UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E ELETRÔNICA

Rodrigo Vaz Pina Cabral Silva

MODELAGEM DE RELIGADOR INTELIGENTE PARA APLICAÇÕES DE RECOMPOSIÇÃO DE SERVIÇO BASEADAS EM AGENTES

Florianópolis

2019

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

Silva, Rodrigo Vaz Pina Cabral
Modelagem de religador inteligente para aplicações de recomposição de serviço baseadas em agentes / Rodrigo Vaz Pina Cabral Silva ; orientador, Diego Issicaba, 2019.
62 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Elétrica, Florianópolis, 2019.

Inclui referências.

1. Engenharia Elétrica. 2. Sistemas multiagentes. 3. Religador automático de potência . 4. Agentes inteligentes. I. Issicaba, Diego. II. Universidade Federal de Santa Catarina. Graduação em Engenharia Elétrica. III. Título.

Rodrigo Vaz Pina Cabral Silva

MODELAGEM DE RELIGADOR INTELIGENTE PARA APLICAÇÕES DE RECOMPOSIÇÃO DE SERVIÇO BASEADAS EM AGENTES

Trabalho de conclusão de curso submetido ao Departamento de Engenharia Elétrica e Eletrônica da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Engenharia Elétrica.
Orientador: Prof. Diego Issicaba, Ph.D

Florianópolis

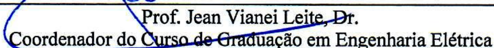
2019

Rodrigo Vaz Pina Cabral Silva

MODELAGEM DE RELIGADOR INTELIGENTE PARA APLICAÇÕES DE RECOMPOSIÇÃO DE SERVIÇO BASEADAS EM AGENTES

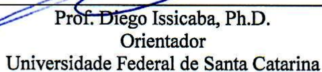
Este Trabalho foi julgado adequado como parte dos requisitos para obtenção do Título de Bacharel em Engenharia Elétrica e aprovado, em sua forma final, pela Banca Examinadora

Florianópolis, 06 de Fevereiro de 2019.

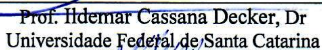


Prof. Jean Vianei Leite, Dr.
Coordenador do Curso de Graduação em Engenharia Elétrica

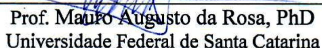
Banca Examinadora:



Prof. Diego Issicaba, Ph.D.
Orientador
Universidade Federal de Santa Catarina



Prof. Ildemar Cassana Decker, Dr
Universidade Federal de Santa Catarina



Prof. Mauro Augusto da Rosa, PhD
Universidade Federal de Santa Catarina

Este Trabalho de Conclusão de Curso en-
contra-se no âmbito do projeto P&D AN-
EEL “Novos Elementos de Automação de
Rede, Com Funções Avançadas de Inte-
ligência Distribuída” fomentado pela em-
presa EdP, código ANEEL PD-00380-00-
27/2018.

AGRADECIMENTOS

Gostaria de agradecer aos meus pais por terem proporcionado a oportunidade de estudar, garantindo que eu tivesse tudo o necessário para que pudesse focar nos estudos e tirar o maior proveito possível. Gostaria também de agradecer aos meus irmãos pelo apoio e companheirismo durante estes anos. Agradeço também à minha namorada, Mia, que me acompanha nas cervejas de segunda-feira à noite e que sempre acreditou em mim e me apoiou em minhas decisões.

Gostaria de agradecer a meus amigos do ensino fundamental e médio com quem tenho feito um longo e enriquecedor percurso. Apesar de nossos caminhos terem se separado, a diversidade de nossos futuros tem nos unido mais do que nunca.Aos meus amigos da Engenharia Elétrica com quem passei dias estudando para provas, compartilhando os ônus e bônus da vida acadêmica e pelas noites de sexta-feira investidas na boêmia.

Agradeço ao meu orientador, Diego Issicaba, pelo apoio durante o desenvolvimento deste trabalho, sanando dúvidas técnicas e oferecendo valiosas dicas de português. Agradeço, também, pela orientação nas dúvidas sobre a pós-graduação. Agradeço ao Professor Mauro Augusto da Rosa pela oportunidade de participar de projetos junto ao INESC P&D Brasil.

Por fim, agradeço especialmente ao professor Ildemar Cassana Decker pela confiança depositada em minha pessoa e por me ajudar a crescer academicamente e pessoalmente durante os três anos que participei do projeto MedFasee BT.

Como a pequena candeia chega longe com seus raios! Desse modo, no mundo corrompido brilha uma boa ação.

William Shakespeare

RESUMO

Os avanços tecnológicos desenvolvidos para os sistemas elétricos de potência têm sido essenciais para manter a qualidade do suprimento de energia frente ao crescimento gradual da demanda. Cada vez mais é estudada a utilização de inteligência artificial na operação de sistemas de distribuição para permitir a proteção de equipamentos e recuperação de sistemas em caso de falhas. Este trabalho tem por objetivo a modelagem de um religador de sistema de potência com o fim de permitir a interface entre um serviço online que obtém informações de hardware embarcado e um sistema multiagente responsável por ações de recomposição de serviço em redes de média tensão. A aplicação da modelagem é exemplificada através de um caso de estudo na plataforma de sistemas multiagente *Jason*, onde é realizada a comunicação via Web para atualização das variáveis do modelo.

Palavras-chave: Sistemas Multiagentes. Jason. Modelos orientados a Objetos.

ABSTRACT

The technological advancements on power systems have been of great relevance to the maintenance of supply quality given the gradual increase of demand. More and more researchers are studying the usage of artificial intelligence in the operation of power distribution systems for protection and recomposition purposes. This work has the target of modeling a power system recloser building an interface between an online service that receives data from reclosing devices and a multiagent system responsible for recomposition actions in medium voltage networks. The application of the modeling is exemplified using a multiagent system built in *Jason*, such that the variables of the model are regularly updated through web requests to a server.
Keywords: Multiagent Systems. Jason. Object Oriented Modeling.

LISTA DE FIGURAS

Figura 1 Camadas da arquitetura de uma *smart grid* 18
Figura 2 Classes de Residência e Cachorro em Java 24
Figura 3 Exemplo da Residência em UML 26
Figura 4 Exemplo do Transformador de Instrumentação em UML 27
Figura 5 Declaração de artefatos e Workspaces 35
Figura 6 Declaração do Enum Phase 39
Figura 7 Diagramas de Classe de Phase e Measurement 40
Figura 8 Diagrama de Classe do Religador 41
Figura 9 Diagrama de Classe do Artefato do religador........... 43
Figura 10 Sistema RBTS 45
Figura 11 Extensão do sistema RBTS para distribuição 46
Figura 12 Esquema de saida de dados do servidor Web........... 48
Figura 13 Diagrama UML da classe WebApi 49
Figura 14 Exemplo de padrão Singleton......................... 50
Figura 15 Diagrama UML da classe PowerSystem................ 51
Figura 16 Agente Cíclico Temporizado.......................... 52
Figura 17 Caso de teste - curto circuito 53
Figura 18 Console do *Jason*...................................... 54
Figura 19 Console do servidor Web 55

SUMÁRIO

1 INTRODUÇÃO 17
1.1 OBJETIVOS 19
1.2 ESTRUTURA DO TEXTO 20
2 MODELAGEM ORIENTADA A OBJETOS 21
2.1 CONSIDERAÇÕES INICIAIS 21
2.2 CLASSES E OBJETOS 22
2.3 ENCAPSULAMENTO E HERANÇA 24
2.4 UNIFIED MODELING LANGUAGE 25
2.5 SÍNTESE DO CAPÍTULO 27
3 SISTEMAS MULTIAGENTES 29
3.1 CONSIDERAÇÕES INICIAIS 29
3.2 AGENTES 30
3.3 AGENTSPEAK 31
3.4 JASON 33
3.5 CARTAGO E ARTEFATOS 34
3.6 SÍNTESE DO CAPÍTULO 35
4 MODELAGEM DE RELIGADOR COM COMUNICAÇÃO VIA WEB 37
4.1 MODELAGEM DO RELIGADOR 37
4.1.1 Modelo Orientado a Objetos 38
4.1.2 Artefato 42
4.2 COMUNICAÇÃO VIA WEB 43
4.2.1 Simulação do sistema 44
4.2.2 Servidor Web 46
4.3 SINCRONIZAÇÃO DOS AGENTES 48
4.3.1 Comunicação HTTP 48
4.3.2 Agente sincronizador 52
4.4 CASO DE TESTE 53
5 CONSIDERAÇÕES FINAIS 57
5.1 CONCLUSÕES 57
5.2 TRABALHOS FUTUROS 57
REFERÊNCIAS 59

1 INTRODUÇÃO

Os primeiros sistemas elétricos de potência surgiram em meados do final do século XIX como demonstrações das tecnologias que estavam por vir, atendendo principalmente aos nichos de iluminação pública [1]. Apesar das limitações técnicas da época, o interesse comercial na energia elétrica fez com que grandes avanços fossem obtidos nos anos seguintes, culminando na adoção da corrente alternada para minimizar perdas por transmissão, permitindo afastar a geração dos centros de consumo [2].

Hoje, depende-se da energia elétrica tanto para atender nossas necessidades básicas individuais aos interesses da sociedade, por exemplo: calefação, refrigeração de alimentos, procedimentos médicos, telecomunicação, transformação de matéria prima e exploração de combustíveis fósseis. Por conta desta relação mutualística, os sistemas elétricos de potência adotaram proporções colossais e de enorme complexidade.

Esses sistemas são comumente divididos em três grandes parcelas: geração de energia, proveniente de centrais geradoras hidrelétricas (CGHs), termelétricas, complexos eólicos, entre outras fontes; transmissão de energia, através de sistemas de alta tensão em corrente alternada ou em corrente continua (HVDC); distribuição de energia, através de sistemas de média e baixa tensão em corrente alternada, sendo esta última a que possui a responsabilidade de atender diretamente aos consumidores cativos e grande parte dos consumidores livres.

Como uma maneira de regulamentar as companhias de prestação de serviços de distribuição de energia elétrica e proteger os consumidores contra os riscos inerentes a um fornecimento instável, a Agência Nacional de Energia Elétrica (ANEEL) criou procedimentos que estabelecem padrões e normas de fornecimento de energia ao sistema elétrico nacional. Esses procedimentos, chamados de Procedimentos de Distribuição de Energia Elétrica no Sistema Elétrico Nacional (PRODIST), contêm no módulo 8 uma série de indicadores de qualidade de produto como: desvio dos níveis de tensão, desvio da frequência, desequilíbrio de tensão entre fases e distorção por harmônicos [3].

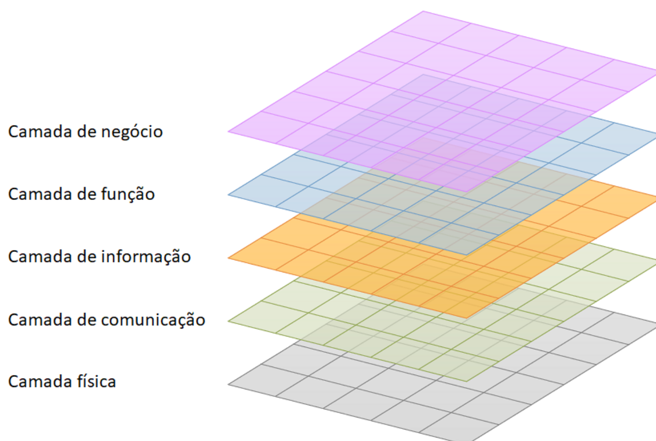
Atrelados aos indicadores de qualidade de produto, estão os indicadores de qualidade de serviço, os quais estão relacionados a problemas de continuidade causadores de interrupções consumidoras. Dentre esses indicadores, destacam-se o tempo médio de atendimento (TMAE) e o percentual de ocorrências onde há interrupção do fornecimento de energia (PNIE). Quanto às interrupções individuais, pode-se citar o in-

dicador de duração equivalente de interrupção (DEC) e de frequência equivalente de interrupção (FEC), ambos por unidade consumidora.

