alguns serviços do *Firebase* são o *Realtime Database*, o *Cloud Messaging*, o *Storage*, o *Authentication*, entre outros.

I.1.1 *Firebase Realtime Database*

O *Realtime Database* é um banco de dados NoSQL[2] na nuvem que sincroniza os dados de todos os clientes em tempo real. Os dados armazenados estão no formato *JavaScript Object Notation* (JSON). Segundo o *Firebase* (2017a), qualquer informação alterada é sincronizada com todos os clientes em milissegundos, até dos clientes que estão *off-line*. Portanto, é um serviço para situações em que há necessidade de sincronização de dados instantaneamente.

I.1.2 *Firebase Cloud Messaging*

O *Cloud Messaging* é um serviço de notificações do *Firebase*. Os principais recursos deste serviço é a possibilidade de mandar notificações programadas ou de forma manual para todos os usuários, para um grupo de usuários ou para um usuário específico. Essas notificações podem ser enviadas pelos clientes da aplicação para outros clientes ou do administrador do aplicativo para os clientes. Segundo o *Firebase* (2017b), existem dois

[1] É um intermediário entre os serviços e as aplicações.
[2] Um banco de dados não relacional.

tipos de mensagem que são enviadas pelo*Cloud Messaging*: as mensagens de dados e as mensagens de notificação. As mensagens de notificação são as mensagens que aparecem no painel de notificação do dispositivo móvel. Uma mensagem de notificação possui três campos:

***body*:** contém o corpo da notificação.

***title*:** contém o título da notificação.

***icon*:** contém a imagem do ícone da notificação.

As mensagens de dados são mensagens que contém uma carga de dados e é entregue para os clientes da aplicação. Uma mensagem de dados possui o campo *data* onde pode ser inserido pares de chaves customizáveis de forma que o aplicativo pode definir ações para certos valores dessas chaves. As mensagens podem ser recolhíveis ou não recolhíveis, ou seja, as antigas podem ser substituídas por mensagens mais recentes ou pode persistir independente de mensagens novas. As mensagens de notificação são recolhíveis por padrão, já as mensagens de dados são não recolhíveis por padrão. Além dessas características, há ainda a prioridade da entrega da mensagem que deve ser definida em alta ou normal. A prioridade alta permite que a entrega da mensagem ocorra mesmo quando o dispositivo estiver suspenso, já a normal é entregue somente nos dispositivos que estiverem em uso, economizando a carga da bateria.

I.1.3 *Firebase Storage*

O *Storage* fornece um serviço de armazenamento de arquivos como fotos, vídeos e aúdios com alta escalabilidade para que os clientes da aplicação possam fazer *upload* e *download* dos mesmos. As operações de *upload* e *download* são realizadas independentemente da qualidade da rede, podendo ser interrompidas e retomadas do ponto em que foram descontinuadas. Para que os clientes acessem os serviços do *Storage* é necessário a autenticação no sistema e a conformidade com as regras de segurança definidas no console da aplicação do *Firebase*, como por exemplo, somente usuários do tipo administradores podem acessar os aúdios de alguma determinada pasta. É integrado com o *Firebase Authentication* para as autenticações.

I.1.4 *Firebase Authentication*

O *Authentication* fornece os serviços de autenticação da aplicação provendo a confirmação da identidade do cliente para acesso a conteúdos restritos. A autenticação pode ser realizada via *e-mail* e senha, por número de telefone ou por redes sociais. Possui integração com várias redes sociais como o *Facebook*,o *Twitter*, o *GitHub* e com contas do

Google. Ações como o *logout*[3], a redefinição de senha, a verificação de cadastro são realizadas automaticamente pelo *Authentication*. Oferece ainda a autenticação anônima, na qual o cliente não precisa se cadastrar de primeira para utilizar a aplicação mas quando realizar o cadastro pode recuperar todas os dados produzidos durante o uso.

I.2 *Amazon Web Services* (AWS)

A *Amazon Web Services* é um *Backend-as-a-Service* (BaaS) desenvolvido pela *Amazon*. São oferecidos serviços em nuvem de hospedagem de aplicações, de *backup* e armazenamento, de banco de dados como o *DynamoDB*, entre outros. De acordo com a Amazon (2018b), os serviços são oferecidos em 55 zonas distribuídas em 18 regiões geográficas no mundo.

