UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL CAVALCANTI LEANDRO DE LIMA

RELATO DE EXPERIÊNCIA:
TESTES EM SISTEMAS DISTRIBUÍDOS

CAMPINA GRANDE - PB

2024

GABRIEL CAVALCANTI LEANDRO DE LIMA

RELATO DE EXPERIÊNCIA: TESTES PARA SISTEMAS DISTRIBUÍDOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Fábio Jorge Almeida Morais

CAMPINA GRANDE - PB

2024

GABRIEL CAVALCANTI LEANDRO DE LIMA

RELATO DE EXPERIÊNCIA: TESTES PARA SISTEMAS DISTRIBUÍDOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Fábio Jorge Almeida Morais, Dr.
Orientador – UASC/CEEI/UFCG

Everton Leandro Galdino Alves, Dr.
Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro, Dr.
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 15 de Maio de 2024.

CAMPINA GRANDE - PB

RESUMO

No cenário atual de desenvolvimento de software, é evidente uma ampla adesão de modelos baseados em sistemas distribuídos, principalmente aqueles relacionados à computação na nuvem. Essa abordagem, embora ofereça benefícios de escalabilidade e flexibilidade, também introduz desafios significativos em relação à segurança, à conformidade, confiabilidade e à qualidade do serviço. Uma forma de garantir esses princípios é mediante a adoção de testes automáticos. No entanto, devido à complexidade inerente da arquitetura dos sistemas distribuídos e à sua necessidade de que todos componentes do sistema existam e persistam simultaneamente, extrair o comportamento do sistema torna-se uma atividade desafiadora. Serão abordados neste trabalho os desafios e soluções relacionados à criação de uma suíte de teste para ambientes naturalmente distribuídos, utilizando como exemplo um sistema open-source formado por servidores SPIRE que trabalham em conjunto com uma unidade federadora Galadriel, para estabelecer e gerenciar a confiança entre as entidades, ou seja, simulam a operação de federação confiável entre entidades, previamente desconhecidas.

EXPERIENCE REPORT:
TESTS FOR DISTRIBUTED SYSTEMS

ABSTRACT

In the current scenario of software development, there is a clear trend towards the widespread adoption of models based on distributed systems, particularly those linked to cloud computing. While this approach offers scalability and flexibility benefits, it also presents significant challenges concerning security, compliance, reliability, and quality of service. One avenue to uphold these principles is through the implementation of automated tests. However, due to the inherent complexity of distributed systems architecture and the requirement for all system components to coexist and persist simultaneously, extracting the system's behavior becomes a daunting task. This work will delve into the challenges and solutions associated with developing a test suite tailored for inherently distributed environments. An open-source system composed of SPIRE servers, working in conjunction with a Galadriel federating unit to establish and manage trust between entities, will be utilized as an example. These components simulate the operation of reliable federation among previously unknown entities.

Relato de experiência: Testes para sistemas distribuídos

Gabriel Cavalcanti Leandro de Lima
Universidade Federal de Campina Grande, UFCG
Campina Grande, Paraíba, BR
gabriel.cavalcanti.lima@ccc.ufcg.edu.br

Fábio Jorge Almeida Morais
Universidade Federal de Campina Grande, UFCG
Campina Grande, Paraíba, BR
fabio@computacao.ufcg.edu.br

RESUMO

No cenário atual de desenvolvimento de software, é evidente uma ampla adesão de modelos baseados em sistemas distribuídos, principalmente aqueles relacionados à computação na nuvem. Essa abordagem, embora ofereça benefícios de escalabilidade e flexibilidade, também introduz desafios significativos em relação à segurança, à conformidade, confiabilidade e à qualidade do serviço. Uma forma de garantir esses princípios é mediante a adoção de testes automáticos. No entanto, devido à complexidade inerente da arquitetura dos sistemas distribuídos e à sua necessidade de que todos componentes do sistema existam e persistam simultaneamente, extrair o comportamento do sistema torna-se uma atividade desafiadora. Serão abordados neste trabalho os desafios e soluções relacionados à criação de uma suíte de teste para ambientes naturalmente distribuídos, utilizando como exemplo um sistema *open-source* formado por servidores SPIRE que trabalham em conjunto com uma unidade federadora Galadriel, para estabelecer e gerenciar a confiança entre as entidades, ou seja, simulam a operação de federação confiável entre entidades, previamente desconhecidas.

PALAVRAS-CHAVE

SPIRE, microsserviços, federação, galadriel, testes funcionais, testes de integração

1 INTRODUÇÃO

A arquitetura de software baseada em microsserviços, sustentada por um contexto formado por uma metodologia ágil aliada ao ideal composto entre desenvolvimento e operações (*DevOps*), acarreta em um desenvolvimento rápido e modular que é o desejado pelo paradigma da Computação em Nuvem (*Cloud Computing*). Este paradigma conquistou notória popularidade nos últimos anos, principalmente por seus recursos, escalabilidade e flexibilidade [1]. Além disso, o paradigma faz uso de um modelo de IaaS (Infrastructure as a Service), que permitem a simplificação de toda a infraestrutura física e a facilidade de gerência, contando com acesso remoto de qualquer lugar do mundo via internet [6, 12].

O desenvolvimento e a realização de testes de aplicações distribuídas possuem uma complexidade intrínseca, devido às suas próprias características essenciais, destacando-se a concorrência de recursos e a distribuição dos componentes, tanto de software quanto de hardware. Entretanto, os processos de teste enfrentam maior dificuldade, uma vez que devem ser capazes, por meio de cenários ou casos de uso, de identificar erros que só ocorrem quando a aplicação está em execução em um ambiente de produção, onde cada componente e/ou dependência da estrutura pode falhar de forma independente.