Esses indicadores têm impacto direto nas distribuidoras uma vez que violações nos limites dos indicadores de qualidade de serviço podem implicar em medidas punitivas por parte da ANEEL. Além das penalizações aplicáveis por violação de indicadores, as concessionárias são responsáveis por danos materiais causados aos consumidores em decorrência de desvios na frequência, subtensões e sobretensões que violem os limiares estabelecidos no PRODIST.

Em adição aos possíveis prejuízos causados pelas eventuais penalizações impostas pelo agente regulador desta área e gastos com compensação de consumidores por avarias ocorridas em seus bens, desvios expressivos na tensão e na frequência, assim como sobrecorrentes, podem causar danos materiais aos ativos das empresas de distribuição, como rompimento de dielétricos e corrosão de óleo isolante, derretimento de enrolamentos de transformadores e diminuição da integridade física de condutores [4]. Esses ativos devem passar por reparos emergenciais e manutenção preventiva constante, encarecendo os custos de operação dos sistemas de distribuição.

Figura 1 – Camadas da arquitetura de uma *smart grid*



Fonte: (Adaptado de Andrén, F, 2017)

Motivadas por esses dois fatores de impacto econômico, as distribuidoras de energia elétrica estão cada vez mais preocupadas com a qualidade do serviço fornecido aos consumidores. Investimentos em

dispositivos de proteção inteligentes tem como principais objetivos prevenir interrupções no fornecimento de energia elétrica, melhorando indicadores de operação, proteger os ativos contra falhas de origem elétrica e prover uma melhor visualização do sistema como um todo.

Recentemente, pesquisadores tem avaliado a utilização de técnicas de inteligência artificial para executar funções frente ao novo paradigma dos sistemas elétricos de potência [5]. A figura *layers* ilustra as camadas de uma arquitetura de *smart grid*, onde a camada física representa toda a rede física de distribuição, as camadas de comunicação e informação representam o aparato colocado em campo para comunicar os componentes e as camadas de função e negócio são responsáveis por operarem o sistema. Este trabalho propõe uma solução para a camada de função que busca simular interação com a camada de comunicação para atuar na camada física.

1.1 OBJETIVOS

Este trabalho tem como objetivo a modelagem de um religador de sistema de potência com o fim de permitir a interface entre um serviço online que obtém informações de *hardware* embarcado e um sistema multiagente responsável por ações de recomposição de serviço em redes de média tensão. Essa modelagem, através de orientação a objetos, servirá de referência para a implementação de uma arquitetura do sistema multiagente.

Os objetivos específicos deste trabalho são:

- Realização de um estudo e escrita de uma síntese sobre fundamentos de orientação a objetos;

- Realização de um estudo e escrita de uma síntese sobre sistemas multiagentes, com foco na linguagem AgentSpeak, plataforma Jason e ambiente CArtAgO.

- Modelagem de religador automático de potência através de orientação a objetos para fins de aplicação em sistema multiagente voltado à recomposição de sistemas de distribuição;

- Modelagem, no âmbito de sistema multiagentes, de artefatos para CArtAgO que se comuniquem com o modelo do religador.

1.2 ESTRUTURA DO TEXTO

O trabalho está estruturado como segue:

- No capítulo 2, apresentam-se definições necessárias para à compreensão de conceitos da modelagem em orientação a objetos, fundamentos da modelagem e estruturas que são utilizadas na composição do modelo do religador. Além disto, faz-se uma breve introdução à linguagem de representação de modelos orientados a objetos.

- No capítulo 3, explora-se conceitos de sistema multiagentes e expõe-se assuntos necessários a compreensão do trabalho desenvolvido. Apresenta-se um breve levantamento das definições da área de sistema multiagente, com foco na linguagem AgentSpeak, plataforma Jason e ambiente CArtAgO.

- No capítulo 4, apresentam-se os desenvolvimentos realizados neste trabalho para a obtenção de um modelo de religador utilizável para as necessidades de estudos de recomposição do sistema. Em adição, descreve-se a comunicação via web desenvolvida, a sincronização dos agentes implementados, além de um caso de teste.

- Finalmente, no capítulo 5, conclusões e trabalhos futuros são apresentados.

2 MODELAGEM ORIENTADA A OBJETOS

Neste capítulo apresentam-se alguns conceitos e exemplos de utilização de modelagem orientada a objetos, como a definição de classes, objetos e noções da *Unified Modeling Language* (UML).

2.1 CONSIDERAÇÕES INICIAIS

Uma estratégia alternativa à modelagem algébrica de sistemas de potência é a utilização de uma linguagem orientada a objetos para modelagem dos componentes, de modo que o resultado final é a obtenção de classes que descrevam os objetos físicos e que implementem estratégias para a simulação de suas grandezas [6].

A orientação a objetos é um paradigma de programação baseada no conceito de objetos. Esses são entidades que possuem posições na memória dedicadas para armazenamento de informações, conhecidas como atributos; e declaração de funções que possuem referência ao próprio escopo (atributos) da instância. Essas funções são tipicamente chamadas de métodos. Linguagens que seguem este paradigma costumam implementar o conceito de classes, ou seja, contratos que regem a criação dos objetos, definindo seu tipo.

No caso da orientação a objetos baseada em classes, tendo de exemplo a linguagem C++, diz-se que um objeto é uma instância de uma classe. Tipicamente, classes permitem a noção de herança, onde uma classe herda características de uma classe pai, sendo obrigada a cumprir a própria especificação e a especificação da classe pai. Além de C++, outras linguagens implementam a orientação a objetos e tem sido aplicadas para a modelagem de sistemas de potência, como Java, Python [7], [8] e Modelica [9].

Na área de distribuição de energia, a reconexão de elementos desconectados é fundamental para os estudos de planejamento e a operação de sistemas de distribuição. Simulações da capacidade de reconfiguração do sistema permitem que equipes da operação avaliem medidas emergenciais para recomposição do sistema em casos críticos de falha, enquanto que permitem também que equipes de planejamento analisem cenários de mudanças na rede frente a possíveis crises [10]. O trabalho desenvolvido utiliza de conceitos de orientação a objetos para segregar as características de modelagem do sistema elétrico e da simulação dos algoritmos de reconfiguração, possibilitando a reutilização dos modelos

para simular outros algoritmos de reconfiguração.

Nesse contexto, em casos onde o foco do trabalho é o estudo dos fenômenos físicos do problema e não o desenvolvimento de aplicações, uma linguagem de programação de mais alto nível pode ser vantajosa para alavancar a construção de modelos e diminuir o tempo de desenvolvimento. Pode-se notar um exemplo do paradigma de orientação a objetos para o desenvolvimento de modelos de compensadores série síncrono estáticos e de limitadores de corrente de curto-circuito supercondutores e utilizando os componentes de sistemas de potência padrões disponíveis na linguagem Modelica[9].

A modelagem orientada a objetos prevê a diminuição de gastos com manutenção da base de código, uma vez que permite que o código comum às varias entidades esteja centralizado em uma classe hierarquicamente superior a elas, fazendo com que maior tempo possa ser investido com o desenvolvimento de aplicações mais ricas, que modelem em maior profundidade os equipamentos, tenham maior performance computacional ou abranjam mais aspectos dos sistemas. Um exemplo disto é a utilização da modelagem orientada a objetos para a simulação da operação com o fim de avaliação de confiabilidade de sistemas de energia com fontes intermitentes[11]. Modelos mais ricos, que levem em consideração a incerteza da disponibilidade de energia a ser convertida, permitem que o impacto da utilização destas fontes de energia seja melhor estudado.

Outros exemplos de aplicações envolvem ferramentas como DOME [7] e Panda Power [8], que permitem a modelagem e simulação de sistemas de potência na linguagem Python, uma linguagem de programação com paradigma de orientação a objetos, leve e simples de ser utilizada, que vem crescendo no meio científico.

Apesar de não ser algo recente, a modelagem orientada a objetos consolidou-se como o paradigma de fato para a codificação de aplicações em engenharia elétrica, já que viabiliza uma adequada abstração de implementações computacionais, podendo abranger diversas aplicações dentro de um mesmo programa [12].

2.2 CLASSES E OBJETOS

Para entender a definição de classe e objeto, primeiro define-se um conceito mais elementar à ciência da computação, o conceito de tipo: tipos são definições que atribuem sentido à informação registrada na memória, definindo possíveis operações a serem realizadas sobre seus

valores e limitando o escopo representativo dessas variáveis. Em sua forma mais elementar, um tipo representa de maneira objetiva um conceito do mundo real. Por exemplo: o conjunto dos números inteiros é representado de maneira aproximada pelo tipo primitivo *Integer*, onde também são definidas suas operações elementares de adição, subtração, multiplicação e divisão; e suas respectivas limitações de representação: apenas números, sem parte fracionária e com máximo e mínimo limitado pelo número de *bits* alocados.

Apesar de classes serem comumente definidas como tipos criados pelos programadores, efetivamente existe uma separação da definição do tipo e da implementação da classe: enquanto que os tipos (primitivos ou construídos pelo programador) definem contratos a serem obedecidos, as classes são estruturas concretas que obedecem tais contratos, implementando código executável (métodos) e contendo dados (atributos). Neste sentido, classes podem implementar contratos (tipos abstratos) já existentes ou podem definir e implementar seu próprio contrato. Você pode ter um tipo definido pelo usuário, de *Animal*, que define em seu contrato informação de peso e altura e uma classe, *Cachorro*, que obedece o contrato de Animal e implementa um método de latido. Ao mesmo tempo, é possível possuir uma classe *Console*, que define o próprio contrato e o implementa, possuindo métodos de escrita e atributos quanto ao tamanho da janela.

Finalmente, pode-se então definir objetos como sendo variáveis de um determinado tipo, que são realizações de uma classe. Ou seja, uma determinada classe cria objetos (de seu próprio tipo ou de um tipo cujo contrato ela segue) e define seus aspectos concretos. Em termos de implementação, as linguagens de programação tipicamente trabalham de maneira que as variáveis de objetos são referências que apontam para onde na memória estão alocados os atributos do objeto e endereços da implementação dos métodos. Esta unificação de informação e execução em uma única entidade faz da orientação a objetos uma ferramenta excelente para a modelagem de fenômenos e coisas do mundo físico.

Uma residência pode servir como exemplo da utilização de objetos: toda casa possui um endereço. É esta informação que permite que cartas e encomendas sejam direcionadas corretamente a seus moradores, atribuindo uma característica que é única por residência. Outro fator que se tem interesse é a quantidade de habitantes da casa e se eles possuem ou não um cachorro de estimação. Sabe-se também que caso eles não possuam um animal de estimação, podem vir a adotar um no canil da cidade. No caso do animal de estimação, há interesse em saber o nome dado ao animal, sua idade, peso e raça. Finalmente,

sabe-se que os cachorros latem e gostam de correr.

