UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MATHEUS KOWALCZUK FERST

ARQUITETURA ORIENTADA A SERVIÇOS APLICADA AO DESENVOLVIMENTO DE UM CONTROLADOR DE RECURSOS DISTRIBUÍDOS DE ENERGIA FOTOVOLTAICA COMPATÍVEL COM O PADRÃO IEC 61850

DISSERTAÇÃO

PATO BRANCO
2022

MATHEUS KOWALCZUK FERST

ARQUITETURA ORIENTADA A SERVIÇOS APLICADA AO DESENVOLVIMENTO DE UM CONTROLADOR DE RECURSOS DISTRIBUÍDOS DE ENERGIA FOTOVOLTAICA COMPATÍVEL COM O PADRÃO IEC 61850

SERVICE ORIENTED ARCHITECTURE APPLIED TO THE DEVELOPMENT OF A PHOTOVOLTAIC DISTRIBUTED ENERGY RESOURCE CONTROLLER COMPATIBLE WITH THE IEC 61850 STANDARD

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná. Área de Concentração: Engenharia Elétrica.

Orientador: Gustavo Weber Denardin

PATO BRANCO

2022

4.0 Internacional

Esta licença permite remixe, adaptação e criação a partir do trabalho, para fins não comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Pato Branco

MATHEUS KOWALCZUK FERST

ARQUITETURA ORIENTADA A SERVIÇOS APLICADA AO DESENVOLVIMENTO DE UM CONTROLADOR DE RECURSOS DISTRIBUÍDOS DE ENERGIA FOTOVOLTAICA COMPATÍVEL COM O PADRÃO IEC 61850

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Sistemas E Processamento De Energia.

Data de aprovação: 31 de Agosto de 2022

Dr. Gustavo Weber Denardin, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Carlos Henrique Barriquello, Doutorado - Universidade Federal de Santa Maria (Ufsm)

Dr. Dalcimar Casanova, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Giovanni Alfredo Guarneri, Doutorado - Universidade Tecnológica Federal do Paraná

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 30/11/2022.

RESUMO

FERST, Matheus Kowalczuk. Arquitetura Orientada a Serviços Aplicada ao Desenvolvimento de um Controlador de Recursos Distribuídos de Energia Fotovoltaica Compatível com o Padrão IEC 61850. 2022. 103 f. Dissertação. Programa de Pós-Graduação em Engenharia Elétrica. Pato Branco. 2022.

A crescente demanda por energia, as perdas relacionadas à transmissão e distribuição e o impacto ambiental das fontes tradicionais de energia têm tornado as fontes Renováveis de Gerações Distribuídas (RDGs) uma opção relevante para a expansão da rede elétrica. O maior uso dessas fontes, entretanto, pode suscitar outros problemas, como aumento da tensão, injeção de harmônicos de corrente e falsos ilhamentos. A superação de tais obstáculos requer o controle de um grande número de Recursos Distribuídos de Energia (DERs) e também representa uma oportunidade para melhorar a qualidade da energia e a confiabilidade da rede. O conjunto de *hardware* e *software* destinado a realizar essa funcionalidade é denominado Sistema de Gerenciamento de Recursos Distribuídos de Energia (DERMS). O presente trabalho aplica os conceitos de Arquiteturas Orientadas a Serviços (SOA) para desenvolver os sistemas de comunicação de um DERMS com foco em fontes de energia fotovoltaica. O sistema implementado foi capaz de monitor mais de quatro centenas de dispositivos sem atrasos significativos no período de *polling*, e a latência média para comunicações com a concessionária ficou abaixo de 2 ms, mesmo com a utilização de criptografia e do protocolo TLS.

Palavras-chave: Sistemas de Comunicação. Geração Distribuída. Energia Fotovoltaica. SOA. Gerenciamento de DER

ABSTRACT

FERST, Matheus Kowalczuk. Service Oriented Architecture Applied to the Development of a Photovoltaic Distributed Energy Resource Controller Compatible with the IEC 61850 Standard. 2022. 103 f. Dissertação. Programa de Pós-Graduação em Engenharia Elétrica. Pato Branco. 2022.

The ever-growing demand for energy, the losses related to transmission and distribution, and the environmental impact of traditional energy sources have made the Renewable Distributed Generations (RDGs) a suitable option for power grid expansion. However, additional problems arise with the increase of RDGs connected to the grid, such as voltage rise, harmonic current emissions, and false islanding. The overcome of such obstacles involve the control of a large number of Distributed Energy Resources (DERs), which additionally provides an opportunity to increase the power quality and reliability of the grid. The combination of hardware and software directed to perform such functionality is called Distributed Energy Resource Management System (DERMS). This work applies the concepts of Service Oriented Architecture (SOA) to develop the communication systems of a DERMS with emphasis on photovoltaic energy resources. The implemented system was able to monitor more than four hundred devices without significant delays in the polling period, and the average latency for communications with the utility was below 2 ms, even with the use of encryption and the TLS protocol.

Keywords: Communication Systems. Distributed Generation. Photovoltaic Energy. SOA. DER Management.

LISTA DE SIGLAS

ANEEL	Agência Nacional de Energia Elétrica
API	*Application Programming Interface*
CA	*Certificate Authority*
CID	*Configured IED Description*
CRC	*Cyclic Redundancy Check*
CSIP	*Common Functions for Smart Inverters*
DER	*Distributed Energy Resource*
DERMS	*Distributed Energy Resource Management System*
DMS	*Distribution Management System*
DNP	*Distributed Network Protocol*
eMIX	*Energy Market Information Exchange*
EMS	*Energy Management System*
EPRI	*Electric Power Research Institute*
ESB	*Enterprise Service Bus*
ESS	*Energy Storage System*
FAM	*Flexible array member*
FDEMS	*Facility DER Management System*
GOSE	*Generic Object Oriented Substation Event*
ICD	*IED Capabilities Description*
IEC	*International Electrotechnical Commission*
IED	*Intelligent Electronic Device*
IPC	*Inter Process Communication*
MMS	*Manufacturing Messaging Specification*
MSS	*Maximum Segment Size*
MTU	*Master Terminal Unit*
NIST	*National Institute of Standards and Technology*
PCC	*Point of Common Coupling*
PRODIST	Procedimentos de Distribuição
PV	*Photovoltaic*
RDG	*Renewable Distributed Generation*
RTU	*Remote Terminal Unit*
SAS	*Substation Automation System*
SCADA	*Supervisory Control and Data Acquisition*
SCD	*Substation Configuration Description*
SEP	*Smart Energy Profile*

SEPA	*Smart Electric Power Alliance*
SGIP	*Smart Grid Interoperability Panel*
SIWG	*Smart Inverter Working Group*
SNL	*Sandia National Laboratories*
SOA	*Service Oriented Architecture*
SSD	*Substation Specification Description*
TC	*Technical Committee*
TLS	*Transport Layer Security*
VEN	*Virtual End Node*
VTN	*Virtual Top Node*
WG	*Working Group*
XML	*Extensible Markup Language*

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A crescente demanda por energia (Bezerra et al., 2012), as perdas relacionadas à transmissão e distribuição (Eid et al., 2016) e o impacto ambiental das fontes tradicionais de energia (Blaabjerg et al., 2017) têm tornado as fontes Renováveis de Gerações Distribuídas (RDG, do inglês *Distributed Renewable Generation*) uma opção relevante para a expansão da rede elétrica (Feng et al., 2018). Dentre essas, sistemas fotovoltaicos (PV, do inglês *Photovoltaic*) são uma solução atrativa para instalações em pequena escala (IEA-PVPS, 2014).

O maior uso de RDG, entretanto, pode suscitar outros problemas, como aumento da tensão, injeção de harmônicos de corrente e falsos ilhamentos (Demirok et al., 2011). A superação de tais obstáculos requer o controle de um grande número de Recursos Distribuídos de Energia (DER, do inglês *Distributed Energy Resource*), e também representa uma oportunidade para melhorar a qualidade da energia e a confiabilidade da rede (Hatziargyriou et al., 2007; EPRI, 2015a). O conjunto de *hardware* e *software* destinado a realizar essa funcionalidade é denominado Sistema de Gerenciamento de Recursos Distribuídos de Energia (DERMS, do inglês *Distributed Energy Resource Management System*).

A implementação de tal sistema de controle depende de sistemas de comunicação e protocolos adequados para troca de dados com DERs e com o Sistema de Gerenciamento de Distribuição (DMS, do inglês *Distribution Management System*). A rede elétrica acrescida de sistemas de comunicação para aumentar suas funcionalidades é denominada de rede inteligente (*smart grid*) (GUNGOR et al., 2012). O desenvolvimento de *smart grids*, por sua vez, requer a criação de padrões que garantam a interoperabilidade entre equipamentos de diferentes fabricantes nos diversos níveis de gerenciamento de transmissão e distribuição energia (California Public Utilities Commission, 2017; SunSpec Alliance, 2016).

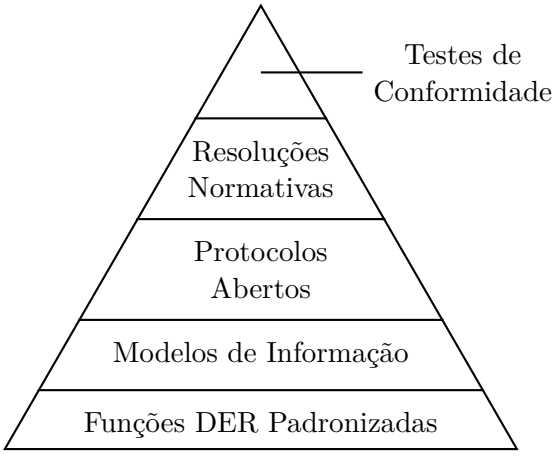


Figura 1 – Relação entre funcionalidades, modelos de informação, protocolos, regulamentações e testes de conformidade.

A Figura 1 apresenta a estratégia prevista por EPRI (2016b) para o desenvolvimento dessas tecnologias necessárias para o controle DERs baseados em inversores inteligentes. Inicialmente são realizados levantamentos a fim de descrever o conjunto de funcionalidades mínimas de um DER. Com base nesse grupo de funções padronizadas é possível construir os modelos de informação, isto é, o conjunto mínimo de informações necessárias para realizá-las. Somado aos requisitos de segurança e tempo real de cada parte do sistema de controle, são então adotados ou desenvolvidos protocolos de comunicação que descrevem como tais informações serão trocadas entre dispositivos. Para garantir a interoperabilidade entre fabricantes os protocolos desenvolvidos devem ser abertos (EPRI, 2016b; SunSpec Alliance, 2016).

Conforme novos produtos chegam ao mercado implementando as novas funcionalidades viabilizadas por tais protocolos, as agências reguladoras podem emitir resoluções normativas especificando novos requerimentos para a conexão destes dispositivos a rede elétrica. Por fim, testes de conformidade devem ser elaborados a fim de garantir um processo único e conciso para verificação do atendimento destes requisitos (EPRI, 2016b).

