UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Matheus Eyng de França

Sistema para Disponibilização de Recursos Computacionais Locais Remotamente

Florianópolis
2022

Matheus Eyng de França

Sistema para Disponibilização de Recursos Computacionais Locais Remotamente

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Jean Everson Martina, Dr.
Coorientador: Pablo Rinco Montezano

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

França, Matheus
Sistema para Disponibilização de Recursos Computacionais Locais Remotamente / Matheus França ; orientador, Jean Martina, coorientador, Pablo Montezano, 2022.
143 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2022.

Inclui referências.

1. Ciências da Computação. 2. Computação Distribuída. 3. Acesso Remoto. 4. Dispositivos Criptográficos. I. Martina, Jean. II. Montezano, Pablo. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Matheus Eyng de França

Sistema para Disponibilização de Recursos Computacionais Locais Remotamente

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 21 de Dezembro de 2022.

_____

Prof. Jean Everson Martina, Dr
Coordenador do Curso

Banca Examinadora:

_____

Prof. Jean Everson Martina, Dr.
Orientador
Universidade Federal de Santa Catarina

_____

Pablo Rinco Montezano
Avaliador
Universidade Federal de Santa Catarina

_____

Lucas Machado da Palma, M.e
Avaliador
Universidade Federal de Santa Catarina

_____

Marlon Trapp
Avaliador

AGRADECIMENTOS

Agradeço primeiramente aos meus pais por sempre me apoiarem incondicionalmente nas minhas escolhas, por prezarem por uma educação de qualidade e por sempre batalhar incansavelmente para me proporcionar a melhor educação ao seu alcance.

Agradeço aos mestres pelos ensinamentos os quais não apenas se restringem ao âmbito acadêmico, cuja dedicação ao ofício eleva a qualidade, não só da instituição, mas da população como um todo. Agradeço especialmente ao meu orientador, Prof. Jean Martina.

Aos colegas e amigos que fiz durante a graduação, que me acompanharam durante esta jornada, os quais dividiram junto comigo muitas risadas, madrugadas de estudo, e momentos de frustração e regojizo. Agradeço também a todos aqueles da equipe do LabSEC, onde passei a maior parte da graduação e foi fundamental no meu crescimento acadêmico e pessoal, em especial ao meu amigo e mentor Pablo Montezano, por sua paciência, seus ensinamentos e por sempre acreditar no meu potencial.

RESUMO

Com o advento da indústria 4.0 e o contexto de um mundo pós pandemia, viu-se a migração de diferentes serviços de software para a nuvem. Sistemas de acesso a dispotivos de hardware no entanto, têm dificuldade para adaptar ao ambiente de sistemas distribuídos sendo em sua maioria desenvolvidos para acesso local. Sendo assim, este trabalho propõe a criação de um sistema de gerenciamento de dispositivos de hardware usando técnicas de computação distribuída, com foco na segurança da transmissão dos dados, que complementa e enriquece uma aplicação previamente desenvolvida chamada Device Server. Como resultado, este trabalho apresenta uma aplicação complementar ao Device Server chamada Device Cliente, melhorias na segurança das comunicações do do Device Server, assim como um novo paradigma de comunicação através de um sistema de mensageria. No mais é apresentado também uma implementação gráfica para a utilização do novo sistema que consiste no conjunto das aplicações Device Client, Device Server Client, ambos integrados através de uma infraestrutura de comunicação por mensagens.

Palavras-chave: Computação Distribuída. Segurança. Gerenciamento de dispositivos. Device Server.

ABSTRACT

With the advent of industry 4.0 and the context of a post-pandemic world, different software services migrated to the cloud. Access systems for hardware devices, however, have difficulty adapting to the environment of distributed systems, most of which are meant for local access. Therefore, this work proposes the creation of a hardware device management system using distributed computing techniques, focusing on data transmission security, which complements and enriches a previously developed application called Device Server. As a result, this work presents a complementary application to Device Server called Device Client, improvements in the security of Device Server communications, as well as a new communication paradigm through a messaging system. Furthermore, a graphical implementation is also presented for the use of the new system, which consists of the set of Device Client and Device Server Client applications, both integrated through a message communication infrastructure.

Keywords: Distributed Computing. Security. Mevice Management. Device Server.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Com a virada do século, ficou claro que a internet desenvolveria um papel essencial não somente no cotidiano do cidadão comum, mas principalmente no mundo dos negócios. A ubiquidade da internet fez com que houvesse uma revolução na computação, principalmente em áreas relacionadas a computação distribuída. Para muitas empresas, não há mais a necessidadeque os funcionários habitem a mesma cidade, ou até mesmo no mesmo país. Graças a várias ferramentas desenvolvidas nas últimas décadas, o trabalho remoto passou a ser não apenas economicamente viável, mas começa a ser visto como parte do futuro de todas as áreas da indústria 4.0 (AVDEEVA et al, 2021).

O ano de 2020 foi crucial para por a prova a viabilidade do trabalho remoto. A pandemia de COVID-19 desafiou não somente a saúde pública mas também o mercado global. Em apenas alguns meses, empresas tiveram que adotar alguma maneira de trabalhar remotamente, devido a necessidade de lockdowns em diversos países para conter a propagação do vírus pandêmico. O impacto destes acontecimentos foi que grande parte da população foi forçada a exercer o trabalho de suas próprias casas, sendo que grande parte destas pessoas não possuem condições propícias para trabalhar, seja por falta de ferramentas, conexão estável ou até mesmo por falta de um ambiente adequado (RALPH et al., 2020). Com isso, ferramentas de trabalho remotos, como salas de reuniões online, passaram a ter um papel integral no dia a dia das pessoas.

Apesar de existirem muitas ferramentas de acesso remoto e de comunicação por rede, nota-se uma falta de softwares que permitam o acesso de recursos locais, como dispositivos USB por exemplo, que sejam agnósticos quanto o tipo e a finalidade destes dispositivos. A Microsoft oferece opções para compartilhar recursos, mas isto limita o ambiente de trabalho ao sistema operacional Windows. Outras empresas como FlexiHub oferecem uma solução mais genérica para acesso a dispositivos USB remotamente, porém por ser uma aplicação voltada ao mercado, é cobrado por uso sendo o preço dos planos oferecidos não transparentes ao cliente a menos que este esteja disposto a entrar em contato com o time de vendas.

Este problema se torna ainda mais evidente quando visto do contexto de desenvolvimento de software. Em um ambiente de desenvolvimento é comum que tenha alguma interação entre recursos hardware e software, como algum dispositivo USB, assim a necessidade de acesso a estes recursos é imprescindível. Surge então a necessidade de se deslocar até o local onde estes recursos estão disponíveis ou, se possível, trazê-los consigo de maneira que eles possam estar sempre acessíveis. Entretanto, comumente o ambiente de desenvolvimento não pode ser facilmente deslocado, seja por questões logísticas ou até mesmo de segurança, tornando o deslocamento do recurso inviável.

Este trabalho tem como objetivo a criação de um sistema onde seja possível acessar recursos computacionais locais de maneira remota e distribuída, provendo meios de autenticação através de certificados digitais e chaves criptográficas para garantir segurança e autenticidade de acesso.

O trabalho está organizado como segue: no capítulo 2 são apresentados os objetivos deste projeto. No capítulo 3 têm-se o referencial teórico necessário para a compreensão das escolhas feitas e dos conceitos discutidos durante a implementação. No capítulo 4 têm-se a

1.1 JUSTIFICATIVA

No Laboratório de Segurança em Computação (LabSEC) da Universidade Federal de Santa Catarina (UFSC) foi desenvolvido, como fruto de um trabalho de conclusão de curso do aluno Marlon Trapp (TRAPP, 2016), um software para comunicação com dispositivos criptográficos. O Device Server é um software de comunicação segura entre cliente e dispositivo criptográfico, mais comumente, tokens criptográficos. Atualmente esta aplicação é usada em conjunto com o projeto Hawa, o qual é responsável pelas autoridades certificadoras finais da ICP-Brasil, e com outras autoridades certificadoras de segundo nível. Apesar do projeto servir seu propósito, se viu necessário fazer certas adaptações para modernizar a abordagem dos processos. Nos últimos anos houve um questionamento quanto a necessidade de aplicações monolíticas, pricipalmente quanto a sua dificuldade de adaptação a diferentes ambientes.

Neste contexto, o projeto existente se mostrou limitado e inflexível. A possibilidade de comunicação com apenas um dispositivo criptográfico limita seus casos de uso em um abiente onde a autenticação pode necessitar mais de um dispositivo. Além disso, a necessidade do recurso estar disponível localmente praticamente impossibilita uma implementação distribuída moderna para os outros projetos dependentes.

Tendo como caso de uso o funcionamento da ICP-Brasil, este trabalho visa desenvolver uma solução distribuída e dinâmica, permitindo a implantação em um ambiente em nuvem, onde seja suportado diversos recursos diferentes atrelados a diversos usuários, de modo que esta solução possa substituir a antiga e proporcionar um novo paradigma nos softwares dependentes da ICP-Brasil.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Este trabalho tem como objetivo a criação de um sistema onde seja possível acessar recursos computacionais locais de maneira remota e distribuída, provendo meios de autenticação através de certificados digitais e chaves criptográficas para garantir segurança e autenticidade de acesso.

2.2 OBJETIVOS ESPECÍFICOS

1. Desenvolver uma solução que permita a autenticação do usuário a um ou mais dispositivos criptográficos.
2. Desenvolver um mecanismo de autenticação de um usuário a um sistema.
3. Permitir que o usuário tenha acesso a estes dispositvos de maneira remota, sem que eles precisem estar na máquina do mesmo.
4. Desenvolver uma solução para gerenciar todos os dispositivos e usuários de maneira dinâmica, onde possam ser adicionados ou removidos tanto usuários quanto dispositivos de acordo com a necessidade.

3 REFERENCIAL TEÓRICO

Neste capítulo serão apresentados conceitos os quais julgou-se necessário ter certo nível de conhecimento para que haja a melhor compreensão das decisões tomadas ao decorrer do desenvolvimento deste trabalho.

3.1 CRIPTOGRAFIA ASSIMÉTRICA

O mundo da criptografia pode er divido em duas categorias básicas, criptografica simétrica e assimétrica. Durante muito tempo conheceu-se apenas a criptografica simétrica, a qual utiliza uma chave única para cifrar e decifrar mensagens. Digamos que João queira mandar uma mensagem para Maria, mas deseja que apenas Maria possa ler esta mensagem. O método simétrico de criptografia consistiria em João cifrar esta mensagem com uma chave criptográfica, e então enviar a mensagem cifrada para Maria. Contudo, Maria apenas conseguiria obter a mensagem original tendo posse da chave que a cifrou, ou seja, tem-se a necessidade de um outro canal seguro para transmitir a chave entre João e Maria, sendo que qualquer tipo de interceptação neste outro canal tornaria todo o processo nulo.

A criptografia assimétrica resolve este problema de maneira elegante. Primeiramente proposta em público por Diffie e Hellman (1976), esta consiste na existência de duas chaves, uma privada, que não deve ser compartilhada como o nome sugere, e uma pública, derivada desta chave privada, que pode ser compartilhada livremente. A proposta original será discutida mais a fundo adiante, mas temos a proposta de Rivest, Shamir e Adleman (1978) a qual se tornou a solução mais amplamente usada, e continua sendo usada atualmente conhecida como RSA.

Voltando ao problema de João e Maria, agora, imaginemos que ambos possuem um par de chaves. João ainda deseja mandar uma mensagem cifrada para Maria, mas desta vez, João conhece a chave pública de Maria, então usa esta para cifrar a mensagem. Segundo o conceito principal do protocolo RSA, apenas o par de chave correspondente pode decifrar uma mensagem. Sendo assim, a mensagem chega até Maria, a qual consegue decifra-la com sua chave privada. Assim, mesmo havendo o compartilhamento de chaves públicas pelo meio de comunicação, o método continua seguro, pois é necessário o seu par correspondente para decifrar a mensagem, diferente do método simétrico, onde qualquer um que obtivesse a chave única poderia decifrar a mensagem.

3.2 FUNÇÃO HASH

Uma função hash define-se por uma transformação onde a entrada é uma sequência de bytes de tamanho qualquer e a saída produzida é uma sequência de

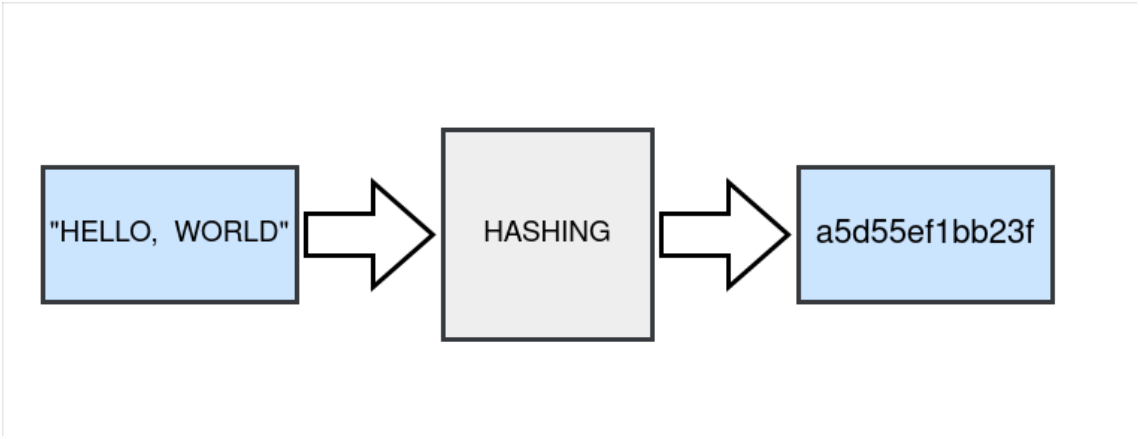

Figura 1 – Funcionamento da função de hashing, autoria própria.

tamanho fixo, como exemplificado na figura 1. A saída desta função é conhecida como resumo criptografico, em inglês *digest*.

Devido ao número de saídas possíveis ser menor do que o número de entradas, pode haver casos em que duas entradas diferentes produzem uma mesma saída, sendo esse fenômeno conhecido como colisão. Podemos entender então que para uma função hash ser considerada "boa", para um grande conjunto de entradas, as saídas serão aparentemente aleatórias (STALLINGS, 2017).

Para uso em criptografia, uma função hash precisa atender alguns requisitos mais específicos. Assim, uma função hash criptografica tem entre suas principais características:

1. Dada a função hash H e uma saída x, é computacionalmente inviável calcular X, tal que H(X) = x.

2. Produzir uma saída, independente do tamanho da entrada, seja relativamente fácil para um computador.

3. A duplicação de saídas iguais para entradas diferentes deve ser mínima.

Assim, no contexto de infraestrutura de chaves públicas, as funções hash são essenciais não somente em otimizar as computações necessárias, mas principalmente em verificar a integridade dos dados manipulados, uma vez que qualquer tipo de alteração implicaria em um resumo diferente do esperado.

3.3 ASSINATURA DIGITAL

A assinatura digital é um algoritmo matemático que permite que um usuário verifque a autenticidade de uma mensagem. Segundo Rivest (1990), um esquema de assinaturas garante uma maneira para que cada usuário assine uma mensagem, de maneira que estas assinaturas possam ser mais tarde verificadas por qualquer

um. Portanto, as assinaturas digitais tem papel importante no mundo da criptografia e servem como base de diversos protocolos criptográficos.

Utilizando a criptografia assimétrica, as assinaturas digitais são cifragens do resultado de uma função hash, o *digest*, a partir de uma chave privada. Desta maneira, permite-se que através do par público desta chave privada, seja possível realizar a decifragem e obter novamente o *digest* original, verificando-se assim a autenticidade da mensagem.

As assinaturas digitias então, garantem três propriedades fundamentais:

- **Autenticidade**: A mensagem recebida, ao ser decifrada com a chave pública de quem a mandou, confirma de maneira incontestávelque o proprietário da chave privada equivalente foi quem originou esta mensagem.
- **Integridade**: É comum haver a necessidade de verificar que dada mensagem não foi alterada durante o seu transporte. Desta maneira, entendendo as propriedades da função hash, pode-se garantir atavés da assinatura digital, que não houve alteração no conteúdo durante o seu caminho até o receptor.
- **Não repúdio**: Uma vez feita a assinatura, as propriedades matemáticas nos garantem que o dentor daquele par de chaves foi quem fez a assinatura, ou seja, o emissor não pode negar a autoria da mensagem.

Apesar de todas essas garantias e proriedades, no âmbito de troca de mensagens, ainda se vê necessário que o possuidor das chaves possa ser identificado de maneira definitiva e incontestável. Uma solução possível e amplamente utilizada que resolve este problema é o uso de certificados digitais, que serão explicados a seguir.

3.4 CERTIFICADO DIGITAL

Um certificado digital é um artefato o qual liga uma entidade a um chave pública. É definido de maneira genérica pelo padrão X.509, elaborado pela União Internacional de Telecomunicações, e de maneira mais rigorosa pela RFC 5280, (BOEYEN et al., 2008). A RFC padroniza a versão 3 da estrutura X.509 quanto aos tipos de dados aceitos, bem como a sua forma.

Um certificado digital é composto de três seções diferentes, uma a qual contém dados que identificam o portador, podendo variar na forma e conteúdo, outra seção contendo a chave pública do portador, e por fim, uma assinatura digital feita por uma terceira parte confiável, chamada de autoridade certificadora (AC). Outra característica integral de um certificado é que este também possui uma data de validade.

Na prática, um certificado deve, inequivocamente, identificar a parte dententora da chave pública (KATZ; LINDELL, 2014). Assumindo-se um certo grau de confiança na AC, que sua chave privada é guardada de maneira segura e não está sujeita

a vazeamentos ou ataques, pode-se garantir a autenticidade do certificado devido a presença da assinatura desta autoridade certificadora. Assim, estes se mostram muito resilientes a fraudes, e portanto são uma maneira segura de confirmar a posse de um certo par de chaves a uma dada entidade.

Sendo assim, os certificados digitais se mostram uma maneira prática de realizar uma troca de chaves, substituindo em muitos casos métodos mais complexos, como o algoritmo de Diffie-Hellman. Podemos entender então por que estes constituem parte integral nos diversos processos de autenticação utitlizados atualmente.

3.5 AUTORIDADE CERTIFICADORA

A autoridade certificadora é a terceira parte que constitui uma infraestrutura de chaves públicas. Segundo Kent (1998), uma autoridade certificadora tem basicamente dois papéis, emitir certificados e emitir uma lista de certificados revogados (LCR).

Autoridades certificadoras são entidades que possuem garantias sobre a vários aspectos quanto a sua integridade de dados e comunicação. Cada autoridade certificadora possui um certificado próprio, o qual foi emitido por outra AC, formando desta maneira uma cadeia que leva, eventualmente, a um certificado auto-assinado, identificando assim uma AC raíz.

As AC's cujo certificado não é autoassinado são chamadas de AC's intermediárias e podem pertencer a empresas privadas ou orgãos governamentais, porém, uma AC raiz quase sempre pertencerá a algum governo. Dessa maneira, garantindo a autenticidade da AC raíz, garante-se também a autenticidade de todos os certificados emitidos por ela, criando uma cadeia de confiança

Quanto a LCR, esta é definida na RFC 5280 por Boeyen et al. (2008) como sendo a lista de todos os certificados não-expirados que foram revogados por um motivo ou outro. Com isso, no processo de verificação de autenticidade de um dado certificado, é preciso consultar a LCR para certificar-se que o mesmo não conste nesta lista.

3.6 SMART CARDS E TOKENS CRIPTOGRÁFICOS

Smart cards e tokens criptográficos são dispositivos físicos usados para guardar informações de maneira segura, podendo também realizar algoritmos criptograficos, graças aos seus pequenos, porém eficientes, processadores internos.

Segundo Rankl e Effing (2010), a característica principal de um smart card é o fato de possuirem embutidos em sua estrutra um circuito integrado, que possui componentes para transimitr, armazenar e processar dados. Sendo uma evolução dos cartões de fita magnéticas, os smart cards além de possuirem uma capacidade maior de armazenamento, possuem ainda um processador capaz de realizar diversas

tarefas. Apesar das limitações de tamanho, estes pequenos processadores são feitos sob medida para cada aplicação diferente, o que os torna inflexíveis porém extremamamente eficientes.

Quando vistos sob um contexto de criptografico, smart cards são uma boa opção, de acordo com Naccache e M'Raihi (1996), devido ao fato que os dados nunca precisam sair do cartão em para serem usados, além de possuirem um chipe resistente a adulteração e capacidade de realizarem algoritmos criptográficos. Como cada processador é feito para uma aplicação específica, mesmo algoritmos criptográficos, como assinatura e geração de chaves por exemplo, que demandam certo poder computacional, podem ser facilmente implementados em hardware, tornando o seu uso muito comum para operações com chaves públicas.

Um dos principais fatores de segurança de um smart card é que este possui diferentes tipos de memórias, sendo algumas delas acessíveis apenas para o próprio sistema operacional. De acordo com Rankl e Effing (2010), os smart cards possuem os diferentes tipos de memória:

- **ROM**: Este tipo de armazenamento permite apenas a leitura de dados, porém apenas para o próprio uso interno. Aqui armazena-se informações sobre o sistema operacional do dispositvo, sendo comumente impossível para o usuário ler ou alteras os dados presentes.
- **RAM**: Assim como a memória usada em computadores pessoais, toda informação escrita aqui é apenas para uso temporário e tudo é perdido após o fim do uso do smart card.
- **EEPROM**: Este é um tipo de memória não-volátil, e é usada para armazenar dados que podem ser alterados mais tarde, como identificação do usuário, o PIN do cartão, entre outros.

Os tokens criptográficos por sua vez, são dispositivos que contém um chipe de smart card, herdando assim todas as características mencionadas anteriormente, porém agora com uma interface de comunicação mais genérica, usualmente USB. O tokens podem possuir algumas características a mais, como uma bateria interna, algum tipo de botão, ou até mesmo um leitor de biometria. Todavia, ainda segundo Rankl e Effing (2010), a única diferença entre um smart card e um token que realizam a mesma funcionalidade é apenas sua aparência.

3.7 AUTENTICAÇÃO

De acordo com Woo e Lam (1992), a autenticação é um processo onde uma entidade ligada a uma certa identidade em um sistema tenta conectar-se ao mesmo, produzindo então um processo de verificação. Assim, em um sistema, autenticação

consiste em determinar se algum serviço ou usuário que deseja conectar-se é, de fato, quem eles afirmar ser.

É importante também esclarecer a diferença entre *autenticação* e *autorização*, dois conceitos comumente tratados como se fossem a mesma coisa porém bem distintos na prática. Autorização é o processo de determinar se uma entidade tem as permissões necessárias para acessar um determinado recurso, necessitando para isso que o passo anterior, a autenticação tenha sido realizado com sucesso (KIM; LEE, 2017). Ambos os conceitos são muito importantes quando se deseja construir uma aplicação que funcione de maneira distribuída, porém neste trabalho abordaremos mais a fundo a questão da autenticação.

A implementação de um serviço de autenticação em um sistema distribuído precisa levar em conta que não apenas tem-se usuários tentando conectar-se ao sistema, mas também as próprias partes do sistema precisam comunicar-se uma com as outras de maneira que estas partes tenham a garantia que as mensagens trocadas não pertencem a um terceiro tentando impersonar uma destas partes a fim de obter informações confidenciais. Desta maneira, os maiores desafios em questão de autenticação está na comunicação entre serviços, onde deve-se levar em conta a complexidade e o custo da implementação destes mecanismos em cada serviço (ALMEIDA; CANEDO, 2022).

3.8 SSL/TLS

A década de 1990 foi marcada pela popularização e rápida adoção da internet pela população em geral, e também ao evento que mais tarde seria conhecido como a bolha da internet. Devido a esta popularização, ficou claro a necessidade de fazer com que as comunicações ocorressem de forma segura. Em 1993 a empresa Netscape, lançou junto com seu navegador de internet o protocolo de segurança SSL, ou em português, camada de soquete seguro, o qual era uma camada acima da pilha TCP/IP.

Como a rede mundial tornava-se cada vez mais complexa, percebeu-se que garantir a segurança da rede como um todo era uma tarefa praticamente impossível, ao invés disso, a melhor maneira de proteger o tráfego de dados seria com cifragem e decifragem nos pontos finais de conexão (CHOU, 2002). O SSL forneceu uma interface de comunicação muito similar ao TCP, permitindo que diversas aplicações pudessem desfrutar desta camada de segurança.

De acordo com Stallings e Brown (2014), o SSL não é apenas um protocolo único mas na verdade duas camadas de protocolos que estão acima da pilha TCP, como mostra a figura 2.

Ainda de acordo com Stallings e Brown (2014), os dois conceitos mais importantes no SSL podem ser descritos da seguinte maneira:

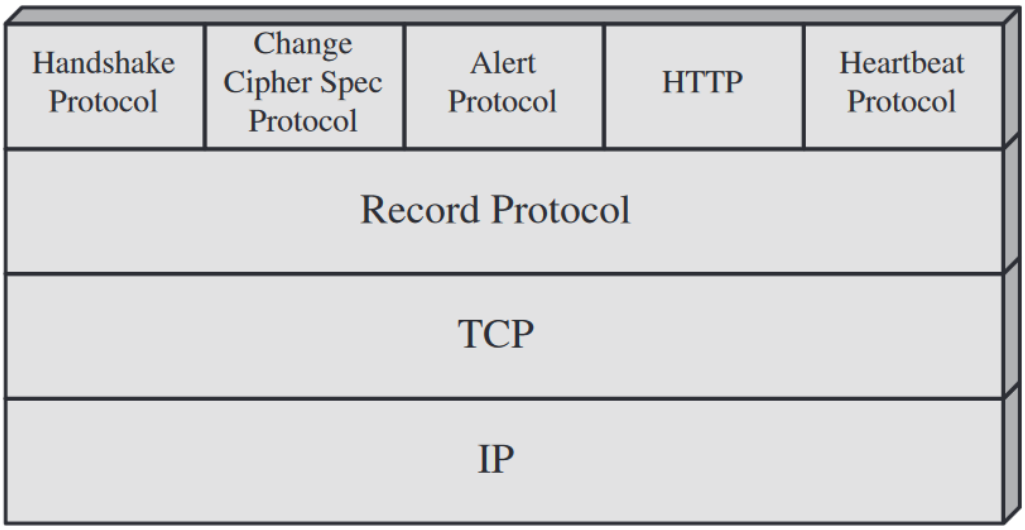

Figura 2 – Pilha do protocolo SSL sobre a pilha TCP, (STALLINGS; BROWN, 2014).