Com base no exemplo anterior, pode-se definir duas classes mostradas na figura 2: a classe Cachorro, em que existem atributos do tipo *Texto* (nome e raça), atributo do tipo *Inteiro* (idade) e atributo do tipo *Numero Real* (peso) e tem-se, também, dois métodos, um para simular o cachorro correndo e outro para o latido. Temos também a classe Residência, onde temos um atributo do tipo *Texto* (endereço), um do tipo *Inteiro* (quantidade de pessoas), um do tipo *Cachorro*, que representa o animal de estimação da família, e temos também um método para adoção do animal.

Figura 2 – Classes de Residência e Cachorro em Java

```java
public class Dog {

    public String name;

    public int age;
    public String breed;
    public float weight;

    public void run() {
        // Run
    }

    public void bark() {
        // Woof Woof
    }
}
```

```java
public class Household {

    public String address;
    public int inhabitants;
    public Dog pet;

    public void adoptPet(){

        pet = new Dog();

        pet.name = "Rex";
        pet.age = 5;
        pet.breed = "Daschund";
        pet.weight = 12.5f;
    }
}
```

Fonte: (Autor, 2019)

2.3 ENCAPSULAMENTO E HERANÇA

Dois conceitos fundamentais da orientação a objetos que devem ser utilizados caso se deseje obter uma boa modelagem são os conceitos de encapsulamento e de herança. Eles permitem que se obtenham modelos que necessitem de menos manutenção e ajudam a prevenir erros gerados por descuidos no momento da implementação dos modelos, além de serem essenciais para a definição de ideias mais complexas, tais como polimorfismo, abstração e desacoplamento,

Encapsulamento é a noção de limitar a capacidade de interferência nos atributos ou métodos de uma classe por influência externa. Isto é, definem-se os atributos ou métodos que se deseja como algo privado à classe, podendo ter estes valores alterados ou métodos executados

apenas através de outros métodos disponíveis publicamente. Tipicamente, isto é utilizado quando se quer realizar alguma verificação do novo valor a ser armazenado, quando não há interesse em expôr métodos de auxílio (por exemplo, *parsing* de informação) ou quando não há necessidade de expôr atributos que contenham objetos, evitando assim vazamentos de memória ou referência.

Herança é o conceito de que uma classe pode herdar definições de atributos e implementações de métodos de uma outra classe, de maneira que os objetos da classe herdeira possuam atributos e método de ambas as classes. Isto possibilita o reaproveitamento de código, uma vez que os métodos da classe base são definidos apenas em um lugar e as mudanças na implementação são propagadas automaticamente para as outras classes herdeiras.

É possível exemplificar o conceito de herança através da seguinte modelagem: define-se a classe e tipo *Máquina Elétrica*, que possui os atributos de corrente e tensão elétrica e implementa o cálculo da potência. Define-se então uma outra classe e tipo *Transformador*, que herda da classe *Máquina Elétrica* os seus atributos e métodos, enquanto adiciona um atributo de relação de transformação e implementa os métodos de conversão de tensão e corrente para o nível de secundário. É possível ter uma terceira classe e tipo *Transformador de Instrumentação*, que herda da classe *Transformador* os seus atributos e métodos, adicionando atributos de classe de precisão e implementando um método de realizar medição. Esta classe de *Transformador de Instrumentação* exemplifica o conceito mais avançado de herança multinível, já que ela herda tanto de *Máquina Elétrica* quanto de *Transformador*.

2.4 UNIFIED MODELING LANGUAGE

A *Unified Modeling Language*, ou UML, é uma linguagem de modelagem de uso geral comumente utilizada nos diversos ramos da ciência da computação e com o objetivo de estabelecer um padrão para a descrição de soluções de *software*. Esta ferramenta permite trabalhar em um maior nível de abstração, ajudando a perceber pontos fracos de uma determinada modelagem antes mesmo de ser investido tempo no desenvolvimento de software. Além da possibilidade de modelar classes e outras estruturas da orientação a objetos, a UML também define diagramas para modelagem de atividades, casos de uso, entre outros.

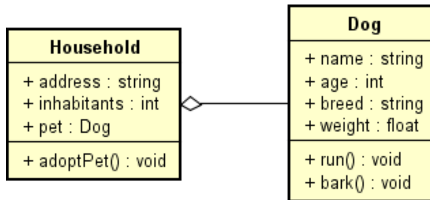
Apesar da sintaxe da classe estar atrelada às características da linguagem sendo utilizada, sua estrutura semântica é agnóstica à imple-

mentação em código, de maneira que a UML permite descrever, especificar e documentar modelos complexos que mesclem diversas tecnologias e ferramentas, de maneira conceitual. Um outro forte aspecto da UML é a possibilidade de sua utilização para modelagem de processos, fluxos de trabalho e sistemas do mundo real. Isto se dá principalmente porque a UML foi desenvolvida focada em modelagem orientada a objetos e, portanto, permite uma representação de entidades e fenômenos físicos.

Hoje existem diversas ferramentas para trabalhar com a linguagem. Algumas são ferramentas livres de criação de esquemas e diagramas que permitem você trabalhar de maneira agnóstica de uma linguagem de programação, ideal para o desenvolvimento inicial dos projetos. Outras ferramentas, comumente *plug-ins* de IDEs, permitem a conversão direta de código fonte para diagramas de classe UML, permitindo gerar, com relativa facilidade esquemas de código já existente.

Com o intuito de exemplificar a notação da UML e como ela é aplicada na modelagem orientada a objetos, foi utilizado o software Astah UML® para representar, em UML, os exemplos da residência e do transformador de instrumentação. Apesar de ter sido empregada a linguagem Java para descrever as classes do primeiro exemplo, a descrição UML aqui apresentada é genérica e poderia ser implementada em qualquer linguagem de programação orientada a objetos como C++, C# e Python.

Figura 3 – Exemplo da Residência em UML



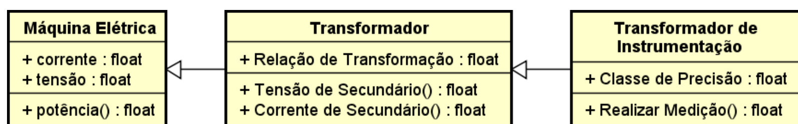
Fonte: (Autor, 2019)

A figura 3 refere-se ao exemplo apresentado na seção 2.2. Na figura, cada classe é representada por um retângulo onde são definidos os atributos e métodos de uma classe. No diagrama, estão disponíveis além dos nomes dos atributos e métodos, os tipos das variáveis e do retorno dos métodos, como *string*, *int* e *Dog* e também estão disponíveis os níveis de acesso. No caso, o símbolo + significa que os atributos e métodos são públicos, isto é, podem ser acessados externamente ao objeto. Além disto, pode-se ver que existe uma ligação entre a classe

Dog e a classe *Household*. O losango transparente indica que há uma ligação do tipo agregação entre as duas classes. A agregação significa que apesar de um objeto do tipo *Household* possuir um objeto do tipo *Dog*, este objeto *Dog* continuará existindo caso o objeto *Household* deixe de existir.

O diagrama de classes da figura 4 ilustra o exemplo da seção 2.3, que descreve o conceito de herança entre classes. Nota-se que as três classes anteriormente definidas estão descritas aqui na mesma notação que na figura 3, com a exceção de que agora não há uma relação de agregação entre as classes, mas sim de herança. As setas transparentes indicam que uma classe é herdeira direta da outra, ou seja, a classe *Transformador* é herdeira direta da classe *Máquina Elétrica* e, por conta disto, herda os atributos de corrente e tensão, assim como o método de cálculo de potência. O mesmo se aplica para a classe *Transformador de Instrumentação*, que é herdeira direta da classe *Transformador* e herda a relação de transformação e os métodos de conversão das grandezas de nível de tensão primário para secundário.

Figura 4 – Exemplo do Transformador de Instrumentação em UML



Fonte: (Autor, 2019)

2.5 SÍNTESE DO CAPÍTULO

Neste capítulo foram apresentados conceitos fundamentais e definições formais de algumas entidades da programação orientada a objetos. Além disto, foi introduzida a UML, uma linguagem utilizada na indústria para escrever diagramas de classes e formalizar modelos e casos de uso dos projetos.

3 SISTEMAS MULTIAGENTES

Este capítulo apresenta uma descrição de sistemas multiagentes, como segue. Define-se o que é um agente e a linguagem de programação tipicamente utilizada para a sua codificação. Além dos agentes e da linguagem AgentSpeak, é apresentada também a plataforma Jason de sistemas multiagentes, que interpreta uma linguagem orientada a agentes bastante similar ao AgentSpeak, sendo destacadas as mudanças principais. Também é apresentado o ambiente CArtAgO e a definição de artefato.

3.1 CONSIDERAÇÕES INICIAIS

Dentre as especificações de inteligência artificial distribuída, tem-se os chamados sistemas multiagentes. São compostos por entidades inteligentes denominadas agentes, que tipicamente trabalham em cooperação para a solução de problemas variados. Esses sistemas são fortemente baseados em suas contrapartidas do mundo real, refletindo conceitos inerentes aos seres vivos [13].

Por causa de sua elevada autonomia na solução de problemas, os sistemas multiagentes são utilizados em situações onde há a descentralização da arquitetura do sistema físico [14], como quando há espalhamento espacial das entidades ou diversificação de suas atribuições. Por conta destas peculiaridades, eles têm se mostrado uma efetiva ferramenta para a solução de problemas na engenharia elétrica.

Uma aplicação onde os sistemas multiagentes se sobressaem é na simulação e modelagem dos sistemas de distribuição de energia elétrica. Esses sistemas notoriamente se estendem por grandes extensões territoriais, atendendo desde consumidores residenciais e comerciais nos centros urbanos até grandes indústrias e consumidores mais afastados nas regiões rurais dos municípios. As redes de distribuição contam com complexos esquemas de proteção, envolvendo diversos equipamentos como relés, chaves fusíveis, religadores e transformadores de regulação, cada um com a sua função de manter a maior confiabilidade possível dos sistemas.

A auto-adaptação dos dispositivos de proteção nos sistemas de distribuição é um bom exemplo de uma aplicação de sistemas multiagentes [15]. Tipicamente, os relés de proteção são configurados de acordo com heurísticas obtidas por observação de faltas nos sistemas,

não sendo incomum que estes valores de ajuste tenham que ser revisitados, uma vez que há mudança na configuração ou expectativa de condições adversas à operação. Uma possibilidade para a realização automática de ajustes é a utilização de agentes descentralizados que procuram, através de métodos matemáticos de otimização, a obtenção de melhores parâmetros para proteção, adaptando-se em tempo real às constantes mudanças, poupando gastos e melhorando a confiabilidade.

As *smart-grids* apresentam-se como solução para a introdução da geração distribuída no sistema elétrico. Esta geração, tipicamente composta de fontes intermitentes, traz consigo a implantação de políticas de remuneração aos consumidores que possuírem geração, além de criar novos desafios por parte da proteção do sistema e da análise de falhas [16]. Uma possibilidade é a distribuição da inteligência computacional necessária nos diversos equipamentos das *smart-grids* de forma que estes equipamentos tornem-se agentes cooperando em um sistema inteligente, trocando informações que permitam a implantação de mercados locais de energia, maior transparência para os operadores descentralizados e correção e aprimoramento da proteção do sistema, tanto por conta dos próprios agentes dos relés de proteção quanto por conta de agentes supervisores da proteção.