I.2.1 *DynamoDB*

O *DynamoDB* é um banco de dados não relacional que armazena valores no formato de documentos e chave-valor que é altamente escalável e rápido. Para alcançar maior rapidez nas respostas, o *DynamoDB* possui o *Amazon DynamoDB Accelerator* (DAX). Segundo a Amazon (2018a), o DAX é um cache de memória que pode reduzir o prazo de retorno do *DynamoDB*, mesmo com várias solicitações, de milissegundos para microssegundos.

As principais vantagens do *DynamoDB* são o rápido desempenho mesmo em grandes escalas, o auto-gerenciamento, ou seja, a instalação e configuração do banco de dados é por conta do serviço, o rigoroso controle de acesso além da possibilidade de ser programado para reagir à eventos rapidamente, por exemplo, reagir a alguma alteração de dados.

I.3 *Virgil Security*

O *Virgil Security* é um *Backend-as-a-Service* (BaaS) que oferece serviços de criptografia em nuvem como a *End-to-end Encryption* (E2EE) (Seção 2.3.1.2).

[3] Encerramento da sessão na aplicação.

APÊNDICE J – Metodologias Ágeis

Metodologias ágeis de desenvolvimento de *software* consistem em um modelo de gestão que prioriza o desenvolvimento do produto em suas atividades para que seja possível uma entrega rápida de forma contínua. É um processo baseado no empirismo, portanto estimula uma melhoria constante para se adaptar ao time proporcionando a entrega imediata de um produto com qualidade. Por ser executado em ciclos, facilita a integração de mudanças tanto no processo quanto no produto satisfazendo os membros da equipe, incluindo os clientes, por ser adaptável aos *feedbacks* de todos. Estimula a produtividade além de aumentar a competitividade por disponibilizar produtos de forma mais rápida no mercado. O *Extreme Programming* (XP), o *Scrum*, o *Microsoft Solutions Framework* (MSF) são exemplos de metodologias ágeis focadas no desenvolvimento de *software*.

J.1 *Extreme Programming* (XP)

O *Extreme Programming* (XP) é uma metodologia ágil de desenvolvimento de *software* que prioriza a implementação do produto durante o processo. Uma das principais características desta metodologia é o desenvolvimento incremental e iterativo, ou seja, é ciclíco e possui entregas contínuas.

De acordo com Teles (2004), essa metodologia é recomendada para projetos com as seguintes características: possui requisitos imaturos que são candidatos à mudanças, uma equipe pequena é responsável pelo desenvolvimento do produto e é orientado à objetos.

Segundo Sommerville (2003), as práticas do XP são:

Estórias de usuário: A escrita dos requisitos em forma de estórias de usuário. Uma estória de usuário descreve o requisito com a visão de um usuário e geralmente segue o padrão: Eu, como <papel>, desejo <ação> para <justificativa>.

Planejamento incremental: As estórias de usuário são priorizadas e o planejamento é realizado em cada iteração para definir quantas e quais estórias serão entregues naquela iteração.

Pequenas *releases*: Disponibilizar o produto mínimo viável (MVP) o mais rápido possível.

Projetos simples: Projetos com o objetivo de solucionar problemas correntes.

Desenvolvimento *test-first*: Escrever testes antes de escrever o código da solução.

Refatoração: Evoluir o código de forma contínua.

Programação em pares: Programar em dupla para transferência de conhecimento entre a equipe e apoio coletivo no desenvolvimento.

Propriedade coletiva: Todos os desenvolvedores devem conhecer todo o código e suas configurações ou seja, não existe especialistas em uma parte específica do sistema.

Integração contínua: Integrar novas funcionalidades ao sistema de forma contínua e executar os testes para garantir que nenhuma parte do sistema foi prejudicada pela integração.

Ritmo sustentável: Evitar a exaustão do trabalho no desenvolvimento para não prejudicar a produtividade do desenvolvedor e nem a qualidade do código.

Cliente no local: Comunicação aberta e contínua com o cliente, que é considerado um integrante do time, para obter *feedback* rápido sobre o produto minimizando as chances de insastifação com o produto final.