Em 2009, trabalhando na construção de demonstrações de *smart grids* de grande porte com foco na implantação de DERs, o *Electric Power Research Institute* (EPRI) identificou uma fragmentação dos protocolos utilizados por diferentes fabricantes e que não havia uma visão comum das funcionalidades que cada parte do sistema deveria prover. A fim de endereçar esse último problema, o instituto associou-se com o Departamento de Energia dos Estados Unidos da América, *Sandia National Laboratories* (SNL) e a *Smart Electric Power Alliance* (SEPA), resultando na elaboração do *Common Functions for Smart Inverters* (CSIP) (EPRI, 2016b).

Os resultados obtidos foram entregues à *International Electrotechnical Commission* (IEC) em contribuição ao desenvolvimento da norma IEC 61850 - Redes de Comunicação e Sistemas em Subestações - que define a comunicação entre Dispositivos Eletrônicos Inteligentes (IED, do inglês *Intelligent Electronic Device*) (IEC, 2013). Essa colaboração originou ainda em 2009 uma nova parte da norma destinada ao controle básico de DERs, a IEC 61850-7-420. A continuidade desses trabalho em conjunto com *Smart Grid Interoperability Panel* (SGIP), *Smart Inverter Working Group* (SIWG) e IEEE 1547.3 adicionou em 2013 uma nova parte a norma, IEC 61850-90-7, para o gerenciamento de DERs inteligentes (HEFNER et al., 2015).

Além de definir o seu próprio protocolo para transmissão dados, a norma IEC 61850 também define modelos de informações baseados em dispositivos físicos, dispositivos lógicos, nós e classes de dados (AMULYA et al., 2017). Diversos outros protocolos mapeiam tais modelos, como *Distributed Network Protocol* (DNP) 3.0 (IEEE. . . , 2016), SunSpec Modbus (SunSpec Alliance, 2015) e *Smart Energy Profile* (SEP) 2.0 (IEEE. . . , 2018b).

Com relação às resoluções normativas, cada país ou região possui suas próprias legislações. Citam-se como de especial relevância ao controle de DERs nos Estados Unidos as normas IEEE 1547 (IEEE..., 2018a) e Regra 21 (California Public Utilities Commission, 2017), e no Brasil os módulos 1 e 3 do Procedimentos de Distribuição (PRODIST) (ANEEL, 2018), atualizados pelas resoluções normativas 482 (ANEEL, 2012) e 687-2015 (ANEEL, 2015) da Agência Nacional de Energia Elétrica (ANEEL) para tratar geração distribuída, bem como as NBRs 16149 (ABNT, 2013) e 16274 (ABNT, 2014). De maneira semelhante, os testes de conformidade devem ser adequados às legislações locais, exemplificados com as normas como a IEEE 1547.1 (IEEE..., 2020) e NBR IEC 61850-10 (ABNT, 2018) para esses mesmos países, respectivamente.

Quanto ao sistema de comunicação com DMS, um caminho semelhante ao ilustrado pela Figura 1 deve ser seguido. Desde 2012, o EPRI tem trabalhado em um documento semelhante ao CSIP, intitulado *Common Functions for DER Group Management*, que está atualmente na sua terceira versão (EPRI, 2016a). Os próximos passos para a padronização, entretanto, ainda estão em desenvolvimento (RENJIT et al., 2019).

O modelo de dados a ser utilizado para comunicação entre centrais de controle é definido nas normas IEC 61968/61970, denominado Modelo de Informações Comum (CIM, do inglês *Common Information Model*), que deve ser harmonizado com a IEC 61850 para suporte a DERs (GREER et al., 2014). Após a consolidação desses modelos, o mapeamento para protocolos adequados pode ser feito, e em seguida a construção de resoluções normativas e testes de conformidade.

Desta forma, o desenvolvimento de um DERMS atualmente deve considerar o possível surgimento de novos requisitos e mudanças em padrões atuais, sendo necessário construir uma arquitetura interna ao DERMS que possa acomodar essas evoluções no futuro. O presente trabalho explora a utilização de uma Arquitetura Orientada a Serviços (SOA, do inglês *Service Oriented Architecture*) para a obtenção de tal flexibilidade.

SOA é um conceito arquitetural que promove reusabilidade, interoperabilidade, agilidade, flexibilidade e acoplamento fraco entre os componentes de um sistema, com foco em reduzir processos em blocos de tarefas e funcionalidades denominados serviços (NIKNEJAD et al., 2020). De maneira tradicional, serviços funcionam como módulos que expõem interfaces que são invocadas por mensagens (MARK, 2006 apud NIKNEJAD et al., 2020).

Dentre as formas de integração de serviços, a utilização de Barramentos de Serviços Empresariais (ESB, do inglês *Enterprise Service Bus*) tem se tornado um padrão *de facto*. Nesse modelo, os diversos serviços do sistema estão ligados indiretamente por meio de um barramento de serviços que realiza a transformação e roteamento de requisições, e pode também concentrar a validação das mensagens e autenticação das partes (Aziz et al., 2020). A utilização de um *middleware* como ESB também viabiliza comunicações

no formato publicar-assinar, possibilitando que parte do sistema seja orientada a eventos (PAPAZOGLOU; HEUVEL, 2007; FERNANDEZ; YOSHIOKA, 2011).

Apresentam-se a seguir os principais objetivos deste trabalho.

1.1 OBJETIVO

Desenvolver os sistemas de comunicação de um DERMS, para troca de mensagens com o DMS e com múltiplos DERs baseados em inversores fotovoltaicos, aplicando os conceitos de arquiteturas orientadas a serviços, observando as normas existentes e mantendo a flexibilidade para a implementação de normas futuras.

1.1.1 Objetivos Específicos

- Levantar as normas e trabalhos relevantes ao desenvolvimento do sistema de comunicação do DERMS;
- Propor uma arquitetura interna para o desenvolvimento do DERMS que promova a flexibilidade para atender normas futuras;
- Implementar os serviços necessários para comunicação DERMS-DER conforme as normas existentes;
- Implementar os serviços necessários para comunicação DMS-DERMS para atender da melhor forma possível as normas em desenvolvimento;
- Verificar o desempenho do sistema conforme parâmetros propostos pelas normas.

1.2 ESTRUTURA E ORGANIZAÇÃO DESTE TRABALHO

Este trabalho está dividido em cinco capítulos. Este primeiro, apresentou as motivações para evolução de *smart grids*e a integração de RDGs, bem como um breve histórico das normas relacionadas ao controle de DERs, os desafios à implementação de um DERMS atualmente e a proposta deste trabalho para resolver tais desafios.

O Capítulo 2 expande o histórico de normas apresentado, retoma os conceitos de SOA e ESB, apresenta algumas arquiteturas de referência encontradas na literatura para a implementação de DERMS, e detalha os protocolos de comunicação que serão utilizados. O Capítulo 3 elabora a arquitetura implementada e apesenta seu desenvolvimento, detalhando os serviços implementados e suas funcionalidades.

Por fim, o Capítulo 4 apresenta e discute os resultados obtidos, e o Capítulo 5 traz as conclusões deste trabalho.

2 REVISÃO DA LITERATURA

Neste capítulo, apresenta-se uma breve revisão sobre a SOA e ESB na Seção 2.1. Em seguida, a apresentação de normas e trabalhos técnicos relacionados ao gerenciamento de DERs é expandido na Seção 2.2, as duas principais arquiteturas de referência são apresentadas na Seção 2.3 e os protocolos a serem utilizados nesse trabalho na Seção 2.4.

2.1 ARQUITETURA ORIENTADA A SERVIÇOS

O termo Arquitetura Orientada a Serviços (SOA, do inglês *Service Oriented Architecture*) pode ser definido por diferentes perspectivas. De um aspecto de negócios, SOA é uma arquitetura conceitual em que a lógica da aplicação é disponibilizada a consumidores por meio de serviços, que podem ser compartilhados entre usuários (MARKS; BELL, 2008 apud NIKNEJAD et al., 2020). É uma forma de aumentar a agilidade nos negócios e reduzir custos, por meio da reutilização de aplicações e funcionalidades, aliado a respostas rápidas a mudanças de mercado e fácil integração entre sistemas (NIKNEJAD et al., 2020).

De um ponto de vista mais prático, SOA é um conceito arquitetural que suscita reusabilidade, interoperabilidade, agilidade, flexibilidade e acoplamento fraco entre os componentes de um sistema. Cada processo do negócio é dividido em blocos de funcionalidades, chamados de serviços, que devem ser bem definidos e organizados para funcionar como unidades independentes, que são conectadas para criar o processo unificado do negócio.

De outra forma, MacKenzie et al. (2006) define SOA como um paradigma para organizar e utilizar capacidades distribuídas entre diferentes entidades. Uma entidade do sistema capaz de realizar uma determinada funcionalidade expõem essa capacidade em uma interface para que outra entidade possa ter suas necessidades atendidas.

Nota-se ainda que SOA não traz nenhuma solução para problemas específicos do domínio da aplicação sendo desenvolvida, mas sim uma forma de organizar e entregar as soluções de maneira a extrair mais valor das capacidades de cada entidade do sistema. A arquitetura oferece uma forma de expressar soluções que facilita modificar e evoluir partes do sistema, ou testar partes alternativas (MACKENZIE et al., 2006).

Outros dois aspectos importantes do paradigma são a visibilidade e a interação entre as partes do sistema. A visibilidade permite que uma entidade que necessita de que uma determinada ação seja realizada possa identificar outras entidades capazes de realizá-la. Para tanto, costuma-se empregar alguma tecnologia que providencie uma descrição das funções que podem ser realizadas, dos requirimentos para a realização, das limitações, etc. Tal descrição deve utilizar um formato conhecido por todas as partes, a fim de garantir que a semântica dos serviçõe expostos seja compreendida (MACKENZIE et al., 2006).

Após identificar a entidade capaz que exponha a ação desejada, deve haver uma forma de interação entre as entidades. Tipicamente a interação é mediada por troca de mensagens ou invocação de métodos (MACKENZIE et al., 2006).

Enterprise Service Bus (ESB) é uma forma de garantir esses dois aspectos da implementação, e tem se tornado um padrão *de facto* em implementações SOA. Nesse modelo de comunicação, os diversos serviços do sistema estão ligados indiretamente por meio de um barramento de serviços que realiza a transformação e roteamento de requisições, e pode também concentrar a validação das mensagens e autenticação das partes (Aziz et al., 2020).

A utilização de um *middleware* como ESB também viabiliza comunicações no formato publicar-assinar, possibilitando, por exemplo, que parte do sistema seja orientada a eventos (PAPAZOGLOU; HEUVEL, 2007; FERNANDEZ; YOSHIOKA, 2011). Por fim, o *middleware* também pode ser utilizado para descoberta de serviços do sistema, contribuindo com o aspecto de visibilidade do sistema.