3.2 AGENTES

Os *agentes*, para a ciência da computação, são entidades que possuem capacidade de sensoriamento e de agir de maneira predeterminada sobre o ambiente onde estão instalados. Esses agentes podem ser tanto entidades físicas quanto virtuais (implementadas em *software*). Exemplos de agentes físicos são relés de proteção, que através de sensores que medem grandezas elétricas, atuam sobre disjuntores caso alguma condição preestabelecida seja verificada. Exemplos de agentes virtuais são programas que organizam a caixa de entrada, marcando mensagens como *spam* ou assinalando sua importância.

Entretanto, não há concordância em uma definição precisa de *agente inteligente* dada a subjetividade do conceito de racionalidade e de inteligência. Alguns pesquisadores [17] se baseiam na teoria da escolha racional, desenvolvida no ramo da economia, para estabelecer que um dos critérios de inteligência é a capacidade de deliberação, ou seja, de tomar decisões com base na reflexão do impacto das ações. Os autores também colocam os *agentes inteligentes* em quatro diferentes categorias de agentes: Agentes reativos, Agentes com memória, Agentes

baseados em objetivos e Agentes baseados na utilidade.

Outros autores [18] sumarizam vários conceitos e definem agentes inteligentes (aqui chamados de agentes autônomos) da seguinte maneira: agente inteligente é aquele agente dotado de sensores que, situado em um ambiente no qual faz parte, é capaz de tomar ações que venham ao encontro com seus objetivos e que possam vir a alterar as percepções obtidas através do sensoriamento. De acordo com esta definição, o ambiente em que o agente está inserido define também a capacidade de agência, uma vez que caso o ambiente mude de forma que os sensores sejam anulados, o programa deixa de ser um agente.

3.3 AGENTSPEAK

Apesar da dificuldade em definir precisamente os conceitos de *agente inteligente* e de *racionalidade*, foram desenvolvidos modelos que buscam diminuir a lacuna entre a abstração dos agentes e de sua implementação computacional. Um modelo tipicamente utilizado é o chamado *Belief Desire Intention* ou BDI. Esse modelo, desenvolvido com base numa exploração da teoria da razão prática para agentes [19], coloca que a racionalidade do agente advém de três conceitos fundamentais:

- Crenças: são a visão de mundo do agente, obtidas através de sensores, comunicação com outros agentes ou de introspecção.
- Desejos: são objetivos, contraditórios ou não, que o agente almeja mas que não necessariamente está ativamente perseguindo.
- Intenções: são as deliberações tomadas pelo agente, de modo que ele cumpra ou termine num estado intermediário de alcançar um objetivo.

Com base na arquitetura BDI e em suas aproximações práticas, como o *Procedural Reasoning System* [20], uma linguagem de programação orientada a agentes denominada AgentSpeak [21] foi criada com o objetivo de alinhar desenvolvimentos teóricos e pragmáticos no ramo, simplificando o processo de descrição de agentes e a codificação de seus programas, através de uma sintaxe similar à programação lógica, uma vez que a linguagem é puramente abstrata.

Em AgentSpeak, toda crença é composta por um nome e um termo, sendo que as crenças onde o *termo* é uma variável são chamadas de crenças literais e as demais, onde *termo* possuí um valor concreto,

são chamadas de crenças básicas. Os objetivos em AgentSpeak são estados que o agente possui interesse e se dividem em objetivos de concretização e objetivos de teste:

- O objetivo de concretização pode ser interpretado como uma crença básica que o agente tem interesse de que se torne presente em sua base e é indicado por um sinal de exclamação antes da crença, da seguinte maneira: `!crença(azul)`

- O objetivo de teste (no sentido de questionar) pode ser interpretado como uma crença básica de que o agente gostaria de questionar, sendo indicada por um sinal de interrogação antes da crença, da seguinte maneira: `?crença(verde)`

A adição de crenças básicas e objetivos ao contexto de um agente dá origem aos chamados eventos de gatilho. Esses eventos tem o propósito de alterar a motivação do agente para que ele reaja às suas percepções e possa atuar em seus objetivos. Complementando estes eventos, a remoção de crenças e objetivos do contexto do agente também constitui eventos de gatilho. Eventos de gatilho causados por adição são indicados pelo sinal `"+"`(adição) antes da crença ou evento, enquanto que eventos de gatilho causados por remoção são indicados pelo sinal `"-"`(subtração).

Apesar de abstrata, AgentSpeak apresenta uma maneira de representar ações que o agente pode vir a executar com base nos seus objetivos e nas crenças que o agente possui. As ações costumam ter por objetivo atuar sobre o ambiente de maneira que as futuras percepções do agente sejam diferentes. A sintaxe para a definição de execução de ações em AgentSpeak é igual à da descrição de uma crença: `ação(termo).`

Finalmente, a linguagem também define uma estrutura chamada *Plano*, que descreve, através das construções anteriormente listadas, a maneira que o agente deve reagir e concretizar algo. Todo plano é dividido em um gatilho, um contexto e um corpo. A sintaxe para a descrição do plano é a seguinte: `{evento} : {contexto} <- {corpo}`.

- O gatilho é o evento que ocasionará a execução do plano. Pode-se aqui utilizar qualquer tipo de evento, seja de adição ou remoção de crenças ou objetivos.

- O contexto é uma condição expressa através de uma equação lógica com base nas crenças que deverá ser atendida para execução do plano.

- O corpo descreve uma sequência composta de ações que o agente irá realizar e de objetivos que o agente irá tentar concretizar ou testar.

3.4 JASON

Para o desenvolvimento de sistemas multiagentes, faz-se interessante a utilização de uma plataforma apropriada, sendo ela responsável não só por interpretar as descrições e os programas de agentes, mas também por manter e atualizar as bases de crenças e disponibilizar uma infraestrutura de execução de planos. Para este trabalho, foi estudada a plataforma open-source *Jason* [22] de desenvolvimento de sistemas multiagentes.

Jason interpreta uma linguagem de programação de agentes baseada em AgentSpeak, introduzindo novas construções com o principal objetivo de tornar a linguagem mais prática e o desenvolvimento de agentes uma tarefa mais simples. Dentre as principais mudanças introduzidas estão as noções de negações absolutas, tratamento de falhas de execução de planos, comunicação baseada em *Speech-Act* e anotações.

A negação absoluta foi introduzida como uma forma de unificar as conclusões tomadas pelos agentes no que diz respeito às suas crenças. Sob a hipótese de mundo fechado, tudo que não for derivado diretamente ou indiretamente das crenças de um agente é necessariamente falso, não havendo a noção de que algo pode vir a ser desconhecido. A negação absoluta garante uma maneira de programar os agentes possibilitando a verificação de desconhecimento de alguma crença (negação trivial) ou da garantia da falsidade de alguma crença (negação absoluta).

Através do tratamento de execução de planos, o desenvolvedor pode definir planos de ações de segurança para o caso de um plano falhar em atingir os objetivos propostos. Isto é essencial em uma situação onde a interrupção de um plano pode criar instabilidade, caso medidas corretivas não sejam tomadas. Um exemplo seria a hipótese onde um disjuntor deixa de abrir por falha mecânica e o agente envia comandos para disjuntores auxiliares abrirem, mitigando a falha.

Speech-Act é a teoria que fundamenta a comunicação entre agentes na plataforma. Para esta teoria, a diferença entre ações concretas e a linguagem é que ações concretas têm a capacidade de impactar diretamente o meio onde o agente se encontra, enquanto que a linguagem tem a capacidade de impactar apenas o estado mental dos agentes, estado

este definido por suas crenças, desejos e intenções.

Além de ser possível desenvolver programas que regem os agentes, *Jason* também permite que o programador interfira em seu ciclo de raciocínio, definindo novas estratégias para a escolha de quais objetivos o agente começará a seguir. Uma das principais maneiras do desenvolvedor interferir no ciclo é através das anotações. Anotações são metadados que podem ser colocados em um plano ou crença para adicionar contexto. Um exemplo de anotação seria a introdução de um grau de probabilidade de uma crença ser verdadeira, de forma que crenças pouco prováveis seriam levadas menos em consideração do que crenças mais prováveis. É importante ressaltar que apenas algumas anotações possuem implementação padrão no *Jason*, sendo necessário introduzir a influência de novas anotações manualmente.

3.5 CARTAGO E ARTEFATOS

Os ambientes onde se inserem os agentes norteiam suas características e alteram as decisões de projeto. Os sensores empregados num agente devem ser compatíveis com o lugar onde ele se encontra. Por exemplo, uma câmera tem pouca utilidade caso o agente opere dentro de uma caverna sem iluminação. Os atuadores do agente também devem ser compatíveis, de maneira que um agente que opere no mar necessita de propulsão, enquanto que um agente terrestre pode utilizar de rodas para a locomoção.

Apesar do ambiente estar relacionado à existência física do agente, faz-se necessário estender esta noção para uma abstração computacional que, de certa forma, mapeia as noções do mundo real para o sistema multiagente. Os ambientes computacionais servem então como uma maneira de traduzir da sintaxe de alto nível da programação de agente para uma sintaxe de baixo nível que opere mais próxima do hardware, permitindo o agente se expressar no ambiente físico.

CArtAgO é um *framework* que possibilita a distribuição do ambiente computacional de um sistema multiagente em unidades chamadas de *workspaces*, permitindo que haja uma separação dos ambientes de acordo com as necessidades dos agentes e das capacidades físicas das interfaces empregadas. Atrelados a um *workspace* estão os artefatos, uma analogia às ferramentas que os seres humanos utilizam para interagir com o meio. Essas entidades são elementos reativos de um sistema multiagente, estando presentes no ambiente computacional e disponíveis para os agentes que ingressarem no *workspace* em que elas

se encontram.

Nesse contexto, artefatos são classes Java que estendem através de herança da classe Artifact do CArtAgO. Cada artefato é composto, como toda a classe, de atributos e métodos próprios e herdados, e é através destes que ele pode se comunicar com a Java Machine e *hardware*, executando código de baixo nível ou com o *software* atualizando a base de crenças e interagindo com outros artefatos para propagar o impacto sobre o ambiente.

A utilização de *workspaces* e artefatos num sistema multiagente em *Jason* é simples: na programação do agente são feitas chamadas aos métodos que o artefato disponibiliza na classe, indicando qual artefato se deseja utilizar por meio de uma identificação. Isto é necessário já que podem existir diversos artefatos da mesma classe em um *workspace*. Após a criação do código do artefato e do agente, basta, na descrição do sistema multiagente, exemplificado na figura 5, declarar os *workspaces* , com seus artefatos e indicar aos agentes quais artefatos eles devem focar.

Figura 5 – Declaração de artefatos e Workspaces

```
agent exampleAgent {
    focus: exampleWorkspace.exampleArtifact
}

workspace exampleWorkspace{
    artifact exampleArtifact: examples.ExampleArtifact("exampleParam")
}
```

Fonte: (Autor, 2019)

3.6 SÍNTESE DO CAPÍTULO

Este capítulo introduz os conceitos necessários para a compreensão do que é um agente inteligente e o que o difere de um simples agente computacional. Além disto, são apresentadas as tecnologias empregadas em sistemas multiagente: AgentSpeak é uma linguagem orientada a agentes, desenvolvida com o objetivo de unificar os desenvolvimentos na área. Jason é um interpretador de uma linguagem baseada em AgentSpeak, que permite a criação de sistemas multiagentes. Finalmente, é apresentado o *framework* CArtAgO, que permite a distribuição de *workspaces* e a implementação de artefatos para interação dos agentes com o meio.