APÊNDICE K – Diagramas de Classes

Abaixo estão os diagramas de classes da aplicação divididas por pacotes a fim de detalhar os encargos de cada classe do sistema.

K.1 *Activity*

A Figura 50 contém o diagrama de classes do pacote *Activity*.

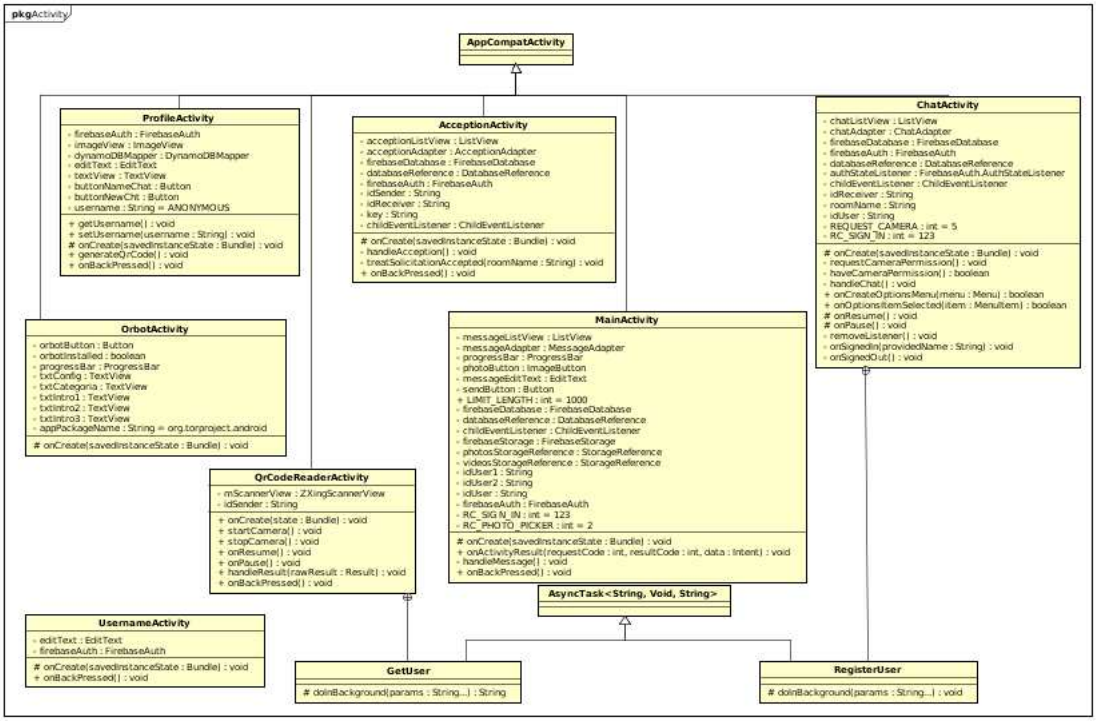


Figura 50 – Diagrama de Classes do Pacote *Activity*.

A seguir, serão detalhadas os papéis principais de cada classe contida no modelo acima.

***AppCompatActivity*:** É a classe base de todas as *activities* e possui suporte para todas as versões do Android, desde as mais antigas até a mais nova. Todas as *activities* devem herdar essa classe.

***AsyncTask<String,void, String>*:** A *AsyncTask* é uma classe do Android que permite executar operações em *background*, isto é, em segundo plano na aplicação. Executa a operação definida e publica o resultado na interface do usuário.

***AcceptionActivity*:** É a classe da tela de solicitações de criação de uma sala. Exibe uma lista de solicitações, cada uma delas possui o nome da sala e dois botões: um para aceitar e um para recusar a solicitação.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* K.1 que é onde a tela é inicializada e onde está definido a *ListView* que contém uma lista de solicitações. Recebe informações da *QrCodeReaderActivity* K.1 de novas solicitações e as salva no banco de dados *Realtime Database*.

***handleAcception*:** É um método que possui um *childEventListener* que é onde está programado a reação dos eventos das solicitações no banco de dados. Os eventos são: adição, alteração, exclusão, mudança da prioridade de local do dado ou falha na requisição.