2.2 NORMAS RELACIONADAS AO GERENCIAMENTO DE RECURSOS DISTRIBUÍDOS DE ENERGIA

Desde 1995, o Grupo de Trabalho (WG, do inglês *Working Group*) 10 do Comitê Técnico (TC, do inglês *Technical Committee*) 57 da IEC tem desenvolvido a norma IEC 61850 (Redes de Comunicação e Sistemas em Subestações) (IEC, 2019), que define a comunicação entre Dispositivos Eletrônicos Inteligentes (IED, do inglês *Intelligent Electronic Device*) (IEC, 2013). Em 2004 o TC57 inaugurou o WG17 para o desenvolvimento de requisitos de comunicação e controle de DERs com IEC 61850.

Os primeiros resultados deste grupo compuseram uma nova parte da norma em 2009, IEC 61850-7-420, para controle básico de DERs. Em 2013, com esforços combinados do EPRI, SGIP, IEEE 1547.3 e SIWG, uma nova parte da norma foi publicada, IEC 61850-90-7, para o gerenciamento de DERs inteligentes (HEFNER et al., 2015). Além de definir um protocolo próprio para transmissão de dados, o modelo de informações definido por essa norma é adotado por diversos outros protocolos. Os detalhes do protocolo seu modelo de informações serão apresentados na Seção 2.4.1.

Para o Brasil, as principais normas relacionadas ao controle de DERs são os módulos 1 e 3 do PRODIST (ANEEL, 2018), atualizados pelas resoluções normativas 482-2012 (ANEEL, 2012) e 687-2015 (ANEEL, 2015) da ANEEL e as NBRs 16149 (ABNT, 2013) e 16274 (ABNT, 2014). Esses documentos, entretanto, não tratam dos sistemas de comunicações que são objeto desse trabalho e, portanto, não serão exploradas em mais detalhes. De forma semelhante, os testes de conformidades são descritos na NBR IEC 61850-10 (ABNT, 2018), que também não inclui os testes a serem feitos nos sistemas de comunicação.

Com relação ao sistema de comunicação entre DERMS e DMS, a padronização das funcionalidades mínimas de um grupo de DERs foi iniciada com *Common Functions for DER Group Management* (EPRI, 2016a). Os próximos passos, como a criação de normas, ainda estão em desenvolvimento (RENJIT et al., 2019).

Com a finalidade de definir os objetivos deste trabalho, os requisitos descritos pela norma IEEE 1547 com relação a DERs serão utilizados para o sistema completo, isto é, para a comunicação desde o DMS até o DER, tendo o DERMS como intermediário. Além de requisitar a devida proteção dos dados durante a transmissão, a norma define 30 s como tempo máximo para responder a leituras (IEEE. . . , 2018a). Outros documentos não normativos, como a lista de requisitos para DERMS apresentada por SEPA (2019), e documentos voltados a outros IEDs, como SGIP (2015), sugerem o tempo de resposta de 4 s para sistemas que operam em tempo real. O presente trabalho procurará atender esse requisito mais restritivo, entendendo que normas futuras podem vir a adotá-lo.

2.3 ARQUITETURAS DE GERENCIAMENTO DE RECURSOS DISTRIBUÍDOS DE ENERGIA

Diversas arquiteturas para a implementação de *smart grids*são propostas tanto em alianças de empresa (SEAL, 2011; CLEVELAND et al., 2014; MESA, 2016; OPENADR, 2017), quanto na literatura técnica disponíveis em artigos de conferências e revistas internacionais (GUNGOR et al., 2011; FREDERICK, 2017; IEEE, 2015). Duas arquiteturas são aqui apresentadas: a Arquitetura Hierárquica de 5 níveis proposta pelo SGIP (parte da SEPA), e a Arquitetura Recursiva proposta pelo EPRI.

Essas arquiteturas foram escolhidas por serem propostas por duas das principais organizações responsáveis pelo *Common Function for Smart Inverters*, principal descrição funcional de DERs baseados em inversores (EPRI, 2016b).

2.3.1 Arquitetura hierárquica de 5 níveis SGIP/SEPA

O SGIP foi criado em 2009 pelo *National Institute of Standards and Technology* (NIST) como uma parceria público-privada para o levantamento de requisitos e criação de especificações para *smart grids*. Em 2013 o SGIP foi transformado em uma instituição sem fim lucrativos. Por fim, em 2017 o SGIP foi incorporado à SEPA (NIST, 2019).

A arquitetura proposta por Cleveland et al. (2014), ilustrada na Figura 2, é voltada aos fabricantes e fornecedores de soluções de gerenciamento da indústria de DERs, buscando criar um modelo de referência a fim de garantir interoperabilidade entre seus diferentes níveis. Sua concepção hierárquica visa a escalabilidade, considerando a possibilidade da existência de milhares de sistemas de geração distribuída por toda a rede.

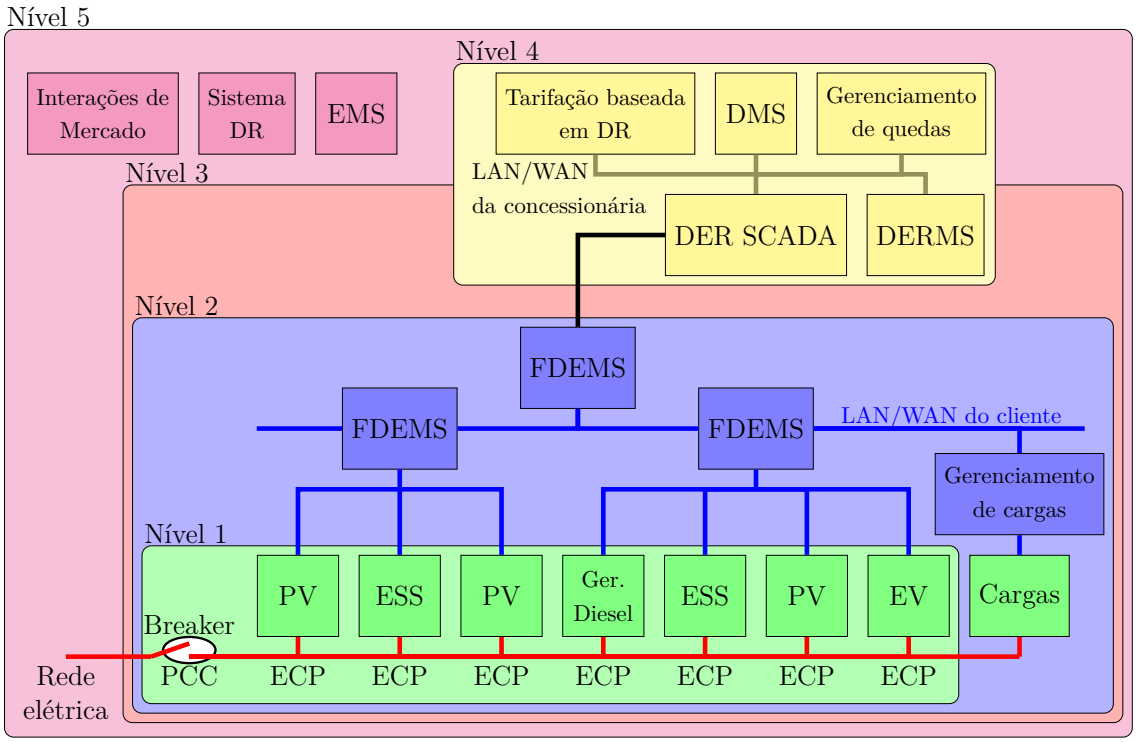


Figura 2 – Arquitetura hierárquica para sistemas DER.

Também considera-se que esses sistemas são gerenciados autonomamente em um nível local com relação às suas atividades de geração e armazenamento, conforme as condições locais, configurações pré-estabelecidas e preferências do proprietário do sistema. A seguir são descritos cada um dos cinco níveis, destacando-se quais domínios de uma rede inteligente podem ser regidos por cada um dos níveis.

Nível 1

No primeiro nível encontra-se os DERs, como sistemas fotovoltaicos, sistema de armazenamento de energia (ESS, do inglês *Energy Storage System*), veículos elétricos (EV, do inglês *Electrical Vehicle*) e geradores diesel. A interação do DER com a rede é amplamente influenciada pelo tipo de conversor utilizado. Classificam-se três tipos de conversores: geradores síncronos, geradores assíncronos (ou indutivos) e geradores estáticos (ou eletrônicos) (BASSO; DEBLASIO, 2004).

ESS é um sistema capaz de absorver energia, armazenando-a por algum tempo e posteriormente despachando-a segundo alguma função de controle pré-definida. A forma de armazenamento de energia pode ser, por exemplo, pelo uso de baterias ou de supercapacitores. A função principal do ESS é a mediação entre as fontes variáveis de energia e as cargas variáveis da rede (IEEE, 2015).

No nível local, esses sistemas devem ser gerenciados autonomamente em relação as suas atividades de geração e armazenamento de energia conforme as condições locais.

Entretanto, os sistemas DER participam ativamente da rede, sendo necessária a coordenação com os demais sistemas DER existentes e com o sistema de transmissão regional.

Múltiplos DERs se conectam à rede local por meio de Pontos Elétricos de Conexão (ECP, do inglês *Electrical Connection Point*) e à rede da concessionária por meio de Pontos de Acoplamento Comum (PCC, do inglês *Point of Common Coupling*). Para prover as funcionalidades básicas ao sistema, esse nível pode utilizar os protocolos Modbus ou SEP1.x (CLEVELAND et al., 2014).

Para prover funcionalidades avançadas à rede inteligente, o padrão IEC 61850 deve ser seguido. Mais especificamente o modelo de objetos descrito pelo padrão IEC 61850-7-420 e IEC 61850-90-7. Neste contexto, os protocolos SEP2.0, DNP3, *Manufacturing Messaging Specification* (MMS) ou um *web service* podem ser utilizado para mapear o modelo e prover as comunicações nesse nível (CLEVELAND et al., 2014).

Nível 2

No segundo nível encontra-se o Sistema de Gerenciamento DER para Clientes (FDEMS, do inglês *Facility DER Management System*). Esse sistema possivelmente estará gerenciando um ou dois DERs em uma aplicação residencial, mas pode gerenciar múltiplos DERs em uma aplicação comercial ou industrial, como o *campus* de uma universidade ou um *shopping*. Esse tipo de sistema também pode ser utilizado pela concessionária para gerenciar sistemas DER presentes em suas instalações, como em subestações e usinas.

O FDEMS também deve ser capaz de gerenciar uma *microgrid* ilhada em caso de falhas na rede da concessionária, incluindo o desligamento de algumas cargas se necessário ou a ativação de outros sistemas DERs. Dentro desse nível é possível ainda a existência de uma hierarquia interna com um FDEMS gerenciando múltiplos FDEMSs. Um exemplo disso seria a instalação em um *campus*, que pode ter um FDEMS em cada prédio que se conectam a um FDEMS principal que tem a conexão com os sistemas do nível superior.