Neste contexto, a estratégia de testes funcionais desempenha um papel crucial, uma vez que são projetados para simular as interações reais, permitindo assim a análise do comportamento do sistema e de seus componentes em relação às expectativas estabelecidas. No entanto, a implementação desses testes enfrenta desafios significativos, especialmente devido ao aumento exponencial do número de cenários de teste, resultantes da complexidade e quantidade das dependências e componentes envolvidos.

Esse grande volume de cenários pode ocasionar riscos à aplicação, visto que provoca atraso nas entregas, lentidão na verificação e validação dos componentes e aumento relativo de custos. Desta forma, muitas vezes é necessário a delimitação de um escopo reduzido de testes que ainda possam trazer a qualidade esperada.

1.1 Caracterização do problema

O foco deste trabalho é o sistema em teste (SUT - *System Under Test*), que consiste em quatro entidades previamente desconhecidas. Três delas são Servidores SPIRE (*Secure Production Identity Framework for Everyone Runtime Environment*), que são a implementação principal do framework SPIFFE (*Secure Production Identity Framework for Everyone*). Esse *framework* oferece documentos de identidade seguros e verificáveis para garantir a segurança das identidades dos componentes em ambientes distribuídos. A quarta entidade é um Servidor Galadriel, atuando como entidade federadora, responsável por gerenciar os relacionamentos entre os servidores SPIRE. O objetivo é que esses componentes trabalhem juntos para estabelecer e gerenciar a confiança entre si, realizando assim a operação de federação.

A federação é fundamentalmente uma prática na qual sistemas independentes e distintos cooperam e confiam uns nos outros com o propósito de colaboração, coordenação e compartilhamento de recursos/serviços. É interessante mencionar que neste contexto, a Galadriel não dispõe de testes funcionais de integração com o SPIRE, e a ausência desses testes pode comprometer a interoperabilidade e a estabilidade do sistema. Portanto, a falta de testes funcionais de integração representam uma preocupação significativa para o desempenho do sistema.

1.2 Proposta

Diante do apresentado, este trabalho se propõe a desenvolver artefatos[1] vinculados ao desenvolvimento de uma suíte de testes funcionais, cujo foco será um componente presente em todos o Servidores SPIRE do SUT: *plugin Federation*.

Esse *plugin* foi desenvolvido por meio de uma cooperação entre o Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG) e a *Hewlett Packard Enterprise* (HPE) cujo objetivo é a de permitir a troca de *bundles* entre um servidor SPIRE e uma entidade federadora (ex: Galadriel), ou seja, permitir a federação entre aplicações/serviços que utilizam o SPIRE.

[1]Artefatos: os testes foram desenvolvidos em um contexto de projeto PD&I, assim, o repositório é privado devido a questões de propriedade intelectual.

Os artefatos produzidos incluem a documentação[2] contendo as decisões de escopo utilizados e cenários de testes construídos, bem como as técnicas de teste utilizadas (testes de integração, dublês de testes, teste de valor e etc.). O código em si será disponibilizado via repositório do *gitlab*[3].

O principal objetivo deste trabalho é atestar a qualidade do código desenvolvido e a de conferir confiabilidade no funcionamento do *plugin* com o auxílio da suíte de testes construída. Os objetivos específicos consistiram em:

- Revisar os cenários de teste de integração fim-a-fim existentes no SPIRE, pois, a adição do *plugin* não pode alterar o resultado de caminhos e funcionalidade já estabelecidas;
- Adicionar, cenários de teste de integração que cubram os casos de uso do *plugin*; e,
- Desenvolver testes de integração, reduzindo o escopo do serviço testado ao *plugin*, ou seja, utilizar de dublês de teste, com o uso de *mock* e *fakes* para os demais componentes e dependências.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção são apresentados os conceitos essenciais para o compreendimento da temática deste trabalho. A área central de discussão do trabalho é sobre testes, com um subtópico exclusivo para seu tratamento. Porém, é necessário entender o SUT, composto pelo SPIRE e o Galadriel. Portanto, será descrita ao menos uma subseção para cada um destes elementos.

2.1 SPIFFE/SPIRE

O SPIFFE/SPIRE são projetos graduados da *Cloud Native Computing Foundation* (CNCF) que traz uma solução *open-source* de padronização, conforme detalhado por Feldman et al. [5], capaz de gerenciar identidades para serviços em ambientes heterogêneos, promovendo a interoperabilidade entre os diferentes sistemas. Suas diretrizes endossam a segurança e a automatização da emissão e gerência de identidades, como reforçado por Falcão et al. [4] e por Goel e Thangaraju [8] .

A fim de compreender como o SPIFFE se propõe a assegurar essas funcionalidades, é fundamental entender seus conceitos básicos. No contexto do SPIFFE, uma carga de trabalho (*Workload*) é um conceito abrangente e granular, podendo ser representada desde uma aplicação inteira até partes dela, como um microsserviço, máquinas físicas ou virtuais, e até mesmo um processo em nó específico. Cada uma dessas entidades é identificada de forma pelo SPIFFE ID, uma representação em cadeia de caracteres que identifica unicamente uma carga de trabalho.

Cargas de trabalho que atuam em um mesmo domínio de confiança (TD - *Trust Domain*), configurado dentro de uma infraestrutura SPIFFE e correspondente à raiz de confiança do sistema, recebem um documento criptograficamente verificável e assinado pela Autoridade Certificadora (CA - *Certificate Authority*) do domínio de confiança. Esse documento é denominado de Documento de Identidade Verificável (SVID - SPIFFE *Verifiable Identity Document*), cujo formato pode ser de certificados X.509 ou JSON *Web Token* (JWT) e contém o SPIFFE ID.