***treatSolicitationAccepted*:** Caso a solicitação seja aceita, este método é acionado e subsequentemente envia as informações para a *ChatActivity* K.1 para que uma nova sala de mensagens seja criada.

***onBackPressed*:** É o método chamada quando o usuário aperta a tecla de retornar no dispositivo. Este método redireciona o usuário para a K.1.

***ChatActivity*:** É a classe da tela principal após a integração com o *Orbot*. Caso o usuário não esteja autenticado, exibe as opções de autenticação. Caso contrário, exibe as salas de mensagens em que o usuário está contido.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* que é onde a tela é inicializada e onde está definido a ListView que contém uma lista de salas de mensagens (*chats*). Verifica se o aparelho possui acesso à câmera do dispositivo. Caso não, aciona o método *requestCameraPermission* K.1 para solicitar permissão ao usuário. O acesso à câmera é crucial para que o leitor de QRCode da *QrCodeReaderActivity* funcione. Recebe informações da *AcceptionActivity* de novas salas e as salva no banco de dados *Realtime Database*. Para que o usuário obtenha acesso às salas de mensagens, define um *listener* para os itens da *ListView* determinando que um item, ao ser clicado, deve redirecionar o usuário para a *MainActivity* com as informações da sala selecionada. Também possui o *listener* do *Firebase Authentication* que verifica se o usuário possui uma sessão válida e ativa no aplicativo. Caso não, exibe a tela de autenticação do aplicativo.

***requestCameraPermission*:** Este método requisita a permissão de acesso à câmera ao usuário. A requisição é feita através do método *requestPermission* da *ActivityCompat* que é uma classe do Android de suporte para acesso às *features* da *Activity*.

***haveCameraPermission*:** Este método verifica se a aplicação possui permissão de acesso à câmera. A verificação é realizada através do método *checkSelfPermission* da *ContextCompat* que é uma classe do Android de suporte para acesso às *features* do Context, cujo também é uma classe do Android que possui informações globais sobre o ambiente do aplicativo.

***handleChat*:** É um método que possui um *childEventListener* que é onde está programado a reação dos eventos das salas de mensagens no banco de dados. Os eventos são: adição, alteração, exclusão, mudança da prioridade de local do dado ou falha na requisição.

***onCreateOptionsMenu*:** Cria o menu de opções do aplicativo.

***onOptionsItemSelected*:** Possui as ações definidas para cada item do menu caso sejam acionados pelo usuário. As ações são: acessar a tela de *profile* (*ProfileActivity* K.1), acessar a tela de solicitações (*AcceptionActivity* K.1) e fazer *logout*.

***onResume*:** É um método polimórfico da *AppCompatActivity* K.1 que é o primeiro a ser acionado quando a *Activity* é reiniciada. Nesta classe, este método invoca o *listener* do *Firebase Authentication* para obter controle do acesso e sessão do usuário.

***onPause*:** É um método polimórfico da *AppCompatActivity* K.1 que é acionado quando a tela do aplicativo não está mais em foco no dispositivo. Nesta classe, esse método remove o *listener* do *Firebase Authentication* e do *Realtime Database* e limpa a lista de salas de mensagens da *Activity*.

***removeListener*:** Remove o *listener* do *Realtime Database*.

***OnSignedIn*:** Método acionado quando o *listener* do *Firebase Authentication* valida a sessão do usuário. Invoca a *inner class*[1] *GetUser*.

***OnSignedOut*:** Método acionado quando o *listener* do *Firebase Authentication* invoca o encerramento da sessão do usuário. Aciona o método *removeListener* e limpa a lista de salas da *Activity*.

***RegisterUser*:** É uma *inner class* que herda a classe *AssyncTask<String, void, String>* e que aciona os métodos da *CryptoEEHelper* para registro das chaves pública e privada do usuário no *Virgil Security*.

***doInBackground*:** É um método polimórfico da *AssyncTask* que aciona o método *createCard* da *CryptoEEHelper* K.3 para registrar as chaves do usuário.