Para prover funcionalidades básicas à rede inteligente, protocolos como SEP1.x, Modbus e OASIS *Energy Market Information Exchange* (eMIX) podem ser utilizados. Para as funcionalidades avançadas da rede, novamente o modelo de objetos do padrão IEC 61850 deve ser seguido e os protocolos SEP2.0, DNP3, MMS além o uso de *Web Services* são opções viáveis para implementar as comunicações desse nível (CLEVELAND et al., 2014).

A descrição usada por Cleveland et al. (2014) de um FDEMS incorpora as funcionalidades atualmente atribuídas ao controlador de *microgrid*, como controle de ilhamento, e DERMS. Em documentos mais recentes publicados pela SEPA sobre o controle de DERs, o conceito de FDEMS tem sido substituído pela separação desses dois sistemas (SEPA, 2019).

Nível 3

O terceiro nível estende o controle local a um escopo global e permite que a concessionária ou o agregador faça requisições aos controladores DER para realizar ações específicas, como ligar, desligar, limitar a saída e outras funcionalidades de gerenciamento da rede. Essas requisições serão geralmente atreladas a cobrança ao cliente, mas também podem estar ligadas às funções de proteção da rede.

Este nível têm os requisitos de tempo real do sistema de distribuição e transmissão, assim como requisitos relacionados a resposta à demanda de preços do mercado.

O sistema desse nível, possivelmente um Sistema de Supervisão e Aquisição de Dados (SCADA, do inglês *Supervisory Control and Data Acquisition*), deve se comunicar com centenas ou até milhares de sistemas DER, sendo necessário adotar algumas técnicas para viabilizar as operações, como monitorar o PCC da rede ao invés de requisitar dados constantemente de todos os DERs, agregar múltiplos DERs em grandes controladores DER a fim de minimizar o número de pontos monitorados ou utilizar outras formas de monitorar a rede na subestação de distribuição.

O grande número de entidades envolvidas também pode inviabilizar o controle direto dos DERs, sendo que muitos dos comandos enviados deverão ser por meio de mensagens de *broadcast* ou *multicast*, como os alertas de situações emergenciais, informações de preço e demanda e atualização nas configurações para ações autônomas dos DERs.

Para prover a comunicação neste nível, protocolos como MMS, DNP3, OASIS eMIX, OpenADR ou *Web Services* podem ser utilizados com mapeamentos para o padrão IEC 61850 (CLEVELAND et al., 2014).

Nível 4

No quarto nível reside o Sistema de Gerenciamento de Distribuição (DMS, do inglês *Distribution Management System*), utilizado pela concessionária para monitorar e gerenciar a rede como um todo, operando com requisitos de tempo real. Neste nível são determinadas as requisições e comandos a serem enviados para os sistemas DER. A concessionária deve monitorar a eficiência e a confiabilidade da rede a fim de verificar se estas podem ser melhoradas com a modificação da operação dos sistemas DER. As ações planejadas pelo Gerenciador de Sistemas DER (DERMS, do inglês *Distributed Energy Resource Management System*) são então enviadas ao sistema SCADA do nível inferior para serem traduzidas em comandos enviados aos sistemas DER.

Para prover a comunicação neste nível, protocolos como OPC UA e MultiSpeak podem ser utilizados. A segurança do protocolo OPC UA é definida pelo padrão IEC 62541 e MultiSpeak apresenta mecanismos de segurança a partir da versão 4 (CLEVELAND et al., 2014).

Nível 5

O quinto nível é o mais alto, em que se encontram as operações do mercado de energia, e informações sobre os sistemas DERs disponíveis podem ser necessárias para algumas operações como transmissões regionais (CLEVELAND et al., 2014).

2.3.2 Arquitetura recursiva EPRI

EPRI (2015b) apresenta uma arquitetura multinível baseada em Nós Virtuais Superiores (VTN, do inglês *Virtual Top Node*) e Nós Virtuais Finais (VEN, do inglês *Virtual End Node*), como mostra a Figura 3. Diferente da arquitetura anterior, essa proposta não limita o número de níveis hierárquicos e privilegia uma abordagem de "informar a motivar" no lugar do controle direto dos ativos da rede.

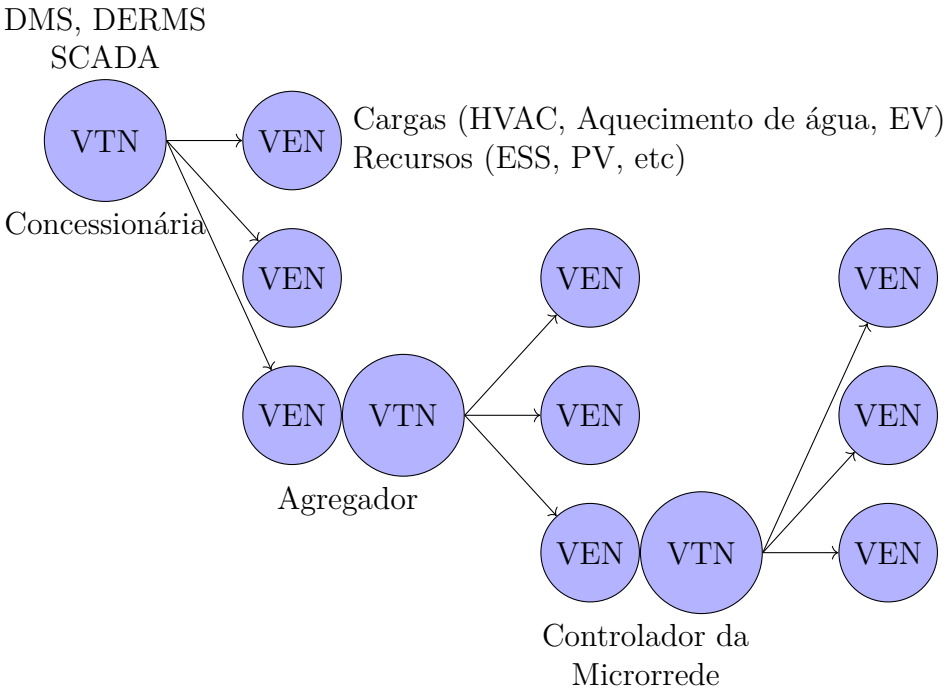


Figura 3 – Arquitetura recursiva baseada em VTN e VEN proposta por EPRI.

Idealmente todo o sistema poderia utilizar um único protocolo, mas na prática diferentes protocolos são utilizados a depender do tipo DER sendo controlado e do nível de agregação. Em níveis mais baixos, por exemplo, SunSpec Modbus pode ser utilizado para controle direto de um DER baseado em PV. Em níveis intermediários, entre a rede do cliente e da concessionária, o protocolo IEC 61850 ou algum de seus mapeamentos, como DNP3 e SEP2.0, seriam aplicados. Por fim, internamente a concessionária e para operações no mercado de energia, os protocolos IEC 60870 e IEC 61970 já são utilizados para interações com o sistema SCADA e demais componentes do *Energy Management System* (EMS), respectivamente (EPRI, 2015b).

Uma visão desta arquitetura baseada na concessionária é apresentada na Figura 4, ilustrando também a relação entre DERMS e o controlador da *microgrid*. É importante notar que o termo DERMS é utilizado por EPRI (2015b) de uma maneira mais genérica do que Cleveland et al. (2014) – tanto o gerenciador posicionado dentro da concessionária quanto o controlador de um grupo de DERs é referenciado como DERMS.

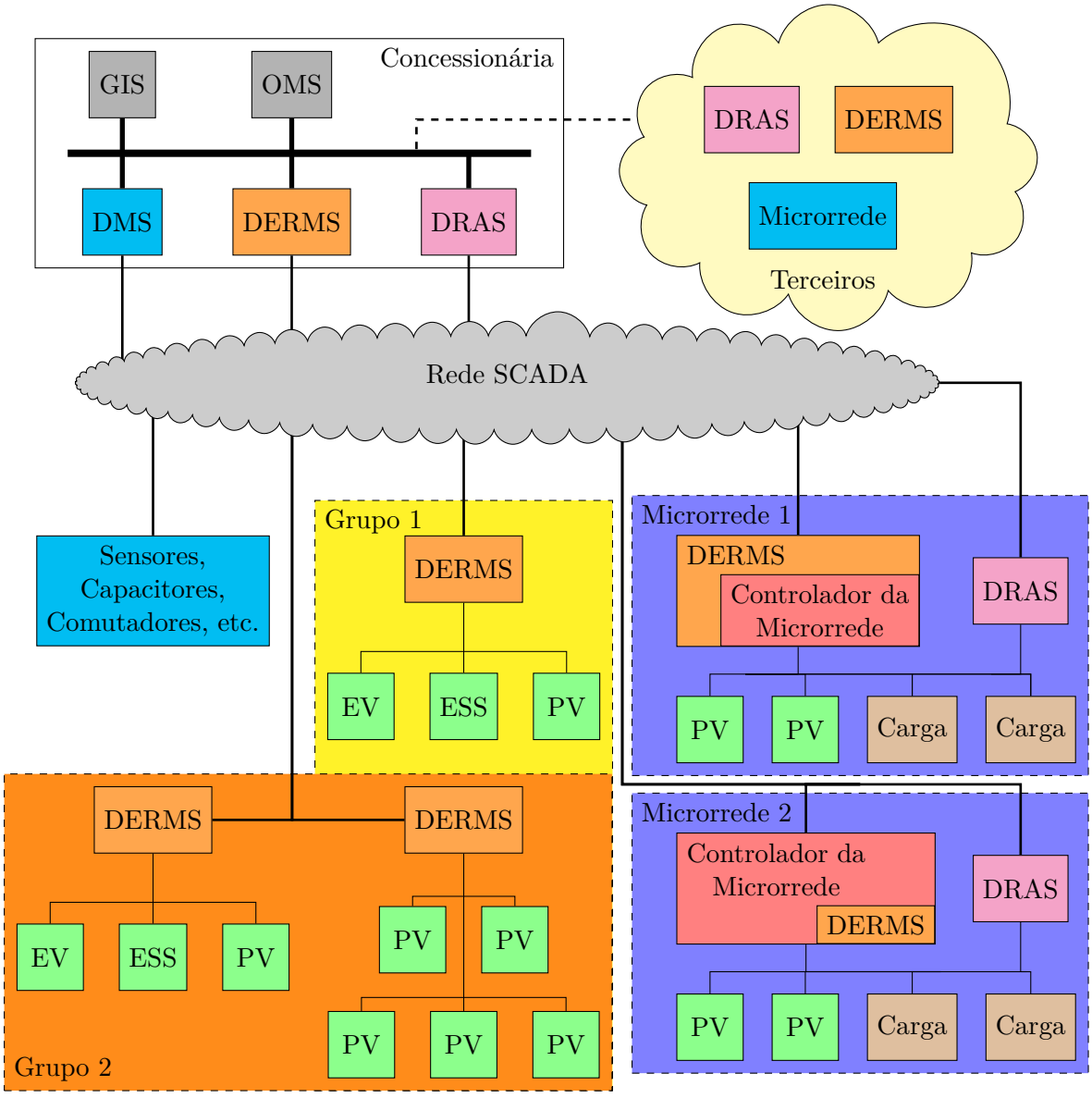


Figura 4 – Visão da concessionária da arquitetura recursiva proposta por EPRI.