4 MODELAGEM DE RELIGADOR COM COMUNICAÇÃO VIA WEB

Este capítulo está dividido em três seções: a primeira seção descreve a modelagem de um religador automático em orientação a objetos na linguagem de programação Java, assim como de um agente Jason e de um artefato CArtAgO. A segunda seção descreve um servidor Web implementado em Javascript que executa uma simulação externa de um sistema teste e disponibiliza os resultados da simulação online. A terceira seção trata da implementação de um agente sincronizador que através de um artefato, sincroniza os artefatos dos agentes religadores com as informações da simulação vindas do servidor Web.

4.1 MODELAGEM DO RELIGADOR

Religadores são componentes fundamentais nos sistemas de distribuição. Tendo em vista que cerca de 80% das falhas em sistemas de distribuição são faltas transitórias[23], é essencial a utilização de equipamentos que realizem a tentativa de reconexão da parte afetada ao resto do sistema, evitando a necessidade de deslocamento de técnicos e diminuindo os índices de duração e frequência de interrupção de fornecimento de energia.

Além da utilidade de atuarem na proteção do sistema, religadores possuem também a funcionalidade de operarem como chaves telecomandadas, permitindo que um operador experiente realize manobras de abertura ou fechamento (salvo casos onde há bloqueio através de configuração) com o intuito de realizar a recomposição do sistema manualmente após falhas ou para desativar parte da linha em caso de manutenção.

Em casos onde existem alimentadores auxiliares ou geração distribuída que possam suprir a carga desconectada, esses religadores, caso posicionados estrategicamente para tal operação, podem atuar como chaves de reconfiguração do sistema [24], conectando a carga neste alimentador alternativo, diminuindo os impactos nos clientes e os custos em multas para a distribuidora.

Entretanto, é importante mencionar que a inserção de religadores no sistema tem o mesmo efeito da inserção de novos elementos de proteção, sendo necessária a coordenação da proteção de modo que apenas os elementos necessários atuem sobre a falta evitando corte des-

necessário de carga. Essa coordenação tem se tornado um desafio [25] com a inserção de geração distribuída já que sua utilização faz com que os sistemas deixam de possuir fluxo de potência unidirecional.

4.1.1 Modelo Orientado a Objetos

A motivação para o desenvolvimento de um modelo de religador surgiu após a análise das propostas de recomposição do sistema elétrico frente a uma falta com interrupção de energia. Este modelo tem a proposta de ser simples mas cobrindo as necessidades do ponto de vista de um sistema multiagente que será desenvolvido na plataforma *Jason*. Após uma análise de modelos comerciais de religadores, foram consideradas as seguintes características e possibilidade de sensorização dos equipamentos como de interesse para a implementação de algoritmos de recomposição:

- **Presença de tensão**: A presença de tensão a jusante e a montante do religador é necessária para diagnosticar regiões do sistema que estão desconectadas ou em curto-circuito.
- **Valor RMS da corrente e tensão**: Os valores RMS de corrente e tensão são importantes para o cálculo da potência requerida pela carga naquele ponto do sistema de distribuição. Com esta informação, permite-se analisar mais precisamente se alimentadores secundários ou uma geração distribuída são capazes de suprir a região desconectada.
- **Estado do bloqueio do religador**: Os religadores possuem configurações de bloqueio de fechamento seja por excesso de tentativas de religamento, seja por trava mecânica no equipamento. A informação de bloqueio é então necessária para indicar religadores que não estão disponíveis para manobra de recuperação do sistema.
- **Condição de Linha Viva**: Linha viva é o nome dado à condição de trabalho de intervenção e manutenção numa linha que está ligada, transferindo energia. Caso o religador venha a abrir a condição de linha viva impede a tentativa de reconexão do religador, impedindo eventual dano aos técnicos de campo caso um acidente tenha causado a atuação da proteção.

Esta lista de forma alguma busca ser uma lista completa de características de um religador [26], sendo estes equipamentos complexos com diversas funções de proteção como sobretensão e subtensão, sobrecorrente temporizada e instantânea para fase e neutro, podem operar em modo chave, implementam protocolos robustos de comunicação além de possuírem proteção contra atuação devido ao fenômeno de Cold Load Pick-up, ou seja, sobrecorrentes na reenergização da linha.

Contudo, estas funcionalidades não foram levadas em consideração na modelagem do religador já que o foco do trabalho é a reposição de serviço. O pequeno conjunto de características que foram escolhidas já é suficiente para o estudo da implementação de sistemas multiagentes que operem sobre os religadores, localmente ou telecomandados, para efetuar a recomposição do sistema.

O modelo do religador de potência depende da definição de outras duas classes para simplificar a sua utilização, diminuindo o trabalho necessário para acessar às variáveis de tensão e de corrente. Um Enum (classe especial do tipo Enum) chamado Phase é responsável por indicar a qual fase o método ou atributo refere-se, enquanto que a classe Measurement é responsável por guardar as grandezas por fase do religador, utilizando instâncias da classe Phase para fazer a indexação das medições.

O Enum Phase é declarado de maneira diferente de uma classe tradicional do Java. A declaração de um Enum na verdade é a simplificação da definição de uma classe que estende *java.lang.Enum* onde todos os atributos são estáticos (pertencem a classe e não às instâncias da classe) e constantes. Aqui, o Enum é empregado com sua finalidade mais tradicional na programação orientada a objetos: declaração de valores constantes, e nele são definidas constantes para representar cada fase de um sistema trifásico e o neutro. A figura 6 demonstra a simplicidade de se definir um Enum para conter as constantes da aplicação.

Figura 6 – Declaração do Enum Phase

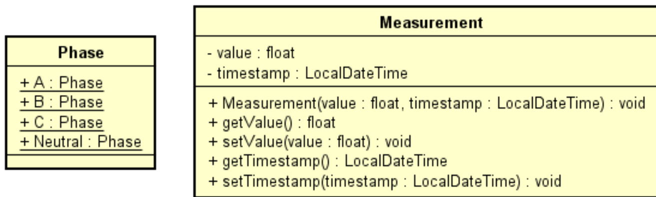
```
public enum Phase {
    A, B, C, Neutral
}
```

Fonte: (Autor, 2019)

A classe Measurement é uma classe que encapsula um atributo do tipo *Float* e um do tipo *LocalDateTime* para unificação das informações do valor e do horário da medição. Como medições individuais

podem falhar, manter um único valor centralizado do horário das medições poderia ocasionar situações onde não se sabe que uma medida está atualizada e a outra não, levando a tomada de decisão equivocada e potencialmente catastrófica. Um exemplo de possível ponto de falha da centralização do horário é o cálculo da carga conectada ao sistema pelo religador, caso o sensor de corrente parasse de enviar dados haveria uma informação errada da potência sendo calculada com tensão atualizada e corrente desatualizada; na hipótese de haver um religador conectando uma certa carga de valor que um alimentador auxiliar nas proximidades ainda consiga suprir com mínima margem para erro, a informação desatualizada devido à conexão de novas cargas ao religador faria com que, na eventual conexão deste religador ao alimentador auxiliar devido a queda de energia, o sistema pudesse ficar sobrecarregado.

Figura 7 – Diagramas de Classe de Phase e Measurement



Fonte: (Autor, 2019)

Com a definição destas duas estruturas, pode-se dar continuidade então à descrição do religador. O modelo apresenta 8 atributos que podem ser divididos em 3 grupos contextuais: Atributos de configuração representam configurações do religador, atributos de estado refletem a condição do religador e atributos de medição armazenam medidas dos sensores do religador. Além deles, existe um atributo que não entra em nenhuma das categorias que é o atributo do tipo String `id` que é um identificador único para diferenciar cada objeto religador.

Os atributos de configuração representam como o religador está configurado na vida real para operação. O primeiro atributo de configuração se chama `liveLine` e é um boolean que indica se o religador está configurado em modo de Linha Viva. Caso ocorra alguma falha no sistema e o religador venha a abrir, ele não pode ser fechado automaticamente. Outro atributo de configuração é o atributo `blocked` também do tipo boolean que indica se o religador está bloqueado para religação. Esse bloqueio pode ser um bloqueio no software de telecomando

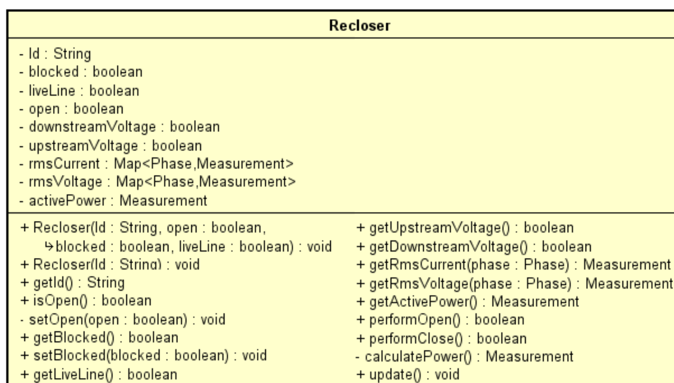
do religador ou então um bloqueio mecânico.

Os três atributos de estado, também do tipo boolean, refletem as condições sob o qual o religador está operando. O primeiro atributo `open` indica se o religador está aberto ou fechado. O atributo `downstreamVoltage` indica se o religador possui tensão elétrica nos terminais a jusante do religador. O atributo `upstreamVoltage` é similar ao anterior e indica se o religador possui tensão nos terminais a montante do religador.

Finalmente, o modelo possuí três atributos que refletem as medições dos sensores de corrente e tensão do religador. Os atributos `rmsCurrent` e `rmsVoltage` do tipo Map<Phase, Measurement>, utilizam uma estrutura de dados simples chamada *HashMap* que permite armazenar e recuperar dados indexados por chaves. Esses atributos guardam respectivamente os valores da medição de corrente e tensão indexados pelas fases disponíveis no Enum Phase do religador. Por fim, o atributo `activePower` do tipo *Measure* armazena o cálculo da potência ativa com base nos valores dos atributos de tensão e corrente.

Quanto aos métodos, a classe possuí os métodos de acesso aos atributos, denominados *Gets* e *Sets*, que garantem o encapsulamento dos valores e expondo eles apenas da maneira apropriada conforme a necessidade. Existem também os métodos de lógica de negócio para a atuação do sistema multiagente sobre os religadores e os métodos relacionados a atualização dos atributos dos modelos com os dados provenientes do servidor Web.

Figura 8 – Diagrama de Classe do Religador



Fonte: (Autor, 2019)

Os dois métodos de lógica de negócio são relacionados à operação do religador: *performOpen* e *performClose* realizam a operação de abertura e fechamento do religador de acordo com a lógica especificada pelos atributos `blocked` e `liveLine`, retornando um boolean que representa o sucesso da operação. Mais especificamente, o método *performOpen* nesta modelagem sempre retorna `true`, enquanto que o método *performClose* está sujeito ao religador estar bloqueado para religamento ou a condição de linha viva estar habilitada. Caso alguma destas duas condições seja verdadeira, a operação falha, retornando `false`.