***MainActivity*:** É a classe da tela de mensagens de uma sala, contém todas as mensagens enviadas na sala com o nome do remetente em cima da mensagem e exibe um campo

[1] Uma *inner class* é uma classe privada implantada dentro de outra classe principal.

para preenchimento de mensagens que o usuário deseja enviar. As mensagens podem ser de texto ou arquivos de fotos.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* K.1 que é onde a tela é inicializada e onde está definido a *ListView* que contém uma lista de mensagens. Recebe informações da *ChatActivity* K.1 sobre a sala de mensagens que está contida. Aciona os *listeners* necessários paro envio e registro de mensagens no *Realtime Database*.

***onActivityResult*:** É o método que acessa a galeria de imagens do dispositivo do usuário e cria as mensagens que contém imagens para serem enviadas e registradas. Além de receber a informação da sessão do usuário e reagir com uma mensagem. Quando a sessão é iniciada, a mensagem exibida é *Ẅlogged in!¨*. Quando a sessão é finalizada, a mensagem exibida é *G̈oodbye!¨*.

***handleMessage*:** É um método que possui um *childEventListener* que é onde está programado a reação dos eventos das mensagens no banco de dados. Os eventos são: adição, alteração, exclusão, mudança da prioridade de local do dado ou falha na requisição.

***onBackPressed*:** É o método chamada quando o usuário aperta a tecla de retornar no dispositivo. Este método redireciona o usuário para a K.1.

***OrbotActivity*:** É a classe da tela de verificação da integração com o *Orbot*. Possui uma mensagem explicativa indicando a necessidade de integração.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* K.1 que é onde a tela é inicializada. É verificado se o *Orbot* está instalado com a função *isAppInstalled* da classe *OrbotHelper* K.3. Se estiver instalado, verifica se o tráfego pelo *Tor* está configurado para o aplicativo e então redireciona para a *ChatActivity*. Senão estiver configurado, redireciona para o aplicativo *Orbot*. Caso o *Orbot* não estiver instalado, redireciona o usuário para a tela de *download* do *Orbot* na *Play Store*.

***ProfileActivity*:** É a classe da tela de criação de salas de mensagens. Possui um campo para escrever o nome da sala de mensagens desejado, um botão para acionar a criação da sala e um QRCode associado ao usuário que é renovado a cada 60 segundos (o tempo restante da duração do QRCode é apresentado na tela). A classe possui métodos de *getter* e *setter* para *username*, pois é o valor *default* sugerido para o nome da sala.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* K.1 que é onde a tela é inicializada. Inicia a comunicação com o *DynamoDB* para registrar

o QRCode do usuário. Além de possuir um contador que a cada 60 segundos aciona o método *generateQrCode*.

***generateQrCode*:** Gera um código aleatório para o QRCode e constrói a imagem do QRCode para ser renderizada na tela. Registra o QRCode no banco contendo o código, o identificador do usuário e o TTL (Time-To-Live) que é calculado a partir da soma de 60 segundos do *timestamp* capturado no momento do registro do QRCode.

***onBackPressed*:** É o método chamada quando o usuário aperta a tecla de retornar no dispositivo. Este método redireciona o usuário para a K.1.

***UsernameActivity*:** É a classe de alteração do nome do usuário. Exibe o nome atual do usuário e se for pressionado ENTER, altera o nome do usuário pelo texto atual contido no campo de edição.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* K.1 que é onde a tela é inicializada e onde está definido o *EditText* que é o campo de alteração.

***onBackPressed*:** É o método chamada quando o usuário aperta a tecla de retornar no dispositivo. Este método redireciona o usuário para a K.1.

***QrCodeReaderActivity*:** É a classe da tela do leitor do QRCode. Aciona a câmera e quando apontado à um QRCode, obtém seu valor e verifica se é válido na aplicação.

***onCreate*:** É um método polimórfico provindo da *AppCompatActivity* K.1 que é onde a tela é inicializada. Recebe informações do solicitante de uma nova sala da *ProfileActivity* e instancia um objeto do *ZXingScannerView*. O *ZXingScannerView* é a biblioteca responsável por decrifrar a imagem apontada na câmera e converter para um código do tipo *String* que pode ser tratado pela aplicação.

***startCamera*:** Aciona o método de iniciar a câmera do *ZXingScannerView* e atribui a *QRCodeReaderActivity* como o *handler* dos resultados obtidos da leitura do QRCode.