O SVID é obtido por intermédio de uma interface de programação de aplicações (API - *Application Programming Interface*) local, a *Workload* API, que não requer autenticação, evitando a utilização de credenciais pelas cargas de trabalho. A API coleta informações do sistema operacional e da carga de trabalho em execução para formar corretamente a SVID. Além disso, a API fornece os pacotes de confiança (TB - *Trust Bundle*), que são uma coleção que contém as chaves públicas dos TDs que uma carga de trabalho considera confiável, bem como providencia a chave privada relacionada ao SPIFFE ID do SVID para assinar dados em nome da carga de trabalho.

Entretanto, é possível que as cargas de trabalho precisem se comunicar com diferentes domínios, devido aos limites organizacionais do sistema ou à interoperabilidade entre organizações. Para viabilizar essa comunicação de forma confiável, é primordial um mecanismo que permita a verificação de identidades entre domínios distintos, conhecido como Federação. O SPIFFE propõe que a Federação seja concebida por meio do compartilhamento de TBs via a exposição de *endpoints*, nomeados de *bundle-endpoint*), utilizando o protocolo de autenticação mútua (mTLS - *mutual Transport Layer Security*). Atualmente o SPIFFE possui uma implementação de referência, o SPIRE, ver Figura 1.

O SPIRE possui dois principais componentes: servidor e o agente. O servidor é responsável por gerenciar e emitir todos os SVIDs do TD para o qual foi configurado, que além de armazenar os registros de entrada, contendo as condições para a validação na emissão de uma identidade, armazena as chaves de assinatura. Os agentes são executados em cada nó que uma ou mais cargas de trabalho está presente e são responsáveis por solicitar ao servidor SPIRE os SVIDs, atestar as carga de trabalho e entregar o seu SVID.

A Federação nativa, descrita no SPIRE, ocorre conforme observado na Figura. 1. Para que ela seja estabelecida, é necessário que cada SPIRE participante, siga estes passos:

(1) Expor seu TB via um *endpoint*;
(2) Mapear os relacionamentos, acrescentando em seus arquivos de configuração os parâmetros relativos aos *endpoints* alvos;
(3) Associar o TD ao *bundle-endpoint*;
(4) Especificar a URI do *bundle-endpoint*;
(5) Descrever o protocolo de transporte e o método de autenticação via o perfil do *endpoint*.

A relação de confiança é estabelecida autenticando os respectivos *bundle-endpoints*, seguida pela troca dos TBs. Isso resulta na criação dos registros de entrada em ambos servidores SPIRE participantes, definindo quais cargas de trabalho serão federadas e poderão estabelecer uma comunicação confiável.

2.2 Galadriel

A Federação provida pelo SPIRE não foi projetada para ser escalável. Há três desafios essa problemática: A Federação SPIRE requer a exposição de um *endpoint* público e seguro, essa operação só pode ser executada por um administrador; Ao federar relacionamento de muitos-para-muitos, têm-se a necessidade de criar manualmente cada entrada da federação; e, o SPIRE não fornece um mecanismo

[2]Documentação: https://drive.google.com/drive/folders/1tw4jdJ4l3GKcs_g-yg_b_fGtLihM5IZ2?usp=sharing
[3]GitLab: https://git.lsd.ufcg.edu.br/galadriel/spire/-/blob/testAll/pkg/server/plugin/federation/v1_test.go?ref_type=heads

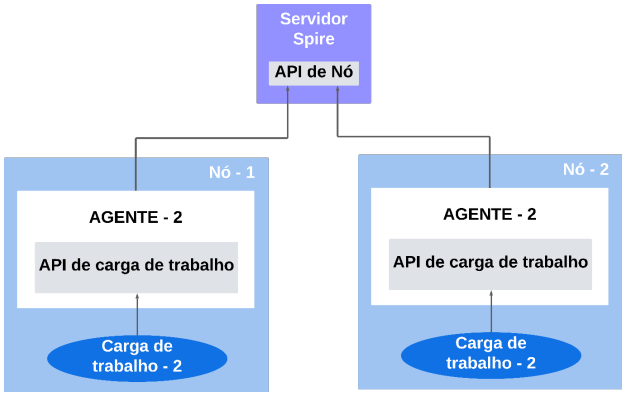

Figura 1: Representação básica do SPIRE

para gerenciar e rastrear o ciclo de vida dos relacionamentos federados, conforme Fuentes (2022) [3].

Por consequência, emerge uma alternativa a essa federação nativa, o Galadriel. Fruto do esforço *open-source* iniciado pela *Hewlett Packard Enterprise* (HPE), é uma solução flexível, que centraliza a gestão da federação e remove a necessidade de configurar um ponto de acesso público no servidor SPIRE. O design dessa solução, visto na Figura 2 introduz dois novos componentes ao sistema para efetuar a Federação, o Servidor Galadriel e o Galadriel Harvester.

O servidor Galadriel, que pode ser interpretado como um hub centralizado, irá gerenciar as relações, expondo uma REST API que será consumida pelo Harvester via HTTPS, que estará operando ao lado da implantação do SPIRE. O Galadriel *Harvester* opera junto com o Servidor SPIRE por meio de uma comunicação utilizando um *Unix Domain Sockets* (UDS). Os papéis assumidos por ele são: a de uma camada intermediária, semelhante a um *proxy*, para a configuração da Federação no servidor SPIRE; e, ser o responsável pelo encaminhamento de *bundles* de e para os servidores SPIRE via o Galadriel.