A arquitetura também considera a construção de grupos de DERMS, não necessariamente disjuntos, conforme diferentes critérios, como localização, tipo de DER sendo controlado, capacidades ou ainda DERMS que pertençam a um mesmo proprietário (EPRI, 2015b). Por fim, caso o sistema tenha a capacidade de operar ilhado, é indicado a possibilidade do DERMS possuir internamente um controlador de *microgrids* ou estar dentro de um controlar de *microgrid*, representado na Figura 4 pelas *microgrids* um e dois, respectivamente.

2.4 PROTOCOLOS PARA GERENCIAMENTO DE RECURSOS DISTRIBUÍDOS DE ENERGIA

Existem inúmeros protocolos que podem ser utilizados para as comunicações necessárias em *smart grids*. Alguns protocolos são definidos para este uso por normas, como o IEC 61850, enquanto outros são desenvolvidos por alianças da indústria, como o SunSpec Modbus e OpenADR. Nas seções a seguir serão detalhados alguns destes protocolos.

2.4.1 IEC 61850

Endereçando os problemas de interoperabilidade de IEDs, o protocolo IEC 61850 surgiu da combinação dos padrões UCA 2.0 e IEC 60870-5-103, tendo as primeiras partes do padrão publicadas em 2004. O protocolo pode ser baseado tanto em redes TCP/IP quanto diretamente sobre Ethernet e define um conjunto regras e de nós lógicos para Sistemas de Automação de Subestações (SAS, do inglês *Substation Automation System*) (AMULYA et al., 2017).

Originalmente, a comunicação entre dispositivos de uma aplicação SAS era feita por conexões diretas de fios de cobre entre os dispositivos (AMULYA et al., 2017). A utilização de Ethernet possibilita uma grande redução na quantidade de fios necessária, e a utilização de TCP/IP possibilita o roteamento de informações entre diversas redes.

Diferentemente dos protocolos antecessores, IEC 61850 define não apenas como os dados devem ser transmitidos, mas também um modelo de organização dos dados, de forma que diferentes fabricantes ainda organizarão as informações de seus dispositivos da mesma forma. O modelo de dados consiste em dispositivos físicos, dispositivos lógicos, nós lógicos e classes de dados (AMULYA et al., 2017).

Cada dispositivo físico possui um ou mais dispositivos lógicos, cada dispositivo lógico possui um ou mais nós lógicos, que possui conjunto predefinido de classes de dados, em que os dados a respeito do dispositivo são armazenados. A interoperabilidade se dá pela padronização das informações de cada nó lógico, de forma que integração de diferentes sistemas é facilitada (AMULYA et al., 2017).

O padrão IEC 61850-6-1 define também uma Linguagem de Configuração de Subestações (SCL, do inglês *Substation Configuration Language*), representada em quatro possíveis formatos: *Substation Specification Description* (SSD), *IED Capabilities Description* (ICD), *Configured IED Description* (CID) e *Substation Configuration Description* (SCD) (AMULYA et al., 2017).

A utilização de várias camadas para comunicação pode afetar a velocidade de comunicações emergenciais. Para resolver este problema o protocolo define um serviço de mensagens baseado diretamente em Ethernet denominado *Generic Object Oriented Substation Event* (GOSE), uma forma rápida e confiável de distribuição de informações. O serviço dispõe de dois tipos de mensagens, *status* e analógicas, que são baseadas em mudanças no estado de conjuntos de dados (AMULYA et al., 2017).

Outros protocolos possuem mapeamentos para o modelo de dados definidos pelo padrão IEC 61850, como MMS (SCHWARZ; EICHBAEUMLE,) e DNP3 (IEEE..., 2016).

2.4.2 Modbus

Inicialmente desenvolvido para comunicações seriais (RS-232 e RS-485) (Modbus Org,), Modbus foi desenvolvido pela Modicon em 1979 como um protocolo aberto e sem royalties visando sistemas de automação industrial (Modbus FAQ, 2017b). A empresa americana, fundada em 1968, tornou-se parte da Gould Electronic em 1977, e foi vendida para a alemã AEG em 1989. Em 1994, a AEG se juntou a empresa Groupe Schneider, da França, que mudou de nome para Schneider Electric em 1999 (Schneider Eletric, 2019).

No mesmo ano, a especificação do protocolo para redes IP foi desenvolvido, denominado Modbus TCP. A mudança de redes seriais para uma rede de pacotes possibilitava o uso de Ethernet, que provia escalabilidade, comunicações a 10/100 Mbps e simplificava a integração com outros sistemas (Modbus FAQ, 2017b). Em 2004 o protocolo foi entregue à Modbus Organization, um grupo independente, sem fins lucrativos de usuários do protocolo, que passou a gerência a evolução do Modbus (Modbus FAQ, 2017a).

O protocolo define comunicações do tipo requisição/resposta e *broadcast* e cada transação Modbus é constituída de um único quadro de requisição, resposta ou *broadcast*. A Figura 5 mostra a distribuição dos dados em um quadro Modbus para redes seriais, também chamado Modbus RTU.

Modbus TCP empacota o quadro Modbus RTU em um quadro TCP, omitindo o campo de endereço do escravo, pois este já é o próprio endereço IP de destino, e o campo de *Cyclic Redundancy Check* (CRC), que já é realizado pelo protocolo TCP. Também é adicionado o campo "Unidade", que possibilita identificar múltiplos escravos em um mesmo endereço IP, possibilitando a operação de um terminal como um *proxy* entre

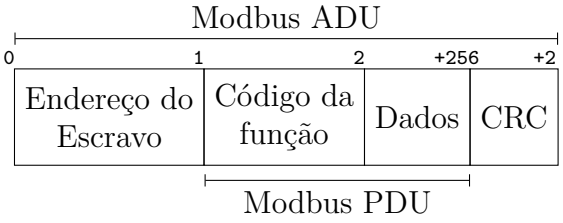


Figura 5 – Quadro Modbus RTU.

escravos que utilizam Modbus RTU e um mestre que utiliza Modbus TCP. A distribuição dos campos no quadro Modbus TCP pode ser vista na Figura 6.

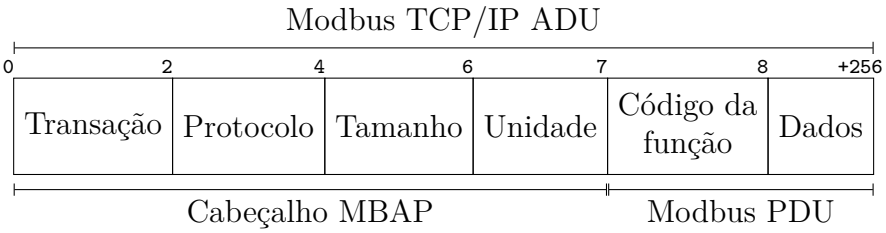


Figura 6 – Quadro Modbus TCP.

Comum a ambos os quadros, os campos de Código da Função e Dados compõem o Modbus Protocol Data Unit (PDU), que implementa o Protocolo de Aplicação Modbus. Essa camada de aplicação específica comunicações no formato requisição-e-resposta, em que um escrevo provê serviços identificados por códigos de função que variam de 1 a 127.

O Modelo de Dados Modbus define quatro tabelas com conteúdos de diferentes tamanhos e propriedades de acesso, apresentadas na Tabela 1. Cada tabela possui até 65.535 elementos, que são acessados e manipulados por códigos de função específicos, conforme exemplificado na Tabela 2.

Tabela	Acesso	Dado
Bobina	R/W	Um bit
Entrada discreta	R	Um bit
Registrador de entrada	R	16 bits
Registrador	R/W	16 bits

Tabela 1 – Tipos de dados do protocolo Modbus.

Ler bobinas	0x01
Ler entradas discretas	0x02
Ler registradores	0x03
Ler registradores de entrada	0x04
Escrever em uma bobina	0x05
Escrever em um registrador	0x06
Ler código de exceção	0x07
Escrever em múltiplas bobinas	0x0F
Escrever em múltiplos registradores	0x10
Ler ID do escravo	0x11
Escrever em registrador com máscara	0x16
Escrever e ler em múltiplos registradores	0x17

Tabela 2 – Exemplos de códigos de funções do protocolo Modbus.

SunSpec Modbus

O protocolo Modbus não define o significado dos dados em cada registrador ou bobina e, portanto, um *Master Terminal Unit* (MTU) Modbus deve saber previamente como estão mapeadas as informações nos endereços de um escravo para obter informações deste ou mandar comandos. A descrição deste mapeamento é chamado de perfil Modbus (SMA, 2015). Para aplicações em sistemas fotovoltaicos, a SunSpec Alliance define Modelo de Informação para compor tal perfil.

Além do mapeamento das informações, são definidos novos tipos de dados estruturados sobre registradores de 16 bits:

- `acc16`, `acc32` e `acc64`: Valores acumulados de 32 e 64 bits. Utilizados para valores sequencialmente crescentes, estritamente positivos. O bit mais significativo é utilizado para sinalizar *overflow*.
- `bitfield16` e `bitfield32`: Combinação de bits individuais. O significado de cada bit é descrito pelo perfil. Se o bit mais significativo estiver ativado, os demais devem ser ignorados.
- `enum16`, `enum32`: Código numérico de 16 e 32 bits especificado pelo perfil.
- `int16` e `int32`: Inteiro com sinal de 16 e 32 bits.
- `string`: Vetor de caracteres terminado com '\0'
- `sunssf`: Fator de escala. Utilizado como expoente de base dez para multiplicar o valor de outros dados do perfil.
- `uint16`, `uint32` e `uint64`: Inteiros sem sinal de 16, 32 e 64 bits.
- `float32`: Valores de ponto flutuante conforme IEEE... (2019)
- `ipaddr` e `ipv6addr`: Endereços IPv4 e IPv6 de 32 e 128 bits, respectivamente.

Cada tipo possui um valor para indicar erro, o *not-a-number* (NaN). O significado exato de um NaN pode variar conforme o dado sendo representado, podendo indicar um parâmetro não configurado, uma medição não implementada pelo equipamento ou um erro no sistema. A Tabela 3 apresenta os valores de NaN para cada tipo.