***stopCamera*:** Aciona o método de fechar a câmera do dispositivo do *ZXingScannerView*.

***onResume*:** É um método polimórfico da *AppCompatActivity* K.1 que é o primeiro a ser acionado quando a *Activity* é reiniciada. Nesta classe, aciona o método de iniciar a câmera do *ZXingScannerView*.

***onPause*:** É um método polimórfico da *AppCompatActivity* K.1 que é acionado quando a tela do aplicativo não está mais em foco no dispositivo. Nesta classe, aciona o método de fechar a câmera do dispositivo do *ZXingScannerView*.

***handleResult*:** É o método onde é tratado o código resultante da leitura. Se estiver vazio, exibe uma mensagem de QRCode inválido. Senão, verifica se o QRCode é válido acionando a *inner class GetUser*.

***onBackPressed*:** É o método chamada quando o usuário aperta a tecla de retornar no dispositivo. Este método redireciona o usuário para a K.1.

***GetUser*:** É uma *inner class* que herda da classe *AssyncTask<String, void, String>* e que consulta se o QRCode obtido é válido.

***doInBackground*:** É um método polimórfico da *AssyncTask* que tenta encontrar o usuário associado ao código do QRCode no *DynamoDB*. Se encontrar, envia as informações para a *AcceptionActivity* K.1 para a criação de uma nova solicitação. Senão, exibe a mensagem Üsuário não encontrado.

K.2 *Model*

A Figura 51 contém o diagrama de classes do pacote *Model*.

Figura 51 – Diagrama de Classes do Pacote *Model*.

A seguir, serão detalhadas os papéis principais de cada classe contida no modelo.

***Acception*:** É a classe modelo de uma solicitação com os seguintes atributos:

***id*:** O identificador da solicitação.

***roomName*:** O nome da sala de mensagens proposta pelo remetente da solicitação.

***identifierReceiver*:** O identificador do destinatário da solicitação.

***response*:** A resposta da solicitação que é falsa por *default*.

***AppMessage*:** É a classe modelo de uma mensagem com os seguintes atributos:

***bodyMessage*:** O corpo da mensagem a ser enviada.

***username*:** O nome do usuário remetente.

***photoUrl*:** A *url* da foto em caso de envio de imagens. Se a mensagem for de texto, esse campo é nulo.

***idUser1*:** O identificador do usuário remetente.

***idUser2*:** O identificador do usuário destinatário.

***Chat*:** É a classe modelo de uma sala de mensagens com os seguintes atributos:

***identifierUser1*:** O identificador de um dos usuários participantes da sala de mensagens.

***identifierUser2*:** O identificador do outro usuário participante da sala de mensagens.

***roomName*:** O nome da sala de mensagens.

NotificationTokenDO: É a classe modelo do *token* da aplicação utilizada no envio de notificações para o dispositivo do usuário com os seguintes atributos:

***userIdNotifcation*:** O identificador do usuário do dispositivo.

***tokenNotifcation*:** O *token* da aplicação instalada no dispositivo do usuário.

***TokenDO*:** É a classe modelo do registro do QRCode com os seguintes atributos:

***qrCode*:** O código do QRCode.

TTL O temporizador que indica quando deve ocorrer a deleção do registro.

***userId*:** O identificador do usuário.

K.3 *Controller*

A Figura 52 contém o diagrama de classes do pacote *Controller*.