De forma simplificada, a troca de TBs será imposta pelas regras de relacionamentos estabelecidas no servidor Galadriel, enquanto que o Harvester fornecerá o encaminhamento dos *bundles*.

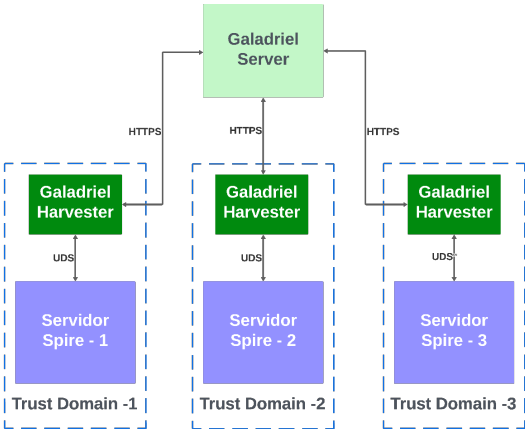

Figura 2: Design básico da solução Galadriel

2.3 Plugin

O SPIRE foi desenvolvido utilizando uma arquitetura orientada a *plugins*, e essa escolha se deve à intenção de torná-lo mais adaptável e extensível para diversas configurações e plataformas. Assim, o SPIRE não só permite, mas também incentiva a criação de *plugins* por parte de seus usuários e contribuidores.

Dessa forma, o *plugin* foi integrado nativamente ao SPIRE e, apresenta comportamento e funcionalidades semelhantes ao Galadriel *Harvester*. A grande vantagem desta proposta é a da diminuição da complexidade e de possíveis pontos de falha de funcionamento, uma vez que, o componente Galadriel *Harvester* deixará de existir. Com isso não será necessário construir toda uma comunicação via UDS entre ele o SPIRE.

O novo fluxo para a determinar a Federação parte da inicialização SPIRE, com a adição das configurações do *plugin*, ou seja, os parâmetros necessários para estabelecer a conexão com o Servidor Galadriel. Após isso, as funcionalidades de atualização e verificação de *bundles*, serão chamadas periodicamente, via rotinas de gerenciamento do controlador do *plugin* no SPIRE. Vale ressaltar que a criação e relacionamentos entre servidores continuará seguindo os padrões da Federação Galadriel.

As principais funcionalidades do *plugin* são:

- *Push bundle*: envia o pacote de confiança, atual do Servidor SPIRE, para o Servidor Galadriel;
- *Pull bundles*: solicita os pacotes dos domínios federados ao Servidor Galadriel atualizando os pacotes confiáveis;
- *Pull relationships*: solicita ao Servidor Galadriel as relações estabelecidas e as atualiza localmente e no próprio Galadriel;
- *Update relationships*: solicita a mudança de status das relações criadas no Servidor Galadriel. O status das relações pode ser alterado de em espera, para aprovadas ou negadas.

2.4 Testes

Testes são um dos pilares de um projeto de desenvolvimento de software. Eles devem ser criados com o propósito de assegurar que o software funcione corretamente e que não apresente comportamentos inesperados. Esses comportamentos podem ter sua origem não apenas no código construído, mas também em falhas em pontos de integração, problemas no ambiente de execução ou em qualquer outra dependência direta ou indireta do sistema.

O processo de criação de testes não é uma ciência exata, tampouco possui uma fórmula única de ser realizada. No entanto, a metodologia proposta por Cohn (2010) [2], que consiste na modelagem do processo em uma pirâmide de testes exemplificada na Figura 3, auxilia na organização e priorização dos testes. Isso possibilita uma cobertura eficiente em diversos aspectos do sistema, além da capacidade de identificar rapidamente áreas que demandam maior atenção durante o desenvolvimento.

No alicerce da pirâmide estão os testes unitários. O princípio desses testes é analisar individualmente pequenas porções do código, tais como, uma função ou uma simples chamada de método. Eles são voltados à tecnologia e não para as regras de negócio, garantindo com certa confiança a robustez do sistema e facilitando o diagnóstico de erros caso algum teste falhe. Por serem pequenos, é possível afirmar que, em termos de tempo de execução, são bem

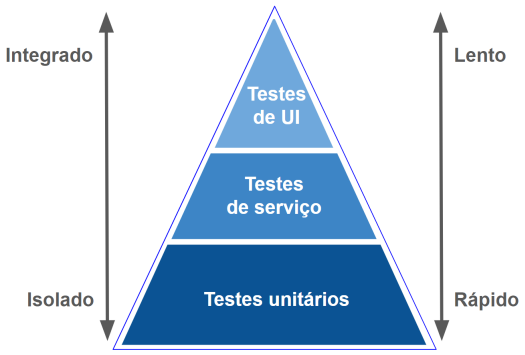

Figura 3: Pirâmide de testes.

rápidos e devem estar em maior quantidade, assim, assegurando uma maior cobertura do código.

No centro da pirâmide estão os testes de serviço, também conhecidos como testes de integração. Os testes desse tipo devem ser construídos para testar um único componente do sistema, com certo isolamento de escopo. Dado que o foco do trabalho são os testes voltados às funcionalidades do *plugin* para avaliar o comportamento do SUT, estes testes são mais relevantes, com isto será incluído no escopo do trabalho, sendo melhor detalhado na próxima subseção.

Por fim, no cume da pirâmide estão os testes de interface com usuário (UI - *User Interface*), incluindo também os testes de fim-a-fim. Esses testes possuem uma tendência voltada para as regras de negócio e devem simular uma interação real do sistema, o que, de certa forma, eleva a confiabilidade quanto ao vínculo entre os componentes do sistema. São testes mais demorados em termos de execução e construção, visto que, além do amplo escopo, todos os componentes do sistema são necessários no processo. Um ponto a se destacar nestes testes é que uma falha em um dos componentes leva à falha de todo o sistema.