- **Conexão**: É o elemento de transporte, segundo a definição OSI do conceito, representado por ligações ponto a ponto. Toda conexão é associada a uma única sessão.
- **Sessão**: Uma sessão é uma associação entre dois pontos de conexão, usualmente um cliente e um servidor. A criação de sessões segue um protocolo chamado de Protocolo de Apresentação (*Handshake Protocol*) onde são definidos parâmetros criptográficos os quais podem ser compartilhados entre várias conexões, evitando assim negociações de novos parâmetros, o que pode ser bastante dispendioso.

A parte principal do SSL é como ocorre o protocolo de apresentação. Para melhor entender como é feito o acordo criptográfico e a segurança da comunicação, podemos dividr este protocolo em 4 fases:

- **Fase 1:** O cliente envia uma mensagem *client_hello*, a qual contém a versão mais atual do SSL entendida pelo cliente, um número aleatório, um identificador de sessão, uma lista de algoritmos de cifragem entendidos pelo cliente e uma lista de métodos de compressão suportadas pelo cliente.

  Com isso, o servidor tem todas as informações básicas necessárias para iniciar a conexão. O servidor deve então responder com um *server_hello*, o qual contém os mesmos parâmetros da mensagem do cliente.

- **Fase 2:** O servidor envia o seu certificado ao cliente, uma requisição de certificado, uma mensagem opcional podendo requisitar mais informações criptográficas, e uma mensagem *server_done* sinalizando o final desta fase.

- **Fase 3:** O cliente envia seu certificado ao servidor, e qualquer outra informação a mais requisitada na fase anterior. Ambos cliente e servidor verificam se os certificados recebidos são válidos ou não. O cliente manda uma mensagem contendo parâmetros de geração de chave, cifrados com a chave pública do servidor. Então ambos cliente e servidor usam estes parâmetros para gerar uma mesma chave em comum.
- **Fase 4:** O cliente então por fim, envia uma mensagem sinalizando o algoritmo de cifragem a ser usado, juntamente com uma mensagem de término, criptografada com o algoritmo escolhido e a chave gerada anteriormente. O servidor faz a decifragem da mensagem, caso nenhuma etapa anterior tenha falhado, então a mensagem é decifrada com sucesso. Com isso o servidor envia uma mensagem confirmando a troca do algoritmo de cifragem e uma mensagem de término criptografada. A partir de agora, todas as mensagens trocadas entre os dois serão criptografadas de acordo com o combinado neste protocolo.

O TLS surgiu da padronização dos protocolos primeiramente definidos no SSL, pela *Internet Engineering Task Force* (IETF), a qual é responsável por definir e documentar os padrões adotados pela internet como um todo. Segundo Rescorla (2018), o objetivo do TLS é prover um canal seguro o qual oferecendo autenticação, confidencialidade e integridade.

O TLS 1.0 foi uma atualização da versão 3.0 do SSL, a qual segundo os autores, Allen e Dierks (1999), não é dramática o suficiente a ponto de mudar a meneira como funciona no geral, mas suficiente para não ser interoperante com o SSL 3.0. Têm-se assim o surgimento do TLS a partir da RFC 2246.

A partir de então, o SSL foi defasado em troca do TLS o qual continua recebendo atualizações e revisões pela IETF, tornando-se o sucessor deste protocolo tão importante para a comunicação segura na web. Diante disso, é muito comum encontrar até mesmo na literatura, referências a SSL e TLS como sendo sinônimos.

3.9 PROTOCOLO DIFFIE-HELLMAN

O Protocolo de Diffie-Hellman consiste um uma maneira de se transmitir uma chave criptográfica entre duas partes de maneira segura através de um meio inseguro de modo que não haja nenhum acordo prévio entre as partes. Tal chave pode então ser usada para criptografar as mensagens usando um esquema de cifragem de chave simétrica.

Em meados dos anos 70, o desnvolvimento de comunicações controladas por redes de computadores prometia contato entre pessoas ao redor do mundo de maneira fácil e barata, contudo com isso surgiu-se a necessidade de tornar estas comunicações seguras ao passo que tais mensagens não poderiam ser bisbilhotadas,

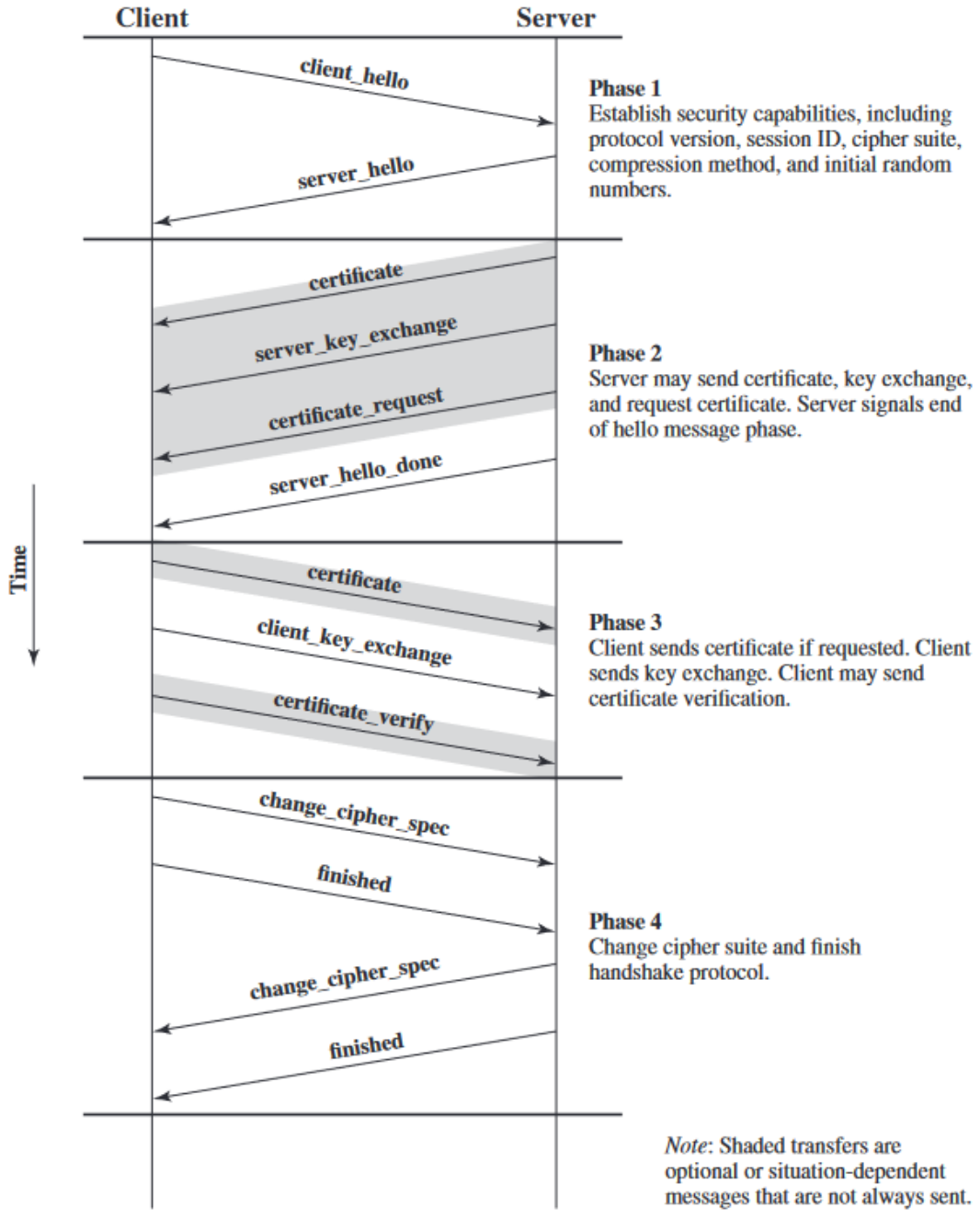

Figura 3 – Protocolo de autenticação do SSL, (STALLINGS; BROWN, 2014)

algo que na época não era factível com o modelo de criptografia (DIFFIE; HELLMAN, 1976).

Tentando solucionar este problema, em 1976 Whitfield Diffie e Martin Hellman publicaram um artigo o qual descrevia o protocolo que ficaria conhecido como Diffie-Hellman e serviria de base para a criptografia de computadores nas próximas décadas até os dias de hoje.

Segundo Diffie e Hellman (1976), o protocolo permite que dois usuários cheguem a uma chave em comum e mantenham suas comunicações criptografadas de maneira que um terceiro agente escutando estas comunicações encontre que é computacionalmente inconcebível calcular esta chave pelas informações capturadas.

De acordo com Li (2010), o método de Diffie-Hellman possui dois grandes atrativos:

- A criação de chaves, algo dispendioso computacionalmente, se da apenas quando necessário. Não há a necessidade de armazenar estas chaves de modo a expô-las a diferentes vulneribilidades.
- A troca de chaves não depende de nenhuma infraestrutura pré-existente, apenas que ambas as partes tenham um acordo básico quantos as variáveis globais do protocolo.

Com isso, a criação deste protocolo foi um marco importante não apenas para a comunicação entre computadores, mas também para o mundo da criptografia como um todo, de maneira que mesmo nos tempos atuais este protocolo ainda é amplamente usado.

A execução do protocolo é relativamente simples e pode ser facilmente demonstrada. Suponha que Alice e Bob desejam se comunicar através de uma rede insegura, porém é do interesse de ambos que as mensagens trocadas não possam ser entendidas por mais ninguém. Para executar o algoritmo de Diffie-Hellman, Alice e Bob precisam seguir apenas 5 passos:

1. Alice e Bob precisam concordar em um parâmetro p, chamado de módulo, o qual é um número primo, e um parâmetro q chamado de base, o qual é uma raíz primitiva de p. Para acordar entre esses parâmetros, suponha que Alice gere então um p e um q e os envie para Bob por meio do canal de comunicação.
2. Alice então escolhe um número secreto a, e envia para Bob A, tal que $A = q^a \mod p$.
3. Bob por sua vez escolhe um número secreto b, e envia para Alice $B = g^b \mod p$.
4. Alice computa então K de maneira que $K = B^a \mod p$

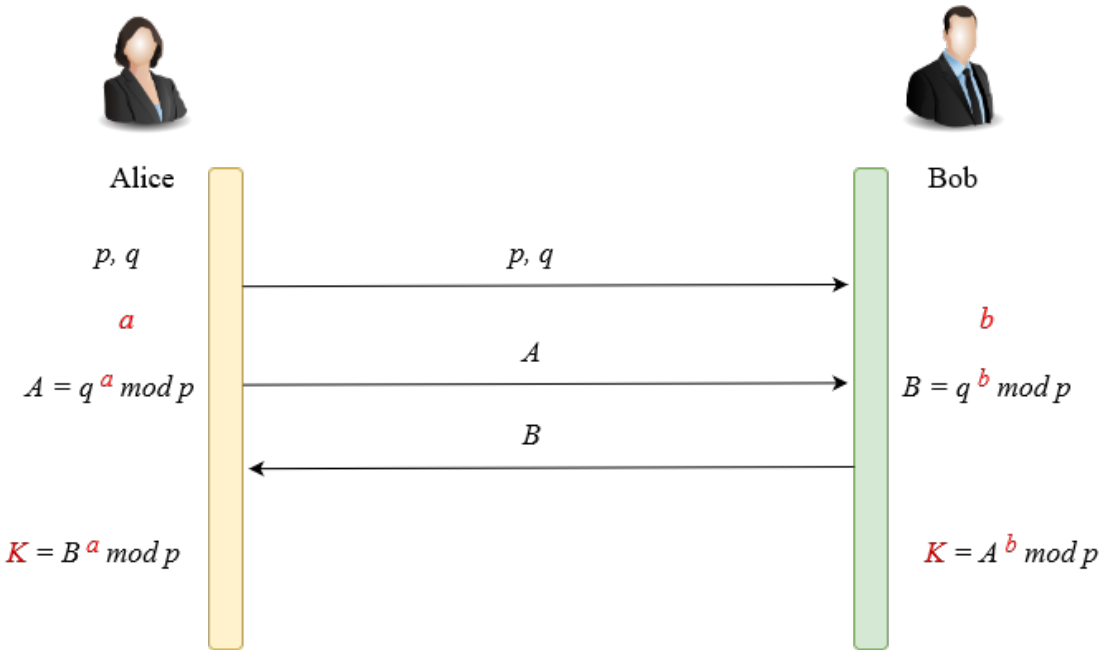

Figura 4 – Protocolo de Diffie-Hellman, autoria própria.

5. Bob faz o mesmo, obtendo K de maneira que $K = A^b \mod p$

Desta maneira ambos Alice e Bob geram a mesma chave K utilizando parâmetros públicos e parâmetros secretos. Como alguns destes parâmetros públicos dependem de segredos gerados por ambos, um atacante que deseja decifrar as mensagens precisaria fazer as operações com logaritmos. Assim, tendo escolhido bons parâmetros para p e q, a descoberta destes de maneira inversa se torna uma tarefa computacionalmente inviável. Com isso, Alice e Bob conseguem então cifrar mensagens de maneira que apenas eles possam decifrar utilizando a mesma chave em comum.

3.10 SISTEMAS DE MENSAGERIA

Softwares modernos raramente vivem em isolamento, sendo que atualmente vemos cada vez mais aplicações adotando o paradigma da computação distribuída. Ao mesmo tempo, tem-se percebido uma tendência crescente na taxa de coleta e transmissão de dados, como o por exemplo o Facebook que maneja cerca de 10^6 requisições por segundo, o que representa uma taxa de aproximadamente 9GB/s (LU et al., 2014).

Com isso surge a necessidade de desenvolver novas maneira de trocar uma alta taxa de mensagens de maneira efetiva. O HTTP, protocolo original para a internet proposto por Tim Berners-Lee, já não atende mais as demandas de um ambiente de alta vazão, sendo um protocolo verboso e demasiadamente pesado (NAIK, 2017).

Desta maneira, desenvolveu-se nos útlimos anos diversos sistemas de mensageria, os quais implementam protocolos mais leves, de comunicação assíncrona, permitindo uma alta vazão de dados. Estes sistemas usualmente são implementados com diversas redundâncias e garantias, permitindo assim uma abordagem *fire-and-forget* das aplicações que os usam, ou seja, o envio correto das mensagens é garantido pelo sistema permitindo assim que as aplicações distribuídas possam escalar seus serviços de maneira descomplicada.

4 DESENVOLVIMENTO

Neste capítulo serão apresentadas as escolhas feitas durante o desenvolvimento do projeto. Em um primeiro momento iremos apresentar o *Device Server* elaborado por Trapp (2016), mostrando algumas características importantes quanto ao seu funcionamento, bem como alguns pontos onde este pode ser melhorada e/ou modificado. Tendo o Device Server como objeto de estudo, será proposto então uma solução para alguns dos problemas mais pertinentes quanto ao escopo deste trabalho. Veremos por último os detalhes de implementação propostos e quais foram os desafios encontrados.

4.1 ANÁLISE DO DEVICE-SERVER

4.1.1 Modelo de Comunicação Remota

Como o objetivo original do Device Server era substituir um applet web, um tipo de extensão de navegador o qual poderia executar código, a idéia central quanto a comunicação era torná-la simples, fácil e segura. Sendo assim, a implementação a qual foi escolhida dá-se diretamente entre aplicação e navegador. Após o usuário instalar a aplicação em sua máquina, esta estaria pronta para trocar informações com um servidor web através de uma comunicação direta com o navegador, como podemos ver na figura 5.

Neste modelo adotado, durante a navegação pela página web, a qual se dá na conexão 1, o navegador poderia requisitar a criação de uma sessão com o Device Server através da conexão 2, o qual reponderia através da mesma conexão. Tendo feita esta comunicação o navegador então enviaria todos os dados pertinentes de volta ao servidor web.

4.1.2 Padrão de Comunicação

Na implementação original foi escolhido um padrão de comunicação por interfaces REST. Como o objetivo do trabalho anterior era possibilitar uma comunicação direta com a aplicação ou site web, este modelo é bastante adequado. No entanto, para tornar esta comunicação segura é necessário um protocolo de cifragem de mensagens. O trabalho original propõe uma implementação SSL no entanto no protótipo temos apenas uma implementação do próprio autor do protocolo Diffie-Hellman.

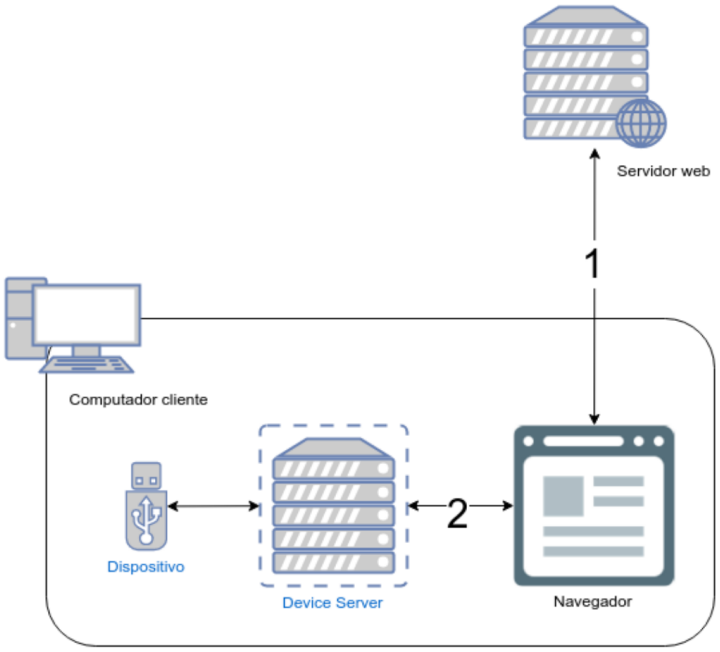

Figura 5 – Proposta original do funcionamento do Device Server, (TRAPP, 2016)

4.1.3 Módulos

Os módulos são implementações as quais seguem a API do *Device Server* e os quais representam a real funcionalidade do projeto. Este por si só realiza apenas a operação de instânciar um módulo e encaminhar as mensagens para o mesmo. No entanto, com a adição de módulos, o *Device Server* ganha diferentes funcionalidades de acordo com as implementações de cada módulo.

Os módulos relevantes para este trabalho são os que implementam operações criptograficas e comunicação com dispositivos critptográficos. No entanto a API não restringe a implementação para questões de criptografia, sendo possível implementar qualquer tipo de funcionalidade que possa ser entendida pelo *Device Server*, sendo que o papel do *Device server* é de apenas carregar os módulos ao iniciar, encaminhar as mensagens recebidas aos módulos correspondentes e devolver as respostas processadas pelos módulos.

4.2 PROPOSTA

Muitos dos modelos adotados na implementação original do Device Server já não atendem mais a demanda de uma aplicação moderna, as quais, por motivos previamente discutidos, requerem uma maior flexibilidade e que não se limitem apenas ao ambiente local.

Sendo assim, podemos perceber que apesar do modelo original de comunicação remota ser simples e fácil de entender, ela apresenta alguns problemas. A esco-

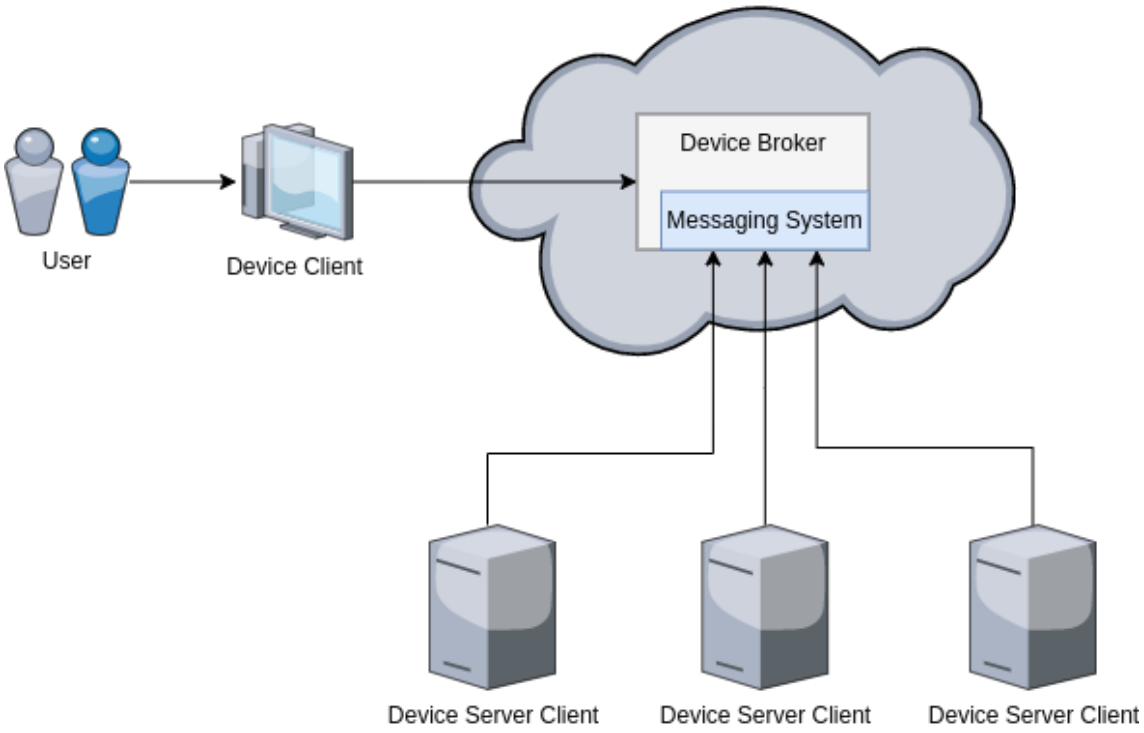

Figura 6 – Proposta do novo modelo de comunicação remota

lha de comunicar-se diretamente com o navegador, aliado ao fato de que é necessário também se comunicar com o dispositivo local, confina a aplicação ao ambiente local. Assim, caso o usuário queira usar um outro dispositivo que esteja conectado em outra máquina, este terá que iniciar uma nova conexão nesta outra máquina, sendo impossível continuar a sessão atual.

Tendo isso, este trabalho propõe a criação de uma nova solução, o *Device Broker* o qual irá fazer um papel de *middleware* entre o navegador e diversas instâncias distribuídas do Device Server, além do *Device Server Client*, o qual aplica melhorias e substitui o Device Server, fazendo a comunicação com o *Device Broker*. Por fim é proposto também a criação de um cliente o qual tem como objetivo fazer a ponte de comunicação entre usuário final e *Device Broker*. O diagrama da proposta encontra-se na figura 6

O principal objetivo deste novo modelo é possiblitar que estas instâncias possam habitar outros ambientes remotos mas ainda assim comportar-se como se estivessem presentes localmente. Deste modo, será possível elminar uma das principais limitações do modelo original, a necessidade de se ter o Device Server localmente. Este modelo novo proporcionará também a possiblidade de agrupar várias instâncias diferentes, sendo possíveis diferentes clientes estarem conectados ao mesmo Broker, cada um com o seu grupo de instâncias do Device Server, sejam elas locais ou remotas. Propõe-se também que o Device Broker seja flexível o suficiente para adicionar ou remover instâncias de maneira dinâmica, sem precisar com que o sistema seja reiniciado ou configurado de alguma forma diferente.

Para alcançar este objetivo no entando, precisamos de um novo padrão de comunicação. A interface REST adotada anteriormente é adequada para quando temos apenas um Device Server local, porém esta abordagem se torna inviável quando necessário gerenciar diversas instâncias deste. O REST por definição não guarda dados quanto ao estado do cliente com o servidor, dificultando a implementação de sessões diferentes para clientes diferentes. No mais, pela natureza síncrona da conexão original, caso ocorra algum erro ou perda na conexão, todos os dados quanto aquela conexão serão perdidos.

Desta maneira, neste trabalho propõe-se um modelo novo baseado em mensageria. Um sistema de mensageria por ter natureza assíncrona permitirá com que mesmo com o exemplo anterior, de perda na conexão, possamos reenviar os mesmos dados em um outro momento. O sistema de mensageria também é bastante flexível, sendo que as mensagens não tem um formato definido, facilitando a implementação de diferentes sessões com diferentes clientes. Além disso, o protocolo de mensageria proporciona uma robustez ao projeto pois as implementações mais populares, como RabbitMQ e Apache Kafka possuem diversos mecanismos para garantir uma boa confiabilidade do serviço como redundância utilizando diferentes nodos, balanceamento de carga, entre outros. Por ser um protocolo simples, com uma implementação específica na camada TCP, é possível alcançar altos níveis de vazão de mensagens, tendo a possiblidade de alcançar uma latência menor que requisições HTTP através da interface REST.

4.3 ESCOLHA DO SISTEMA DE MENSAGERIA

Como mencionado anteriormente, o modelo de comunicação escolhido para esta implementação foi o a comunicação por mensagens, ou mensageria. Este paradigma de comunicação apresenta algumas importantes vantagens, principalmente no contexto de ambientes distribuídos. A principal vantagem destes sistemas é sua alta confiabilidade não somente quanto ao envio de mensagens mas quanto ao tempo de atividade do sistema, isto é, o tempo o qual a infraestrutura necessária encontra-se disponível. Devido ao fato das mensagens serem enviadas a um nodo agregador e não diretamente ao possível destinatário (o que a primeira vista pode parecer uma desvantagem em questão de desempenho) estes nodos conseguem implementar políticas de replicação e redundâncias a falhas que tornam o sistema como um todo mais robusto e consequentemente têm-se um maior desempenho geral, principalmente quanto a vazão de mensagens.

Desta maneira os sistemas de mensageria são por natureza distribuídos e consequentemente facilmente escaláveis. Um outro aspecto importante é que estes oferecem diversas garantias quanto ao envio e chegada das mensagens, podendo ser configurados de acordo com as necessidades de cada infraestutura. No nível mais

básico, todas as implementações de sistemas de mensageria oferecem a garantia que, dado o envio de uma mensagem, esta chegará ao seu nodo de destino salvo um problema grave como por exemplo, todos os nodos estarem indisponíveis ao mesmo tempo.

Com isso, uma das principais escolhas quanto ao projeto é a de qual sistema de mensageria usar. Há diversas opções para se escolher, mesmo grandes empresas de tecnologia como Amazon, Google e IBM oferecem seu próprio sistema de mensageria, porém todos de código fechado e limitados em suas funcionalidades caso opte-se pelo modelo gratuito.