O perfil Modbus de um dispositivo é formado por dois ou mais Modelos de Informação. Cada modelo inicia com um identificador único e seu tamanho, seguido por um ou mais blocos. Um bloco é composto por uma coleção de pontos, que são valores codificados conforme os tipos de dados SunSpec. Blocos fixos possuem pontos únicos nos

Tipo	Tamanho (bits)	NaN
`acc16`	16	0x0000
`int16`		0x8000
`uint16`		0xFFFF
`bitfiled16`		0xFFFF
`enum16`		0xFFFF
`acc32`	32	0x0000 0000
`int32`		0x8000 0000
`uint32`		0xFFFF FFFF
`bitfield32`		0xFFFF FFFF
`enum32`		0xFFFF FFFF
`acc64`	64	0x0000 0000 0000 0000
`int64`		0x8000 0000 0000 0000
`ipaddr`	32	0.0.0.0
`ipv6addr`	128	::0
`string`	Múltiplo de 16	"\0\0"

Tabela 3 – Valores de NaN para os diferentes tipos de dados SunSpec.

modelos, enquanto blocos de repetição podem possuir múltiplas instâncias de seus pontos. A definição dos modelos padronizados é distribuída em formato *Extensible Markup Language* (XML) (SunSpec Alliance, 2015).

Os modelos do perfil são mapeados nos registradores conforme ilustrado pela Figura 7. Preferencialmente, o mapeamento deve iniciar no registrador 40.000 e, alternativamente, em 50.000 ou 0. Os valores 0x5375 e 0x6e53, que formam a *string* 'SunS', aparecem no início do mapa para identificar que o dispositivo é compatível com o padrão SunSpec. O mapeamento segue com o Modelo Comum, obrigatório a todos os dispositivos SunSpec, e então os demais modelos padronizados implementados pelo equipamento. Em seguida, o fornecedor pode inserir modelos próprios para funcionalidades não descritas pelos modelos SunSpec. O fim do mapa é identificado pelo Modelo Final, que possui tamanho zero e identificador 0xFFFF (SunSpec Alliance, 2015).

Em uma aplicação típica, o MTU Modbus estaria conectado ao barramento serial ou rede local com múltiplos *remote terminal units* (RTUs) Modbus. Para comunicações seriais, o MTU pode utilizar a função "Reportar ID do Escravo" (0x11) em *broadcast* para descobrir o endereço de todos os RTUs presentes. Para Modbus TCP essa função não está disponível e, portanto, o MTU deve possuir conhecimento prévio dos dispositivos daquela rede ou utilizar outro protocolo para descobri-los.

Quando um novo RTU é descoberto, o MTU irá escanear os possíveis endereços-base do mapeamento de modelos, buscando pela *string* 'SunS'. Se encontrada, os registradores seguintes são lidos, esperando encontrar o cabeçalho com ID do Modelo Comum

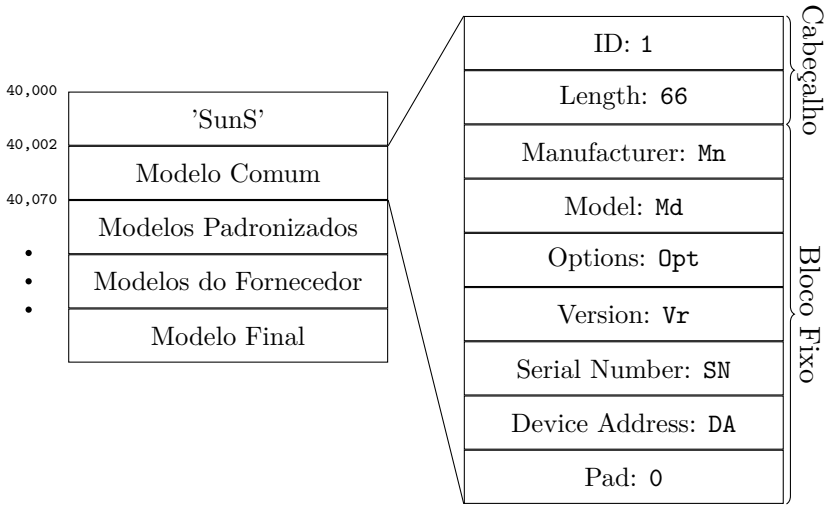


Figura 7 – Mapeamento dos Modelos de Informação SunSpec em Registradores Modbus.

e seu tamanho. Esse tamanho é somado ao endereço do último registrador lido para encontrar o cabeçalho do próximo modelo. O processo é repetido até encontrar o Modelo Final. Como resultado desse procedimento, o RTU possuirá o perfil desse MTU e poderá começar a enviar comandos e realizar leituras.

2.4.3 DNP3

O protocolo DNP3 foi criado em 1993 pela GE Harris Canada baseado na então incompleta norma IEC 60870-5 como um protocolo completo e aberto para atender as demandas do mercado imediatamente para aplicações SCADA e de controle, com foco em sistemas de energia. Em novembro do mesmo ano a empresa entregou ao DNP3 User Group a propriedade sobre o protocolo e a responsabilidade pela evolução e novas especificações deste.

O protocolo foi desenvolvido para transportar dados por comunicações seriais ou sobre redes IP. Essencialmente, o protocolo define as camadas física, de enlace e de aplicação do modelo OSI e uma pseudo-camada de transporte para tratar a fragmentação dos dados, quando necessário. A camada física do protocolo foi concebida para comunicações ponto-a-ponto serial como redes RS485 ou para comunicações em redes de rádio que trabalham com *broadcast* (IEEE. . . , 2012).

A camada de aplicação do protocolo DNP3 representa os dados de uma *outstation* por meio de Pontos. Cada Ponto pode ser unicamente identificado por seu Índice e Tipo,

Tipo	Acesso	Descrição
Analog Input	R	Valores representados por inteiros de 16 ou 32-bits, com ou sem sinal, ponto flutuante de precisão simples ou dupla. Pode possuir uma *flag* para reportar a validade dos dados, um limiar para reportar eventos e uma estampa de tempo.
Analog Output	R/W	Valores representados por inteiros de 16 ou 32-bits, com ou sem sinal, ou ponto flutuante de precisão simples ou dupla. Pode possuir uma *flag* para reportar a validade dos dados e uma estampa de tempo.
Binary Input	R	Valores booleanos. Pode possuir uma *flag* para reportar a validade dos dados e uma estampa de tempo.
Binary Output	R/W	Valores booleanos. Pode possuir uma *flag* para reportar a validade dos dados e uma estampa de tempo.
Counter	R	Valores monotonicamente crescentes, representados por inteiros de 16 ou 32-bits. Pode possuir uma *flag* para reportar a validade dos dados e uma estampa de tempo.
Octet String	R/W	Bloco de bytes. Pode ser usado para transmitir *strings* ASCII, *dumps* de memória, etc.

Tabela 4 – Exemplos de tipos de dados do protocolo DNP3.

Função	Código	Descrição
Confirm	0	Utilizado para confirmar o recebimentos de mensagens da *outstation*.
Read	1	Utilizado para requisitar dados da *outstation*.
Write	2	Utilizado para copiar dados do mestre para a *outstation*.
Select	3	Parte do processo em duas etapas para modificar um Ponto, "*select-before-operate*". A resposta da *outstation* deve possuir exatamente os mesmos Objetos da requisição para que a operação possa prosseguir.
Operate	4	Segunda parte do processo "*select-before-operate*". Ao receber a requisição, a *outstation* deverá verificar se houve uma requisição de Select anteriormente, e se essa requisição possuía exatamente os mesmos pontos da requisição atual. Do contrário, a operação deverá ser abortada.
Direct Operate	5	Quando a redundância do processo de "*select-before-operate*" não é necessária, o mestre DNP3 pode utilizar esse código para modificar Analog Outputs e Binary Outputs.
Immediate Freeze	7	Cria uma cópia o valor atual do ponto, o valor congelado.
Freezee-at-Time	11	Agenda a criação de valores congelados.
Enable Unsolicited	20	Autoriza a *outstation* a enviar Respostas Não Solicitadas para um conjunto de Pontos ou Classe.
Assign Class	22	Atribui uma Classe a um Ponto.

Tabela 5 – Exemplos de Código de Função do protocolo DNP3.

exemplificados na Tabela 4. Um mestre DNP3 pode operar sobre os Pontos por meio de Códigos de Função. A Tabela 5 apresenta uma breve descrição de alguns códigos.

O protocolo também apresenta o conceito de Eventos, que fazem com que a *outstation* armazene o valor do Ponto em decorrência algum fato relevante, como, por exemplo, um horário, uma leitura analógica que ultrapassa um determinado limiar ou uma requisição com código 7 do mestre DNP3. Caso o mestre tenha habilitado Respostas Não Solicitadas para o Ponto, a *outstation* pode iniciar uma comunicação para reportar o Evento. Do contrário, os dados do Evento devem ser armazenados até que mestre realize um *polling* por Eventos do Ponto.

Para transmitir o valor atual de um Ponto (chamado de Estático pelo protocolo), ou de um Evento associado ao Ponto, um Objeto DNP3 deve ser criado. Além do Tipo e Índice, o Objeto carrega as informações de Grupo e Variação. O Grupo indica se o dado representa o valor Estático ou o tipo de Evento que o gerou. A Variação indica como o valor está codificado (com ou sem sinal, com ou sem estampa de tempo, quantidade de bits, etc.). A Tabela 6 exemplifica algumas combinações de Grupo e Variação. Após a transmissão, o Objeto DNP3 deve ser armazenado até que a outra parte da comunicação confirme o recebimento.

Tipo	Grupo	Variação	Descrição
Binary Input	1	1	Conjunto de Binary Inputs empacotadas
		2	Uma Binary Input com *flags* de qualidade
Binary Output	11	1	Mudança no valor de uma Binary Output
		2	Mudança no valor de uma Binary Output com *flags* de qualidade e estampa de tempo
Analog Input	30	1	Valor estático representado como inteiro de 32-bits com sinal
		4	Valor estático representado como inteiro de 16-bits com sinal e *flag* de qualidade
		5	Valor estático representado em ponto flutuante de precisão simples, com *flag* de qualidade
		6	Valor estático representado em ponto flutuante de precisão dupla, com *flag* de qualidade
	33	3	Valor congelado representado como inteiro de 32-bits com sinal, *flag* de qualidade e estampa de tempo
Analog Output	43	5	Comando sobre uma Analog Output, representado em ponto flutuante de precisão simples

Tabela 6 – Exemplos combinações de Grupo e Variação.

O protocolo ainda define quatro Classes, utilizadas para organizar dados Estáticos e de Eventos. O mestre DNP3 pode requisitar que um ponto seja associado a uma

determinada Classe (com o Código de Função 22), e mais tarde realizar o *polling* de todos os dados de uma Classe. A Classe 0 é reservada para valores Estáticos, enquanto as demais Classes são reservadas para Eventos.