Encontrar um equilíbrio adequado entre essas três camadas de testes é crucial para o desenvolvimento de software. Duas situações podem destacar essa importância: o excesso de testes UI, tornando a suíte de testes custosa e demorada, e fornecendo um *feedback* limitado; e um volume absurdo de testes unitários, comprometendo o *feedback* rápido e aumentando os custos de manutenção.

2.4.1 *Teste de Integração.*

Os testes de integração são uma estratégia de teste cujo o cerne é a interação entre os componentes internos e/ou externos do sistema, visando averiguar se esses componentes funcionam corretamente em conjunto. Segundo [9], os testes de integração podem ser descritos no contexto dos testes normais de aceitação. Contudo, para [11] e [7], não há uma definição concreta para o que é de fato um teste de integração. Entre os diversos contextos, destacam-se dois: contexto amplo e o contexto estrito.

O contexto amplo, os testes de integração devem englobar todos os componentes da aplicação simultaneamente, exigindo que cada componente esteja disponível e que seu funcionamento seja similar ao real durante a execução do teste. Esses tipo de teste é essencialmente de fim-a-fim, abrangendo todo o sistema em um ambiente de execução real.

Por outro lado, no contexto estreito, o foco dos testes está em pontos de interação individuais entre componentes ou serviços específicos. As demais partes que compõem o sistema, poderão ser substituídas por simulações, como dublês, a fim de isolar o ponto de interação. Essa abordagem assemelha-se ao processo de um teste unitário, porém, com um escopo um pouco mais amplo visto que abrange mais componentes.

2.4.2 *Teste Dublês.*

Os dublês de testes são uma estratégia de testes, na qual se propõe a utilização de objetos simulados para representar o comportamento de algum componente. Isso proporciona maior controle sobre as interações durante os testes, reduz a complexidade de execução, já que não é necessário um ambiente de execução real, resultando em uma execução mais rápida dos testes. De acordo com [10], há cinco categorias de dublês:

- *Dummies*: são objetos que apenas existem e não possuem uma funcionalidade específica, estão lá apenas para preencher um espaço necessário. Não produzem efeitos no teste, portanto, seu comportamento é irrelevante;
- *Stub*: são objetos que simulam iterações de chegada de algum componente externo ao SUT, ou seja, providenciam respostas prontas, nada além do que foi programado para o teste;
- *Mock*: são objetos utilizados para simular as interações de saída do SUT. Eles possuem expectativas sobre seu comportamento, podendo lançar exceções caso recebam alguma chamada errônea, bem como conferir recebeu todas chamadas que deveriam;
- *Spy*: são objetos que permitem o registro de qualquer interação entre eles e o SUT e podem ser compreendidos como umas mescla entre *mock* e *stub*; e,
- *Fake*: são objetos que possuem implementações funcionais bem próximas aos dos objetos reais, porém, bem mais simples.

A escolha de qual categoria utilizar depende de alguns fatores, enfatizando o contexto no qual se está realizando o teste, a complexidade do componente e o conhecimento prévio sobre o SUT. Neste trabalho, optou-se pela técnica de *fake* para construir um *fake* Galadriel, a fim de simular o comportamento do Servidor Galadriel, garantindo que esse objeto funcione de forma coerente e sem acoplamento a condições específicas de teste. Além disso, foram utilizados *fakes* já disponibilizados para os bancos de dados do Galadriel e dos SPIREs que compõem o SUT.

3 PLANEJAMENTO DE TESTES

O planejamento do processo de teste seguiu as diretrizes do SPIRE, as quais destacam a importância e exigência de testes unitários, de integração e de fim-a-fim para as funcionalidades significativas, como também, é recomendado que os testes sejam isolados, acessando apenas os recursos do próprio teste. As diretrizes sugerem o uso de dublês de teste *fakes* em detrimento dos *mocks*, devido a sua maior flexibilidade de uso, uma vez que *mocks* estão fortemente

acoplados com os detalhes de uma dependência e requerem maiores modificações em cada caso de teste, os *fakes* oferecem uma solução mais robusta e menos propensa a mudanças nos padrões de uso de cada caso de teste.

Notou-se que dentro da própria estrutura de diretórios do projeto do SPIRE, há um diretório denominado "*test*". Este diretório contém os *mocks*, *fakes* e funções auxiliares para testes, como também, testes de integração. É interessante destacar que os testes de integração dispostos neste diretório, são automatizados e tendem a ser testes de caixa preta, seguindo o tipo de teste fim-a-fim. Outro destaque é quanto a não utilização de um *framework* específico para escrita de testes, em vez disso, é utilizado pacote *testing*, nativo da linguagem Go.

Quanto ao Galadriel, ele segue o modelo adotado no SPIRE. Porém, por ser solução ainda em amadurecimento, ainda lhe faltam testes de funcionalidades, como por exemplo teste para o cliente Galadriel, como também, lhe faltam testes de integração do Galadriel com o servidor SPIRE.

3.1 Entendendo o SUT

O cerne do escopo do SUT, observado na Figura 4, é a capacidade de simular a funcionalidade de Federação entre muitos Servidores SPIRES, com o uso do *plugin* e um único servidor Galadriel. Além disso, esse sistema deve ser capaz de cumprir com os seguintes requisitos: cada Servidor SPIRE terá sua própria unidade de armazenamento; os servidores devem ser capazes de se comunicar com o Servidor Galadriel; os Servidores SPIRES devem manter registros de suas relações e dos pacotes confiáveis trocados; e, os Servidores SPIRES devem conseguir realizar a Federação entre si, por intermédio do Servidor Galadriel.