Assim, realisticamente existem duas alternativas de código aberto e maduras, as quais oferecem todas as suas funcionalidadas prontas para uso, Apache Kafka e RabbitMQ. Ambos são ótimas implementações porém diferem em alguns aspectos importantes. O principal ponto onde estes dois sistemas diferem é quanto ao "armazenamento" das mensagens, no Apache Kafka as mensagens chegam e são escritas em sequências em uma estrutura que se assemelha a um log, ou seja, depois de recebida, a mensagem fica para sempre gravada neste log, chamado de tópico, armazenado em disco. Os logs, por padrão permanecem por 30 dias, sendo este período, chamado de período de retenção, facilmente configurável. Apesar de parecer uma desvantajoso armazenar os logs em disco, principalmente em um caso de uso onde possivelmente milhares de mensagens podem chegar a cada segundo, o Kafka oferece também diversas formas de compressão para o armazenamento dos logs.

Comparativo RabbitMQ vs. Kafka		
Característica	**RabbitMQ**	**Kafka**
Entrega de Mensagens	Garante entrega mas não garante atomicidade da operação	Garante entrega de uma mensagem exatamente uma vez, sem duplicação.
Ordernação de Mensagens	Não garante ordem	Mensagens são sempre ordenadas
Prioridade de Mensagens	Pode definir níveis de prioridade, sendo mensagens de maior prioridade consumidas primeiro	Não há suporte para priorização de mensagens
Tempo de Vida de Mensagens	Mensagens são descartadas depois de consumidas	Mensagens permanecem no log, sendo o tempo de vida do log definido por configuração.

Tabela 1 – Comparativo entre Kafka e RabbitMQ.

Por outro lado, no RabbitMQ temos um sistema mais simples de filas. As mensagens chegam e são armazenadas em uma fila, onde cada consumidor, após

consumir esta mensagem tira a mesma da fila, efetivamente apagando esta mensagem do nodo de maneira que não é mais possível recupera-la. Outro ponto onde as duas implementações diferem é na topologia oferecida, enquanto no Kafka é oferecido apenas *publisher/subscriber*, o RabbitMQ oferece tipos como *fan-out* onde uma lista de filas recebe uma mesma mensagem, *Header exchange* onde a fila destino é baseada no cabeçalho da mensagem, entre outros.

Apesar destas diferenças ambos os sistemas são extremamanete capazes e oferecem uma alta vazão de dados, segundo a própria documentação do RabbitMQ este é capaz de processar de 30.000 a 40.000 mensagens por segundo, enquanto o Kafka, também de acordo com a própria documentação, é capaz de processar milhões de mensagens por segundo.

Dado estas diferenças fica claro que qualquer um dos dois serviços seria mais que capaz de atender as necessidade que possam se apresentar, visto que este projeto inicialmente não prevê um grande volume de mensagens sendo trocadas. Mesmo assim, foi escolhido o Kafka como serviço de mensageria. O fato das mensagens não desaparecerem oferece maior flexibilidade quanto a implementação tendo que desta maneira podemos ter vários consumidores consumindo mensagens de um mesmo tópico sem preocupação quanto a apenas o destinatário receber a mensagem. Ter logs permanentes também é útil para diagnosticar possíveis problemas durante a implementação ou também para possíveis fins de auditoria em casos de uso real do projeto. Uma possível desvantagem é quanto ao acesso apenas ao modelo de *publisher/-subscriber* porém, a simplicidade do modelo mostrou-se na verdade uma vantagem, devido ao modelo simplista é necessário dedicar mais tempo ao planejamento da topologia do projeto, porém uma vez definido, as preocupações quanto a escalabilidade resumem-se em apenas adicionar mais nodos.

Um conceito que precisa ser esclarecido no entanto é o de mensagens. Como foi escolhido Apache Kafka trataremos de mensagens no âmbito deste software. Assim, uma mensagem é um simples par de chave-valor, onde a chave é opcional e pode assumir diferentes estruturas de acordo com a necessidade, e o valor é o conteudo da mensagem em si, a informação que se quer enviar. As mensagens são também limitadas em seu tamanho, porém neste trabalho isto não será relevante.

O motivo pelo o qual é possível processar tantas mensagens se da pelo fato do Kafka implementar um protocolo próprio para o envio destas mensagens. Diferente do RabbitMQ e outros serviços de mensageria que implementam o protocolo AMPQ, o Kafka se destaca por usar uma implementação própria que busca otimizar o máximo o possível o envio e recebimento de um grande volume de mensagens, tornado-o assim extremamente capaz e com alto desempenho.

O conceito de tópicos do Kafka também mereçe uma explicação melhor, como dito anteriormente os tópicos armazenam as mensagens muito semelhante a um log, onde é feito apenas um *append* quando uma nova mensagem chega. Assim, cada

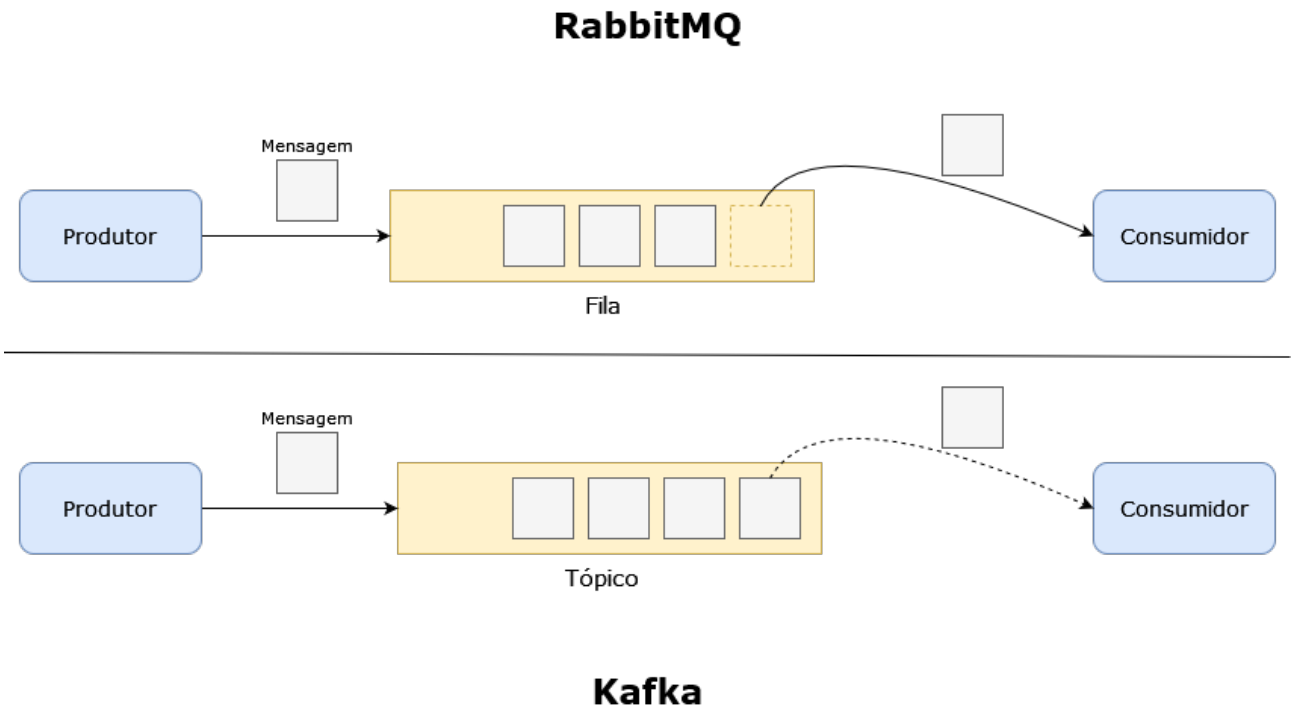

Figura 7 – Diferentes abordagens de consumo de mensagem. No RabbitMQ a mensagem é perdida após o consumo, já no Kafka, o consumidor recebe um cópia da mensagem, sendo que a original continua no tópico

consumidor precisa manter informação sobre o seu offset em relação o começo do log para saber se uma mensagem é nova ou não. Felizmente todo esse mecanismo é gerenciado pelo próprio Kafka e suas bibliotecas disponíveis, como veremos mais a frente. A única preocupação que teremos em relação ao offset neste projeto é quanto ao ponto de partida quando um consumidor se conecta ao sistema, temos basicamente três opções: assumir o offset inicial sempre, assumir o offset como sendo o da última mensagem lida, ou assumir o offset como sendo sempre o último disponivel.

Outra característica dos tópicos são as partições. Cada tópicos pode ser dividido em uma ou mais partição, estas partições por sua vez podem, e na maioria das vezes, irão possuir um conjunto diferente de mensagens. Cada mensagem enviada pode ser destinada explicitamente a uma partição, sendo assim, uma mensagem destinada a partição 0 do tópico “foo” não estará disponível a um consumidor ligado a partição 1 do mesmo tópico. Caso nenhuma partição seja explicitada na mensagem, é feito então um *hash* da chave da mensagem e uma divisão pelo número de partições, onde o resto então é a partição escolhida. Por último, se a mensagem for enviada sem nenhuma chave, é feito então um *round-robin* sobre as partições.

Apesar do Kafka possuir diversas opções de customização, isso também se mostra uma desvantagem pois aumenta bastante a complexidade da configuração da infraestrutura básica. O Apache Kafka também conta com um software a parte chamado Zookeper, necessário para o funcionamento. O Zookeper faz o gerenciamento

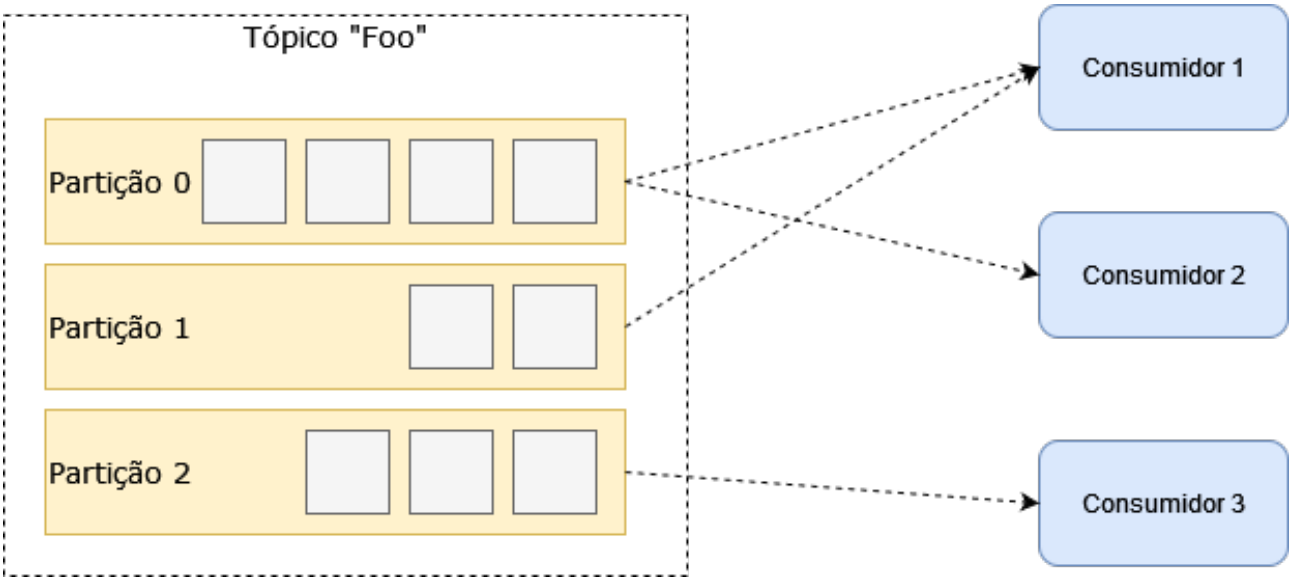

Figura 8 – Modelo de partições dos tópicos no Kafka. Cada consumidor pode consumir de um ou mais tópicos.

dos nodos, tópicos, balançeamento de carga e outras tarefas importantes, mas isso significa também mais um software para configurar aumentando assim ainda mais a complexidade do projeto.

Uma vantagem não aparente encontrada no entanto, se deve pelo fato do Apache Kafka ser amplamente utilizado em diversos setores da indústria tecnológica, sendo assim é um software extremamente maduro que conta com diversas aplicações complementares que servem os mais diferentes propósitos, como facilitar conexão direta com banco de dados, processamento de mensagens antes destas chegarem ao seu tópico destino entre outros. Sendo assim seria possível expandir este trabalho para os mais variados casos de uso, sem preocupar-se com a intereção dos diferentes componenetes.

Desta maneira fica claro a vantagem, em termos gerais, do Apache Kafka sobre o RabbitMQ. Mesmo utilizando apenas uma fração do seu pontencial, o Kafka oferece toda a flexibilidade de um sistema de mensageria, junto com um ecossistema maduro repleto de recursos, documentação, e que pode resistir ao teste do tempo.

4.4 ARQUITETURA DO SISTEMA

Após escolhida a implementação do sistema de mensageria, é preciso então definir a arquitetura do projeto no nível de troca de mensagens, ou seja, a topologia da rede de mensagens. Como discutido anteriormente o Kafka oferece somente o padrão *publisher/subscriber* o que é o suficiente para atender aos requisitos do trabalho. Como mostrado na figura 6, um cliente estará ligado a diversos *Device Servers*, isso significa que precisaremos que o *device* alvo da mensagem receba esta mensagem

corretamente. Uma primeira idéia seria organizar os *Device Servers* em um grupo de consumidores. No Apache Kafka, cada consumidor deve pertencer a um grupo, se nenhum for definido nas configurações, então o próprio Kafka designará um grupo para o consumidor. O conceito de grupos é importante pois afeta a maneira como as mensagens são recebidas, e consequentemente, consumidas. Quando uma mensagem chega a um tópico, apenas um membro de cada grupo irá consumir efetivamente esta mensagem. A política de escolha de qual dos consumidores em um grupo receberá a mensagem se dá de maneira simples, seguindo um algoritmo de round-robin , de maneira que a cada mensagem, um consumidor diferente do grupo será escolhido até todos os consumidores terem recebido ao menos uma mensagem, repetindo assim o ciclo.

Com isso em mente podemos perceber que a escolha de agrupar todos os nossos consumidores (Device Servers) em um grupo não é ideal, visto o consumidor a receber a mensagem pode não ser o destinatário pretendido. Assim, optou-se por definir diferentes grupos para diferentes consumidores. Não há evidências na documentação do Apache Kafka que relacionem a criação de diversos grupos diferentes com algum tipo de degradação de desempenho, desta maneira, não teremos nesta área com a criação de diversos grupos. Assim o esquema definido por ser visto na figura 9.

4.4.1 Serviço de Banco de Dados

Uma consequência de criar um projeto distribuído é que precisamos armazenar diferentes informações sobre um cliente, relacionadas a um usuário ou um device. Desta maneira, inevitavelmente surge a necessidade da criação de um banco de dados que armazene informações de login de um cliente, e quais devices estão vinculados a ele. Resta então a decisão de onde inserir este banco de dados na estrutura atual.

No caso de cada *Device Server*, eles apenas se comportam como produtores e consumidores de mensagens, recebem uma mensagem e enviam uma resposta. Assim, não há a necessidade de conecta-los diretamente a um banco de dados. Já o cliente, aqui chamado de *Device Server Client* precisa obter informações de login, visto que este deve comportar mais de um usuário, e adicionar e remover dispositivos conectados a ele.

Apesar de ser uma solução simples, usar um banco de dados localmente não é factível, visto que a intenção do *Device Server Client* é ser instalado na máquina do usuário final, de modo que além de adicionar muita complexidade para o usuário lidar, poderíamos ter problemas como ambientes restritos onde simplesmente não seria possível adicionar um banco de dados.

Com isso em mente resta adicionar um serviço a mais na topologia do tra-

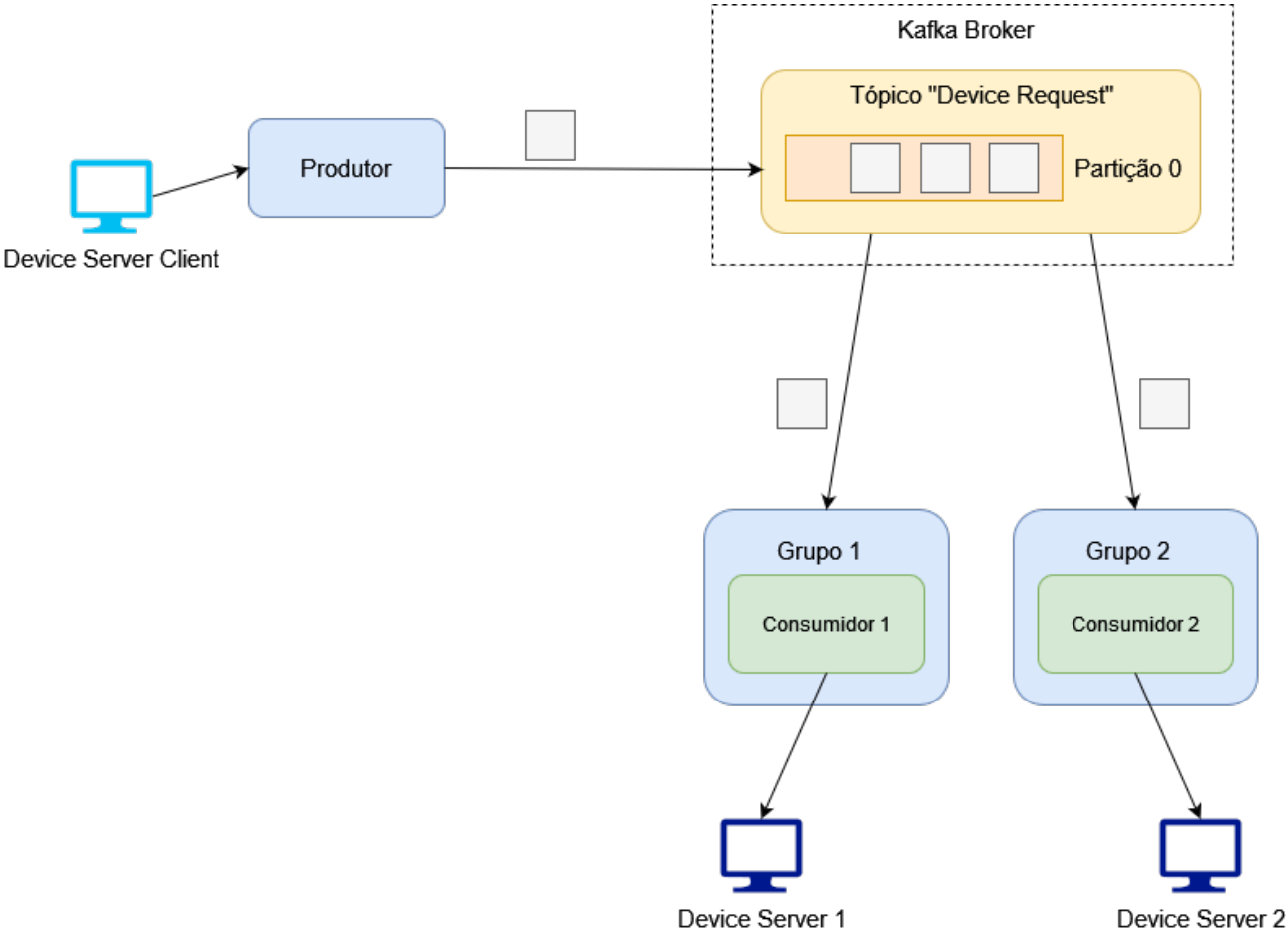

Figura 9 – Esquema adotado para consumidores.

balho, sendo este um microsserviço que apenas realiza operações CRUD sobre um banco de dados. Uma outra opção seria utilizar um serviço complementar ao Kafka chamado ksqlDB. Com o ksqlDB é possível usar tópicos como *schemas* e armazenar dados permanentemente nos tópicos. O ksqlDB ainda oferece uma sintaxe muito semelhante ao SQL para realizar as operações sobre os dados. Apesar de parecer uma boa solução, infelizmente o nível de complexidade deste software é muito alto e como o serviço de banco de dados não faz parte do escopo do trabalho, optou-se por criar um microsserviço simples, utilizando MariaDB que consome e produz em tópicos do Kafka, semelhante aos devices.

Como dito anteriormente, a criação deste serviço foge do escopo do projeto, sendo assim este foi construido de maneira simples, apenas o básico que permita o funcionamento devido do trabalho, sem as preocupações que normalmente ocorrem quando se tem um banco de dados distribuído como replicação, concorrência, fragmentação, etc.

Assim, foi escolhido MariaDB como o sistema de gerenciamento de banco de dados devido a maior familiaridade do autor com este programa e com a sintaxe SQL.

4.5 SEGURANÇA

Como se trata de um trabalho que lida com dispositvos seguros e que contém informações sensíveis, há uma preocupação em manter a aplicação segura, assume-se que em ambientes distrbuídos os canais de comunicação são sempre inseguros e ativamente interceptados por atores maliciosos, portanto surge uma necessidade de manter as comunicações seguras.

4.5.1 Módulos

O funcionamento do Device Server depende da adição de módulos os quais realizam tarefas como a comunicação com token criptográfico. Cada módulo implementa uma API definida pelo próprio Device Server para comunicação com o mesmo. Apesar da adição de um módulo ser simples, basta gerar uma versão em formato JAR e adicionar no diretório *modules*, visto que um módulo pode lidar com informações sensíveis é necessário que haja uma verificação dos módulos.

Na implementação original, o Device Server faz uma checagem criptográfica de cada JAR antes de carregá-los. Para um módulo ser válido este necessita estar assinado com uma chave pública vinculada a um certificado válido. Ou seja, a AC emissora do certificado do módulo precisa ser conhecida e aceita como uma âncora de confiança pelo Device Server, para que esse então possa verificar a integridade do módulo através de uma validação de sua assinatura.

4.5.2 Comunicação

O modelo anterior do Device Server também garantia autenticação através da comunicação por TLS/SSL, o modo como foi implementado não se adapta bem ao novo modelo distribuído. Na implementação original, durante a inicialização da aplicação, um servidor é iniciado localmente, e uma URL é gerada e requisita-se ao usuário o acesso a esta página. Ao fazer isso, o usuário pode então aceitar a conexão do seu navegador com o servidor, aceitando o certificado do Device Server como um certificado confiável, estabelecendo assim a possibilidade de comunicação segura através do navegador.

Diante disso, como este trabalho visa desvincular a interação direta com o navegador por parte do Device Server, foi retirado do código tudo relacionado a sessões e comunicação TLS através de um *web-server*. Porém ainda é preciso garantir a autenticação segura em um servidor, no caso deste trabalho, um broker Kafka. Para isso, felizmente esta autenticação pode ser feita de maneira mais simples.

Primeiro é necessário que se tenha um certificado de uma AC e seu par de chaves correspondente. Este *keystore* será usado como âncora de confiança, ou seja,

todo cliente que tentar autenticar-se com um broker deverá possuir um certificado que pertença a uma cadeia contendo esta AC. Cada broker então terá um certificado próprio que o identifica, emitido pela esta AC de confiança. O mesmo é verdade para cada consumidor e produtor que deseja se conectar ao Kafka, para uma conexão com o broker é exigido um certificado que faça parte da mesma cadeia de confiança.

Desta maneira temos então um canal de comunicação seguro e autenticado, onde todos os agentes conectados são realmente quem eles afirmam ser, garantindo a mesma robustez do sistema anterior e com todas as garantias oferecidas pelo protocolo TLS.

4.5.3 Sigilo

Tendo estabelecida a conexão segura por TLS temos que todas as mensagens trocadas entre o Kafka passam a ser cifradas, garantindo o sigilo da aplicação. Assim, além de servir como fator de autenticação no sistema, o TLS também garante que mesmo em uma rede comprometida, onde agentes maliciosos possam interceptar os pacotes de comunicação, estas mensagens não serão decifradas.

Desta maneira podemos usar estas conexões para comunicação com o Kafka sem preocupar-mos com agentes maliciosos na rede. Mesmo com a rede comprometida, como o conteúdo das mensgens está criptografado, seria praticamente impossível decifrar estas mensgens sem as chaves corretas. Desta maneira, mesmo que ocorra interceptção de pacotes, ataques do tipo *man in the middle*, temos a garantia que a informação contida nesses pacotes está segura. Isso facilita em alguns aspectos da implementação, como veremos mais a frente, onde precisamos passar informações como senhas e PIN's de tokens criptográficos de um serviço a outro.

4.6 ADAPTAÇÃO DO DEVICE SERVER

O Device Server foi concebido para ser um programa de adaptável a diferentes ambientes e funções, suas funcionalidades dependem do carregamento de módulos desenvolvidos separadamente os quais, seguindo a API descrita pelo Device Server, são possíveis de realizar diferentes tarefas. Além disso, a linguagem utilizada, Java, também demonstra este compromisso com a flexibilidade e portabilidade do projeto original.

Com isso em mente, o objetivo deste trabalho é atualizar o Device Server para um âmbito moderno de microsserviços, mas ainda mantendo as funcionalidades e a proposta original de ser um projeto flexivel e portátil.

4.6.1 Spring Boot

Para poder alcançar este objetivo foi escolhido adaptar o projeto para um novo framework de desenvolvimento, o *Spring Boot*. O projeto *Spring* é o framework mais conhecido e mais utilizado atualmente no ambiente de desenvolvimento Java, isso se deve principalmente a suas inúmeras abstrações que facilitam o desenvolvimento, suporte de primeira classe a mais diversas aplicações, como Apache Kakfa e bancos de dados SQL, e seu constante aprimoramento.

Assim, trazendo o *Spring* para o projeto proporciona diversas facilidades, principalmente quanto a adaptação para o serviço de mensageria. Apesar do Apache Kafka possuir uma documentação extensa para o seu manuseio, a biblioteca Java disponível pelo instituto Apache não possui a mesma riqueza de informações em sua documentação. Felizmente o *Spring* possui uma biblioteca própria para comunicação por mensageria, específica para o Kafka. Esta biblioteca simplifica diversos aspectos para o desenvolvedor disponibilizando diferentes interafaces e anotações as quais é possível aplicar configurações muito semelhantes a documentação original do Kafka.

Além de conexões com sistema de mensageria o *Spring* também oferece uma API própria para comunicação por serviços REST, porém com uma implementação mais madura e com mais funcionalidades quando comparado a interface REST proporcionada pela biblioteca padrão do Java. O Spring também auxilia no desenvolvimento evitando erros comuns como expor um *stack trace* para o cliente acidentalmente, algo que pode ser considerado um risco de segurança.