Os métodos *getRmsCurrent* e *getRmsVoltage* são métodos auxiliares para expor os objetos Measurement das variáveis `rmsCurrent` e `rmsVoltage` de forma controlada. Como não há motivo para expor o atributo inteiro, estes métodos fazem o acesso aos objetos do mapa e no caso de acessar uma fase que não foi lida até então, retornando o valor `null`.

Por sua vez, o método *calculatePower* é responsável por realizar o cálculo da potência ativa com base nos valores de tensão e corrente oriundos das medições. Este método verifica se todas as medições são existentes e válidas (diferentes de `null`) e verifica também se todas as medidas estão atualizadas: caso alguma medida esteja desatualizada, o cálculo é cancelado e o valor atual é mantido já que a nova medida não seria confiável.

Finalmente, o método *update* é responsável por fazer a atualização do modelo do religador com os valores obtidos através do servidor Web. Este método acessa as informações mais recentes disponíveis na instância da classe *PowerSystem* e atualiza os atributos *upstreamVoltage* e *downstreamVoltage*. Em seguida, o método itera por cada fase disponível no Enum *Phase* e tenta acessar a nova medição, atualizando seu atributo caso o valor seja existente. Finalmente, após atualizar tanto o atributo *rmsCurrent* quando o atributo *rmsVoltage*, o método *calculatePower* é executado para atualizar a potência suprida pelo religador.

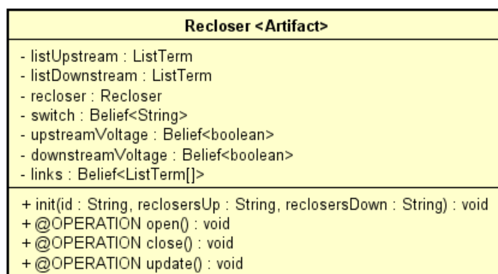
4.1.2 Artefato

O artefato do religador expõe o modelo aos agentes religadores, fornecendo operações com as quais ele pode interagir com o ambiente. Como foi decidido empregar uma arquitetura onde existe um artefato por agente religador, representando um equipamento físico, o artefato

é responsável por definir quais serão as crenças relacionadas ao agente. As crenças de presença de tensão a jusante e montante ajudam, na hipótese de uma falha, a localizar a zona afetada enquanto que a lista de vizinhos permite que o religador saiba qual agente comunicar para informar da falha.

Uma segunda função do artefato é efetuar a abertura ou fechamento do religador e atualizar a crença dos agentes do novo estado da chave, uma vez que não necessariamente a operação será bem sucedida dependendo da implementação de regra de negócio do modelo do religador. Em um ambiente real, este método iria acionar os telecomandos para abertura ou fechamento do religador, atualizando a crença com base no sensoriamento disponível após manobra.

Figura 9 – Diagrama de Classe do Artefato do religador



Fonte: (Autor, 2019)

Finalmente, a ultima função do artefato é realizar a atualização do modelo do religador através do método *update* descrito anteriormente. Após a atualização dos atributos do religador, é feita a atualização da base de crenças conforme os valores disponíveis no modelo. O artefato do religador possuí três atributos: O primeiro é um atributo do tipo Recloser que armazena um modelo instanciado conforme o String Id passado no método de inicialização do artefato, o segundo atributo e terceiro são atributos de uma classe própria do Jason chamada ListTerm que armazena uma lista de objetos. No caso, estas duas listas armazenam os Ids dos vizinhos a jusante e a montante do religador.

4.2 COMUNICAÇÃO VIA WEB

Por mais que seja interessante trabalhar com o sistema multiagente operando de maneira distribuída onde cada religador possuiria o

hardware embarcado necessário para rodar o seu próprio agente, uma solução centralizada onde um único computador é responsável por concentrar as informações e realizar o processamento pode representar uma economia no custo dos equipamentos físicos do sistema e na diminuição da complexidade da solução de maneira geral. Nas hipóteses onde há recurso disponível para a implementação de um sistema distribuído, um segundo sistema multiagente centralizado pode funcionar como um ambiente de simulação ou de validação da execução do sistema principal, permitindo que engenheiros e desenvolvedores estudem a resposta dos algoritmos empregado em campo.

Neste trabalho, foi avaliada a proposta de um sistema multiagente centralizado para validação de algoritmos de recomposição do sistema que serão implementados nos agentes religadores. Com o intuito de contribuir com uma proposta mais prática, foi considerado necessário reaproveitar as estruturas de agrupamento de medição e atuação em equipamentos que as redes de distribuição tipicamente possuem na forma de sistemas SCADA [27]. Portanto, este trabalho teve como foco não a comunicação com os religadores, mas sim a conexão remota via protocolo de internet para obtenção de informações e atuação num supervisório através de um servidor web.

Para o desenvolvimento do código de comunicação do sistema multiagente e validação através de dados de grandezas do sistema, seria necessário um servidor SCADA disponível com dados e que se comunicasse via Web. Por falta de um disponível, foi utilizada a simulação de um sistema de distribuição fictício como fonte de dados de corrente e tensão e foi desenvolvido um servidor Web para expor as informações da simulação para o sistema multiagente através de uma API.

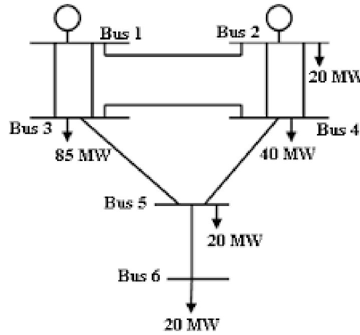
4.2.1 Simulação do sistema

Por não ser o foco deste trabalho, foi decidido utilizar um programa de simulação que já estivesse desenvolvido, apenas fazendo as alterações necessárias para a integração dele no resto do sistema. O programa de simulação utilizado foi uma implementação [28] do método Backward Fordward Sweep onde a saída de dados foi alterada para ser um arquivo de texto estruturado no formato JSON. O sistema utilizado na simulação foi uma modificação do Sistema Teste Roy Billinton (RBTS) [29], [30] com a inclusão de simples redes de distribuição [31].

O RBTS foi um sistema de transmissão proposto em 1989 com

a prerrogativa de ser um sistema pequeno de seis barras mas de fácil utilização para o ensino e validação dos conceitos de confiabilidade em sistemas de transmissão e geração, já que até então o sistema mais utilizado era o Reliability Test System [32] (IEEE-RTS). Apesar de possibilitar o desenvolvimento e comparação de técnicas diversas, o nível elevado de detalhe apresentava-se como intempérie para estudos sem a utilização de computadores.

Figura 10 – Sistema RBTS



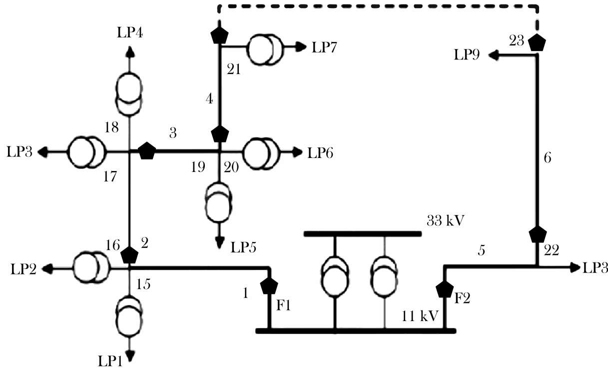
Fonte: (R. Aazami, 2016)

Em 1991, foi publicado um novo artigo com a finalidade de propor sistemas de distribuição simples para serem estudados sem a necessidade de computadores. Dando continuidade ao trabalho proposto em [29], as barras 2 e 4 do sistema RBTS original foram escolhidas como o ponto de partida para a criação destes sistemas de distribuição. O sistema utilizado neste trabalho é um sub-sistema da barra 2, onde apenas os alimentadores superiores (F1 e F2) foram levados em consideração.

Para fins práticos, foi pressuposto a instalação de 8 religadores, indicados por pentágonos na figura 11, sendo estes instalados no começo dos ramos próximos aos nós de saída de corrente. Como a saída do programa de simulação é a tensão nos nós e a corrente nos ramos, foi criado um mapeamento do sistema para obter as grandezas de cada religador no fim da simulação. Este mapeamento serve de configuração para o servidor Web, de modo que é possível selecionar as grandezas da simulação com base no identificador da classe do religador.

Devido à natureza radial do sistema de distribuição, as técnicas tradicionais de fluxo de potência costumam apresentar problemas de convergência [33], características como a alta relação entre a resistência

Figura 11 – Extensão do sistema RBTS para distribuição



Fonte: Adaptado de (V. Balaji, 2017)

e reatância das linhas e a operação frequentemente desbalanceada do sistema criam problemas mal condicionados para os algoritmos de fluxo de potência tradicionais. Por conta disto, para calcular as tensões e correntes foi utilizado uma implementação já consolidada na análise de sistemas de distribuição, o método Forward/Backward Sweep.

Este método divide a implementação em duas etapas: Na etapa Backward, a corrente nos ramos é calculada com base nos valores das impedâncias (constantes) e nos valores de tensão nas barras (no começo é admitido uma tensão igual a 1,05 P.U.), indo da subestação em direção ao ramo mais afastado. Na etapa Forward, a tensão nas barras é atualizada com base nos valores calculados na etapa anterior mas em ordem inversa. O critério para convergência do algoritmo é que a norma dos vetores resposta entre duas iterações seja menor que um limiar preestabelecido. Ao final da convergência, o programa escreve os valores de tensão nas barras e corrente nos ramos em um arquivo `.json`.

4.2.2 Servidor Web

Para o desenvolvimento do servidor Web, optou-se por utilizar uma tecnologia de desenvolvimento leve, rápida e que seja de fácil utilização. Com isto em vista, foi escolhido Javascript, uma linguagem de programação interpretada, multi-paradigma, orientada a objetos com protótipos que foi adotada como ferramenta para execução de código

nos browsers de internet. Suas principais utilizações são: permitir interatividade com páginas web e validação de formulários, obtenção e envio de estatísticas de utilização da página (Analytics) e obtenção assíncrona de conteúdo.

Diferente de uma linguagem compilada cujo produto é código de máquina que pode ser executado diretamente, linguagens interpretadas como Javascript precisam utilizar um interpretador para executar o código. Para este projeto, foi escolhido o interpretador de código aberto NodeJs. Este interpretador é baseado na *V8 Engine* do Google e foi inicialmente desenvolvido com a proposta de trazer o Javascript dos Browsers (client-side) para os servidores (server-side).

O interpretador NodeJs é utilizado na indústria para o desenvolvimento de aplicações em rede facilmente escaláveis já que funciona de maneira assíncrona e opera através de eventos. Isto significa que quando uma operação começa a ser executada mas necessita esperar por uma resposta externa, o interpretador começa a processar outras operações (assincronismo) até que a resposta esteja pronta, entregando o controle de volta a operação interrompida (eventos).

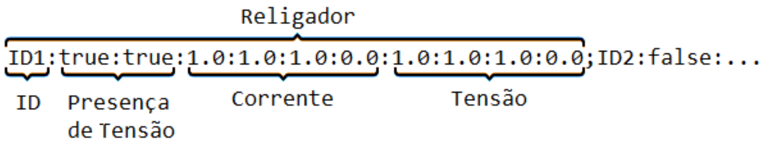
Apesar da fácil utilização do NodeJs, um servidores Web que implementa uma API necessitaria de uma quantidade considerável de código base até se tornar estável. Por conta disto, optou-se por utilizar uma biblioteca que já nos proporcionasse esta estrutura básica essencial para APIs Web, permitindo focar na integração com a simulação e na disponibilização dos dados dos religadores.