Por questões de simplificação, não foi implementada e implantada nenhuma carga de trabalho para os servidores, pois a simples troca de TBs entre servidores já é suficiente para demonstrar e comprovar o funcionamento da Federação. Quanto à quantidade de Servidores SPIRES no SUT, não será fixa, pois existem funcionalidades que não requerem interações entre os servidores, mas sim entre um Servidor SPIRE e o Servidor Galadriel. Portanto, a depender do caso de teste considera-se de um e a três servidores.

Essa quantidade limite foi determinada, após etapas de testes exploratórios, que visavam analisar o fluxo e os comportamentos esperados das funcionalidades construídas para o *plugin* e da própria Federação. Um número superior a três não traria nenhum benefício, bem como dificultaria a manutenção e legibilidade da suíte de testes.

3.2 *Fake* Galadriel

Durante o planejamento dos casos de testes, ficou evidente a importância de criar um dublê para o Galadriel. Realizar testes diretamente com o Servidor Galadriel apresentaria desafios significativos, pois à uma complexidade envolvida na manutenção do servidor em execução e na realização de operações de requisições HTTPS repetidas. Além do mais, seria necessário reiniciar o servidor após cada teste para garantir o isolamento adequado entre os casos de testes. Portanto, a criação do dublê irá proporcionar um maior controle no ambiente de testes, garantindo que os erros sejam limitados ao que de fato se deseja testar - o *plugin* - e não a por exemplo, problemas de rede.

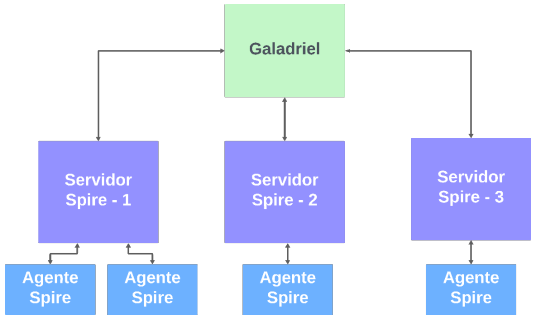

Figura 4: Sistema em teste- SUT

```
type client struct {
    trustDomain spiffeid.TrustDomain
    harvester   *endpoints.HarvesterAPIHandlers
    fakeDB      *fakedatastore.FakeDatabase
}
```

Código 2: Simplificação do cliente para o *fake* Galadriel

A escolha por construir um *fake* foi dada pelo entendimento do que se queria alcançar com os testes e o caráter investigativo da pesquisa, ambos visam compreender e garantir que o fluxo e os comportamentos dos componentes são os esperados, além de seguir com as diretrizes de contribuição, já mencionadas.

Foram consideradas algumas limitações e adaptações na construção do *fake* Galadriel, uma vez que *fake*s devem ser uma implementação simplificada do objeto real. A primeira é quanto ao processo de *onboard* entre o SPIRE e o Galadriel, decidindo-se não implementá-lo e nem verificá-lo, considerando que a comunicação entre ocorrerá e estará devidamente registrada.

A segunda é um pouco mais complexa, deve-se entender o contexto da conexão entre um Servidor SPIRE e o Servidor Galadriel. Essa conexão é estabelecida por um cliente Galadriel, configurado e instanciado pelo gerenciador do *plugin*, e é então passado como argumento para executar as funcionalidades. Internamente, a versão real desse cliente utiliza outro cliente internamente, o cliente *Harvester* (Código. 1).

Dito isto, existem duas possíveis soluções: construir um dublê para esse cliente, aumentando a complexidade e dificuldade na construção dos testes, ou optar por fazer com que o *fake* Galadriel use a API do *Harvester* diretamente. A última opção foi a escolhida, ver Código. 2, por ser mais condizente com o propósito e o escopo do trabalho.

```
type client struct {
    client      harvester.ClientInterface
    trustDomain spiffeid.TrustDomain
    jwtStore    *jwtStore
    logger      logrus.FieldLogger
}
```

Código 1: Cliente Gabriel composto por um cliente *Harvester*

4 TESTES FUNCIONAIS

Foram conduzidos testes de serviço, ou seja, testes funcionais de integração para validar o *plugin*, perante os comportamentos e as interações dos componentes do SUT. Aplicou-se uma metodologia sistemática com a abordagem de casos de teste, como apresentado na Tabela. 1, onde se incluem cenários abrangentes, gerados manualmente, que cumpram com os requisitos do SUT, já detalhados, e que permita a avaliação do fluxo de execução do *plugin*. Nesta seção, será utilizado a nomenclatura simplificada de SPIRE e Galadriel para o Servidor SPIRE e o Servidor Galadriel, respectivamente.

O fluxo de execução do teste é praticamente o mesmo para todos os casos, seguindo as seguintes etapas:

(1) Instanciação dos SPIREs e do Galadriel, com as configurações definidas nos parâmetros de cada caso de teste;
(2) Geração e armazenamento dos *bundles* por parte de cada SPIRE em seu *datastore*;
(3) Armazenamento das informações dos domínios federados, incluindo o TD, relacionamento e bundles, pelo Galadriel;
(4) Execução da funcionalidade do *plugin* através dos SPIREs;
(5) Validação dos resultados.

Nos casos de testes dos *Pull* e *UpdateRelationship* há um passo adicional que ocorre logo após a execução da funcionalidade do *plugin*, caso o cenário do caso de teste necessite realizar alguma atualização em algum elemento, ele será realizado neste instante e só então irá para a última etapa. Vale salientar que a execução da suíte é feita de forma manual utilizando o seguinte comando: `go test ./pkg/server/plugin/federation/ -run ^{Funcionalidade}$`.