4.6.2 Novo Modelo de Autenticação

Como o projeto original previa comunicação por interface REST, e essa comunicação deveria se dar de maneira segura, utilizando TLS. O protótipo do Device Server no entando foi simplificado, sendo a parte do TLS substituída por uma implementação própria do protocolo de *Diffie-Hellman*.

A troca de chaves de *Diffie-Hellman* proporciona sigilo na troca de mensagens em um meio comprometido, porém este protocolo também é bastante antigo e já passou por diferentes revisões para evitar uma maior gama de ataques que se tornaram populares na útlimas décadas. Além disso, a efetividade deste protocolo depende na escolha “criptograficamente boa” de números primos para realizar a troca e a cifragem de mensagens.

Um ponto fraco da impelementação original é a necessidade do uso de sessões para guardar o estado da autenticação do usuário devido substituição do protocolo TLS pelo *Diffie-Hellman*. A implementação do TLS oferece uma solução mais madura de autenticação e sigilo, além eliminar a necessidade de manter e gerenciar sessões.

Outro ponto fraco da implementação anterior era a necessidade de embutir um servidor web junta a aplicação para expor os recursos REST. Além de aumentar a potencial superfície de ataque, a necessidade de haver um serivdor para cada instância de um Device Server dificulta a adaptação para os fins deste trabalho, sendo que cada cliente teria que se autenticar em diversas sessões diferentes e manter um estado de cada conexão.

Sendo assim, neste trabalho optou-se por remover completamente a impelmentação de interfaces REST, visto que será utilizado o sistema de mensageria. Assim também não temos mais a necessidade de expor essas interfaces portanto também foi removido o servidor web embutido.

Sem um servidor web e endpoints para acesso HTTP, pode-se então remover completamente também a implementação de autenticação usando *Diffie-Hellman* e toda a parte de sessões. O projeto irá se comunicar apenas através do serviço de mensagens, e apenas receberá mensagens do cliente, sendo assim não há a necessidade de expor nenhum outro canal de comunicação com outra aplicação ou com o usuário. Além disso a autenticação passa a ser feita diretamente através do Kafka, sendo a conexão de cada instância do Device Server autenticada e protegida por TLS.

Como o cliente será uma implementação nova por si só, o design deste serviço já leva em consideração o fatoque cada instância do cliente deve se autenticar na rede do Kafka através de TLS.

4.6.3 Mudanças na API

Em preparação para a nova implementação do serviço de mensageria, é preciso fazer algumas pequenas alterações na API do Device Server.

Como teremos várias instâncias conectadas na rede, precisamos que as mensagens enviadas possuam algum campo para distinguir uma instância da outra. Uma vez que uma mensagem chega a um tópico, todos os Device Servers que consomem deste tópico irão receber a mesma mensagem, o que significa que todos eles produzirão uma resposta, enquanto o cliente espera que haja apenas uma resposta para cada mensagem enviada. No mais, seria um erro fazer com que todas as instâncias conectadas executem o mesmo comando, sendo que estas podem ter módulos diferentes carregados.

A maneira como será feita essa identificação ainda será discutida mais a frente, mas independente de como será a implementação, segue que precisa-se enviar metadados sobre o Device Server junto com a mensagem, portanto foi criada a classe *MessageMetadata.java* a qual armazena, de maneira simples, dados como o remetente da mensagem, o destinatário, e qual módulo deve executar a mensagem (no caso de uma mensagem para um Device Server). A definição desta classe é a seguinte:

```
public class MessageMetadata {
    private String sender;
    private String receiver;
    private String targetModule;
}
```

Segundo a API do Device Server, temos que as mensagens trocadas sempre são do seguinte formato:

```
public class DeviceMessage {
    private String method;
    private String message;
    private String iv;
}
```

No entanto agora precisamos adaptar esta mensagem para possuir os metadados. Simplesmente adicionar estes dados novos a API antiga parece simples, porém qualquer alteração na forma da mensagem *DeviceMessage* irá fazer com que módulos criados antes destas mudanças deixem de funcionar corretamente. Para evitar isso foi criado então uma nova classe, essa classe representa as mensagens que realmente são trocadas entre os serviços dentro do Kafka, ou seja, estas são as mensagens que são escritas nos tópicos e consumidas pelos consumidores. Assim garantimos a retrocompatibilidade com os módulos antigos mas conseguimos adaptar para o novo modelo, onde facilmente podemos extrair a *DeviceMessage*.

O modelo novo de mensagem é definido de maneira:

```
public class TopicMessage {
    private MessageMetadata metaData;
    private DeviceMessage message;
}
```

4.6.4 Adaptação para Serviço de Mensageria

Com as devidas preparações feitas, pode-se então inciar a adaptação do Device Server para o serviço de mensageria. Graças a escolha de utilizar o framework *Spring* esta adaptação é relativamente simples. Como cada instância deverá não apenas consumir mensagens de tópicos mas também produzir mensagens de resposta, sendo assim é necessário criar configurações diferentes para consumidores e produtores.

Antes de criar as classes que conectam ao serviço do Kafka, é necessário definir alguma estrutura para configurações de conexão como endreço, porta, e nome

dos tópicos de interesse. Para isso, o *Spring* possui um arquivo de configuração por padrão, que fica na raiz do projeto, onde podemos aplicar várias configurações diferentes ou criar nossas próprias variáveis de configuração. Neste caso, optou-se por definir todas variáveis de conexão baseadas em varíaveis de ambiente. Desta maneira descartamos a necessidade de arquivos de configuração, visto que estes podem ser bastante inconvenientes de lidar, fáceis de configurar errado etc. Utilizando variáveis de ambiente têm-se uma maior facilidade não apenas de alterar uma destas variáveis como também facilmente portar para uma imagem Docker por exemplo.

Assim, o arquivo *application.properties* fica definido da seguinte maneira:

```
spring.kafka.bootstrap-servers=${KAFKA_HOST:localhost}:${KAFKA_PORT:9093}
consumer.topic=${CONSUMER_TOPIC:client-device}
producer.topic=${PRODUCER_TOPIC:device-client}
```

As variáveis de ambiente neste contexto são KAFKA_HOST, KAFKA_PORT para o endereço e a porta da conexão com o servidor Kafka respectivamente, e CONSUMER_TOPIC, PRODUCER_TOPIC para os nome dos tópicos do consumidor e produtor sendo que cada uma destas variáveis assumem um valor padrão caso nenhuma variável de ambiente esquivalente esteja definida.

Podemos então configurar nosso consumidor, o componenente que irá ativamente requisitar o broker Kafka por novas mensagens no tópico de maneira assíncrona. Como o Kafka nos permite enviar qualquer tipo de estrutura na mensagem, precisamos definir a configuração do desserializador para o objeto correto. Como mostrado anteriormente este objeto será uma instância da classe *TopicMessage* porém para simplificar o envio, veremos que o produtor irá converter este objeto em formato JSON, portanto precisamos de um desserializador de JSON para a mensagem, e um desserializador de String para a chave da mensagem correspondente. Assim temos que a configuração do consumidor será:

```
public Map<String, Object> consumerConfig() {
    HashMap<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        JsonDeserializer.class);
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
    return props;
}
```

Uma configuração importante a se pontuar aqui é relacionada ao offset do tópico. É preferível que toda vez que uma instância do Device Server fique online

esta então tenha que seu offset de mensagens seja o maior possível. A razão disto é pelo fatoque não queremos que caso esta instância fique offline por problema de rede enquanto mensagens estão sendo enviadas, não queremos que estas mensagens sejam processadas quando o Device Server voltar a ficar online, pois elas podem estar expiradas ou não serem mais necessárias.

Com esta configuração podemos então criar o consumidor em si. Diferente do que se espera normalmente, através do *Spring* é possível abstrair toda a configuração manual e deixar o framework instanciar e gerenciar os objetos necessários para a criação do consumidor. Para definir um consumidor basta usar a anotação *@KafkaListener* em um método o qual irá lidar com a mensagem recebida.

Seguindo as boas práticas de programação, foi criada então uma classe que implementa um método o qual lida apenas com o recebimento das mensagens. Como sabemos o formato da mensagem será uma *TopicMessage* podemos então definir este objeto como argumento do método. Este método irá receber toda e qualquer mensagem enviada ao tópico cujo nome foi definido na variável de ambiente CONSUMER_TOPIC, então aqui precisa-se filtrar estas mensagens de acordo com os metadados da mesma. Sendo assim, é feito uma checagem básica, caso o ID do destinatário for igual ao ID do Device Server atual, então esta mensagem é processada de acordo. A definição do consumidor fica da forma:

```
    @KafkaListener(
        topics = "${consumer.topic}",
        groupId = "device-server",
        containerFactory = "deviceMessageConsumerFactory")
public void senderListener(TopicMessage message) {
    MessageMetadata metadata = message.getMetaData();

    if (metadata.getReceiver().equals(deviceConfiguration.getUuid())) {

        TopicMessage response = new TopicMessage();
        MessageMetadata responseMetadata = new MessageMetadata();
        responseMetadata.setSender(deviceConfiguration.getUuid());
        responseMetadata.setReceiver(metadata.getSender());
        response.setMetaData(responseMetadata);

        DeviceMessage moduleResponse =
                messageService.resolveMessage(message.getMessage(),
                    metadata.getTargetModule());

        response.setMessage(moduleResponse);
```

```
            producer.send(producerTopic, response);
            logger.info("Responded to " + metadata.getSender());
        }
    }
```

Já no produtor as configurações são idênticas a do consumidor, apenas precisamos trocar os desserializador por serializador. Como discutido anteriormente, a conversão do objeto mensagem será para JSON, então será preciso de um serializador de JSON para a mensagem em si e um serializador de String para a chave da respectiva mensagem. Como o produtor não precisa ter conhecimento do offset no tópico, pode-se omitir esta informação do objeto de configuração:

```
public Map<String, Object> producerConfig() {
    HashMap<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        JsonSerializer.class);
    return props;
}
```

Diferente do consumidor, o produtor requer que seja definido um método o qual seja possível instancia-lo. No entando podemos usar anotações do framework para que este mesmo instâncie e gerencie o objeto, permitindo que ele possa ser injetado em classes sem a instânciação explícita. Este produtor irá apenas publicar no tópico a mensagem retornada pelo módulo alvo, o único tratamento que esta mensagem sofrerá será a inversão nos metadados de remetente e destinatário, indicando uma resposta a um pedido.

O método que permite instânciar um produtor fica definido da seguinte forma:

```
 @Bean
public KafkaTemplate<String, TopicMessage> kafkaTemplate(
        ProducerFactory<String, TopicMessage> producerFactory) {
    return new KafkaTemplate<>(producerFactory);
}
```

Desta maneira pode-se perceber que houve uma grande simplificação do projeto Device Server. Anteriormente este atuava como um programa monolítico que era responsável por autenticar o usuário, expor rotas através de um servidor web e encaminhar mensagens para os módulos destino. Com a simplificação da autenticação, remoção da necessidade de rotas, tem-se um verdadeiro microsserviço o qual apenas

recebe mensagens de um tópico, encaminha estas mensagens ao módulo correto, e retorna as respostas a outro tópico.

Estas mudanças tornam a manutenção do software mais fácil e aumentam sua portabilidade, é possível converter em uma imagem Docker[1] a qual pode ser feito o deploy em outro ambiente facilmente, ainda mantendo todas as funcionalidades e segurança da versão anterior.

4.7 IMPLEMENTAÇÃO DO CLIENTE

O Device Server Client é o serviço o qual será responsável pelo gerenciamento da comunicação com os Device Servers e também a interface para o cliente. Com as mudanças feitas no Device Server, temos agora que tal serviço atua de maneira orientada a eventos, ou seja, caso não chegue uma mensagem no tópico o serviço fica ocioso, aguardando o próximo evento. Assim as reponsabilidados quanto a funcionalidade do sistema foram passadas ao cliente, onde este deve realizar a tarefas como comunicação com o serviço de banco de dados, interação com o usuário e gerenciamento de dispositivos e usuários conectados.

As configurações básicas do cliente seguem o modelo adotado no Device Server, ou seja, temos uma opção de instalação a qual cria um arquivo de configuração que armazena o atributo ID do programa. Este atributo então é usado como identificador de origem de cada mensagem enviada, permitindo que o cliente identifique mensagens destinadas ao mesmo no tópico de resposta.

Os mecanismos de comunicação com o Kafka também são idênticos aos implementados no Device Server. Cada instância do cliente tem um produtor que envia mensagens ao tópico de requisições e um consumidor o qual consome mensagens do tópico de resposta.

Como interface para o cliente considerou-se implementar um modelo web baseado em chamadas REST, muito parecido com o conceito que existia anteriormente no Device Server. A vantagem de um sistema deste tipo seria a possibilidade de manter uma funcionalidade idêntica a proposta original, onde outras aplicações pudessem interagir com cliente e acessar os dispositivos criptográficos. Esta abordagem no entando requer que a aplicação tenha um próprio servidor web para expor as rotas REST, o que leva a mais considerações sobre autenticação e segurança de rede, uma vez que a autenticação através do TLS tratada neste trabalho refere-se apenas as conexões do Kafka.

Este trabalho no entanto optou por uma opção mais visual, utilizando recursos da linguagem para criar uma interface gráfica. Desta maneira evitamos esta camada de complexidade a mais que uma interface web nos leva a considerar, na interface

[1] Aplicação que permite entrega de softwares em um ambiente virtualizado e isolado, chamado de contâiner.

gráfica temos total controle sobre como é o fluxo do usuário, evitando assim preocupações com segurança neste aspecto.

No entanto, a aplicação foi desenvolvida levando em consideração que a interface gráfica não deve ser parte do núcleo do projeto. Todas as funcionalidades principais como mandar mensagens, receber mensagens, gerenciar opções do Kafka, gerenciar dispositivos e usuários são implementadas fazendo uso de interfaces genéricas. Sendo assim, a parte gráfica apenas implementa tais interfaces de maneira a gerar painéis gráficos para o usuário.

4.7.1 Sincronismo

Como todas as comunicações foram migradas para serviço de mensageria e considerando que este é naturalmente assíncrono, para a implementação do cliente com interface gráfica achou-se necessário criar alguns mecanismos de sincronismo. Quando uma mensagem é enviada a um tópico, não se tem uma respostas, em essência estamos apenas criando um evento o qual não se espera um retorno. Na interface com o usuário tem-se grande interesse em reportar falhas que possam ocorrer, sejam por erro do usuário ou mesmo do sistema, durante o fluxo de uso. Seria um erro avançar para a tela que gerencia os dispositivos de um usuário se os dados para autenticação informados estiverem errados por exemplo.

4.7.2 Protótipo

Sendo o foco deste trabalho dispositivos a aplicação direcionada a dispositivos criptográficos, foi protótipo desenvolvido tem como finalidade mostrar a funcionalidade de um dos módulos existentes quanto a assinatura de dados.

Como mencionado, o cliente implementa todas as questões de comunicação com o Kafka de maneira idêntica ao Device Server, sendo assim, o cliente autentica-se na rede através de um certificado próprio conhecido pelo broker do Kafka, estabelecendo uma conexão protegida pro TLS.

O cliente no entanto precisa lidar com outro tipo de autenticação, a autenticação do usuário no sistema. O usuário tem um usuário e senha e uma lista de dispositivos pertencentes a ele. Sendo assim, no cliente temos que ao iniciar a execução do programa uma tela de login simples, como mostra a figura 10.

Concluída a etapa de autenticação do cliente, temos em seguida uma tela a qual apresenta os dispositivos conectados. Existem botões para adicionar e remover dispositivos a qualquer momento. Ambas ações geram mensagens para atualizar as informações do usuário no banco de dados, assim caso ocorra algum erro ou não haja uma resposta do banco, então uma mensagem de erro é exibida para o usuário e a operação cancelada. A figura 11 mostra esta tela de gerenciamento de dispositivos.

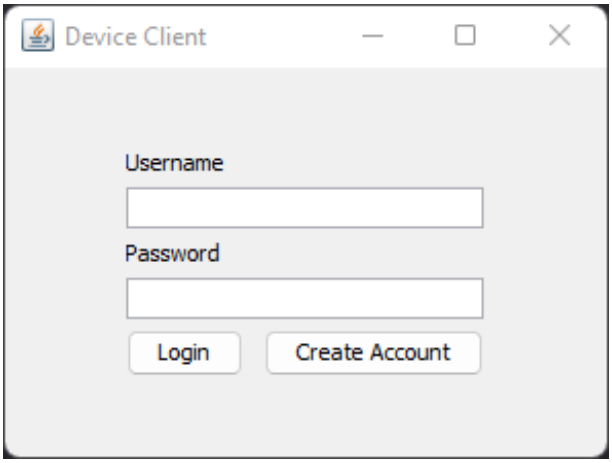

Figura 10 – Tela de login no Device Server Client

Para adicionar um dispositivo é preciso informar o ID que o mesmo apresenta ao ser inicializado e um label para melhor identifica-lo. Nesta etapa a aplicação não faz nenhuma checagem quanto a existência ou não do dispositivo informado, porém o dispositivo só estará visualmente disponível após uma mensagem de confirmação do banco de dados.

Apenas após o usuário ter selecionado o dispositivo desejado é que então o cliente faz a primeira comunicação com o dispositivo. Uma mensagem é enviada com o ID do dispositivo e caso não haja uma resposta, o usuário é informado do erro e não é permitido avançar para a próxima tela. Na arquitetura atual o sistema não consegue distinguir dispositivos offline de dispositivos inexistentes, sendo em que ambos os casos o erro gerado será um de timeout na comunicação.

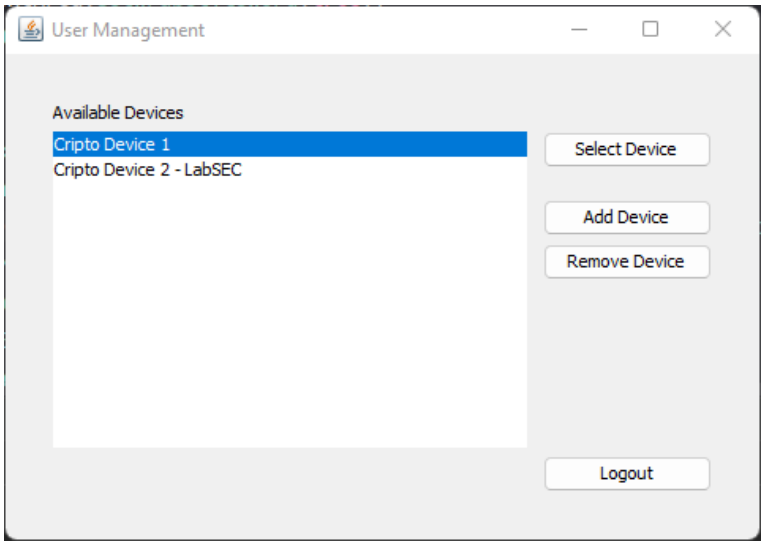

Figura 11 – Tela de gerenciamento de dispositivos

Caso o dispositivo esteja disponível no entanto, na próxima tela temos algumas seleções disponíveis. A mensagem resposta de um dispositivo disponível é uma

listagem de todos os seus módulos. Cada mensagem, como vimos anteriormente, são destinadas a um dispositivo e a um módulo deste dispositivo, portanto é importante que o usuário possa decidir qual módulo deseja usar.

Como este trabalho é destinado a operações com dispositivos criptográficos, após selecionado o o módulo uma mensagem é enviada requisitando todos os certificados disponíveis. O usuário é apresentado com uma lista de *alias* de certificados, permitindo-o facilmente identifiar cada um e selecionar o desejado, como mostrado na figura 12.

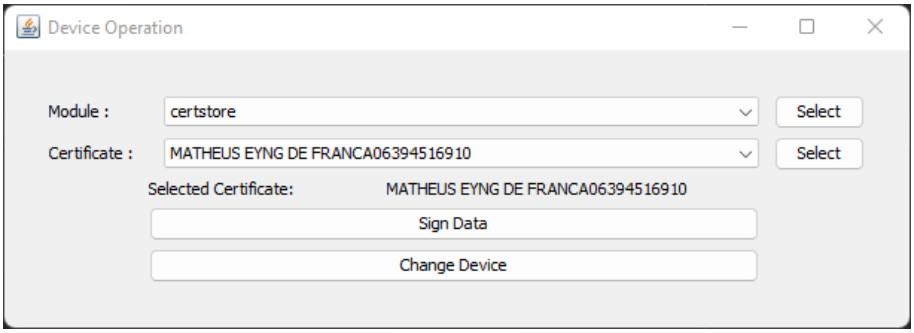

Figura 12 – Tela de escolha de módulo e assinatura de arquivo.

Após selecionado o módulo e o certificado a ser usado, temos quase todas as informações necessárias para realizar uma assinatura. Ao clicar o botão de assinatura, o usuário então é solicitado a escolher um arquivo contendo as informações as quais deseja assinar. Feito isso basta apenas obter a senha ou PIN do dispositivo criptográfico. Assim a mensagem de requisição de assinatura é formada e enviada para o dispositvo. Aqui confiamos no sigilo oferecido pelo TLS para enviar o PIN na mensagem, que precisamos lembrar, chega ao tópico em formato JSON. Com isso é apresentado ao usuário então o resultado da assinatura como podemos ver na figura 13.

Em geral a aplicação se comporta de maneira indistinguível de uma aplicação cuja implementação seja totalmente local. Apesar da escolha de utilizar-se uma interface gráfica, graças a combinação da natureza assíncrona do serviço de mensageria e da implementação permitindo comunicação síncrona, é possível substituir esta interface por qualquer outra, oferecendo grande flexibilidade ao sistema como um todo.

4.8 SERVIDOR KAFKA

Grande parte da configuração do servidor Kafka já foi discutida neste trabalho, porém alguns apectos ainda requerem um esclarecimento. Para se ter um servidor Kafka executando, é preciso de 2 serviços diferentes: Zookeper e o próprio Apache Kafka. Enquanto o Kafka foi bem explorado, o Zookeper foi apenas mencionado na seção 4.3, porém este tem um papel essencial no funcionamento do trabalho.

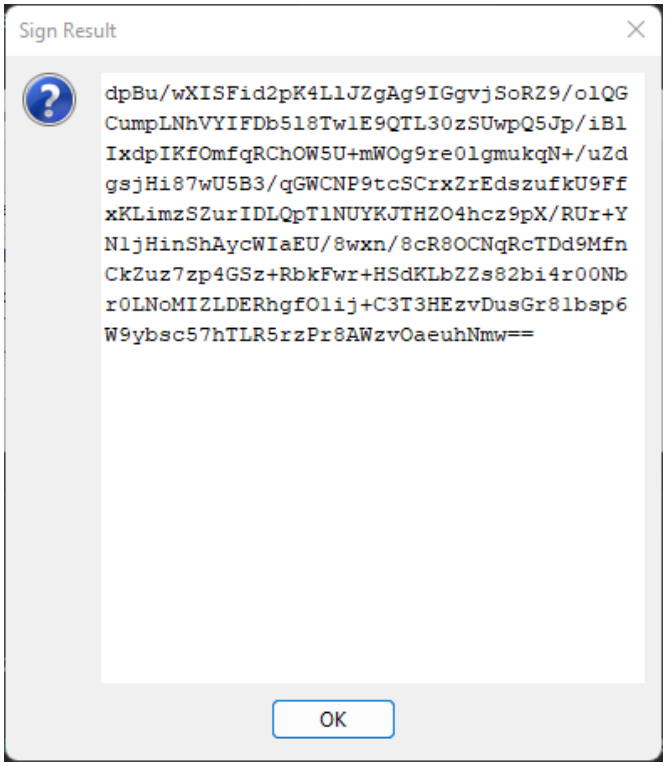

Figura 13 – Tela de resultado da assinatura.

O Zookeper é um serviço complementar, porém essencial, ao funcionamento do serviço de mensageria. Este é responsável pelas principais tarefas essenciais ao funcionamento de um sistema distribuído, como gerenciamento de nodos, eleição de líder, *timestamps*, gerenciamento de cache, entre outros. Apesar do Kafka depender deste serivço para funcionar, o Zookeper no entanto é um projeto a parte, sendo ele parte integral de outros projetos da fundação Apache os quais lidam com manuseio de dados de forma distribuída. Assim, um cluster Kafka é a combinação deste dois serviços independentes.

Neste trabalho discutimos sobre autenticação e sigilo entre conexões de consumidores e produtores a nodos do Kafka, porém nada foi dito quanto a comunicação entre Zookeper e Kafka. O motivo disso é que optou-se por se utilizar estes dois serviços da maneira mais comum encontrada em aplicações reais, através de containers. Desta maneira, a comunicação entre eles se dá através de uma camada de rede própria e isolada do resto do sistema operacional. Assim, como containers usam recursos do próprio sistema operacional para realizar este isolamento das comunicações, assume-se que esta se dá de maneira segura.

Com isso, temos então para o funcionamento da rede, um container contendo o Zookeper, que gerencia nodos de serviço de mensageria, e um ou mais containers representando nodos Kafka diferentes, sendo estes abertos apenas para comunicações com TLS.

5 CONCLUSÃO

Neste trabalho foi desenvolvida uma solução para a implantação do *Device Server* em um ambiente distribuído. O movimento atual de migração para microsserviços traz grandes vantagens, principalmente em um âmbito pós-pandêmico onde houve uma grande demanda para soluções remotas distribuídas. Deste ponto de vista no entanto, o *Device Server* apresentava algumas limitações, como a necessidade de um servidor web local, que inviabilizavam a migração para uma arquitetura distribuída.

Este trabalho então apresentou um novo paradigma de comunicação, o serviço de mensageria, o qual tem natureza distribuída e assíncrona, de forma que este pudesse substituir o padrão de mensagens HTTP através de uma interface REST. Com isso foi escolhido então um sistema maduro e usado amplamente na indústria de tecnologia, o Apache Kafka.

Foi implementado também uma solução de autenticação e sigilo por TLS, removendo assim a implementação anterior de um protocolo Diffie-Hellman, o qual não havia sido testado contra estratégias comuns de ataques, por um protocolo mais confiável.

Por último foi desenvolvido uma nova aplicação, a qual diferentes usuários podem se autenticar e gerenciar suas instâncias de *Device Server*. Desta maneira tem-se então uma solução completamente distribuída e escalável para acesso a recursos locais através de instâncias do novo *Device Server*.