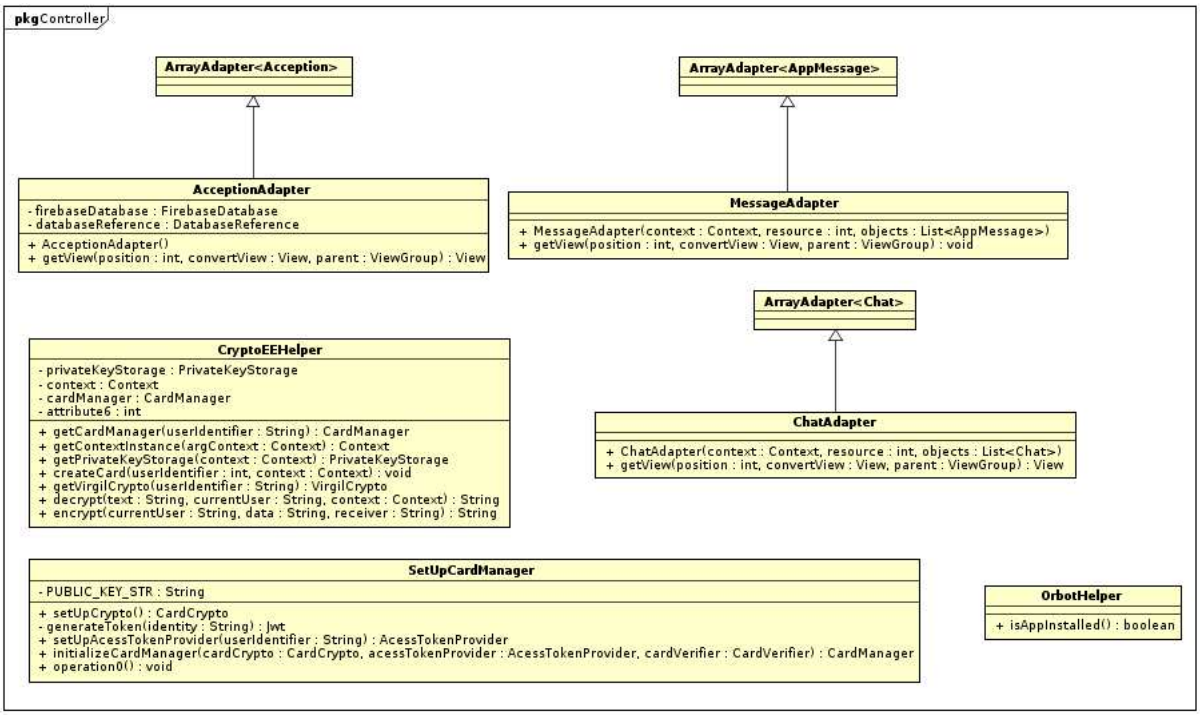


Figura 52 – Diagrama de Classes do Pacote *Controller*.

A seguir, serão detalhadas os papéis principais de cada classe contida no modelo.

***ArrayAdapter<Acception>*:** A *ArrayAdapter* é a classe do Android que retorna a interface de cada objeto de uma lista de objetos *View* da *ListView*. Neste caso, é um *ArrayAdapter* dedicado à objetos da classe *Acception*.

***AcceptionAdapter*:** É a classe intermediária entre o XML das solicitações e a *Acception*. Essa classe fornece as solicitações criadas à *View* que as exibe aos usuários.

- ***getView*:** É um método polimórfico da *ArrayAdapter*. Atribui o *layout* para cada solicitação e adiciona os *listeners* nos botões de aceitação e recusa das solicitações. Se o usuário aceitar a solicitação, atualiza o atributo response da solicitação para *true*. Se recusar, atualiza para *false*.

***ArrayAdapter<Chat>*:** A *ArrayAdapter* é a classe do Android que retorna a interface de cada objeto de uma lista de objetos *View* da *ListView*. Neste caso, é um *ArrayAdapter* dedicado à objetos da classe *Chat*.

***ChatAdapter*:** É a classe intermediária entre o XML das salas de mensagens e a *Chat*. Essa classe fornece as salas de mensagens criadas à *View* que as exibe aos usuários.

***getView*:** É um método polimórfico da *ArrayAdapter*. Atribui o *layout* para cada sala de mensagens.

***ArrayAdapter<AppMessage>*:** A *ArrayAdapter* é a classe do Android que retorna a interface de cada objeto de uma lista de objetos *View* da *ListView*. Neste caso, é um *ArrayAdapter* dedicado à objetos da classe *AppMessage*.

***MessageAdapter*:** É a classe intermediária entre o XML das mensagens e a *AppMessage*. Essa classe fornece as mensagens enviadas à *View* que as exibe aos usuários.

***getView*:** É um método polimórfico da *ArrayAdapter*. Atribui o *layout* para cada mensagem. Se a mensagem não for de texto, trata a imagem para exibi-la ao usuário.