Optou-se por uma biblioteca já consolidada para o desenvolvimento de APIs chamada Express que simplifica o processamento de requisições HTTP através da definição de rotas e funções que atendem estas rotas. O protocolo HTTP estabelece tipos de requisições a serem implementadas pelos servidores, de modo que as rotas trocam de funcionalidade conforme a requisição: uma rota */users* pode retornar todos os usuários na requisição GET e cadastrar novo usuário na requisição POST, por exemplo.

O servidor implementado tem duas principais funções: A primeira é realizar a execução da simulação periodicamente, atualizando as informações carregadas na memória do servidor. A segunda função é disponibilizar os dados da simulação de acordo com o arquivo de configuração que foi criado com base na estrutura do sistema. Por conta disto, o servidor possui seis endpoints (rotas) diferentes, cada uma respondendo apenas a requisições do tipo GET. Destes endpoints, dois são relacionados a configuração do sistema, três são relacionados a simulação e um é relacionado a obtenção das medidas.

O endpoint de configuração */config* retorna a configuração carregada atual para a requisição, servindo como debug do que está carregado no sistema. Já o endpoint */reload* recarrega o arquivo de configuração e retorna a nova configuração na requisição. Os três endpoints de simulação são */start*, */run* e */stop*. O primeiro serve para começar a execução periódica da simulação e carregando seu resultado na memória. O segundo serve para forçar uma execução da simulação, o terceiro serve para parar a execução periódica da simulação. Finalmente, o endpoint */recloser* retorna as medições de corrente e tensão de cada religador de acordo com o arquivo de configuração. Por motivos de praticidade, as informações são retornadas como um String estruturado de acordo com a figura 12

Figura 12 – Esquema de saida de dados do servidor Web



Fonte: (Autor, 2019)

4.3 SINCRONIZAÇÃO DOS AGENTES

4.3.1 Comunicação HTTP

Para a comunicação do sistema multiagente com o servidor Web desenvolvido em Javascript, foi desenvolvida uma classe Java chamada *WebApi* que realiza a comunicação com o servidor Web através do protocolo HTTP. Para indicar o endereço onde o servidor Web está localizado, a classe apresenta dois atributos do tipo String chamados `host` e `port` responsáveis por guardar respectivamente a informação do hostname (ou ip) do servidor e a porta aonde ele está escutando as requisições. Um terceiro atributo chamado `baseURL` armazena uma URL gerada a partir da concatenação do hostname e da porta, evitando realizar esta concatenação em toda requisição. Um quarto atributo `status` do tipo String armazena o status da conexão com o servidor.

Além dos métodos de exposição e alteração dos atributos da classe, existem três métodos implementados: um método público *request*, um método privado *buildURL* e um método estático público *par-*

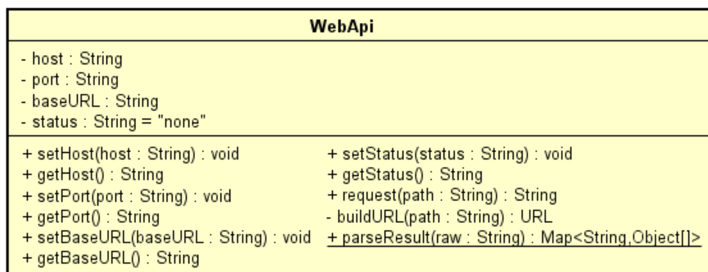
seResult.

- O método *buildURL* é utilizado para criar um objeto da classe *URL* (nativa do Java) a partir do atributo `baseURL` e de um parâmetro `path`. Caso a criação do objeto URL falhe devido a má formação (termos que não podem compor uma URL válida), o objeto é retornado como `null` e o atributo `status` recebe o valor de `bad url` para indicar que a requisição falhou por conta da URL mal formatada.

- O método *request* recebe um parâmetro `path` e através do método *buildURL*, constrói a URL da requisição. Após, ele através de um objeto da classe *HttpUrlConnection* (nativa do java), faz a requisição GET para o servidor Web e retorna os resultados da requisição como uma String. Caso a requisição seja bem sucedida, o atributo `status` recebe o valor `ok`, caso contrário recebe o valor `error`.

- O método *parseResult* é diferente dos outros uma vez que ele é um método estático. Métodos estáticos estão disponíveis no contexto da classe e podem ser invocados sem a necessidade de criação de um objeto da classe. Ele é um método de utilidade para realizar a conversão dos dados estruturados entregues pelo servidor Web (uma string) para uma variável do tipo *Map*. Aqui, as chaves são do tipo String (o ID do religador) e os valores são arrays de *Object*.

Figura 13 – Diagrama UML da classe WebApi



Fonte: (Autor, 2019)

Apesar da capacidade desta classe de fazer a chamada à API do servidor Web e fazer a conversão dos dados, ela segue o princípio de

única responsabilidade, sendo responsável apenas por realizar as requisições, não mantendo contexto do sistema multiagente nem interpretando os resultados da requisição, apenas os retornando para o método que a invocou. A classe que se responsabiliza de guardar as informações e as fornecer para os agentes religadores é uma classe chamada *PowerSystem*.

Esta classe implementa um padrão de design de programação chamado *Singleton*. Este padrão garante a existência de uma única instância de um objeto da classe, de tal maneira que cada religador acessa esta mesma instância da classe *PowerSystem* e obtém a informação sincronizada com os outros agentes. A figura 14 ilustra uma classe Singleton onde o atributo estático `instance` armazena um objeto que é instanciado assim que a classe é carregada. O acesso ao objeto é feito através do método estático *getInstance* que retorna o atributo `instance`.

Figura 14 – Exemplo de padrão Singleton

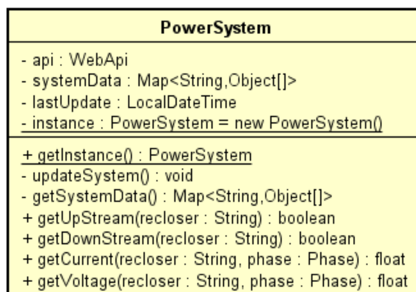
```
package system;

public class SingletonClass {

    private static final SingletonClass instance = new SingletonClass();

    private SingletonClass(){}

    public static SingletonClass getInstance(){
        return instance;
    }
}
```

Fonte: (Autor, 2019)

Além de seu atributo de instância a classe possui três atributos (o nome dos atributos está entre parenteses): um atributo do tipo `WebApi (api)` que é utilizado para fazer as requisições HTTP. Um atributo do tipo `Map<String, Object[]> (systemData)` que serve de cache dos dados já convertidos obtidos pela chamada ao servidor Web. Um último atributo do tipo `LocalDateTime (lastUpdate)` armazena o horário da última requisição feita ao servidor Web.

Quanto aos métodos, a classe não apresenta nenhum método set ou get já que não há interesse em substituir os valores dos atributos externamente, nem de obter acesso aos objetos em si. Além do método estático de retornar a instância do Singleton, existem outros seis métodos, sendo eles *getSystemData*, *updateSystem*, *getUpStream*, *getDownStream*, *getCurrent* e *getVoltage*.

Figura 15 – Diagrama UML da classe PowerSystem



Fonte: (Autor, 2019)

- O método privado *updateSystem* é responsável por receber os dados do servidor Web, converte-los com o método estático da classe WebApi e guardar o resultado no atributo `systemData`. Caso esta requisição e a conversão ocorram sem falhas, o atributo `lastUpdate` é atualizado com a hora local atual.

- O método privado *getSystemData* implementa a rotina de acesso ao atributo `systemData` com base em cache. Caso tenha se passado mais de um minuto desde a ultima vez que o método *updateSystem* foi executado com sucesso, ele é chamado, atualizando assim o valor do atributo `systemData`. Finalmente, independente dos dados terem sido atualizados ou não, é retornado o valor de `systemData`. Este padrão é utilizado para evitar que sejam feitas requisições demais ao servidor Web, idealmente deve ser sincronizado com a taxa de atualização dos dados no servidor.

- Os métodos públicos *getUpStream* e *getDownStream* são métodos de acesso às informações de presença de tensão a montante e a jusante, respectivamente, do religador. Estes métodos recebem o valor da leitura do sistema através do método *getSystemData* e recuperam os dados indexados no `Map`, sendo a chave um parâmetro `recloser` do tipo String que contem a identificação única do religador.

- Os métodos *getCurrent* e *getVoltage* são similares aos métodos *getUpStream* e *getDownStream* mas em vez da presença de tensão, retorna as medições de corrente e de tensão RMS por fase do religador. Além do parâmetro `recloser` do tipo String, há um

parâmetro `phase` do tipo Phase que indica de qual fase deseja-se obter a medição de corrente ou tensão. Similarmente aos outros dois métodos de acesso a informação, estes métodos obtem os seus valores do `Map` através do método *getSystemData*.

4.3.2 Agente sincronizador

Para realizar a sincronização dos agentes religadores com as informações da instância da classe PowerSystem, foi criado um agente homônimo cujo objetivo é periodicamente executar os métodos de atualização do artefato de cada agente. O agente sincronizador é iniciado com um objetivo de concretização para realizar uma atualização dos agentes e após o fim deste objetivo de concretização, uma crença idêntica é adicionada ao agente com o intuito de recomeçar o objetivo de concretização.

Figura 16 – Agente Cíclico Temporizado

```
!start.

+!start
   <-
      .wait(60000);
      // time consuming/blocking operations
      !start;
   .
```

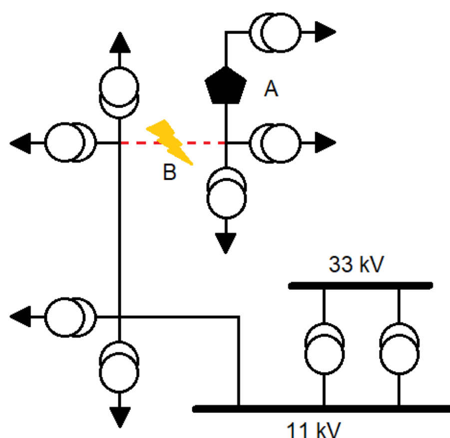
Fonte: (Autor, 2019)

Este objetivo de concretização é um objetivo simples que faz uma chamada a uma ação primitiva *wait* que espera um determinado tempo até a chamada do próximo método e após isso itera por cada artefato registrado no sistema de potência e faz uma chamada ao método de *update* do artefato, atualizando os religadores. Este método depende do nome de projeto dos artefatos, sendo que em um ambiente de produção este método deveria conseguir encontrar e fazer a chamada a artefatos descobertos em tempo de execução.

4.4 CASO DE TESTE

Como maneira de demonstrar a operação da plataforma *Jason* e a integração com o servidor Web, foi implementado um pequeno caso de teste de acordo com a figura 17. Alguns instantes após o começo da simulação, um curto circuito, indicado em B, faz com que a proteção da linha (aqui considerada como uma chave fusível) atue, isolando as três últimas cargas do restante do sistema.