6 TRABALHOS FUTUROS

Durante o desenvolvimento, algumas decisões foram tomadas que não necessariamente eram as melhores disponíveis, mas sim as mais viáveis no momento. Sendo assim este trabalho ainda pode ser evoluído em algumas áreas como:

- Substituição do serviço de banco de dados pelo Ksql, o qual oferece uma forma de tratar tópicos tais como se fossem schemas de banco de dados, permitindo fazer-se consultas SQL a tais tópicos e armazenar dados.
- Fazer uso de Kafka Streams, ferramenta que permite que uma mensagem seja transformada e inserida em outro tópico, permitindo assim um mecanismo de adição de dispositvos sem a necessidade de inserir seus identificadores manualmente.
- Melhorar o mecanismo de autenticação do usuário, substituindo o modelo de nome de usuário e senha, por algo mais moderno, ou talvez por um mecanismo de certificados digitais.

REFERÊNCIAS

ALLEN, C.; DIERKS, T. **The TLS Protocol Version 1.0**. RFC Editor, 1999. RFC 2246. (Request for Comments, 2246). Disponível em: https://www.rfc-editor.org/info/rfc2246. Acesso em: 12 set. de 2022.

ALMEIDA, M. G. de; CANEDO, E. D. Authentication and authorization in microservices architecture: A systematic literature review. **Applied Sciences**, v. 12, n. 6, 2022. ISSN 2076-3417. Disponível em: https://www.mdpi.com/2076-3417/12/6/3023.

BOEYEN, S. et al. **Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile**. RFC Editor, 2008. RFC 5280. (Request for Comments, 5280). Disponível em: https://www.rfc-editor.org/info/rfc5280. Acesso em: 03 nov. de 2022.

CHOU, W. Inside ssl: the secure sockets layer protocol. **IT Professional**, v. 4, p. 47–52, 2002.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. **IEEE Transactions on Information Theory**, v. 22, n. 6, p. 644–654, 1976.

KATZ, J.; LINDELL, Y. **Introduction to modern cryptography, second edition**. 2. ed. Philadelphia, PA: Chapman & Hall/CRC, 2014. (Chapman & Hall/CRC Cryptography and Network Security Series).

KENT, S. Evaluating certification authority security. In: IEEE. **1998 IEEE Aerospace Conference Proceedings (Cat. No. 98TH8339)**. [S.l.], 1998. v. 4, p. 319–327.

KIM, H.; LEE, E. A. Authentication and authorization for the internet of things. **IT Professional**, v. 19, n. 5, p. 27–33, 2017.

LI, N. Research on diffie-hellman key exchange protocol. In: **2010 2nd International Conference on Computer Engineering and Technology**. [S.l.: s.n.], 2010. v. 4, p. V4–634–V4–637.

LU, R. et al. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In: **2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing**. [S.l.: s.n.], 2014. p. 69–78.

NACCACHE, D.; M'RAIHI, D. Cryptographic smart cards. **IEEE Micro**, v. 16, n. 3, p. 14–24, 1996.

NAIK, N. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: **2017 IEEE International Systems Engineering Symposium (ISSE)**. [S.l.: s.n.], 2017. p. 1–7.

RALPH, P. et al. Pandemic programming: How COVID-19 affects software developers and how their organizations can help. **CoRR**, abs/2005.01127, 2020. Disponível em: https://arxiv.org/abs/2005.01127.

RANKL, W.; EFFING, W. **Smart Card Handbook**. 4th. ed. [S.l.]: Wiley Publishing, 2010. ISBN 0470743670.

RESCORLA, E. **The Transport Layer Security (TLS) Protocol Version 1.3**. RFC Editor, 2018. RFC 8446. (Request for Comments, 8446). Disponível em: https://www.rfc-editor.org/info/rfc8446. Acesso em: 03 nov. de 2022.

RIVEST, R.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, v. 21, p. 120–126, 1978.

RIVEST, R. L. Cryptography. In: **Algorithms and complexity**. [S.l.]: Elsevier, 1990. p. 717–755.

STALLINGS, W. **Cryptography and Network Security Principles and Practice, 7/E**. [S.l.]: Pearson Education, 2017.

STALLINGS, W.; BROWN, L. **Computer Security: Principles and Practice**. 3rd. ed. USA: Prentice Hall Press, 2014. ISBN 0133773922.

TRAPP, M. Uma alternativa aos applets em java para acesso a dispositivos. **UNIVERSIDADE FEDERAL DE SANTA CATARINA**, 2016.

WOO, T.; LAM, S. Authentication for distributed systems. **Computer**, v. 25, n. 1, p. 39–52, 1992.

APÊNDICE A – ARTIGO SBC

Sistema para Disponibilização de Recursos Computacionais Locais Remotamente

Matheus E. de França[1]

[1]Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC)
CEP: 88040-370 – Florianópolis – SC – Brazil

matheus.eyng@grad.ufsc.br

Abstract. *With the advent of industry 4.0 and the context of a post-pandemic world, different software services migrated to the cloud. Access systems for hardware devices, how- ever, have difficulty adapting to the environment of distributed systems, most of which are meant for local access. Therefore, this work proposes the creation of a hard- ware device management system using distributed computing techniques, focusing on data transmission security, which complements and enriches a previously developed application called Device Server.*

Resumo. *Com o advento da indústria 4.0 e o contexto de um mundo pós pandemia, viu-se a migração de diferentes serviços de software para a nuvem. Sistemas de acesso a dispotivos de hardware no entanto, têm dificuldade para adaptar ao ambiente de sistemas distribuídos sendo em sua maioria desenvolvidos para acesso local. Sendo assim, este trabalho propõe a criação de um sistema de gerenciamento de dispositivos de hardware usando técnicas de computação distribuída, com foco na segurança da transmissão dos dados, que complementa e enriquece uma aplicação previamente desenvolvida chamada Device Server.s*

1. Introdução

Com a virada do século, ficou claro que a internet desenvolveria um papel essencial não somente no cotidiano do cidadão comum, mas principalmente no mundo dos negócios. A ubiquidade da internet fez com que houvesse uma revolução na computação, principalmente em áreas relacionadas a computação distribuída. O ano de 2020 foi crucial para por a prova a viabilidade do trabalho remoto. A pandemia de COVID-19 desafiou não somente a saúde pública mas também o mercado global. Em apenas alguns meses, empresas tiveram que adotar alguma maneira de trabalhar remotamente, devido a necessidade de lockdowns em diversos países para conter a propagação do vírus pandêmico. O impacto destes acontecimentos foi que grande parte da população foi forçada a exercer o trabalho de suas próprias casas, sendo que grande parte destas pessoas não possuem condições propícias para trabalhar, seja por falta de ferramentas, conexão estável ou até mesmo por falta de um ambiente adequado [Ralph et al. 2020]. Apesar de existirem muitas ferramentas de acesso remoto e de comunicação por rede, nota-se uma falta de softwares que permitam o acesso de recursos locais, como dispositivos USB por exemplo, que sejam agnósticos quanto o tipo e a finalidade destes dispositivos. A Microsoft oferece opções para compartilhar recursos, mas isto limita o ambiente de trabalho ao sistema operacional Windows. Outras empresas como FlexiHub oferecem uma solução mais genérica para

acesso a dispositivos USB remotamente, porém por ser uma aplicação voltada ao mercado, é cobrado por uso sendo o preço dos planos oferecidos não transparentes ao cliente a menos que este esteja disposto a entrar em contato com o time de vendas.

No Laboratório de Segurança em Computação (LabSEC) da Universidade Federal de Santa Catarina (UFSC) foi desenvolvido, como fruto de um trabalho de conclusão de curso do aluno Marlon Trapp [Trapp 2016], um software para comunicação com dispositivos criptográficos. O Device Server é um software de comunicação segura entre cliente e dispositivo criptográfico, mais comumente, tokens criptográficos. Atualmente esta aplicação é usada em conjunto com o projeto Hawa, o qual é responsável pelas autoridades certificadoras finais da ICP-Brasil, e com outras autoridades certificadoras de segundo nível. Apesar do projeto servir seu propósito, se viu necessário fazer certas adaptações para modernizar a abordagem dos processos. Nos últimos anos houve um questionamento quanto a necessidade de aplicações monolíticas, pricipalmente quanto a sua dificuldade de adaptação a diferentes ambientes.

Neste contexto, o projeto existente se mostrou limitado e inflexível. A possibilidade de comunicação com apenas um dispositivo criptográfico limita seus casos de uso em um abiente onde a autenticação pode necessitar mais de um dispositivo. Além disso, a necessidade do recurso estar disponível localmente praticamente impossibilita uma implementação distribuída moderna para os outros projetos dependentes.

Tendo como caso de uso o funcionamento da ICP-Brasil, este trabalho visa desenvolver uma solução distribuída e dinâmica, permitindo a implantação em um ambiente em nuvem, onde seja suportado diversos recursos diferentes atrelados a diversos usuários, de modo que esta solução possa substituir a antiga e proporcionar um novo paradigma nos softwares dependentes da ICP-Brasil.

2. Sistemas de Mensageria

Softwares modernos raramente vivem em isolamento, sendo que atualmente vemos cada vez mais aplicações adotando o paradigma da computação distribuída. Ao mesmo tempo, tem-se percebido uma tendência crescente na taxa de coleta e transmissão de dados, como o por exemplo o Facebook que maneja cerca de 10^6 requisições por segundo, o que representa uma taxa de aproximadamente 9GB/s [Lu et al. 2014].

Com isso surge a necessidade de desenvolver novas maneira de trocar uma alta taxa de mensagens de maneira efetiva. O HTTP, protocolo original para a internet proposto por Tim Berners-Lee, já não atende mais as demandas de um ambiente de alta vazão, sendo um protocolo verboso e demasiadamente pesado [Naik 2017].

Desta maneira, desenvolveu-se nos útlimos anos diversos sistemas de mensageria, os quais implementam protocolos mais leves, de comunicação assíncrona, permitindo uma alta vazão de dados. Estes sistemas usualmente são implementados com diversas redundâncias e garantias, permitindo assim uma abordagem *fire-and-forget* das aplicações que os usam, ou seja, o envio correto das mensagens é garantido pelo sistema permitindo assim que as aplicações distribuídas possam escalar seus serviços de maneira descomplicada.

3. Análise do Device Server

Neste capítulo será feita uma breve análise do Device Server, pontuando algumas características importantes do seu funcionamento, bem como falhas as quais este trabalho almeja corrigir.

3.1. Modelo de Comunicação Remota

Como o objetivo original do Device Server era substituir um applet web, um tipo de extensão de navegador o qual poderia executar código, a idéia central quanto a comunicação era torná-la simples, fácil e segura. Sendo assim, a implementação a qual foi escolhida dá-se diretamente entre aplicação e navegador. Após o usuário instalar a aplicação em sua máquina, esta estaria pronta para trocar informações com um servidor web através de uma comunicação direta com o navegador, como podemos ver na figura 1.

Neste modelo adotado, durante a navegação pela página web, a qual se dá na conexão 1, o navegador poderia requisitar a criação de uma sessão com o Device Server através da conexão 2, o qual reponderia através da mesma conexão. Tendo feita esta comunicação o navegador então enviaria todos os dados pertinentes de volta ao servidor web.

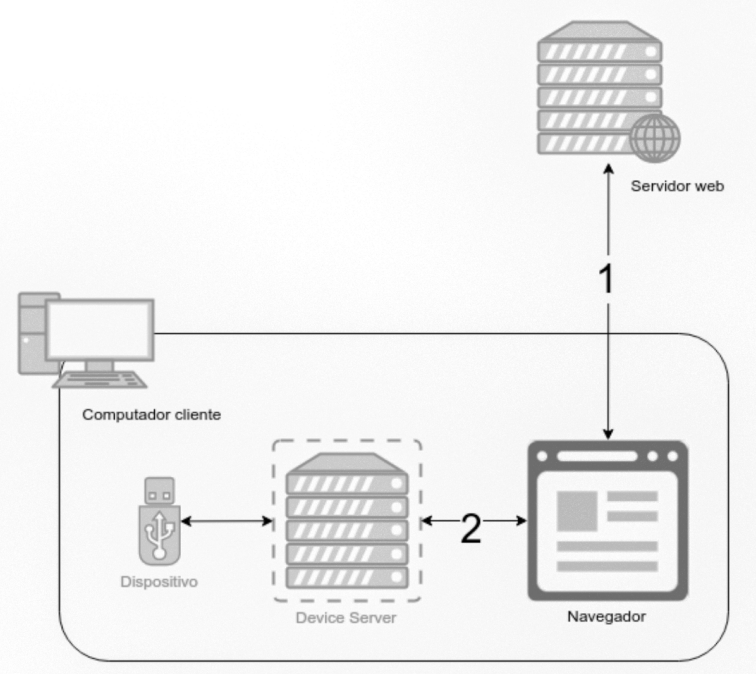

Figura 1. Proposta original do funcionamento do Device Server, [Trapp 2016]

3.2. Padrão de Comunicação

Na implementação original foi escolhido um padrão de comunicação por interfaces REST. Como o objetivo do trabalho anterior era possibilitar uma comunicação direta com a aplicação ou site web, este modelo é bastante adequado. No entanto, para tornar esta comunicação segura é necessário um protocolo de cifragem de mensagens. O trabalho original propõe uma implementação SSL no entanto no protótipo temos apenas uma implementação do próprio autor do protocolo Diffie-Hellman.

3.3. Módulos

Os módulos são implementações as quais seguem a API do Device Server e os quais representam a real funcionalidade do projeto. Este por si só realiza apenas a operação de instânciar um módulo e encaminhar as mensagens para o mesmo. No entanto, com a adição de módulos, o Device Server ganha diferentes funcionalidades de acordo com as implementações de cada módulo.

Os módulos relevantes para este trabalho são os que implementam operações criptograficas e comunicação com dispositivos critptográficos.

4. Proposta

Podemos perceber que apesar do modelo original de comunição remota ser simples e fácil de entender, ela apresenta alguns problemas. A escolha de comunicar-se diretamente com o navegador, aliado ao fato de que é necessário também se comunicar com o dispositivo local, confina a aplicação ao ambiente local. Assim, caso o usuário queira usar um outro dispositivo que esteja conectado em outra máquina, este terá que iniciar uma nova conexão nesta outra máquina, sendo impossível continuar a sessão atual.

Tendo isso, este trabalho propõe a criação de uma nova solução, o *Device Broker* o qual irá fazer um papel de *middleware* entre o navegador e diversas instâncias distribuídas do Device Server, além do *Device Server Client*, o qual aplica melhorias e substitui o Device Server, fazendo a comunicação com o *Device Broker*. Por fim é proposto também a criação de um cliente o qual tem como objetivo fazer a ponte de comunicação entre usuário final e *Device Broker*. O diagrama da proposta encontra-se na figura 2

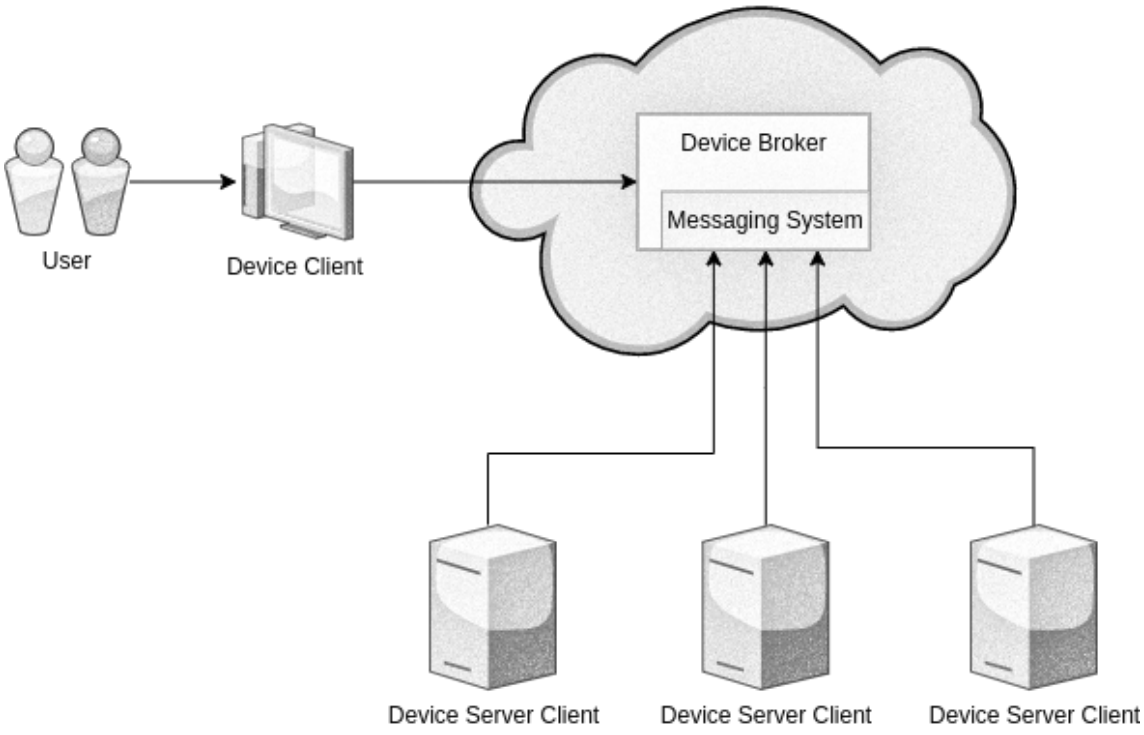

Figura 2. Proposta do novo modelo de comunicação remota

O principal objetivo deste novo modelo é possiblitar que estas instâncias possam

habitar outros ambientes remotos mas ainda assim comportar-se como se estivessem presentes localmente. Deste modo, será possível elminar uma das principais limitações do modelo original, a necessidade de se ter o Device Server localmente.

Para alcançar este objetivo no entando, precisamos de um novo padrão de comunicação. A interface REST adotada anteriormente é adequada para quando temos apenas um Device Server local, porém esta abordagem se torna inviável quando necessário gerenciar diversas instâncias deste. O REST por definição não guarda dados quanto ao estado do cliente com o servidor, dificultando a implementação de sessões diferentes para clientes diferentes. No mais, pela natureza síncrona da conexão original, caso ocorra algum erro ou perda na conexão, todos os dados quanto aquela conexão serão perdidos.

5. Escolha do sistema de Mensageria

Como mencionado anteriormente, o modelo de comunicação escolhido para esta implementação foi o a comunicação por mensagens, ou mensageria. Este paradigma de comunicação apresenta algumas importantes vantagens, principalmente no contexto de ambientes distribuídos. A principal vantagem destes sistemas é sua alta confiabilidade não somente quanto ao envio de mensagens mas quanto ao tempo de atividade do sistema, isto é, o tempo o qual a infraestrutura necessária encontra-se disponível.

Desta maneira os sistemas de mensageria são por natureza distribuídos e consequentemente facilmente escaláveis. Um outro aspecto importante é que estes oferecem diversas garantias quanto ao envio e chegada das mensagens, podendo ser configurados de acordo com as necessidades de cada infraestutura. No nível mais básico, todas as implementações de sistemas de mensageria oferecem a garantia que, dado o envio de uma mensagem, esta chegará ao seu nodo de destino salvo um problema grave como por exemplo, todos os nodos estarem indisponíveis ao mesmo tempo.

Assim, realisticamente existem duas alternativas de código aberto e maduras, as quais oferecem todas as suas funcionalidadas prontas para uso, Apache Kafka e RabbitMQ. Ambos são ótimas implementações porém diferem em alguns aspectos importantes. O principal ponto onde estes dois sistemas diferem é quanto ao "armazenamento" das mensagens, no Apache Kafka as mensagens chegam e são escritas em sequências em uma estrutura que se assemelha a um log, ou seja, depois de recebida, a mensagem fica para sempre gravada neste log, chamado de tópico, armazenado em disco. Os logs, por padrão permanecem por 30 dias, sendo este período, chamado de período de retenção, facilmente configurável. Apesar de parecer uma desvantajoso armazenar os logs em disco, principalmente em um caso de uso onde possivelmente milhares de mensagens podem chegar a cada segundo, o Kafka oferece também diversas formas de compressão para o armazenamento dos logs.

Por outro lado, no RabbitMQ temos um sistema mais simples de filas. As mensagens chegam e são armazenadas em uma fila, onde cada consumidor, após consumir esta mensagem tira a mesma da fila, efetivamente apagando esta mensagem do nodo de maneira que não é mais possível recupera-la. Outro ponto onde as duas implementações diferem é na topologia oferecida, enquanto no Kafka é oferecido apenas *publisher/subscriber*, o RabbitMQ oferece tipos como *fan-out* onde uma lista de filas recebe uma mesma mensagem, *Header exchange* onde a fila destino é baseada no cabeçalho da mensagem, entre

outros.

Dado estas diferenças fica claro que qualquer um dos dois serviços seria mais que capaz de atender as necessidade que possam se apresentar, visto que este projeto inicialmente não prevê um grande volume de mensagens sendo trocadas. Mesmo assim, foi escolhido o Kafka como serviço de mensageria. O fato das mensagens não desaparecerem oferece maior flexibilidade quanto a implementação tendo que desta maneira podemos ter vários consumidores consumindo mensagens de um mesmo tópico sem preocupação quanto a apenas o destinatário receber a mensagem. Ter logs permanentes também é útil para diagnosticar possíveis problemas durante a implementação ou também para possíveis fins de auditoria em casos de uso real do projeto.

6. Arquitetura do sistema

Após escolhida a implementação do sistema de mensageria, é preciso então definir a arquitetura do projeto no nível de troca de mensagens, ou seja, a topologia da rede de mensagens. Como discutido anteriormente o Kafka oferece somente o padrão *publisher/subscriber* o que é o suficiente para atender aos requisitos do trabalho. Como mostrado na figura 2, um cliente estará ligado a diversos *Device Servers*, isso significa que precisaremos que o *device* alvo da mensagem receba esta mensagem corretamente. Uma primeira idéia seria organizar os *Device Servers* em um grupo de consumidores. No Apache Kafka, cada consumidor deve pertencer a um grupo, se nenhum for definido nas configurações, então o próprio Kafka designará um grupo para o consumidor. O conceito de grupos é importante pois afeta a maneira como as mensagens são recebidas, e consequentemente, consumidas. Quando uma mensagem chega a um tópico, apenas um membro de cada grupo irá consumir efetivamente esta mensagem. A política de escolha de qual dos consumidores em um grupo receberá a mensagem se dá de maneira simples, seguindo um algoritmo de round-robin , de maneira que a cada mensagem, um consumidor diferente do grupo será escolhido até todos os consumidores terem recebido ao menos uma mensagem, repetindo assim o ciclo.

Com isso em mente podemos perceber que a escolha de agrupar todos os nossos consumidores (Device Servers) em um grupo não é ideal, visto o consumidor a receber a mensagem pode não ser o destinatário pretendido. Assim, optou-se por definir diferentes grupos para diferentes consumidores. Não há evidências na documentação do Apache Kafka que relacionem a criação de diversos grupos diferentes com algum tipo de degradação de desempenho, desta maneira, não teremos nesta área com a criação de diversos grupos.

7. Segurança

7.1. Comunicação

O modelo anterior do Device Server também garantia autenticação através da comunicação por TLS/SSL, o modo como foi implementado não se adapta bem ao novo modelo distribuído. Na implementação original, durante a inicialização da aplicação, um servidor é iniciado localmente, e uma URL é gerada e requisita-se ao usuário o acesso a esta página. Ao fazer isso, o usuário pode então aceitar a conexão do seu navegador com o servidor, aceitando o certificado do Device Server como um certificado confiável, estabelecendo assim a possibilidade de comunicação segura através do navegador.

Diante disso, como este trabalho visa desvincular a interação direta com o navegador por parte do Device Server, foi retirado do código tudo relacionado a sessões e comunicação TLS através de um *web-server*. Porém ainda é preciso garantir a autenticação segura em um servidor, no caso deste trabalho, um broker Kafka. Para isso, felizmente esta autenticação pode ser feita de maneira mais simples.

Primeiro é necessário que se tenha um certificado de uma AC e seu par de chaves correspondente. Este *keystore* será usado como âncora de confiança, ou seja, todo cliente que tentar autenticar-se com um broker deverá possuir um certificado que pertença a uma cadeia contendo esta AC. Cada broker então terá um certificado próprio que o identifica, emitido pela esta AC de confiança. O mesmo é verdade para cada consumidor e produtor que deseja se conectar ao Kafka, para uma conexão com o broker é exigido um certificado que faça parte da mesma cadeia de confiança.

Desta maneira temos então um canal de comunicação seguro e autenticado, onde todos os agentes conectados são realmente quem eles afirmam ser, garantindo a mesma robustez do sistema anterior e com todas as garantias oferecidas pelo protocolo TLS.

7.2. Sigilo

Tendo estabelecida a conexão segura por TLS temos que todas as mensagens trocadas entre o Kafka passam a ser cifradas, garantindo o sigilo da aplicação. Assim, além de servir como fator de autenticação no sistema, o TLS também garante que mesmo em uma rede comprometida, onde agentes maliciosos possam interceptar os pacotes de comunicação, estas mensagens não serão decifradas.

Desta maneira podemos usar estas conexões para comunicação com o Kafka sem preocupar-mos com agentes maliciosos na rede. Mesmo com a rede comprometida, como o conteúdo das mensgens está criptografado, seria praticamente impossível decifrar estas mensgens sem as chaves corretas. Desta maneira, mesmo que ocorra interceptção de pacotes, ataques do tipo *man in the middle*, temos a garantia que a informação contida nesses pacotes está segura. Isso facilita em alguns aspectos da implementação, como veremos mais a frente, onde precisamos passar informações como senhas e PIN's de tokens criptográficos de um serviço a outro.

8. Adaptação do Device Server

O Device Server foi concebido para ser um programa de adaptável a diferentes ambientes e funções, suas funcionalidades dependem do carregamento de módulos desenvolvidos separadamente os quais, seguindo a API descrita pelo Device Server, são possíveis de realizar diferentes tarefas. Além disso, a linguagem utilizada, Java, também demonstra este compromisso com a flexibilidade e portabilidade do projeto original.

Com isso em mente, o objetivo deste trabalho é atualizar o Device Server para um âmbito moderno de microsserviços, mas ainda mantendo as funcionalidades e a proposta original de ser um projeto flexivel e portátil.

8.1. Novo Modelo de Autenticação

Como o projeto original previa comunicação por interface REST, e essa comunicação deveria se dar de maneira segura, utilizando TLS. O protótipo do Device Server no entando

foi simplificado, sendo a parte do TLS substituída por uma implementação própria do protocolo de *Diffie-Hellman*.