4.1 Testes para *Pushbundle*

Para a funcionalidade de *PushBundle* do *plugin*, elaborou-se quatro cenários de teste, estes foram pensados para os casos de sucesso na publicação do *bundle* do SPIRE no Galadriel, falha na publicação por má formação do *bundle* devido a omissão ou falta de algum parâmetro, e a não publicação do *bundle* por falta de autorização. Os cenários foram executados em SUT simplificado, formado por um SPIRE e um Galadriel, pois isso já é o suficiente para atestar o funcionamento da funcionalidade e do sistema.

4.2 Testes para *Pull*

As funcionalidades *Pullbundle* e *PullRelationship* foram testadas no mesmo conjunto de testes, visto que o *PullBundles* realiza internamente o *PullRelationships* para sincronizar os *bundles* do SPIRE com os do Galadriel. Projetou-se ao total oito casos de teste que permitem investigar a sincronização de *bundles* dado a mudança ou criação de relacionamento entre SPIREs por intermédio do Galadriel.

4.3 Testes para *UpdateRelationship*

Por último foram criados três casos de teste para o *UpdateRelationship*, apesar da quantidade diminuta, estes casos conseguem captar o comportamento quanto à gerência dos TBs por parte dos SPIREs, para as possíveis mudanças no *status* dos relacionamentos entre eles.

Título	Caso de Teste - 2: Erro na publicação do *bundle*: o TD do *bundle* não é igual ao TD autenticado
Propósito	Verificar se o Galadriel nega a requisição de publicação do *bundle* quando o TD do *bundle* enviado pelo SPIRE não é igual ao TD autenticado.
Fluxo	(1) Inicia-se o SPIRE, com o TD igual a "a.org", porém modifica-se o TD de seu *bundle* para "c.org"; (2) O Galadriel é iniciado com suas respectivas configurações; (3) SPIRE realiza a requisição, via chamada da função Push*bundle* do *plugin*, para publicação do *bundle* no Galadriel; e, (4) Galadriel nega a requisição, retornando o status Unauthorized com a seguinte mensagem: *"trust domain in request bundle "c.org"does not match authenticated trust domain: "a.org".*
Critérios de aceitação	• O banco de dados do Galadriel não deve conter o *bundle* requisitado; • A resposta recebida pelo SPIRE deve ser no formato HTTP, com status *Unauthorized* e mensagem igual à *"trust domain in request bundle "c.org"does not match authenticated trust domain: "a.org".*

Tabela 1: Descrição do Caso de Teste 2

5 RESULTADOS

A suíte de testes teve um desempenho satisfatório, alcançando uma cobertura[4] de 77,4%, que corresponde ao percentual de declarações no código do *plugin* que foram testadas. Como mostrado na Tabela 2, a cobertura apresentada é superior à de *plugins* já consolidados para o SPIRE. Além do mais, a suíte abrangeu um total de 15 testes, distribuídos entre as quatro funcionalidades, conforme discutido na seção anterior, cuja cobertura para cada funcionalidade é disposta na Tabela 3.

Pugins	Cobertura (%)
credentialcomposer	95,8
upstreamauthority	91,5
federation	77,4
notifier	76,9
nodeattestor	75,0
keymanager	71,8
bundlepublusher	62,5

Tabela 2: Comparação de cobertura de código entre os *plugins* do SPIRE

Função	Cobertura (%)
PullBundles	72,9
PushBundle	75,0
PullRelationships	83,3
UpdateRelationships	72,7

Tabela 3: Cobertura das funções do *plugin*

As áreas em que houve falta de cobertura estavam principalmente relacionadas a métodos privados do *plugin*. A maneira correta de cobri-los seria com testes unitários, o que está além do escopo deste trabalho. Além disso, houve falhas de cobertura em exceções

[4]Cobertura: os valores foram computados manualmente por meio dos comandos `go test -coverprofile=coverage.out -coverpkg=./pkg/server/plugin/federation/`, `go tool cover -func=coverage.out` e `go tool cover -html=coverage.out`.

mais complexas de alcançar, como por exemplo, quando o TD da requisição feita pelo o *plugin* ao Galadriel não é o autorizado para a solicitação.

Durante as etapas de desenvolvimento do *plugin*, a suíte foi capaz de reconhecer um problema de desenvolvimento referente à lógica de operação da funcionalidade de *PullBundles*. O defeito encontrado estava relacionado à forma como o SPIRE reagiria caso o *status* do relacionamento fosse alterado de aprovado para negado no Galadriel, ou se os registros de cadastramento de TD fossem removidos do Galadriel. Essas situações representam a quebra de confiança ou a finalização de uma federação entre duas ou mais entidades.

O comportamento esperado para esses casos era que, quando o *plugin* sincronizasse os *bundles* com o Galadriel, ele interpretasse essas alterações e, portanto, removesse os TBs das relações inexistentes e/ou negadas do armazenamento do SPIRE. Porém, em ambas situações, o *plugin* não identificava a mudança, não apresentava erros ou *logs* dessas alterações.

Consequentemente, o SPIRE permaneceria com os TBs das relações negadas ou removidas, o que seria uma grande falha e poderia comprometer a sua segurança. O problema foi tratado com urgência, e a sua correção foi realizada ao adicionar uma função que verifica se houve alterações nos relacionamentos criados e armazenados no SPIRE quando comparados aos armazenados no Galadriel.

Embora a detecção deste problema e a sua correção tenham sido efetuadas com sucesso, e a métrica da cobertura de código tenha sido expressiva, é importante destacar, que como esperado, os testes realizados não capturaram todas as nuances e cenários do sistema. Em geral, as limitações reconhecidas estão associadas à ausência de verificação do processo de autorização e *onboard* entre o SPIRE e o Galadriel.