Finalmente, a norma IEEE 1815 define quatro níveis de implementação do protocolo que especificam um grupo mínimo de Códigos de Função, Grupos e Variações que devem suportados por uma determinada implementação do protocolo. Cada nível inclui todos os requisitos do nível anterior, e procedimentos de testes são especificados. Ao ser submetido a testes em laboratórios associados do DNP3 User Group, um produto que utiliza o protocolo pode receber a certificação para um desses níveis.

Nota de Aplicação 2018-001

DNP Users Group (2019) define o perfil de uma *outstation* DNP3 para aplicações DERs utilizando o modelo de dados IEC 61850. Embora o nome do documento sugira um caráter informativo, a especificação é considerada normativa para fins de interoperabilidade, incluindo três níveis de implementação do perfil proposto e procedimentos para teste de conformidade. A implementação DNP3 que utiliza essa Nota de Aplicação deve ser compatível com ao menos o nível dois do protocolo.

O perfil assume que a *outstation* é um *proxy* de um ou mais DERs, que existe ao menos um medidor no sistema e que cada DER possui ao menos uma fonte de energia e um inversor inteligente. São definidas quatro categorias de pontos:

- SCADA e Configuração: Pontos relacionados ao controlador dos DERs, como funções suportadas e a caracterização dos limites de operação de cada DER, e leituras do medidor do sistema. Essa categoria é exigida em todos os níveis de implementação.

- Pontos de Agendamento: Pontos que permitem programar a alteração de parâmetro do sistema com base nas condições de operação, como curvas Volt-VAr e Volt-Freq. Essa categoria é exigida a partir do segundo nível de implementação.

- Pontos de Histórico: Medições detalhadas que podem servir para análise do desempenho do sistema ao longo do tempo, mas não são imediatamente necessárias para controle em operações normais. Essa categoria só é exigida para o terceiro nível de implementação.

- Pontos do Fornecedor: Pontos adicionais definidos pelo Fornecedor do sistema. Qualquer nível pode incluir Pontos do Fornecedor no final do mapeamento.

Cada categoria possui um ou mais blocos de Tipos DNP3 que devem aparecer na ordem definida pela Nota de Aplicação. Para representar múltiplos dispositivos, alguns blocos são repetidos, como os Pontos de Históricos relacionados a inversores. Outros

blocos, como o de SCADA e Configuração, são mapeados uma única vez no perfil. A Tabela 7 apresenta um resumo dos blocos definidos pela norma. A Figura 8 ilustra como esses blocos se organizam nos Índices de um Tipo DNP3 qualquer.

		Quantidade de Pontos				
Categoria	Repetição	BI	BO	Count.	AI	AO
SCADA e Configuração	Bloco único	102	38	4	534	432
Agendamento	Bloco único	9	8		12	9
	Para cada configuração de agendamento				2	2
	Para cada agendador				3	
Histórico	Para cada medidor	13		4	37	12
	Para cada DER	4			14	
	Para cada inversor	35			33	12
	Para cada banco de baterias	54			27	3

Tabela 7 – Resumo dos blocos de Pontos definidos pela Nota de Aplicação 2018-001.

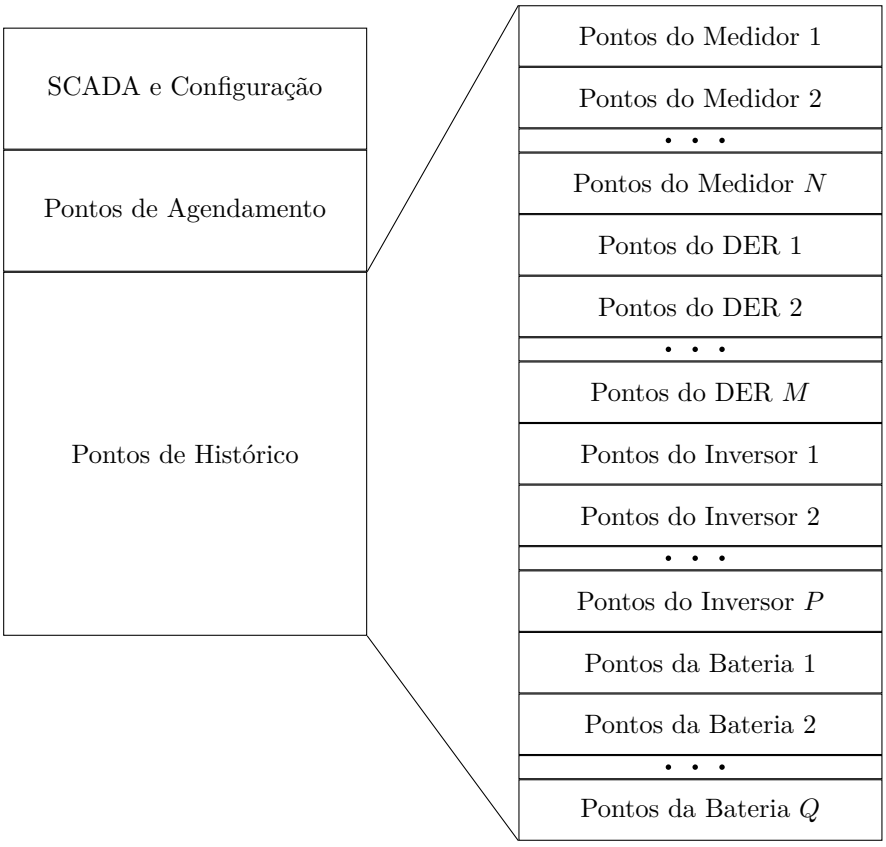


Figura 8 – Organização dos Pontos do perfil de um *outstation* com N medidores, M DERs, P inversores e Q bancos de baterias.

A Nota de Aplicação ainda define o uso da Classe 1 para reportar Eventos críticos e alarmes, da Classe 2 para Eventos de *feedback* e da Classe 3 para Eventos de medidas e

configurações. Por fim, o documento apresenta uma relação entre os Pontos mapeados e as funções descritas por EPRI (2016b).

2.5 CONSIDERAÇÕES FINAIS

Este capítulo retomou o histórico de normas do Capítulo 1 para detalhar as normas relevantes para a comunicação entre DERs e DERMS, e entre DERMS e DMS. Para esses últimos, observou-se que o processo de padronização das funcionalidades mínimas ainda está em curso, sendo necessário adaptar os requisitos de outras normas para possibilitar a definição dos objetivos deste trabalho.

Também foram apresentados os protocolos que serão utilizados nesse trabalho, Modbus e DNP3, com como seus mapeamentos para o modelo de dados IEC 61850, SunSpec Modbus e a Nota de Aplicação 2018-001.

Finalmente, foram apresentadas duas das principais arquiteturas ligadas aos protocolos listados. A escolha pela utilização de uma ou outra é feita pela especificação do protocolo de comunicação e do modelo de dados. Entretanto, como será visto no Capítulo 3, isso não impede que os serviços do sistema implementado sejam classificados com base em ambas as arquiteturas.

3 DESENVOLVIMENTO

Neste capítulo a arquitetura do sistema e os protocolos escolhidos serão apresentados, na Seção 3.1, e o desenvolvimento dos serviços que compõem o sistema serão detalhados. Os componentes relacionados ao DBus são utilizados em ambos os serviços de comunicação, e portanto são separados em uma biblioteca nomeada WDBus, que é apresentada na Seção 3.2.

Em seguida, a `sunspec-daemon` e `der-daemon` são apresentadas nas Seções 3.3 e 3.4, respectivamente. A Seção 3.5 apresenta a operação do sistema completo, e propõem os testes a serem realizados para validar as capacidades do sistema.

3.1 MÉTODO

Os protocolos IEC 61850 e DNP3 são em geral recomendados para integração com sistemas SCADA, enquanto SEP2.0 e Modbus são indicados para comunicação direta com DERs. (EPRI, 2013; Smart Inverter Working Group, 2015). Assim sendo, os seguintes protocolos foram escolhidos:

- Para comunicação entre DMS e DERMS, optou-se pela utilização do protocolo DNP3 devido sua capacidade de comunicações iniciadas pelo DERMS, por meio de respostas não solicitadas, que reduzem a largura de banda utilizada.
- Para comunicação entre DERMS e DER optou-se pela utilização do protocolo SunSpec Modbus, por ser um protocolo bem aceito em cenários industriais de um modo geral e ser definido pela principal aliança de empresas do setor fotovoltaico.
- Ambos os protocolos escolhidos aproximam-se mais da arquitetura hierárquica descrita na Seção 2.3.1 e, portanto, ela será utiliza como referência da implementação.

Para a implementação da SOA, o D-Bus foi escolhido como ESB. Desenvolvido pelo projeto de interoperabilidade ambientes *desktop*, o freedesktop.org, D-Bus é adotado por projetos como GNOME e KDE, facilitando a integração dos serviços a serem desenvolvidos com interfaces gráficas.

Uma visão geral do sistema proposto é apresentada na Figure 9. O DERMS é composto por quatro serviços que se conectam ao barramento DBus:

- `der-daemon`: responsável por comunicações com o DMS utilizando o protocolo DNP3 com o *profile* da Nota de Aplicação 2018-001 (DNP Users Group, 2019).
- `sunspec-daemon`: gerencia o estado da conexão e realiza o *polling* de DERs utilizando SunSpec Modbus.

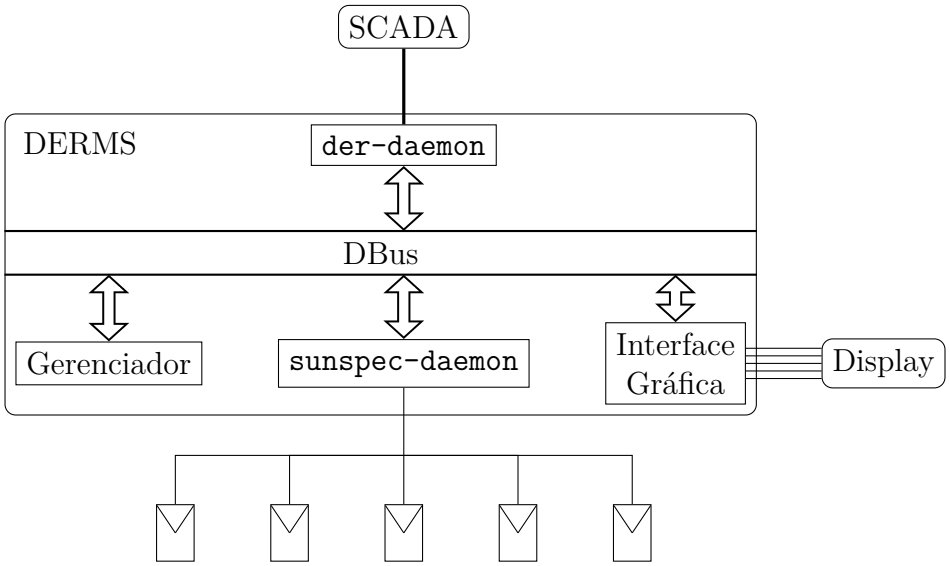


Figura 9 – Visão geral do sistema proposto.