A troca de chaves de *Diffie-Hellman* proporciona sigilo na troca de mensagens em um meio comprometido, porém este protocolo também é bastante antigo e já passou por diferentes revisões para evitar uma maior gama de ataques que se tornaram populares na útlimas décadas. Além disso, a efetividade deste protocolo depende na escolha "criptograficamente boa" de números primos para realizar a troca e a cifragem de mensagens.

Sendo assim, neste trabalho optou-se por remover completamente a impelmentação de interfaces REST, visto que será utilizado o sistema de mensageria. Assim também não temos mais a necessidade de expor essas interfaces portanto também foi removido o servidor web embutido.

Sem um servidor web e endpoints para acesso HTTP, pode-se então remover completamente também a implementação de autenticação usando *Diffie-Hellman* e toda a parte de sessões. O projeto irá se comunicar apenas através do serviço de mensagens, e apenas receberá mensagens do cliente, sendo assim não há a necessidade de expor nenhum outro canal de comunicação com outra aplicação ou com o usuário. Além disso a autenticação passa a ser feita diretamente através do Kafka, sendo a conexão de cada instância do Device Server autenticada e protegida por TLS.

8.2. Mudanças na API

Como teremos várias instâncias conectadas na rede, precisamos que as mensagens enviadas possuam algum campo para distinguir uma instância da outra. Uma vez que uma mensagem chega a um tópico, todos os Device Servers que consomem deste tópico irão receber a mesma mensagem, o que significa que todos eles produzirão uma resposta, enquanto o cliente espera que haja apenas uma resposta para cada mensagem enviada. No mais, seria um erro fazer com que todas as instâncias conectadas executem o mesmo comando, sendo que estas podem ter módulos diferentes carregados.

Segue que precisa-se enviar metadados sobre o Device Server junto com a mensagem, portanto foi criada a classe *MessageMetadata.java* a qual armazena, de maneira simples, dados como o remetente da mensagem, o destinatário, e qual módulo deve executar a mensagem (no caso de uma mensagem para um Device Server).

No entanto agora precisamos adaptar esta mensagem para possuir os metadados. Simplesmente adicionar estes dados novos a API antiga parece simples, porém qualquer alteração na forma da mensagem *DeviceMessage* irá fazer com que módulos criados antes destas mudanças deixem de funcionar corretamente. Para evitar isso foi criado então uma nova classe, essa classe representa as mensagens que realmente são trocadas entre os serviços dentro do Kafka, ou seja, estas são as mensagens que são escritas nos tópicos e consumidas pelos consumidores. Assim garantimos a retrocompatibilidade com os módulos antigos mas conseguimos adaptar para o novo modelo, onde facilmente podemos extrair a *DeviceMessage*.

9. Implementação do Cliente

O Device Server Client é o serviço o qual será responsável pelo gerenciamento da comunicação com os Device Servers e também a interface para o cliente. Com as

mudanças feitas no Device Server, temos agora que tal serviço atua de maneira orientada a eventos, ou seja, caso não chegue uma mensagem no tópico o serviço fica ocioso, aguardando o próximo evento. Assim as reponsabilidados quanto a funcionalidade do sistema foram passadas ao cliente, onde este deve realizar a tarefas como comunicação com o serviço de banco de dados, interação com o usuário e gerenciamento de dispositivos e usuários conectados.

Como interface para o cliente considerou-se implementar um modelo web baseado em chamadas REST, muito parecido com o conceito que existia anteriormente no Device Server. A vantagem de um sistema deste tipo seria a possibilidade de manter uma funcionalidade idêntica a proposta original, onde outras aplicações pudessem interagir com cliente e acessar os dispositivos criptográficos. Esta abordagem no entando requer que a aplicação tenha um próprio servidor web para expor as rotas REST, o que leva a mais considerações sobre autenticação e segurança de rede, uma vez que a autenticação através do TLS tratada neste trabalho refere-se apenas as conexões do Kafka.

Este trabalho no entanto optou por uma opção mais visual, utilizando recursos da linguagem para criar uma interface gráfica. Desta maneira evitamos esta camada de complexidade a mais que uma interface web nos leva a considerar, na interface gráfica temos total controle sobre como é o fluxo do usuário, evitando assim preocupações com segurança neste aspecto.

9.1. Protótipo

Como mencionado, o cliente implementa todas as questões de comunicação com o Kafka de maneira idêntica ao Device Server, sendo assim, o cliente autentica-se na rede através de um certificado próprio conhecido pelo broker do Kafka, estabelecendo uma conexão protegida pro TLS.

O cliente no entanto precisa lidar com outro tipo de autenticação, a autenticação do usuário no sistema. O usuário tem um usuário e senha e uma lista de dispositivos pertencentes a ele. Sendo assim, no cliente temos que ao iniciar a execução do programa uma tela de login simples, como mostra a figura 3.

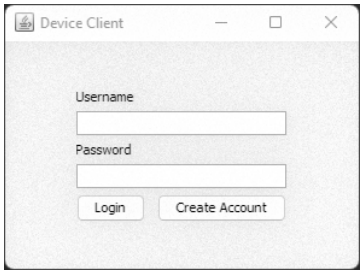

Figura 3. Tela de login no Device Server Client

Concluída a etapa de autenticação do cliente, temos em seguida uma tela a qual apresenta os dispositivos conectados. Existem botões para adicionar e remover dispositivos a qualquer momento. Ambas ações geram mensagens para atualizar as informações do usuário no banco de dados, assim caso ocorra algum erro ou não haja uma resposta do banco, então uma mensagem de erro é exibida para o usuário e a operação cancelada. A figura 4 mostra esta tela de gerenciamento de dispositivos.

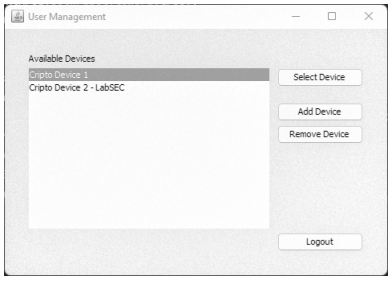

Figura 4. Tela de gerenciamento de dispositivos

Como este trabalho é destinado a operações com dispositivos criptográficos, após selecionado o o módulo uma mensagem é enviada requisitando todos os certificados disponíveis. O usuário é apresentado com uma lista de *alias* de certificados, permitindo-o facilmente identifiar cada um e selecionar o desejado, como mostrado na figura 5.

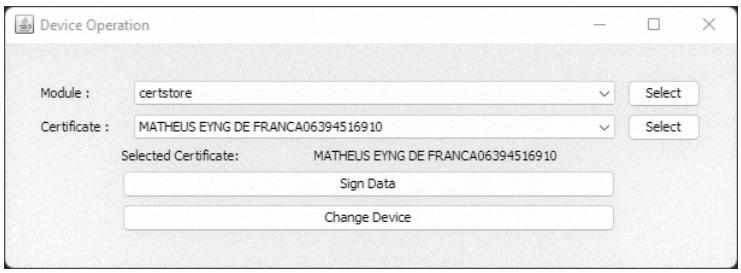

Figura 5. Tela de escolha de módulo e assinatura de arquivo.

Após selecionado o módulo e o certificado a ser usado, temos quase todas as informações necessárias para realizar uma assinatura. Ao clicar o botão de assinatura, o usuário então é solicitado a escolher um arquivo contendo as informações as quais deseja assinar. Feito isso basta apenas obter a senha ou PIN do dispositivo criptográfico. Assim a mensagem de requisição de assinatura é formada e enviada para o dispositvo. Aqui confiamos no sigilo oferecido pelo TLS para enviar o PIN na mensagem, que precisamos lembrar, chega ao tópico em formato JSON. Com isso é apresentado ao usuário então o resultado da assinatura como podemos ver na figura 6.

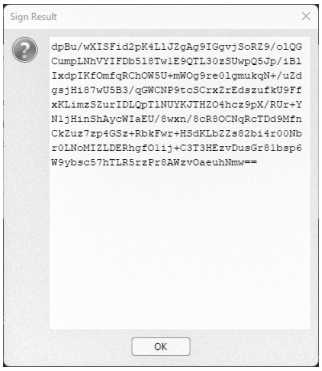

Figura 6. Tela de resultado da assinatura.

10. Conclusão

Neste trabalho foi desenvolvida uma solução para a implantação do *Device Server* em um ambiente distribuído. O movimento atual de migração para microsserviços traz grandes vantagens, principalmente em um âmbito pós-pandêmico onde houve uma grande demanda para soluções remotas distribuídas. Deste ponto de vista no entanto, o *Device Server* apresentava algumas limitações, como a necessidade de um servidor web local, que inviabilizavam a migração para uma arquitetura distribuída.

Este trabalho então apresentou um novo paradigma de comunicação, o serviço de mensageria, o qual tem natureza distribuída e assíncrona, de forma que este pudesse substituir o padrão de mensagens HTTP através de uma interface REST. Com isso foi escolhido então um sistema maduro e usado amplamente na indústria de tecnologia, o Apache Kafka.

Foi implementado também uma solução de autenticação e sigilo por TLS, removendo assim a implementação anterior de um protocolo Diffie-Hellman, o qual não havia sido testado contra estratégias comuns de ataques, por um protocolo mais confiável.

Por último foi desenvolvido uma nova aplicação, a qual diferentes usuários podem se autenticar e gerenciar suas instâncias de *Device Server*. Desta maneira tem-se então uma solução completamente distribuída e escalável para acesso a recursos locais através de instâncias do novo *Device Server*.

Referências

Lu, R., Wu, G., Xie, B., and Hu, J. (2014). Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 69–78.

Naik, N. (2017). Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7.

Ralph, P., Baltes, S., Adisaputri, G., Torkar, R., Kovalenko, V., Kalinowski, M., Novielli, N., Yoo, S., Devroey, X., Tan, X., Zhou, M., Turhan, B., Hoda, R., Hata, H., Robles, G., Fard, A. M., and Alkadhi, R. (2020). Pandemic programming: How COVID-19 affects software developers and how their organizations can help. *CoRR*, abs/2005.01127.

Trapp, M. (2016). Uma alternativa aos applets em java para acesso a dispositivos. *UNIVERSIDADE FEDERAL DE SANTA CATARINA*.

APÊNDICE B – CÓDIGO DATABASE-SERVICE

Classe User.java

```
package com.example.databaseservice.entity;

import java.util.Set;
import javax.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "users", schema = "${database.schema}")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(unique = true)
    private String username;

    private String passwordHash;

    @ElementCollection(fetch = FetchType.EAGER)
    private Set<String> registeredDevices;

    @Override
    public String toString() {
        return String.format("User[id=%s, username=%s, registeredDevices=%s]",
            id, username, registeredDevices);
    }
}
```

Classe KafkaConnectionParams.java

```
package com.example.databaseservice.kafka.config;

import lombok.Data;
import org.springframework.beans.factory.annotation.Value;
import
    org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;
```

```
@Data
@Configuration
@EnableConfigurationProperties
public class KafkaConnectionParams {
    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServer;
}
```

Classe KafkaConsumerConfig.java

```
package com.example.databaseservice.kafka.config;

import com.example.databaseservice.kafka.message.UserManagementMessage;
import java.util.HashMap;
import java.util.Map;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.KafkaListenerContainerFactory;
import org.springframework.kafka.core.*;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.support.serializer.JsonDeserializer;

@Configuration
public class KafkaConsumerConfig {

    private final KafkaConnectionParams connectionConfig;

    public KafkaConsumerConfig(KafkaConnectionParams connectionConfig) {
        this.connectionConfig = connectionConfig;
    }

    public Map<String, Object> consumerConfig() {
        HashMap<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionConfig.getBootstrapServer());
        // Key and Value are Strings here
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
```

```
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        JsonDeserializer.class);
    return props;
}

@Bean
public ConsumerFactory<String, UserManagementMessage>
    userManageConsumerFactory() {
    return new DefaultKafkaConsumerFactory<>(
            consumerConfig(), new StringDeserializer(), new
                JsonDeserializer<>(UserManagementMessage.class));
}

@Bean
public
    KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,
    UserManagementMessage>>
        userManagefactory(ConsumerFactory<String, UserManagementMessage>
            consumerFactory) {
    ConcurrentKafkaListenerContainerFactory<String, UserManagementMessage>
        factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory);
    return factory;
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String,
    UserManagementMessage> userManagementConsumerFactory(
        ConsumerFactory<String, UserManagementMessage> consumerFactory,
        ProducerFactory<String, String> producerFactory) {
    ConcurrentKafkaListenerContainerFactory<String, UserManagementMessage>
        factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory);
    factory.setReplyTemplate(kafkaTemplate(producerFactory));
    return factory;
}

@Bean
public KafkaTemplate<String, String> kafkaTemplate(ProducerFactory<String,
    String> kafkaProducerFactory) {
```

```
        return new KafkaTemplate<>(kafkaProducerFactory);
    }

    public Map<String, Object> simpleProducerConfig() {
        HashMap<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionConfig.getBootstrapServer());
        // Key and Value are Strings here
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        return props;
    }

    @Bean
    public ProducerFactory<String, String> simpleProducerFactory() {
        return new DefaultKafkaProducerFactory<>(simpleProducerConfig());
    }
}
```

Classe KafkaProducerConfig.java

```
package com.example.databaseservice.kafka.config;

import java.util.HashMap;
import java.util.Map;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.support.serializer.JsonSerializer;

@Configuration
public class KafkaProducerConfig {

    private final KafkaConnectionParams connectionConfig;

    public KafkaProducerConfig(KafkaConnectionParams connectionConfig) {
        this.connectionConfig = connectionConfig;
    }

    public Map<String, Object> producerConfig() {
        HashMap<String, Object> props = new HashMap<>();
```

```
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionConfig.getBootstrapServer());
        // Key and Value are Strings here
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            JsonSerializer.class);
        return props;
    }
}
```

Classe KafkaTopicConfig.java

```
package com.example.databaseservice.kafka.config;

import org.apache.kafka.clients.admin.NewTopic;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.TopicBuilder;

@Configuration
public class KafkaTopicConfig {

    /*
    FIXME: topics should not be created at runtime.
     */
    @Value("${consumer.topic}")
    private String receiverTopicName;

    @Value("${producer.topic}")
    private String senderTopicName;

    public KafkaTopicConfig(
            @Value("${consumer.topic}") String receiverTopicName,
                @Value("${producer.topic}") String senderTopicName) {
        this.receiverTopicName = receiverTopicName;
        this.senderTopicName = senderTopicName;
    }

    @Bean("${sender.topic}")
    public NewTopic senderTopic() {
        return TopicBuilder.name(senderTopicName).build();
```

```
    }

    @Bean("${receiver.topic}")
    public NewTopic receiverTopic() {
        return TopicBuilder.name(receiverTopicName).build();
    }
}
```

Classe PublishException.java

```
package com.example.databaseservice.kafka.exception;

public class PublishException extends RuntimeException {
    private static final String FAILED_TO_PUBLISH_TOPIC_OFFSET = "Failed to
        publish message to topic %s with offset %d";
    private static final String FAILED_TO_PUBLISH_TOPIC = "Failed to publish
        message \"%s\" to topic %s";
    private static final String FAILED_TO_PUBLISH_MSG = "Failed to publish
        message \"%s\"";

    public PublishException(String message) {
        super(String.format(FAILED_TO_PUBLISH_MSG, message));
    }

    public PublishException(String message, String topic) {
        super(String.format(FAILED_TO_PUBLISH_TOPIC, message, topic));
    }

    public PublishException(String topic, long offset) {
        super(String.format(FAILED_TO_PUBLISH_TOPIC_OFFSET, topic, offset));
    }

    public PublishException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Classe MessageOperation.java

```
package com.example.databaseservice.kafka.message;

public enum MessageOperation {
    CREATE(0),
```

```
    GET(1),
    UPDATE(2),
    DELETE(3);

    private int opNum;

    MessageOperation(int opNum) {
        this.opNum = opNum;
    }
}
```

Classe UserManagementMessage.java

```
package com.example.databaseservice.kafka.message;

import lombok.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class UserManagementMessage {
    private MessageOperation operation;
    private String username;
    private String password;
    private String[] devices;
}
```

Classe KafkaListeners.java

```
package com.example.databaseservice.kafka;

import com.example.databaseservice.kafka.message.UserManagementMessage;
import com.example.databaseservice.kafka.service.MessageService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Component;

@Component
public class KafkaListeners {
```

```
    private final MessageService service;

    private final Logger logger = LoggerFactory.getLogger(KafkaListeners.class);

    public KafkaListeners(MessageService service) {
        this.service = service;
    }

    @KafkaListener(
            topics = "${consumer.topic}",
            groupId = "amadeus",
            containerFactory = "userManagementConsumerFactory")
    @SendTo
    public String userManagementListener(UserManagementMessage message) {
        return service.receiveMessage(message);
    }
}
```

Classe UserRepository.java

```
package com.example.databaseservice.repository;

import com.example.databaseservice.entity.User;
import java.util.List;
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long> {

    List<User> findByUsername(String username);

    User findById(long id);
}
```

Classe DatabaseServiceApplication.java

```
package com.example.databaseservice;

import com.example.databaseservice.kafka.service.MessageService;
import com.example.databaseservice.repository.UserRepository;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.kafka.core.KafkaTemplate;

@SpringBootApplication
public class DatabaseServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(DatabaseServiceApplication.class, args);
    }

    @Bean
    public CommandLineRunner runner(
            UserRepository userRepository,
            MessageService service,
            KafkaTemplate<String, String> simpleProducer,
            @Value("${consumer.topic}") String topic) {
        return args -> {
        };
    }
}
```

Arquivo application.properties.java

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${DB_HOST:localhost}:${DB_PORT:3306}/${DB_SCHEMA:userdat
spring.datasource.username=${DB_USER:labsec}
spring.datasource.password=${DB_PASSWORD:labsec}
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

## KAFKA
spring.kafka.bootstrap-servers=${KAFKA_HOST:localhost:9092}
consumer.topic=db-req
producer.topic=db-resp
```

APÊNDICE C – CÓDIGO DEVICE CLIENT

Classe AvailableDevice.java

```
package br.ufsc.labsec.serverclient.config;

import java.util.ArrayList;
import java.util.List;
import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.stereotype.Component;

@Data
@Component
@AllArgsConstructor
public class AvailableDevice {
    private String uuid;
    private String label;
    private List<String> modules;

    public AvailableDevice() {
        this.modules = new ArrayList<>();
    }
}
```

Classe ClientConfiguration.java

```
package br.ufsc.labsec.serverclient.config;

import java.util.List;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Data
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
@NoArgsConstructor
@AllArgsConstructor
public class ClientConfiguration {
```

```
    private String uuid;
    private String username;
    private String password;
    private List<AvailableDevice> availableDevices;
}
```

Classe ConfigUtil.java

```
package br.ufsc.labsec.serverclient.config;

import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.net.URI;
import org.springframework.stereotype.Component;
import org.yaml.snakeyaml.DumperOptions;
import org.yaml.snakeyaml.Yaml;

@Component
public class ConfigUtil {

    public void writeToConfigFile(URI path, ClientConfiguration config) {
        Yaml yaml = buildYaml();
        try (FileWriter fileWriter = new FileWriter(path.getPath())) {
            yaml.dump(config, fileWriter);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public ClientConfiguration loadConfigFromFile(URI path) {
        Yaml yaml = buildYaml();
        try (InputStream inputStream = new FileInputStream(path.getPath())) {
            return yaml.load(inputStream);
        } catch (IOException e) {
            throw new RuntimeException("Invalid configuration file!",
                e.getCause());
        }
    }

    private Yaml buildYaml() {
        DumperOptions options = new DumperOptions();
```

```
        options.setIndent(2);
        options.setPrettyFlow(true);
        options.setDefaultFlowStyle(DumperOptions.FlowStyle.BLOCK);

        return new Yaml(options);
    }
}
```

Classe ProgramArguments.java

```
package br.ufsc.labsec.serverclient.config;

import java.util.Arrays;
import java.util.List;

public enum ProgramArguments {
    INSTALL(new String[] {"--install", "-i"});

    private final String[] cmd;

    ProgramArguments(String[] cmd) {
        this.cmd = cmd;
    }

    public String[] getCmd() {
        return this.cmd;
    }

    public List<String> getCmdList() {
        return List.of(this.cmd);
    }

    public static ProgramArguments matchArgument(String arg) {
        return Arrays.stream(ProgramArguments.values())
                .filter(cmdArg -> cmdArg.getCmdList().contains(arg))
                .findFirst()
                .orElseThrow(() -> new RuntimeException("No argument \"" + arg +
                    "\" found"));
    }
}
```

Classe KafkaConnectionParams.java

```
package br.ufsc.labsec.serverclient.kafka.config;

import lombok.Data;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.properties.ConfigurationProperties;
import
    org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Data
@Configuration
@ConfigurationProperties("kafka")
// @PropertySource(value = "file:/home/eyng/kafka.yml", factory =
    YamlPropertySourceFactory.class)
@EnableConfigurationProperties
public class KafkaConnectionParams {
    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServer;

    @Value("${consumer.group}")
    private String consumerGroup;
}
```

Classe KafkaConsumerConfig.java

```
package br.ufsc.labsec.serverclient.kafka.config;

import java.util.HashMap;
import java.util.Map;

import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.KafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.support.serializer.JsonDeserializer;

@Configuration
```

```
public class KafkaConsumerConfig {

    private final KafkaConnectionParams connectionConfig;

    public KafkaConsumerConfig(KafkaConnectionParams connectionConfig) {
        this.connectionConfig = connectionConfig;
    }

    public Map<String, Object> consumerConfig() {
        HashMap<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionConfig.getBootstrapServer());
        // Key and Value are Strings here
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            JsonDeserializer.class);
        return props;
    }

    @Bean
    public ConsumerFactory<String, TopicMessage> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(
                consumerConfig(), new StringDeserializer(), new
                    JsonDeserializer<>(TopicMessage.class));
    }

    @Bean
    public
        KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,
        TopicMessage>> factory(
            ConsumerFactory<String, TopicMessage> consumerFactory) {
        ConcurrentKafkaListenerContainerFactory<String, TopicMessage> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        return factory;
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, TopicMessage>
        deviceMessageConsumerFactory(
            ConsumerFactory<String, TopicMessage> consumerFactory) {
```

```
        ConcurrentKafkaListenerContainerFactory<String, TopicMessage> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        factory.setConcurrency(1);
        return factory;
    }
}
```

Classe KafkaProducerConfig.java

```
package br.ufsc.labsec.serverclient.kafka.config;

import java.util.HashMap;
import java.util.Map;

import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.support.serializer.JsonSerializer;

@Configuration
public class KafkaProducerConfig {

    private final KafkaConnectionParams connectionConfig;

    public KafkaProducerConfig(KafkaConnectionParams connectionConfig) {
        this.connectionConfig = connectionConfig;
    }

    public Map<String, Object> producerConfig() {
        HashMap<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionConfig.getBootstrapServer());
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            JsonSerializer.class);
        return props;
```

```
    }

    @Bean
    public ProducerFactory<String, TopicMessage> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfig());
    }

    @Bean
    public KafkaTemplate<String, TopicMessage>
        kafkaTemplate(ProducerFactory<String, TopicMessage> producerFactory) {
        return new KafkaTemplate<>(producerFactory);
    }
}
```

Classe KafkaReplyingTemplateConfig.java

```
package br.ufsc.labsec.serverclient.kafka.config;

import java.util.HashMap;
import java.util.Map;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.listener.ContainerProperties;
import org.springframework.kafka.listener.KafkaMessageListenerContainer;
import org.springframework.kafka.requestreply.ReplyingKafkaTemplate;

@Configuration
public class KafkaReplyingTemplateConfig {

    final KafkaConnectionParams connectionParams;

    public KafkaReplyingTemplateConfig(KafkaConnectionParams connectionParams) {
        this.connectionParams = connectionParams;
    }
```

```
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionParams.getBootstrapServer());
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
            connectionParams.getConsumerGroup());

        return props;
    }

    @Bean
    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            connectionParams.getBootstrapServer());
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        return props;
    }

    @Bean
    public ProducerFactory<String, String> requestProducerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    @Bean
    public ConsumerFactory<String, String> replyConsumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs(), new
            StringDeserializer(), new StringDeserializer());
    }

    @Bean
    public KafkaMessageListenerContainer<String, String>
        replyListenerContainer() {
        ContainerProperties containerProperties = new
            ContainerProperties("db-resp");
```

```
        return new KafkaMessageListenerContainer<>(replyConsumerFactory(),
            containerProperties);
    }

    @Bean
    public ReplyingKafkaTemplate<String, String, String> replyKafkaTemplate(
            ProducerFactory<String, String> pf,
                KafkaMessageListenerContainer<String, String> lc) {
        return new ReplyingKafkaTemplate<>(pf, lc);
    }
}
```

Classe KafkaTopicConfig.java

```
package br.ufsc.labsec.serverclient.kafka.config;

import org.apache.kafka.clients.admin.NewTopic;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.TopicBuilder;

@Configuration
public class KafkaTopicConfig {

    /*
    FIXME: topics should not be created at runtime.
     */
    @Value("${consumer.topic}")
    private String consumerTopicName;

    @Value("${producer.topic}")
    private String producerTopicName;

    public KafkaTopicConfig(
            @Value("${consumer.topic}") String consumerTopicName,
            @Value("${producer.topic}") String producerTopicName) {
        this.consumerTopicName = consumerTopicName;
        this.producerTopicName = producerTopicName;
    }

    @Bean("${producer.topic}")
    public NewTopic producerTopic() {
```

```
        return TopicBuilder.name(producerTopicName).build();
    }

    @Bean("${consumer.topic")
    public NewTopic consumerTopic() {
        return TopicBuilder.name(consumerTopicName).build();
    }
}
```

Classe YamlPropertySourceFactory.java

```
package br.ufsc.labsec.serverclient.kafka.config;

import java.util.Objects;
import java.util.Properties;
import org.springframework.beans.factory.config.YamlPropertiesFactoryBean;
import org.springframework.core.env.PropertiesPropertySource;
import org.springframework.core.env.PropertySource;
import org.springframework.core.io.support.EncodedResource;
import org.springframework.core.io.support.PropertySourceFactory;

public class YamlPropertySourceFactory implements PropertySourceFactory {
    @Override
    public PropertySource<?> createPropertySource(String name, EncodedResource
        resource) {
        YamlPropertiesFactoryBean factory = new YamlPropertiesFactoryBean();
        factory.setResources(resource.getResource());

        Properties properties = factory.getObject();

        return new PropertiesPropertySource(
                Objects.requireNonNull(resource.getResource().getFilename()),
                    properties);
    }
}
```