***SetUpCardManager*:** É a classe responsável pelo gerenciamento da comunicação com os serviços do *Virgil Security*.

***setupCrypto*:** Instancia o objeto da classe *VirgilCardCrypto* do *Virgil Security*.

***generateToken*:** Gera o *token* através do identificador do usuário para poder realizar chamadas HTTP para os serviços do *Virgil Security*.

***setupAccessTokenProvider*:** É o método acionado sempre que uma chamada HTTP é realizada com destino ao *Virgil Security*. Se o *token* gerado pelo método *setupCrypto* for válido, completa a chamada. Senão, joga uma exceção.

***initializeCardManager*:** Instancia o objeto da classe *CardManager* do *Virgil Security* que possui como parâmetro a função *setupAccessTokenProvider* para defini-la como intermediária das requisições.

***CryptoEEHelper*:** É a classe responsável pela criptografia das mensagens da aplicação.

***getCardManager*:** Método de um *singleton* para obter uma única instância do objeto da classe *CardManager* do *Virgil Security*.

***getContextInstance*:** Método de um *singleton* para obter uma única instância do objeto da classe *Context* do Android.

***getPrivateKeyStorage*:** Método de um *singleton* para obter uma única instância do objeto da classe *PrivateKeyStorage* do *Virgil Security*.

***getVirgilCrypto*:** Método de um *singleton* para obter uma única instância do objeto da classe *VirgilCrypto* do *Virgil Security*.

***createCard*:** É o método onde é criado e publicado os cartões no *Virgil Security*. Um cartão possui um identificador do usuário e suas chaves públicas. Além de ser assinado pela chave privada do usuário.

***encrypt*:** Método responsável pela encriptação das mensagens com as chaves públicas dos participantes da sala de mensagens.

***decrypt*:** Método responsável pela decriptação das mensagens com a chave privada do usuário.

***OrbotHelper*:** É a classe que possui metódos relacionados ao *Orbot* que possam ser acessíveis de qualquer classe da aplicação.

***isAppInstalled*:** Acessa o sistema de arquivos do dispositivo e verifica se o *Orbot* está instalado.

K.4 *Service*

A Figura 53 contém o diagrama de classes do pacote *Service*.

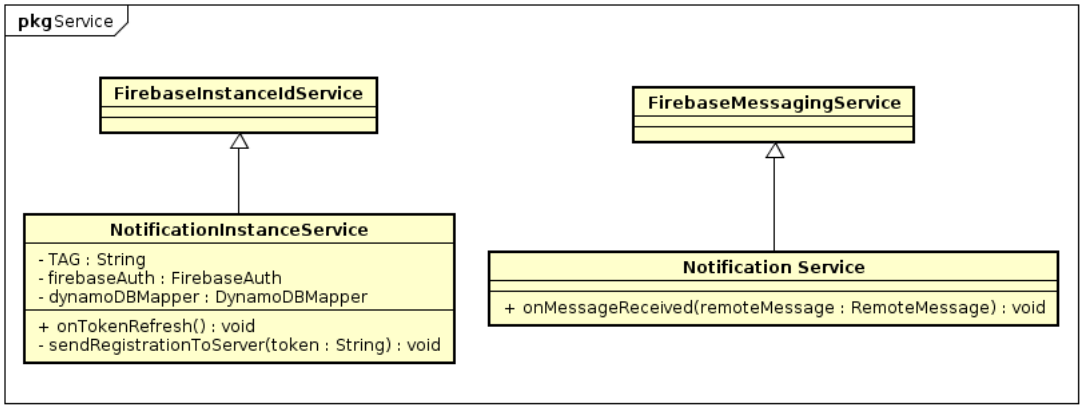


Figura 53 – Diagrama de Classes do Pacote *Service*.

A seguir, serão detalhadas os papéis principais de cada classe contida no modelo.

***NotificationInstanceService*: *onTokenRefresh*:** Invoca o método *sendRegistrationToServer* e é acionado sempre que o aplicativo é instalado no dispositivo.

***sendRegistrationToServer*:** Registra no *DynamoDB* o identificador do usuário associado com o *token* do aplicativo no dispositivo.

***NotificationService*: *onMessageReceived*:** É acionado sempre que a aplicação recebe uma notificação. Recebe o conteúdo da notificação e adiciona ao *log* do sistema.