6 CONCLUSÕES

Este trabalho explorou conceitos de testes em sistemas distribuídos, com o objetivo de desenvolver uma suíte de testes para o *plugin Federation*, o que foi o principal objeto da análise do SUT. A intenção é garantir seu correto comportamento em diversos cenários construídos e executados no SUT. Para tanto, fez-se uso da estratégia de testes funcionais de integração, com o apoio de dublês do tipo *fake* e das diretrizes de contribuição para o desenvolvimento dessa suíte, essas decisões foram fundamentadas na delimitação do escopo do projeto.

A suíte foi desenvolvida e utilizada no processo de desenvolvimento do *plugin*, conseguindo identificar desvios do funcionamento esperado no código desenvolvido, isso faz com que aumente-se a confiabilidade e qualidade do desenvolvimento. Como o *plugin* está em constante evolução, com novas funcionalidades e mudanças na arquitetura, os testes também devem acompanhar o processo e devem ser continuamente atualizados, corrigidos e expandidos, para manter e evoluir a qualidade do produto.

A cobertura dos testes, em termos de linhas de código, foi satisfatória, levando em consideração a quantidade de casos de uso e o escopo estipulado para este trabalho. A opção pela prevalência de testes de caixa branca se deve ao caráter de pesquisa e desenvolvimento, pois eles são mais úteis quando se quer identificar falhas de lógica e entender o fluxo de operações. Uma desvantagem de utilizar este tipo de técnica, é a sua necessidade do domínio do código e dos componentes que irão participar do sistema, então seu desenvolvimento é mais lento e complexo.

Quanto ao desenvolvimento de testes em si, é necessário um processo de aprendizado gradual. A biblioteca de testes do *Go* (*testing*) é bem documentada e oferece exemplos para praticamente todos seus recursos. Além do que, o próprio SPIRE dispõe de excelentes testes que servem como referência e possui ferramentas desenvolvidas especificamente para esse fim, como o *fake datastore*.

Trabalhos futuros podem ser orientados a investigar as lacunas deixadas pelas simplificações realizadas na construção do *Fake* Galadriel, ao refatoramento e aprimoramento do código, como também na expansão dos casos de testes. Por exemplo, pode-se considerar a inclusão de testes de checagem e validação do processo de *onboard* entre o Servidor SPIRE e o Servidor Galadriel ou a extensão de testes para o gerenciador do *plugin*. Adicionalmente, a criação de testes fim-a-fim com o intuito de automatizar o processo de testagem.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer especialmente à minha avó Josefa, infelizmente já não mais presente, pelo seu olhar do céu e por ter sido a personificação das palavras amor e força.

Agradeço também a toda minha família, que é o porto seguro, sempre cheios de carinho e amor, confiança de que eu vou atingir os objetivos. Em especial à minha mãe e pai, Maria Gilvaneide Cavalcanti de Lima, por todo amor, cuidado, confiança, dedicação e instrução investidos em mim durante estes anos de vida, e a minha companheira Ana Michelle, por ser essa fonte de amor e companheirismo para todas as horas, seu incentivo e suporte foram essenciais para conclusão deste percurso.

Ao meu orientador Fábio Jorge Almeida Morais, por sua orientação, cobranças moderadas, e com o qual muito aprendi e compartilhei dessa jornada acadêmica.

Agradeço aos meus colegas de curso e do LSD, por dividir momentos de angústia e também de extrema alegria, durante esses anos de formação, pelo companheirismo nos estudos, pela torcida sempre presente e pelo sucesso um do outro. Enfim, agradeço a cada pessoa que sempre esteve comigo, sempre me dedicou palavras de carinho e de força, cada um de vocês foi essencial para a minha trajetória até aqui.

REFERÊNCIAS

[1] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco AS Netto, et al. 2018. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)* 51, 5 (2018), 1–38.

[2] Mike Cohn. 2010. *Succeeding with agile: software development using Scrum*. Pearson Education.

[3] William e Barrera Fuentes. 2022. *Galadriel - A SPIRE Federation Alternative*. Retrieved Abril 28, 2024 from https://developer.hpe.com/blog/galadriel-a-spire-federation-alternative/

[4] Eduardo Falcão, Matteus Silva, Ariel Luz, and Andrey Brito. 2022. Supporting confidential workloads in spire. In *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 186–193.

[5] Daniel Feldman, Emily Fox, Evan Gilman, Ian Haken, Frederick Kautz, Umair Khan, Max Lambrecht, Brandon Lum, Agustín M Fayó, Eli Nesterov, et al. 2020. Solving the Bottom Turtle: a SPIFFE way to establish trust in your infrastructure via universal identity. *Sprint Lab, Nova Zelândia* (2020).

[6] Claudio Fiandrino, Dzmitry Kliazovich, Pascal Bouvry, and Albert Y Zomaya. 2015. Performance and energy efficiency metrics for communication systems of

cloud computing data centers. *IEEE Transactions on Cloud Computing* 5, 4 (2015), 738–750.

[7] Martin Fowler. 2018. *Integration Test.* Retrieved Abril 28, 2024 from https://martinfowler.com/bliki/IntegrationTest.html

[8] Akarsh Goel and B Thangaraju. 2022. Authenticating distributed systems using SPIRE over kubernetes cluster. In *2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT).* IEEE, 1–6.

[9] J. Humble and D. Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education.

[10] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[11] Sam Newman. 2021. *Building microservices.* "O'Reilly Media, Inc.".

[12] Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. 2020. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing* 76, 12 (2020), 9493–9532.