Classe PublishException.java

```
package br.ufsc.labsec.serverclient.kafka.exception;

public class PublishException extends RuntimeException {
    private static final String FAILED_TO_PUBLISH_TOPIC_OFFSET = "Failed to
        publish message to topic %s with offset %d";
```

```
    private static final String FAILED_TO_PUBLISH_TOPIC = "Failed to publish
        message \"%s\" to topic %s";
    private static final String FAILED_TO_PUBLISH_MSG = "Failed to publish
        message \"%s\"";

    public PublishException(String message) {
        super(String.format(FAILED_TO_PUBLISH_MSG, message));
    }

    public PublishException(String message, String topic) {
        super(String.format(FAILED_TO_PUBLISH_TOPIC, message, topic));
    }

    public PublishException(String topic, long offset) {
        super(String.format(FAILED_TO_PUBLISH_TOPIC_OFFSET, topic, offset));
    }

    public PublishException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Classe SyncException.java

```
package br.ufsc.labsec.serverclient.kafka.exception;

public class SyncException extends Exception {
    public SyncException(String message, Throwable throwable) {
        super(message, throwable);
    }

    public SyncException(String message) {
        super(message);
    }
}
```

Classe KafkaListeners.java

```
package br.ufsc.labsec.serverclient.kafka;

import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import br.ufsc.labsec.serverclient.service.DeviceMessageProcessor;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
public class KafkaListeners {

    private final ClientConfiguration clientConfiguration;
    private final DeviceMessageProcessor<TopicMessage> processingService;

    private final Logger logger = LoggerFactory.getLogger(KafkaListeners.class);

    public KafkaListeners(
            @Value("${consumer.topic}") String consumerTopic,
            @Value("${producer.topic}") String producerTopic,
            ClientConfiguration clientConfiguration,
            DeviceMessageProcessor<TopicMessage> processingService) {
        this.clientConfiguration = clientConfiguration;
        this.processingService = processingService;
    }

    @KafkaListener(
            topics = "${consumer.topic}",
            groupId = "${consumer.group}",
            containerFactory = "deviceMessageConsumerFactory")
    public void listenerListener(TopicMessage message) {
        var metadata = message.getMetaData();
        if (metadata.getReceiver().equals(clientConfiguration.getUuid())) {
            processingService.processMessage(message);
        }
    }
}
```

Classe DatabaseMessageFactory.java

```
package br.ufsc.labsec.serverclient.message;

import br.ufsc.labsec.serverclient.config.ClientConfiguration;

public interface DatabaseMessageFactory<T> {
    T createUserMessage(String username, String password);
```

```
    T deleteUserMessage(ClientConfiguration configuration);

    T getUserMessage(String username, String password);

    T updateUserMessage(ClientConfiguration configuration);
}
```

Classe KafkaDatabaseMessageFactory.java

```
package br.ufsc.labsec.serverclient.message;

import br.ufsc.labsec.serverclient.config.AvailableDevice;
import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import com.google.gson.Gson;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;
import org.springframework.stereotype.Component;

@Component
public class KafkaDatabaseMessageFactory implements
    DatabaseMessageFactory<String> {

    private Gson gson;

    public KafkaDatabaseMessageFactory(Gson gson) {
        this.gson = gson;
    }

    @Override
    public String createUserMessage(String username, String password) {
        var encodedPassword = hashPassword(password);
        var messageMap = basicMessage(username, encodedPassword);
        messageMap.put("operation", 0);
        return gson.toJson(messageMap);
    }

    @Override
```

```
    public String deleteUserMessage(ClientConfiguration configuration) {
        var messageMap = basicMessage(configuration.getUsername(),
            configuration.getPassword());
        messageMap.put("operation", 3);
        return gson.toJson(messageMap);
    }

    @Override
    public String getUserMessage(String username, String password) {
        var encodedPassword = hashPassword(password);
        var messageMap = basicMessage(username, encodedPassword);
        messageMap.put("operation", 1);
        return gson.toJson(messageMap);
    }

    @Override
    public String updateUserMessage(ClientConfiguration configuration) {
        var encodedPassword = hashPassword(configuration.getPassword());
        var messageMap = basicMessage(configuration.getUsername(),
            encodedPassword);
        var devices = configuration.getAvailableDevices().stream()
                .map(device -> device.getUuid() + ";" + device.getLabel())
                .collect(Collectors.toList());
        String[] devicesArray = new String[devices.size()];
        devicesArray = devices.toArray(devicesArray);

        messageMap.put("operation", 2);
        messageMap.put("devices", devicesArray);
        return gson.toJson(messageMap);
    }

    private Map<String, Object> basicMessage(String username, String password) {
        var basicMessage = new HashMap<String, Object>();
        basicMessage.put("username", username);
        basicMessage.put("password", password);
        return basicMessage;
    }

    private String hashPassword(String password) {
        try {
            var digester = MessageDigest.getInstance("SHA-256");
            var hashedPassword = digester.digest(password.getBytes());
```

```
            return Base64.getEncoder().encodeToString(hashedPassword);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Classe DatabaseException.java

```
package br.ufsc.labsec.serverclient.service.exception;

public class DatabaseException extends Exception {

    public DatabaseException(String message, Throwable cause) {
        super(message, cause);
    }

    public DatabaseException(String message) {
        super(message);
    }
}
```

Classe DatabaseServiceInterface.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import br.ufsc.labsec.serverclient.service.exception.DatabaseException;

public interface DatabaseServiceInterface {

    void createUser(String username, String password) throws DatabaseException;

    void updateUser(ClientConfiguration client) throws DatabaseException;

    void deleteUser(ClientConfiguration client) throws DatabaseException;

    ClientConfiguration getUser(String username, String password) throws
        DatabaseException;
}
```

Classe DatabaseException.java

```
package br.ufsc.labsec.serverclient.service;
```

```
public interface DeviceMessageProcessor<T> {
    void processMessage(T message);
}
```

Classe StartupService.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import br.ufsc.labsec.serverclient.config.ConfigUtil;
import br.ufsc.labsec.serverclient.config.ProgramArguments;
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.Arrays;
import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;
import org.springframework.stereotype.Service;

@Service
public class StartupService {

    //    private final String DEFAULT_CONFIG_LOCATION = "";
    private final String DEFAULT_CONFIG_FILE = ".client-conf.yml";
    private final ConfigUtil configUtil;
    private final ClientConfiguration clientConfiguration;

    public StartupService(ConfigUtil configUtil, ClientConfiguration
        clientConfiguration) {
        this.configUtil = configUtil;
        this.clientConfiguration = clientConfiguration;
    }

    public void start(String[] args) {
        if (args.length != 0) {
            parseArguments(args);
        }

        loadConfiguration();
    }
```

```
private void parseArguments(String[] args) {
    List<ProgramArguments> argumentsList =
            Arrays.stream(args).map(ProgramArguments::matchArgument).collect(Collectors.

    for (ProgramArguments arg : argumentsList) {
        resolveArgument(arg);
    }
}

private void resolveArgument(ProgramArguments argument) {
    switch (argument) {
        case INSTALL:
            install();
            break;
        default:
            throw new RuntimeException();
    }
}

private void loadConfiguration() {
    URI configFileUri = getConfigFileURI();

    var config = configUtil.loadConfigFromFile(configFileUri);

    clientConfiguration.setUuid(config.getUuid());
}

public void install() {
    URI configFileURI = getConfigFileURI();

    ClientConfiguration newConfig = new ClientConfiguration();
    newConfig.setUuid(UUID.randomUUID().toString());

    configUtil.writeToConfigFile(configFileURI, newConfig);
}

private URI getConfigFileURI() {
    URI configURI;
    try {
        String configFilePath = new File(StartupService.class
                        .getProtectionDomain()
                        .getCodeSource()
```

```
                        .getLocation()
                        .toURI())
                .getParentFile()
                .getPath();
        configURI = formatFilePathWindows(configFilePath);
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }
    return configURI;
}

private URI formatFilePathWindows(String path) throws URISyntaxException {
    final String filePrepend = "file:/";
    String filePath = path + "/" + DEFAULT_CONFIG_FILE;
    filePath = filePath.replace("\\", "/");
    filePath = filePrepend + filePath;
    return new URI(filePath);
}
}
```

Classe SyncDatabaseMessageService.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.serverclient.config.AvailableDevice;
import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import br.ufsc.labsec.serverclient.message.DatabaseMessageFactory;
import br.ufsc.labsec.serverclient.service.exception.DatabaseException;
import java.time.Duration;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.concurrent.ExecutionException;
import java.util.stream.Collectors;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.header.internals.RecordHeader;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.requestreply.ReplyingKafkaTemplate;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.stereotype.Service;
```

```
import org.springframework.util.concurrent.ListenableFutureCallback;

@Service
public class SyncDatabaseMessageService implements DatabaseServiceInterface {

    private final ReplyingKafkaTemplate<String, String, String>
        synchronousProducer;
    private final DatabaseMessageFactory<String> databaseMessageFactory;
    private final ClientConfiguration clientConfiguration;
    private final String replyTopic;
    private final String producerTopic;

    private final Duration TIMEOUT_DURATION = Duration.ofSeconds(8);
    private final Logger logger =
        LoggerFactory.getLogger(SyncDatabaseMessageService.class);

    @Autowired
    public SyncDatabaseMessageService(
            ReplyingKafkaTemplate<String, String, String> synchronousProducer,
            @Value("${database.consumer.topic}") String replyTopic,
            @Value("${database.producer.topic}") String producerTopic,
            DatabaseMessageFactory<String> databaseMessageFactory,
            ClientConfiguration clientConfiguration) {
        this.synchronousProducer = synchronousProducer;
        this.replyTopic = replyTopic;
        this.producerTopic = producerTopic;
        this.databaseMessageFactory = databaseMessageFactory;
        this.clientConfiguration = clientConfiguration;
    }

    @Override
    public void createUser(String username, String password) throws
        DatabaseException {
        synchronousProducer.setDefaultReplyTimeout(TIMEOUT_DURATION);
        String message = databaseMessageFactory.createUserMessage(username,
            password);
        createRecordAndSend(message); // Ignore result
    }

    @Override
    public void updateUser(ClientConfiguration updateRequest) throws
        DatabaseException {
```

```
        synchronousProducer.setDefaultReplyTimeout(TIMEOUT_DURATION);
        String message = databaseMessageFactory.updateUserMessage(updateRequest);
        createRecordAndSend(message); // Ignore result
        clientConfiguration.setUsername(updateRequest.getUsername());
        clientConfiguration.setPassword(updateRequest.getPassword());
        clientConfiguration.setAvailableDevices(updateRequest.getAvailableDevices());
    }

    @Override
    public void deleteUser(ClientConfiguration client) throws DatabaseException
        {
        synchronousProducer.setDefaultReplyTimeout(TIMEOUT_DURATION);
        String message = databaseMessageFactory.deleteUserMessage(client);
        createRecordAndSend(message); // Ignore result
    }

    @Override
    public ClientConfiguration getUser(String username, String password) throws
        DatabaseException {
        synchronousProducer.setDefaultReplyTimeout(TIMEOUT_DURATION);
        String message = databaseMessageFactory.getUserMessage(username,
            password);
        var requestRecord = createRecord(message);
        var future = synchronousProducer.sendAndReceive(requestRecord);
        future.addCallback(getCallback());
        try {
            var responseRecord = future.get();
            if (!responseRecord.value().startsWith("[")) { // bery ugly
                throw new DatabaseException(responseRecord.value());
            }
            clientConfiguration.setUsername(username);
            clientConfiguration.setPassword(password);
            updateClientConfiguration(clientConfiguration,
                responseRecord.value());
        } catch (ExecutionException | InterruptedException e) {
            throw new DatabaseException(e.getMessage(), e.getCause());
        }
        return clientConfiguration;
    }

    private void updateClientConfiguration(ClientConfiguration config, String
        response) {
```

```
    response = response.replace("[", "").replace("]", "");
    String[] devices = response.split(",");
    if (devices.length == 0 || devices[0].isBlank()) {
        config.setAvailableDevices(new ArrayList<>());
        return;
    }
    var deviceList = Arrays.stream(devices)
            .map(deviceUID -> new AvailableDevice(getDeviceUid(deviceUID),
                getDeviceLabel(deviceUID), new ArrayList<>()))
            .collect(Collectors.toList());
    config.setAvailableDevices(deviceList);
}

private String getDeviceUid(String deviceString) {
    var strArray = deviceString.split(";");
    return strArray[0];
}

private String getDeviceLabel(String deviceString) {
    var strArray = deviceString.split(";");
    return strArray[1];
}

private String createRecordAndSend(String message) throws DatabaseException
    {
    var requestRecord = createRecord(message);
    var future = synchronousProducer.sendAndReceive(requestRecord);
    future.addCallback(getCallback());

    try {
        var responseRecord = future.get();
        String response = responseRecord.value();
        if (!response.isBlank()) {
            throw new DatabaseException("Database error: " + response);
        }

        return response;

    } catch (InterruptedException | ExecutionException e) {
        throw new DatabaseException(e.getMessage(), e.getCause());
    }
}
```

```
    private ProducerRecord<String, String> createRecord(String message) {
        var record = new ProducerRecord<String, String>(producerTopic, message);
        record.headers().add(new RecordHeader(KafkaHeaders.REPLY_TOPIC,
            replyTopic.getBytes()));
        return record;
    }

    private ListenableFutureCallback<ConsumerRecord<String, String>>
        getCallback() {
        return new ListenableFutureCallback<>() {
            @Override
            public void onSuccess(ConsumerRecord<String, String> result) {
                // do nothing
            }

            @Override
            public void onFailure(Throwable ex) {
                logger.error("FAILED TO RECEIVE MESSAGE: " + ex.getMessage());
            }
        };
    }
}
```

Classe SyncDeviceMessageService.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.deviceserver.api.DeviceMessage;
import br.ufsc.labsec.deviceserver.api.messaging.MessageMetadata;
import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import br.ufsc.labsec.serverclient.kafka.exception.SyncException;
import br.ufsc.labsec.serverclient.util.MessageUtil;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonArray;
import com.google.gson.JsonObject;
import org.apache.tomcat.util.codec.binary.Base64;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
```

```
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service
public class SyncDeviceMessageService {

    private final SynchronizingBean<TopicMessage> synchronizingBean;
    private final ClientConfiguration clientConfiguration;
    private final KafkaTemplate<String, TopicMessage> producer;
    private final String producerTopic;

    private final Logger logger =
        LoggerFactory.getLogger(SyncDeviceMessageService.class);

    public SyncDeviceMessageService(
            SynchronizingBean<TopicMessage> synchronizingBean,
            ClientConfiguration clientConfiguration,
            KafkaTemplate<String, TopicMessage> producer,
            @Value("${producer.topic}") String producerTopic) {
        this.synchronizingBean = synchronizingBean;
        this.clientConfiguration = clientConfiguration;
        this.producer = producer;
        this.producerTopic = producerTopic;
    }

    public DeviceMessage sendMessageToDevice(String deviceUID, String module,
        DeviceMessage deviceMessage) {
        MessageMetadata metadata = new
            MessageMetadata(clientConfiguration.getUuid(), deviceUID, module);
        TopicMessage topicMessage = new TopicMessage();
        topicMessage.setMetaData(metadata);
        topicMessage.setMessage(deviceMessage);

        producer.send(producerTopic, topicMessage);

        TopicMessage response = null;
        try {
            response = synchronizingBean.getResult();
        } catch (SyncException e) {
            throw new RuntimeException(e);
```

```
        }

        return response.getMessage();
    }

    public List<String> sendListModulesMessage(String deviceUID) throws
        SyncException {
        List<String> modulesList;
        var message = createTopicMessage(deviceUID, "");
        message.setMessage(new DeviceMessage("", ""));

        producer.send(producerTopic, message);

        TopicMessage response = synchronizingBean.getResult();

        modulesList =
            MessageUtil.listFromString(response.getMessage().getMessage());

        if (modulesList.isEmpty()) {
            logger.error("No modules found for device: " + deviceUID);
            logger.debug("No modules found for device: " + deviceUID);
        }
        return modulesList;
    }

    public List<String> sendListCertsMessage(String deviceID, String module)
        throws SyncException {
        List<String> certificates;
        TopicMessage message = createTopicMessage(deviceID, module);

        DeviceMessage deviceMessage = new DeviceMessage("listCertificates", "");
        message.setMessage(deviceMessage);

        producer.send(producerTopic, message);

        TopicMessage response = synchronizingBean.getResult();

        Gson gson = new Gson();
        JsonObject jsonObject =
            gson.fromJson(response.getMessage().getMessage(), JsonObject.class);
        String certAliases = jsonObject.get("certificateList").getAsString();
        certificates = MessageUtil.listFromString(certAliases);
```

```
        if (certificates.isEmpty()) {
            logger.error("No certificates found for device: " + deviceID + "
                with module " + module);
            logger.debug("No certificates found for device: " + deviceID + "
                with module " + module);
        }

        return certificates;
    }

    public String sendSignMessage(String deviceID, String module, byte[] data,
        String pin, String certificateAlias) throws SyncException {
        TopicMessage message = createTopicMessage(deviceID, module);

        var gsonBuilder = new GsonBuilder();
        var gson = gsonBuilder.disableHtmlEscaping().create();
        var signRequest = new JsonObject();
        signRequest.addProperty("alias", certificateAlias);
        signRequest.addProperty("signatureAlgorithm", "SHA256WithRSA");
        signRequest.addProperty("data", Base64.encodeBase64String(data));
        signRequest.addProperty("pin", pin);

        DeviceMessage deviceMessage = new DeviceMessage("sign",
            gson.toJson(signRequest));
        message.setMessage(deviceMessage);

        producer.send(producerTopic, message);

        TopicMessage response = synchronizingBean.getResult();

        if (response.getMessage().getMethod().equals("error")) { // ugly
            throw new SyncException(response.getMessage().getMessage());
        }

        JsonObject jsonObject = new
            Gson().fromJson(response.getMessage().getMessage(),
            JsonObject.class);

        String signature = jsonObject.get("signatureBase64").getAsString();

//        System.out.println(signature);
```

```
        return signature;
    }

    private TopicMessage createTopicMessage(String deviceUID, String module) {
        MessageMetadata metadata = new
            MessageMetadata(clientConfiguration.getUuid(), deviceUID, module);
        TopicMessage topicMessage = new TopicMessage();
        topicMessage.setMetaData(metadata);
        return topicMessage;
    }
}
```

Classe SynchronizingBean.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import br.ufsc.labsec.serverclient.kafka.exception.SyncException;

public interface SynchronizingBean<T> {

    T getResult() throws SyncException;

    void setResult(TopicMessage topicMessage);

    boolean hasWaiting();
}
```

Classe SynchronizingTopicMessage.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import br.ufsc.labsec.serverclient.kafka.exception.PublishException;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

import br.ufsc.labsec.serverclient.kafka.exception.SyncException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
```

```
import org.springframework.stereotype.Component;

@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class SynchronizingTopicMessage implements
    SynchronizingBean<TopicMessage> {
    private final Logger logger =
        LoggerFactory.getLogger(SynchronizingTopicMessage.class);

    private final Semaphore semaphore;
    private TopicMessage result;

    public SynchronizingTopicMessage() {
        this.semaphore = new Semaphore(1);
    }

    @Override
    public TopicMessage getResult() throws SyncException {
        try {
            boolean acquiredQueueSpot = semaphore.tryAcquire(8L,
                TimeUnit.SECONDS);
            if (!acquiredQueueSpot) {
                logger.error("Failed trying to wait for result");
                logger.debug("Someone already waiting?");
                throw new RuntimeException("Concurrency error!");
            }
            boolean hasMessage = semaphore.tryAcquire(8L, TimeUnit.SECONDS);
            if (!hasMessage) {
                logger.debug("Timed out while trying to get result from topic!");
                throw new SyncException("Failed to send message, connection timed
                    out");
            }
            return result;
        } catch (InterruptedException e) {
            throw new RuntimeException(e.getMessage(), e.getCause());
        } finally {
            semaphore.release();
        }
    }

    @Override
    public void setResult(TopicMessage topicMessage) {
```

```
        result = topicMessage;
        semaphore.release();
    }

    @Override
    public boolean hasWaiting() {
        return semaphore.availablePermits() == 0;
    }
}
```

Classe SyncMessageProcessor.java

```
package br.ufsc.labsec.serverclient.service;

import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import org.springframework.stereotype.Service;

@Service
public class SyncMessageProcessor implements
    DeviceMessageProcessor<TopicMessage> {

    private SynchronizingBean<TopicMessage> synchronizingBean;

    public SyncMessageProcessor(SynchronizingBean<TopicMessage>
        synchronizingBean) {
        this.synchronizingBean = synchronizingBean;
    }

    @Override
    public void processMessage(TopicMessage message) {
        if (synchronizingBean.hasWaiting()) {
            synchronizingBean.setResult(message);
        }
    }
}
```

Classe MessageUtil.java

```
package br.ufsc.labsec.serverclient.util;

import java.util.ArrayList;
import java.util.List;
```

```
public class MessageUtil {

    public static List<String> listFromString(String listStr) {
        List<String> ret = new ArrayList<>();

        String temp = listStr.replace("[", "").replace("]", "");
        if (!temp.isBlank()) {
            String[] strArray = temp.split(",");
            ret = List.of(strArray);
        }
        return ret;
    }
}
```

Classe DeviceBrokerApplication.java

```
package br.ufsc.labsec.serverclient;

import br.ufsc.labsec.serverclient.config.ClientConfiguration;
import br.ufsc.labsec.serverclient.service.StartupService;
import br.ufsc.labsec.serverclient.service.SyncDatabaseMessageService;
import br.ufsc.labsec.serverclient.service.SyncDeviceMessageService;
import br.ufsc.labsec.serverclient.view.LoginGUI;
import javax.swing.*;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.annotation.Bean;

@SpringBootApplication()
public class DeviceBrokerApplication {

    public static void main(String[] args)
            throws UnsupportedLookAndFeelException, ClassNotFoundException,
                InstantiationException,
                    IllegalAccessException {
        //       SpringApplication.run(DeviceBrokerApplication.class, args);
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new SpringApplicationBuilder(DeviceBrokerApplication.class)
                .headless(false)
                .web(WebApplicationType.NONE)
                .run(args);
```

```
    }

    @Bean
    public CommandLineRunner runner(
            SyncDatabaseMessageService syncDatabaseMessageService,
            ClientConfiguration clientConfiguration,
            StartupService startupService,
            SyncDeviceMessageService deviceMessageService,
            LoginGUI gui) {
        return (args -> {
            startupService.start(args);
            gui.start("Device Client");
        });
    }
}
```

application.properties

```
spring.kafka.bootstrap-servers=${KAFKA_HOST:localhost}:${KAFKA_PORT:9092}

## Topic configuration
producer.topic=client-device
consumer.topic=device-client
consumer.group=${CONSUMER_GROUP:foo123}

database.producer.topic=db-req
database.consumer.topic=db-resp

spring.kafka.security.protocol=SSL
spring.kafka.ssl.trust-store-location=classpath:kafka.truststore.jks
spring.kafka.ssl.trust-store-password=112233
spring.kafka.ssl.key-store-location=classpath:wildcard.kafka.keystore.jks
spring.kafka.ssl.key-store-password=112233
```

APÊNDICE D – CÓDIGO DEVICE SERVER

Classe ConfigurationService.java

```
package br.com.trapp.deviceserver.controller;


import br.com.trapp.deviceserver.model.DeviceConfiguration;
import java.io.*;
import java.net.URI;
import org.springframework.stereotype.Service;
import org.yaml.snakeyaml.DumperOptions;
import org.yaml.snakeyaml.Yaml;

@Service
public class ConfigurationService {

    public void writeToConfigFile(URI path, DeviceConfiguration config) {
        Yaml yaml = buildYaml();
        try (FileWriter fileWriter = new FileWriter(path.getPath())) {
            yaml.dump(config, fileWriter);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public DeviceConfiguration loadConfigFromFile(URI path) {
        Yaml yaml = buildYaml();
        try (InputStream inputStream = new FileInputStream(path.getPath())) {
            return yaml.load(inputStream);
        } catch (IOException e) {
            throw new RuntimeException("Invalid configuration file!",
                e.getCause());
        }
    }

    private Yaml buildYaml() {
        DumperOptions options = new DumperOptions();
        options.setIndent(2);
        options.setPrettyFlow(true);
        options.setDefaultFlowStyle(DumperOptions.FlowStyle.BLOCK);

        return new Yaml(options);
```

```
    }
}
```

Classe DeviceService.java

```
package br.com.trapp.deviceserver.controller;


import br.com.trapp.deviceserver.model.exception.ModuleManagerException;
import br.com.trapp.deviceserver.model.module.ModuleInfo;
import br.com.trapp.deviceserver.model.module.ModuleManagerImpl;
import br.ufsc.labsec.deviceserver.api.DeviceException;
import br.ufsc.labsec.deviceserver.api.DeviceMessage;
import br.ufsc.labsec.deviceserver.api.DeviceModule;
import java.lang.reflect.InvocationTargetException;
import java.util.Set;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DeviceService {

    private Logger logger = LoggerFactory.getLogger(DeviceService.class);

    private ModuleManagerImpl moduleManagerImpl;

    @Autowired
    public DeviceService(ModuleManagerImpl moduleManagerImpl) {
        this.moduleManagerImpl = moduleManagerImpl;
    }

    public Set<ModuleInfo> getAvailableModules() {
        return moduleManagerImpl.getAvailableModules();
    }

    public DeviceMessage sendMessage(DeviceMessage message, String moduleId)
            throws ModuleManagerException {
        ModuleInfo targetModule;
        try {
            var moduleOptional =
                    moduleManagerImpl.getAvailableModules().stream()
```

```
                        .filter(moduleInfo ->
                            moduleInfo.getIdendifier().equals(moduleId))
                        .findFirst();

            targetModule =
                    moduleOptional.orElseThrow(
                            () ->
                                    new ModuleManagerException(
                                            "No module found for id: " + moduleId));

            var module = moduleManagerImpl.getModules().get(targetModule);

            DeviceModule moduleInstance = ((DeviceModule)
                module.getConstructor().newInstance());

            var returnMessage = moduleInstance.incomingMessage(message);

            logger.debug(
                    String.format(
                            "Sent message %s to module %s", message.getMessage(),
                            targetModule));

            return returnMessage;
        } catch (InvocationTargetException
                | InstantiationException
                | IllegalAccessException
                | NoSuchMethodException
                | DeviceException e) {
            throw new ModuleManagerException(e.getMessage(), e.getCause());
        }
    }
}
```