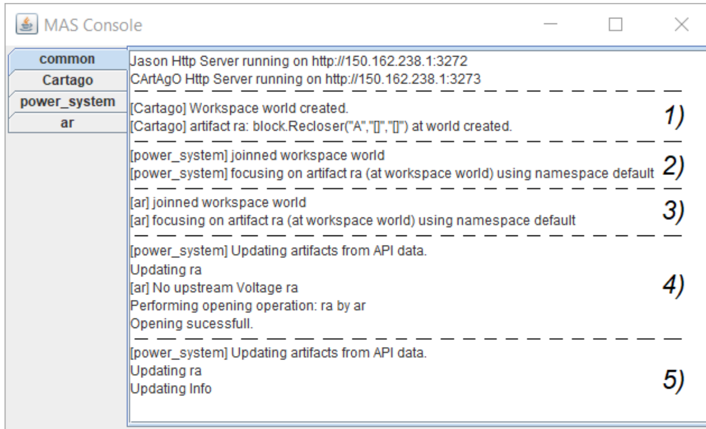
Figura 17 – Caso de teste - curto circuito



Fonte: (Autor, 2019)

A partir da atuação da proteção, o servidor Web irá disponibilizar informações atualizadas do sistema onde não há presença de tensão a jusante e a montante do religador indicado em A. O agente religador foi programado para que, caso ele obtenha a crença de que perdeu tensão a montante e caso seu contexto indique que ele ainda esteja fechado, ele execute uma ação de abertura para isolar as cargas. A figura 18 apresenta o console de *output* da plataforma *Jason* para este caso.

Figura 18 – Console do *Jason*



Fonte: (Autor, 2019)

Por questões práticas, a figura 18 foi dividida em cinco partes que representam instantes temporais distintos:

1. O CArtAgO cria o *workspace* `world` onde serão inseridos os artefatos e logo em seguida cria o artefato do religador em `world`.

2. *Jason* cria o agente responsável por fazer a sincronização dos artefatos dos religadores. O agente é então inserido no *workspace* `world`.

3. O agente religador é criado em `world` e ele então foca o seu artefato de religador, de modo que a sua base de crenças é atualizada conforme as informações disponíveis no artefato.

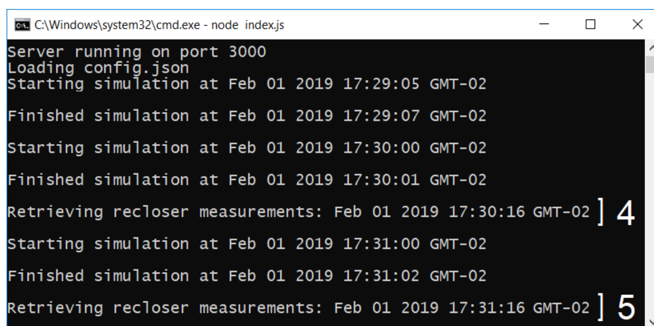
4. Um minuto após o começo do programa, o agente sincronizador faz a primeira atualização dos artefatos. Após a conclusão da requisição, o agente atualiza o artefato `ra` com as novas informações. Este novo estado faz com que o agente perceba a adição da crença de que não existe tensão a montante do religador, iniciando a operação de abertura. Esta operação é bem sucedida, trocando a crença de que o religador está fechado para aberto.

5. Finalmente, o agente sincronizador realiza uma nova etapa de atualização, onde ele percebe que seu cache está desatualizado e faz uma nova requisição para baixar as informações. Após isto,

ele atualiza o artefato sem haver nenhuma nova intervenção do agente religador.

Em paralelo, podemos visualizar o *output* do servidor Web, onde é possível ver que a simulação é executada a cada minuto e toda vez que uma requisição é feita para obter informações dos religadores, o servidor Web avisa que foi feito um acesso às informações e o horário da requisição. Os instantes assinalados por 4 e 5 no *output* do servidor Web são os mesmos instantes 4 e 5 em que o *Jason* fez a atualização dos artefatos.

Figura 19 – Console do servidor Web



Fonte: (Autor, 2019)

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÕES

Este trabalho teve como proposta o desenvolvimento de um modelo de Religador para sistemas de distribuição que utilizasse de técnicas de modelagem orientada a objetos com o objetivo de diminuir o custo de manutenção do código e permitir a expansão do modelo conforme necessidade. Este modelo, baseado em especificações práticas de religadores, busca suprir as necessidades do ponto de vista da recomposição do sistema, deixando para um segundo momento os aspectos de proteção.

Considerando a natureza conectada do sistema elétrico e a utilização da internet como forma de comunicação entre as diversas entidades, elaborou-se uma arquitetura de testes, empregando técnicas de simulação com o objetivo de produzir medições coerentes com as de um sistema de distribuição e utilizando de tecnologias atuais na computação para criar um servidor que pudesse conectar os diversos componentes.

Por fim, foi demonstrada a possibilidade de integrar os modelos e o servidor Web com uma infraestrutura de sistema multiagente. Esta integração permite que os agentes atualizem seus modelos de religadores com informações atualizadas do sistema através de simples requisições Web, recriando os típicos cenários que serão encontrados em campo.

5.2 TRABALHOS FUTUROS

Como proposta de um eventuais trabalhos futuros, coloca-se:

- Implementação de técnicas de recomposição do sistema na plataforma Jason;
- Expansão do algoritmo de simulação para levar em consideração os comandos de abertura e fechamento dos agentes religadores;
- Remodelagem da API do servidor Web para uma abordagem RESTful, que possibilite a configuração de novos religadores durante a execução.

REFERÊNCIAS

[1] The American Society of Mechanical Engineers. *The Folsom Powerhouse No. 1.* Setembro 1976.

[2] M. Brown and R. Sedano. *Electricity Transmission, A Primer.* National Council on Electric Policy, Junho 2004.

[3] ANEEL – Agência Nacional de Energia Elétrica. PRODIST – Módulo 8 – Qualidade de Energia Elétrica, 10ª Revisão, 2018.

[4] B. Pahlavanpour and A. Wilson. Analysis of transformer oil for transformer condition monitoring. In *IEEE Colloquium on An Engineering Review of Liquid Insulation (Digest No. 1997/003)*, pages 1/1–1/5, Jan 1997.

[5] Filip Pröstl Andrén, Thomas I. Strasser, and Wolfgang Kastner. Engineering smart grids: Applying model-driven development from use case design to deployment. *Energies*, 10(3), 2017.

[6] C. R. Fuerte-Esquivel, E. Acha, S. G. Tan, and J. J. Rico. Efficient object oriented power systems software for the analysis of large-scale networks containing facts-controlled branches. *IEEE Transactions on Power Systems*, 13(2):464–472, May 1998.

[7] F. Milano. A python-based software tool for power system analysis. In *2013 IEEE Power Energy Society General Meeting*, pages 1–5, July 2013.

[8] L. Thurner, A. Scheidler, F. Schäfer, J. Menke, J. Dollichon, F. Meier, S. Meinecke, and M. Braun. Pandapower—an open-source python tool for convenient modeling, analysis, and optimization of electric power systems. *IEEE Transactions on Power Systems*, 33(6):6510–6521, Nov 2018.

[9] K. Hongesombut, Y. Mitani, Y. Tada, T. Takazawa, and T. Shishido. Object-oriented modeling for advanced power system simulations. In *2005 IEEE Russia Power Tech*, pages 1–6, June 2005.

[10] I. Drezga, R. P. Broadwater, and A. J. Sugg. Object-oriented analysis of distribution system reconfiguration for power restoration. In *2001 Power Engineering Society Summer Meeting. Con-*

ference Proceedings (Cat. No.01CH37262), volume 2, pages 1215–1220 vol.2, July 2001.

[11] Júlio Alberto Silva Dias and C. L. T. Borges. Object oriented model for composite reliability evaluation including time varying load and wind generation. In *2010 IEEE 11th International Conference on Probabilistic Methods Applied to Power Systems*, pages 767–772, June 2010.

[12] Marcelo Neujahr Agostini. Nova filosofia para o projeto de software para sistemas de energia elétrica usando modelagem orientada a objetos, 2003.

[13] Jomi F. Hübner Rafael H. Bordini and Michael Woolridge. *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley, 2007.

[14] V. S. Koritarov. Real-world market representation with agents. *IEEE Power and Energy Magazine*, 2(4):39–46, July 2004.

[15] S. J. Lee, B. W. Min, K. H. Chung, M. S. Choi, S. H. Hyun, and S. H. Kang. An adaptive optimal protection of a distribution system using a multi-agent system. In *2004 Eighth IEE International Conference on Developments in Power System Protection*, 2004.

[16] R. Abedini, T. Pinto, H. Morais, and Z. Vale. Multi-agent approach for power system in a smart grid protection context. In *2013 IEEE Grenoble Conference*, pages 1–6, June 2013.

[17] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[18] Stan Franklin and Art Graesser. Is it an agent, or just a program?:a taxonomy for autonomous agents. In *Third International Workshop on Agent Theories,Architectures, and Languages*, 1996.

[19] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, 1988.

[20] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, Dec 1992.

[21] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away*, pages 42–55, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[22] Rafael Bordini and Jomi Hübner. An overview of jason. 03 2012.

[23] O.A. Quiroga, Joaquim Meléndez, and Sergio Herraiz. Fault causes analysis in feeders of power distribution networks. *Renewable Energy and Power Quality Journal*, pages 1269–1272, 05 2011.

[24] Y. L. Lo, C. H. Wang, and C. N. Lu. A multi-agent based service restoration in distribution network with distributed generations. In *2009 15th International Conference on Intelligent System Applications to Power Systems*, pages 1–5, Nov 2009.

[25] Muhammad Yousaf and Tahir Mahmood. Protection coordination for a distribution system in the presence of distributed generation. *Turkish Journal of Electrical Engineering & Computer Sciences*, 25(1):408–421, 2017.

[26] Companhia Paranaense de Energia - COPEL. *Especificação Técnica para Religadores Automáticos Para Rede De Distribuição*, January 2018. REL - 02.

[27] E. . Chan and H. Ebenhoh. The implementation and evolution of a scada system for a large distribution network. *IEEE Transactions on Power Systems*, 7(1):320–326, Feb 1992.

[28] L. Venturini, G. Silva, P. Pauletti, and Z Alves. Algorithm for the calculation of power flow for unbalanced distribution grids through the backward/forward sweep method. *Brazilian Technology Symposium - BTSym'18*, 2017.

[29] Roy Billinton, Sudhir Kumar, Nurul Chowdhury, Kelvin Chu, Kamal Debnath, Lalit Goel, Easin Khan, P. Kos, Ghavameddin Nourbakhsh, and J. Oteng-Adjei. A reliability test system for educational purposes-basic data. *IEEE Transactions on Power Systems*, 4(3):1238–1244, August 1989.

[30] R. Billinton, S. Kumar, N. Chowdhury, K. Chu, L. Goel, E. Khan, P. Kos, G. Nourbakhsh, and J. Oteng-Adjei. A reliability test system for educational purposes-basic results. *IEEE Transactions on Power Systems*, 5(1):319–325, Feb 1990.

[31] R. N. Allan, R. Billinton, I. Sjarief, L. Goel, and K. S. So. A reliability test system for educational purposes-basic distribution system data and results. *IEEE Transactions on Power Systems*, 6(2):813–820, May 1991.

[32] P. M. Subcommittee. Ieee reliability test system. *IEEE Transactions on Power Apparatus and Systems*, PAS-98(6):2047–2054, Nov 1979.

[33] M. S. Srinivas. Distribution load flows: a brief review. In *2000 IEEE Power Engineering Society Winter Meeting. Conference Proceedings (Cat. No.00CH37077)*, volume 2, pages 942–945 vol.2, Jan 2000.