Classe MessageService.java

```
package br.com.trapp.deviceserver.controller;


import br.com.trapp.deviceserver.model.exception.ModuleManagerException;
import br.com.trapp.deviceserver.model.module.ModuleInfo;
import br.ufsc.labsec.deviceserver.api.DeviceMessage;
import java.util.List;
import java.util.Set;
```

```
import java.util.stream.Collectors;
import org.apache.commons.lang.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;

@Service
public class MessageService {

    private final DeviceService deviceService;

    private final Logger logger = LoggerFactory.getLogger(MessageService.class);

    public MessageService(DeviceService deviceService) {
        this.deviceService = deviceService;
    }

    public DeviceMessage resolveMessage(DeviceMessage message, String moduleId)
        {
        if (StringUtils.isBlank(moduleId)) {
            logger.info("Listing Modules");
            return getListModulesResponse();
        }

        try {
            return deviceService.sendMessage(message, moduleId);
        } catch (ModuleManagerException e) {
            logger.error("Module message returned with error: " +
                e.getMessage());
            e.printStackTrace();
            return new DeviceMessage("fail", "Operation failed! " +
                e.getMessage());
        }
    }

    public DeviceMessage getListModulesResponse() {
        DeviceMessage response = new DeviceMessage();

        Set<ModuleInfo> moduleInfoSet = deviceService.getAvailableModules();
        List<String> modules =
                moduleInfoSet.stream().map(ModuleInfo::getIdendifier).collect(Collectors.toList()
        response.setMessage(modules.toString());
```

```
        return response;
    }
}
```

Classe ProgramArguments.java

```
package br.com.trapp.deviceserver.controller;


import java.util.Arrays;
import java.util.List;

public enum ProgramArguments {
    INSTALL(new String[] {"--install", "-i"});

    private final String[] cmd;

    ProgramArguments(String[] cmd) {
        this.cmd = cmd;
    }

    public String[] getCmd() {
        return this.cmd;
    }

    public List<String> getCmdList() {
        return List.of(this.cmd);
    }

    public static ProgramArguments matchArgument(String arg) {
        return Arrays.stream(ProgramArguments.values())
                .filter(cmdArg -> cmdArg.getCmdList().contains(arg))
                .findFirst()
                .orElseThrow(() -> new RuntimeException("No argument \"" + arg +
                    "\" found"));
    }
}
```

Classe StartupService.java

```
package br.com.trapp.deviceserver.controller;
```

```
import br.com.trapp.deviceserver.model.DeviceConfiguration;
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.Arrays;
import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;
import org.springframework.stereotype.Service;

@Service
public class StartupService {

    //    private final String DEFAULT_CONFIG_LOCATION = "";
    private final String DEFAULT_CONFIG_FILE = ".device-conf.yml";
    private final ConfigurationService configurationService;
    private final DeviceConfiguration deviceConfiguration;

    public StartupService(
            ConfigurationService configurationService, DeviceConfiguration
                deviceConfiguration) {
        this.configurationService = configurationService;
        this.deviceConfiguration = deviceConfiguration;
    }

    public void start(String[] args) {
        if (args.length != 0) {
            parseArguments(args);
        }

        loadConfiguration();
    }

    private void parseArguments(String[] args) {
        List<ProgramArguments> argumentsList =
                Arrays.stream(args)
                        .map(ProgramArguments::matchArgument)
                        .collect(Collectors.toList());

        for (ProgramArguments arg : argumentsList) {
            resolveArgument(arg);
```

```
    }
}

private void resolveArgument(ProgramArguments argument) {
    switch (argument) {
        case INSTALL:
            install();
            break;
        default:
            throw new RuntimeException();
    }
}

private void loadConfiguration() {
    URI configFileUri = getConfigFileURI();

    var config = configurationService.loadConfigFromFile(configFileUri);

    deviceConfiguration.setUuid(config.getUuid());
}

public void test() {
    DeviceConfiguration deviceConfig = new
        DeviceConfiguration(UUID.randomUUID().toString());
    URI configFile = getConfigFileURI();
    configurationService.writeToConfigFile(configFile, deviceConfig);
}

public void install() {
    URI configFileURI = getConfigFileURI();

    DeviceConfiguration newConfig = new
        DeviceConfiguration(UUID.randomUUID().toString());

    configurationService.writeToConfigFile(configFileURI, newConfig);
}

private URI getConfigFileURI() {
    URI configURI;
    try {
        String configFilePath =
                new File(
```

```
                                StartupService.class
                                        .getProtectionDomain()
                                        .getCodeSource()
                                        .getLocation()
                                        .toURI())
                            .getParentFile()
                            .getPath();
            configURI = formatFilePathWindows(configFilePath);
        } catch (URISyntaxException e) {
            throw new RuntimeException(e);
        }
        return configURI;
    }

    private URI formatFilePathWindows(String path) throws URISyntaxException {
        final String filePrepend = "file:/";
        String filePath = path + "/" + DEFAULT_CONFIG_FILE;
        filePath = filePath.replace("\\", "/");
        filePath = filePrepend + filePath;
        return new URI(filePath);
    }
}
```

Classe KafkaConsumerConfig.java

```
package br.com.trapp.deviceserver.kafka.config;


import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import java.util.HashMap;
import java.util.Map;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.support.serializer.JsonDeserializer;

@Configuration
public class KafkaConsumerConfig {
```

```
    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServers;

    public Map<String, Object> consumerConfig() {
        HashMap<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        // Key and Value are Strings here
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            JsonDeserializer.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
        return props;
    }

    @Bean
    public ConsumerFactory<String, TopicMessage> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(
                consumerConfig(),
                new StringDeserializer(),
                new JsonDeserializer<>(TopicMessage.class));
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, TopicMessage>
            deviceMessageConsumerFactory(ConsumerFactory<String, TopicMessage>
                consumerFactory) {
        ConcurrentKafkaListenerContainerFactory<String, TopicMessage> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        return factory;
    }
}
```

Classe KafkaProducerConfig.java

```
package br.com.trapp.deviceserver.kafka.config;


import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import java.util.HashMap;
import java.util.Map;
```

```
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.support.serializer.JsonSerializer;

@Configuration
public class KafkaProducerConfig {

    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServers;

    public Map<String, Object> producerConfig() {
        HashMap<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        // Key and Value are Strings here
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            JsonSerializer.class);
        return props;
    }

    @Bean
    public ProducerFactory<String, TopicMessage> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfig());
    }

    @Bean
    public KafkaTemplate<String, TopicMessage> kafkaTemplate(
            ProducerFactory<String, TopicMessage> producerFactory) {
        return new KafkaTemplate<>(producerFactory);
    }
}
```

Classe KafkaTopicConfig.java

```
package br.com.trapp.deviceserver.kafka.config;
```

```
import org.apache.kafka.clients.admin.NewTopic;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.TopicBuilder;

@Configuration
public class KafkaTopicConfig {

    @Value("${consumer.topic}")
    private String receiverTopicName;

    @Value("${producer.topic}")
    private String senderTopicName;

    public KafkaTopicConfig(
            @Value("${consumer.topic}") String receiverTopicName,
            @Value("${producer.topic}") String senderTopicName) {
        this.receiverTopicName = receiverTopicName;
        this.senderTopicName = senderTopicName;
    }

    @Bean("${producer.topic}")
    public NewTopic senderTopic() {
        return TopicBuilder.name(senderTopicName).build();
    }

    @Bean("${consumer.topic}")
    public NewTopic receiverTopic() {
        return TopicBuilder.name(receiverTopicName).build();
    }
}
```

Classe MessageListener.java

```
package br.com.trapp.deviceserver.kafka.listener;


import br.com.trapp.deviceserver.controller.MessageService;
import br.com.trapp.deviceserver.model.DeviceConfiguration;
import br.ufsc.labsec.deviceserver.api.DeviceMessage;
import br.ufsc.labsec.deviceserver.api.messaging.MessageMetadata;
```

```
import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class MessageListener {

    private String producerTopic;
    private MessageService messageService;
    private DeviceConfiguration deviceConfiguration;
    private KafkaTemplate<String, TopicMessage> producer;

    private final Logger logger =
        LoggerFactory.getLogger(MessageListener.class);

    public MessageListener(
            @Value("${producer.topic}") String producerTopic,
            MessageService messageService,
            DeviceConfiguration deviceConfiguration,
            KafkaTemplate<String, TopicMessage> producer) {
        this.producerTopic = producerTopic;
        this.messageService = messageService;
        this.deviceConfiguration = deviceConfiguration;
        this.producer = producer;
    }

    @KafkaListener(
            topics = "${consumer.topic}",
            groupId = "device-server",
            containerFactory = "deviceMessageConsumerFactory")
    public void senderListener(TopicMessage message) {
        MessageMetadata metadata = message.getMetaData();

        if (metadata.getReceiver().equals(deviceConfiguration.getUuid())) {

            TopicMessage response = new TopicMessage();
            MessageMetadata responseMetadata = new MessageMetadata();
            responseMetadata.setSender(deviceConfiguration.getUuid());
```

```
            responseMetadata.setReceiver(metadata.getSender());
            response.setMetaData(responseMetadata);

            DeviceMessage moduleResponse =
                    messageService.resolveMessage(message.getMessage(),
                        metadata.getTargetModule());

            response.setMessage(moduleResponse);

            producer.send(producerTopic, response);
            logger.info("Responded to " + metadata.getSender());
        }
    }
}
```

Classe AuthorizationException.java

```
package br.com.trapp.deviceserver.model.exception;

public class AuthorizationException extends Exception {

    /** */
    private static final long serialVersionUID = 1L;

    public static final AuthorizationException NOT_LOADED =
            new AuthorizationException("Module not installed or loaded");

    public AuthorizationException(String msg, Throwable throwable) {
        super(msg, throwable);
    }

    public AuthorizationException(String msg) {
        super(msg);
    }
}
```

Classe ModuleManagerException.java

```
package br.com.trapp.deviceserver.model.exception;

public class ModuleManagerException extends Exception {

    /** */
```

```
    private static final long serialVersionUID = 1L;

    public static final ModuleManagerException NOT_A_MODULE =
            new ModuleManagerException("The given jar is not a valid module");

    public ModuleManagerException(String msg, Throwable throwable) {
        super(msg, throwable);
    }

    public ModuleManagerException(String msg) {
        super(msg);
    }
}
```

Classe JarVerifier.java

```
package br.com.trapp.deviceserver.model.module;


import java.io.File;
import java.io.IOException;
import java.security.cert.X509Certificate;

public interface JarVerifier {

    /**
     * Verifies the signature of the module .jar against the target certificate.
     *
     * @param moduleJar the module file object
     * @param targetCert the certificate containing the public key to verify
     * @return
     */
    Class<?> verifySignature(File moduleJar, X509Certificate targetCert) throws
        IOException;
}
```

Classe JarVerifierImpl.java

```
package br.com.trapp.deviceserver.model.module;


import br.ufsc.labsec.deviceserver.api.DeviceModule;
import java.io.File;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.net.URLClassLoader;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.Vector;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;
import java.util.jar.Manifest;
import org.springframework.stereotype.Component;

@Component
public class JarVerifierImpl implements JarVerifier {

    /**
     * Compare it to the expected X509Certificate. If everything went well and
         the certificates are
     * the same, no exception is thrown.
     */
    @Override
    public Class<?> verifySignature(File file, X509Certificate targetCert)
        throws IOException {

        ClassLoader cl =
               URLClassLoader.newInstance(
                      new URL[] {new URL("jar:file:" + file.getAbsolutePath() +
                          "!/")});
        JarFile jarFile = new JarFile(file);
        Class<?> portClass = null;
        // Sanity checking
        if (targetCert == null) {
            jarFile.close();
            throw new SecurityException("Provider certificate is invalid");
        }

        Vector<JarEntry> entriesVec = new Vector<JarEntry>();

        // Ensure the jar has a manifest file.
        Manifest man = jarFile.getManifest();
```

```
if (man == null) {
    jarFile.close();
    throw new SecurityException("The jar is not signed, missing manifest
        file");
}

// Ensure all the entries' signatures verify correctly
byte[] buffer = new byte[8192];
Enumeration<JarEntry> entries = jarFile.entries();

while (entries.hasMoreElements()) {
    JarEntry je = entries.nextElement();

    // Skip directories.
    if (je.isDirectory()) continue;
    entriesVec.addElement(je);
    InputStream is = jarFile.getInputStream(je);

    // Read in each jar entry. A security exception will
    // be thrown if a signature/digest check fails.
    while (is.read(buffer, 0, buffer.length) != -1) {
        // Don't care
    }
    is.close();
}

// Get the list of signer certificates
Enumeration<JarEntry> e = entriesVec.elements();

while (e.hasMoreElements()) {

    JarEntry je = e.nextElement();
    try {

        if (je.getName().endsWith(".class")) {
            // -6 because of .class
            String className = je.getName().substring(0,
                je.getName().length() - 6);
            className = className.replace('/', '.');
            Class<?> clazz;
            clazz = cl.loadClass(className);
```

```
                    if
                        (Arrays.asList(clazz.getInterfaces()).contains(DeviceModule.class))
                        {
                        portClass = clazz;
                    }
                }
            } catch (ClassNotFoundException e1) {
                // Ignore
            }
            // Every file must be signed except files in META-INF.
            Certificate[] certs = je.getCertificates();
            if ((certs == null) || (certs.length == 0)) {
                if (!je.getName().startsWith("META-INF")) {
                    jarFile.close();
                    throw new SecurityException("The jar " + "has unsigned " +
                        "class files.");
                }
            } else {
                // Check whether the file is signed by the expected
                // signer. The jar may be signed by multiple signers.
                // See if one of the signers is 'targetCert'.
                boolean signedAsExpected = false;

                for (int i = 0; i < certs.length; i++) {
                    if (certs[i].equals(targetCert)) {
                        // Stop since one trusted signer is found.
                        signedAsExpected = true;
                        break;
                    }
                }

                if (!signedAsExpected) {
                    jarFile.close();
                    throw new SecurityException(
                            "The jar " + "is not signed by a " + "trusted signer");
                }
            }
        }
        jarFile.close();
        return portClass;
    }
}
```

Classe ModuleID.java

```
package br.com.trapp.deviceserver.model.module;

public class ModuleID {

    private String name;
    private String identifier;

    public ModuleID() {
        // TODO Auto-generated constructor stub
    }

    public ModuleID(String name, String identifier) {
        this.name = name;
        this.identifier = identifier;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIdentifier() {
        return identifier;
    }

    public void setIdentifier(String identifier) {
        this.identifier = identifier;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((identifier == null) ? 0 :
            identifier.hashCode());
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
```

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        ModuleID other = (ModuleID) obj;
        if (identifier == null) {
            if (other.identifier != null) return false;
        } else if (!identifier.equals(other.identifier)) return false;
        if (name == null) {
            if (other.name != null) return false;
        } else if (!name.equals(other.name)) return false;
        return true;
    }
}
```

Classe ModuleInfo.java

```
package br.com.trapp.deviceserver.model.module;

public class ModuleInfo {

    private String name;
    private String idendifier;
    private String version;

    public ModuleInfo(String name, String idendifier, String version) {
        this.name = name;
        this.idendifier = idendifier;
        this.version = version;
    }

    public ModuleInfo() {}

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public String getIdendifier() {
        return idendifier;
    }

    public void setIdendifier(String idendifier) {
        this.idendifier = idendifier;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((idendifier == null) ? 0 :
            idendifier.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        ModuleInfo other = (ModuleInfo) obj;
        if (idendifier == null) {
            if (other.idendifier != null) return false;
        } else if (!idendifier.equals(other.idendifier)) return false;
        return true;
    }

    @Override
    public String toString() {
        String str = "[ModuleInfo:[name : %s, identifier %s, version: %s]]";
        return String.format(str, name, idendifier, version);
    }
```

```
}
```

Classe ModuleManager.java

```
package br.com.trapp.deviceserver.model.module;


import br.ufsc.labsec.deviceserver.api.DeviceModule;
import java.io.File;
import java.util.Set;

public interface ModuleManager {

    /**
     * Loads a module, returning the module name on success.
     *
     * @param moduleJar the module file object
     * @return the name of the module
     */
    String loadModule(File moduleJar);

    Set<ModuleInfo> getAvailableModules();

    ModuleInfo getModuleIdentifier(Class<?> module);

    DeviceModule getInstanceOfModule(String identifier);
}
```

Classe DeviceConfiguration.java

```
package br.com.trapp.deviceserver.model;


import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Data
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
```

```
@NoArgsConstructor
@AllArgsConstructor
public class DeviceConfiguration {
    private String uuid;
}
```

Classe PKIXUtils.java

```
package br.com.trapp.deviceserver.utils;


import br.com.trapp.deviceserver.model.TrustedCertificates;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;
import java.security.cert.CertPath;
import java.security.cert.CertPathValidator;
import java.security.cert.CertStore;
import java.security.cert.CertStoreParameters;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.CollectionCertStoreParameters;
import java.security.cert.PKIXParameters;
import java.security.cert.TrustAnchor;
import java.security.cert.X509CRL;
import java.security.cert.X509Certificate;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import org.bouncycastle.asn1.ASN1Primitive;
import org.bouncycastle.asn1.DERIA5String;
import org.bouncycastle.asn1.x509.CRLDistPoint;
import org.bouncycastle.asn1.x509.DistributionPoint;
import org.bouncycastle.asn1.x509.DistributionPointName;
import org.bouncycastle.asn1.x509.Extension;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.GeneralNames;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
```

```
import org.bouncycastle.jce.provider.X509CRLParser;
import org.bouncycastle.x509.extension.X509ExtensionUtil;

public class PKIXUtils {

    /**
     * Verify if the certificate is valid
     *
     * @param pemEncodedCert The PEM encoded certificate
     * @throws CertificateException if the certificate is not valid or no
         encoded correctly
     */
    public void verifyCertificate(String pemEncodedCert) throws
        CertificateException {
        try {
            ByteArrayInputStream byteArrIStream =
                    new ByteArrayInputStream(pemEncodedCert.getBytes());
            CertificateFactory cf = CertificateFactory.getInstance("X.509");
            this.verifyCertificate((X509Certificate)
                cf.generateCertificate(byteArrIStream));
        } catch (CertificateException e) {
            throw new CertificateException("The certificate is not correctly
                encoded", e);
        }
    }

    public void verifyCertificate(X509Certificate cert) throws
        CertificateException {
        try {

        } catch (Exception e) {
            throw new CertificateException("The certificate is incorrectly
                encoded", e);
        }
    }

    /**
     * Verifies the validity of the certificate
     *
     * @param certificate
     * @throws CertificateException if some error occurred during the
         validations, meaning that the
```

```
     *     certificate is invalid
     */
    @SuppressWarnings("unchecked")
    public void verifyCertificateValidity(InputStream certificatesInputStream)
            throws CertificateException {
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        Collection<? extends Certificate> certs =
            cf.generateCertificates(certificatesInputStream);
        X509Certificate trust = TrustedCertificates.getCaCert();
        List<URL> urls = this.getCRLURLs(trust);
        if (urls.isEmpty()) {
            throw new CertificateException("None CRL was found");
        }
        Collection<X509CRL> crls = new ArrayList<X509CRL>();
        CertPath cp = cf.generateCertPath(new ArrayList<>(certs));
        X509CRLParser parser = new X509CRLParser();
        int index = 0;
        int max = urls.size() - 1;
        try {
            parser.engineInit(urls.get(index).openStream());
            crls.addAll(parser.engineReadAll());
        } catch (Exception e) {
            if (index == max) throw new CertificateException("All CRLs are
                invalids", e);
        }
        this.verifyCertificate(trust, cp, crls);
    }

    /**
     * Method to get the URL of the CRL from the certificate
     *
     * @param cert the certificate where the CRLs are
     * @return an list o URL of CRLs
     * @throws CertificateException if is not possible to get the
         cRLDistributionPoints extension
     */
    private List<URL> getCRLURLs(X509Certificate cert) throws
        CertificateException {
        List<URL> crlStreames = new ArrayList<URL>();
        byte[] crlDistPointBytes =
            cert.getExtensionValue(Extension.cRLDistributionPoints.getId());
        try {
```

```
            ASN1Primitive ext =
                X509ExtensionUtil.fromExtensionValue(crlDistPointBytes);
            CRLDistPoint crlsDistPoint = CRLDistPoint.getInstance(ext);
            DistributionPoint[] distPoints =
                crlsDistPoint.getDistributionPoints();
            for (DistributionPoint distributionPoint : distPoints) {
                DistributionPointName distPointName =
                    distributionPoint.getDistributionPoint();
                GeneralNames generalNames =
                    GeneralNames.getInstance(distPointName.getName());
                GeneralName[] names = generalNames.getNames();
                for (GeneralName generalName : names) {
                    if (generalName.getTagNo() ==
                        GeneralName.uniformResourceIdentifier) {
                        URI uri = new URI(((DERIA5String)
                            generalName.getName()).getString());
                        URL urlFromURI = uri.toURL();
                        crlStreames.add(urlFromURI);
                    }
                }
            }
        } catch (IOException e) {
            throw new CertificateException("The CRLDistributionPoint extension
                is invalid", e);
        } catch (URISyntaxException e) {
            throw new CertificateException(
                    "The URI in CRLDistributionPoint extension is invalid", e);
        }
        return crlStreames;
    }

    /**
     * Verify the cert path
     *
     * @param trust the trusted anchor
     * @param cp the cert path to validate
     * @param crls the list of crls
     * @throws CertificateException if the certificate path is not valid
     */
    public void verifyCertificate(X509Certificate trust, CertPath cp,
        Collection<X509CRL> crls)
            throws CertificateException {
```

```
        try {
            TrustAnchor anchor = new TrustAnchor(trust, null);
            PKIXParameters params = new
                PKIXParameters(Collections.singleton(anchor));
            params.setRevocationEnabled(true);
            CertStoreParameters revoked = new
                CollectionCertStoreParameters(crls);
            params.addCertStore(CertStore.getInstance("Collection", revoked));
            CertPathValidator cpv =
                    CertPathValidator.getInstance("PKIX",
                        BouncyCastleProvider.PROVIDER_NAME);
            cpv.validate(cp, params);
        } catch (Exception e) {
            throw new CertificateException("The certificate is not valid", e);
        }
    }
}
```

Classe MainRunner.java

```
package br.com.trapp.deviceserver;


import br.com.trapp.deviceserver.model.module.ModuleManager;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FilenameFilter;
import java.net.URISyntaxException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.Security;
import java.util.Optional;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class MainRunner {

    Logger logger = LoggerFactory.getLogger(MainRunner.class);

    private static final String MODULES_FOLDER = File.separator + "modules" +
```

```
    File.separator;

private final ModuleManager moduleManager;

public MainRunner(ModuleManager moduleManager) {
    this.moduleManager = moduleManager;
}

public void loadModules() throws FileNotFoundException {
    File modulesDirectory = getModulesDirectory();
    Security.addProvider(new BouncyCastleProvider());
    String[] modules = listModules(modulesDirectory);
    logger.info("Loading modules...");
    for (String moduleFilename : modules) {
        logger.debug("Loading module: " + moduleFilename + ".....");

        File module =
                new File(modulesDirectory.getAbsolutePath() + File.separator
                    + moduleFilename);
        moduleManager.loadModule(module);

        logger.info("Module loaded: " + moduleFilename);
    }
}

private File getModulesDirectory() throws FileNotFoundException {
    String modulesDirectoryPath = getModulesDirectoryPath();
    File modulesDirectory = new File(modulesDirectoryPath);
    if (!modulesDirectory.exists()) {
        logger.error("Modules folder does not exist.");
        logger.debug("Failed to find modules folder on path: " +
            modulesDirectoryPath);
        throw new FileNotFoundException("Modules directory not found");
    }
    return modulesDirectory;
}

private String getModulesDirectoryPath() {
    try {
        Path deviceServerJarPath =
                Paths.get(
                        MainRunner.class
```

```
                                .getProtectionDomain()
                                .getCodeSource()
                                .getLocation()
                                .toURI());
            String deviceServerJarPathStr = deviceServerJarPath.toString();
            return deviceServerJarPathStr.substring(
                            0, deviceServerJarPathStr.lastIndexOf(File.separator))
                    + MODULES_FOLDER;
        } catch (URISyntaxException e) {
            throw new RuntimeException("Failed to get Device Server path");
        }
    }

    private String[] listModules(File modulesDirectory) {
        FilenameFilter jarFileNameFilter = (dir, name) -> name.endsWith(".jar");
        Optional<String[]> modulesListOptional =
                Optional.ofNullable(modulesDirectory.list(jarFileNameFilter));
        return modulesListOptional.orElseThrow(() -> new RuntimeException("No
            modules found"));
    }
}
```

Classe SpringMain.java

```
package br.com.trapp.deviceserver;


import br.com.trapp.deviceserver.controller.StartupService;
import br.com.trapp.deviceserver.model.DeviceConfiguration;
import br.ufsc.labsec.deviceserver.api.messaging.TopicMessage;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.kafka.core.KafkaTemplate;

@SpringBootApplication
public class SpringMain {

    public static void main(String[] args) {
        SpringApplication.run(SpringMain.class, args);
    }
```

```
    @Bean
    public CommandLineRunner runner(
            MainRunner mainRunner,
            StartupService startupService,
            KafkaTemplate<String, TopicMessage> testProducer,
            @Value("${consumer.topic}") String topic,
            DeviceConfiguration deviceConfiguration) {
        return (args -> {
            startupService.start(args);
            mainRunner.loadModules();

            //          TopicMessage topicMessage = new TopicMessage();
            //          MessageMetadata metadata = new MessageMetadata("raul",
            // deviceConfiguration.getUuid(), "");
            //          topicMessage.setMetaData(metadata);
            //          DeviceMessage message = new DeviceMessage("", "");
            //          topicMessage.setMessage(message);
            //
            //          testProducer.send(topic, topicMessage);

            System.out.println(deviceConfiguration.getUuid());
        });
    }
}
```

application.properties

```
spring.kafka.bootstrap-servers=localhost:9092
server.port=8081
spring.mvc.servlet.path=/device-server

consumer.topic=client-device
producer.topic=device-client

logging.level.br.com.trapp.deviceserver = DEBUG

spring.kafka.security.protocol=SSL
spring.kafka.ssl.trust-store-location=classpath:kafka.truststore.jks
spring.kafka.ssl.trust-store-password=112233
spring.kafka.ssl.key-store-location=classpath:wildcard.kafka.keystore.jks
spring.kafka.ssl.key-store-password=112233
```