- Gerenciador: Recebe os comandos do DMS por sinais emitidos por `der-daemon` e os redistribui aos DERs invocando métodos de `sunspec-daemon`. Além disso, agrega as mudanças de estados dos DERs, recebidas em sinais emitidos por `sunspec-daemon`, e atualiza esses dados invocando métodos de `der-daemon`.
- Interface Gráfica: Apresenta gráficos em tempo real de dados de cada DER, conforme seleção feita pelo usuário. Os dados são recebidos de sinais emitidos por `sunspec-daemon`.

Este trabalho concentra-se na implementação dos serviços de comunicação, `der-daemon` e `sunspec-daemon`. O Gerenciador utilizado é descrito em Santos et al. (2020), e por questões de espaço, a interface gráfica não será apresentada. O sistema completo será executado em uma Raspberry Pi 3b, executando a distribuição Raspbian.

A fim de verificar o desempenho do sistema, dois testes serão utilizados. O primeiro, voltado à escalabilidade, verificará o comportamento da `sunspec-daemon` quando utilizada para monitorar uma grande quantidade de DERs. O segundo, voltado ao atendimento dos parâmetros da norma IEEE 1547, irá medir o tempo de resposta da `der-daemon` para reportar mudanças nos valores de pontos específicos.

3.2 BIBLIOTECA WDBUS

Embora a implementação de referência do D-Bus forneça a biblioteca `libdbus`, sua documentação informa que a utilização direta da API dessa deve ser evitada, em favor de interfaces de mais alto nível chamadas *bindings* (FREEDESKTOP, 2020c). Dentre as opções listadas para a linguagem C encontram-se GDBus (FREEDESKTOP, 2020a), Eldbus (The Enlightenment Project, 2020a) e sd-bus (SYSTEMD, 2020a).

Todas essas opções, entretanto, depende de grandes bibliotecas que fazem parte de projetos maiores. A GDBus depende da GLib (GNOME Foundation, 2020), principal biblioteca do GTK, *toolkit* para construção de interfaces gráficas. A Eldbus depende da EFL (The Enlightenment Project, 2020b), conjunto de bibliotecas do gerenciador de janelas *Enlightenment*. Por fim a sd-bus é parte do systemd (SYSTEMD, 2020b), gerenciador sistemas e serviços.

Desta forma, a fim de evitar que os serviços de comunicação dependam de componentes da interface gráfica ou de um sistema de inicialização específico, optou-se pelo desenvolvimento de um *binding* próprio, nomeado como WDBus. A biblioteca é dividida em três módulos: o módulo de contexto, que gerencia a conexão com o barramento por meio de um laço de eventos; o módulo de objetos, para construção de objetos e suas interfaces; e o módulo de mensagens, para construção de mensagens a serem enviadas e extração de dados de mensagens recebidas. A seguir, cada um desses módulos é apresentado, em ordem de complexidade.

3.2.1 Módulo de Mensagens

A `libdbus` representa mensagens por meio da estrutura opaca `DBusMessage`. Um ponteiro para uma instância dessa estrutura pode ser obtido por meio de chamadas ao método `dbus_message_new`, que recebe um inteiro representando se a mensagem é uma chamada a um método, uma resposta a uma chamada, um sinal ou um erro. Alternativamente, os métodos `dbus_message_new_method_call`, `dbus_message_new_method_return`, `dbus_message_new_signal` e `dbus_message_new_error` podem ser utilizado, já recebendo como argumento outras informações, como interface e método para chamadas, a mensagem a que se responde, ou nome e mensagem de erro.

Para adicionar argumentos a uma mensagem, a biblioteca `libdbus` apresenta duas opções. A primeira é com o método `dbus_message_append_args`, que recebe um número variável de argumentos, intercalando o tipo do dado a ser anexado e o ponteiro para esse dado, finalizando com o tipo especial `DBUS_TYPE_INVALID`:

```
1 dbus_int32_t v_INT32 = 42;
2 const char *v_STRING = "Hello World";
3 dbus_message_append_args (message, DBUS_TYPE_INT32, &v_INT32,
    ↪ DBUS_TYPE_STRING, &v_STRING, DBUS_TYPE_INVALID);
```

Este método, entretanto, possui duas limitações: todos os argumentos precisam ser anexados em uma única chamada e tipos mais complexos, como estruturas e variantes, não são suportados. Para esses casos a opção de iteradores deve ser utilizada.

Um iterador é representado pela estrutura opaca `DBusMessageIter` que, no caso de adição de argumentos, pode ser obtido com chamadas ao método `dbus_message_iter_init_append`. Tipos básicos podem ser anexados a mensagem com

`dbus_message_iter_append_basic`, que recebe o iterador, o tipo e um ponteiro para o dado.

```
1 dbus_int32_t v_INT32 = 42;
2 const char *v_STRING = "Hello World";
3 dbus_message_iter_init_append(message, &iter);
4 dbus_message_iter_append_basic(&iter, DBUS_TYPE_INT32, &v_INT32);
5 dbus_message_iter_append_basic(&iter, DBUS_TYPE_STRING, &v_STRING);
```

Tipos complexos requerem a criação de subiteradores com o método `dbus_message_iter_open_container`, que recebe o iterador original, o tipo de contêiner, a assinatura caso o tipo necessite, e um ponteiro para o subiterador. Todos os subiteradores abertos devem ser fechados com `dbus_message_iter_close_container`, que recebe o iterador original e o subiterador.

```
1 dbus_int32_t v_INT32 = 42;
2 const char *v_STRING = "Hello World";
3 dbus_message_iter_init_append(message, &iter);
4 dbus_message_iter_open_container(&iter, DBUS_TYPE_STRUCT, NULL, &subiter
    ↪ );
5 dbus_message_iter_append_basic(&subiter, DBUS_TYPE_INT32, &v_INT32);
6 dbus_message_iter_append_basic(&subiter, DBUS_TYPE_STRING, &v_STRING);
7 dbus_message_iter_close_container(&iter, &subiter);
```

Outro caso de uso da estrutura `DBusMessage` é o de extração de argumentos, quando a mensagem é recebida do barramento. De maneira semelhante ao uso anterior, duas alternativas são oferecidas. A primeira baseia-se no método de argumentos variáveis `dbus_message_get_args`, com a mesma sintaxe de tipos e ponteiros intercalados de `dbus_message_append_args`. Limitações semelhantes se aplicam a tipos complexos.

A alternativa a esse método também se baseia na estrutura `DBusMessageIter`, dessa vez obtida com o método `dbus_message_iter_init`. Tipos básicos podem ser obtidos com `dbus_message_iter_get_basic`, cujos argumentos são o iterador e um ponteiro para receber o dado. O método `dbus_message_iter_next` deve ser utilizado para avançar o iterador para o próximo argumento.

```
1 dbus_message_iter_init(message, &iter);
2 dbus_message_get_basic(&iter, &v_INT32);
3 dbus_message_iter_next(&iter);
4 dbus_message_get_basic(&iter, &v_STRING);
```

Tipos complexos também requerem subiteradores, agora obtidos com `dbus_message_iter_recurse`. Diferente de `dbus_message_iter_open_container`, entretanto, iteradores obtidos com esse método não devem ser fechados.

```
1 dbus_message_iter_init(message, &iter);
2 dbus_message_recurse(&iter, &subiter);
3 dbus_message_get_basic(&iter, &v_INT32);
```

Importante notar que esses métodos para obtenção de argumentos com iteradores não recebem tipo ou assinatura dos dados, de forma que a função chamadora é quem deve se certificar de que o iterador aponta para o tipo esperado. Para tanto, o método `dbus_message_iter_get_arg_type` pode ser utilizado.

A fim de facilitar o uso desses mecanismos, a WDBus define as estruturas `WDBusContainer` e `WDBusMessage`. Ambas as estruturas são não opacas, para que possam ser alocadas na pilha da função chamadora, sem a necessidade de alocações com `malloc`. Buscando uma API mais concisa, apenas uma das formas de anexa e obter argumentos é utilizada, sendo escolhida a baseada em iteradores por ser a mais genérica.

A primeira estrutura consiste em uma lista ligada simples para manter registro da ordem de abertura de subiteradores:

```
1 struct _WDBusContainer {
2   DBusMessageIter iter;
3   struct _WDBusContainer *next;
4 };
```

A segunda armazena o ponteiro para a estrutura `DBusMessage` e também serve de cabeça para a lista ligada de `WDBusContainer`.

```
1 struct _WDBusMessage {
2   WDBusMessageType type;
3   DBusMessage *msg;
4   WDBusContainer *top, bottom;
5 };
```

O membro `bottom` armazena o iterador inicial, obtido com `dbus_message_iter_init` ou `dbus_message_iter_init_append`, e é também utilizado para inicializar o ponteiro `top`. O enumerador `WDBusMessageType` é definido como

```
1 typedef enum _WDBusMessageType {
2   WDBUS_MESSAGE_INVALID,
3   WDBUS_MESSAGE_IN,
4   WDBUS_MESSAGE_OUT,
5   WDBusMessageTypeSize
6 } WDBusMessageType;
```

servindo para indicar se a mensagem foi recebida ou está sendo construída para ser enviada. Com base nesse valor é possível detectar chamadas incorretas, como abertura de contêiner, ao invés de recursão, em uma mensagem recebida.

Por possuir a lista de `WDBusContainer`, apenas a estrutura `WDBusMessage` é necessária para anexar ou obter dados da mensagem, sempre utilizando `message->top.iter` como argumento para os métodos que requerem um iterador. Um novo subiterador, obtido por qualquer um dos métodos, pode ser adicionado a lista com

```
1 WDBusContainer cont;
2 cont.iter = subiter;
3 cont.next = msg.top;
4 msg.top = &cont;
```

Contêineres obtidos por `dbus_message_iter_recurse` podem ser simplesmente removidos da lista com

```
1 msg.top = msg.top->next;
```

O mesmo pode ser feito para contêineres obtidos com `dbus_message_iter_open_container`, mas com uma chamada à `dbus_message_iter_close_container` antes.

Essas manipulações das estruturas são abstraídas com uma série de métodos prefixados com `wdbus_message_`. O primeiro deles, `wdbus_message_init`, inicializa uma estrutura do tipo `WDBusMessage`, recebendo um ponteiro para esta, outro para `DBusMessage` e uma das constantes `WDBUS_MESSAGE_IN` ou `WDBUS_MESSAGE_OUT`. O método apropriado é utilizado para abertura do iterador que inicializa o membro `bottom` de `WDBusMessage`.

Os métodos `wdbus_message_get_arg_type`, `wdbus_message_next_arg` e `wdbus_message_append` baseiam-se em chamadas a `dbus_message_iter_get_arg_type`, `dbus_message_iter_next` e `dbus_message_iter_append_basic`, respectivamente, acrescidas das verificações oportunas com relação ao tipo da mensagem.

O método `wdbus_message_get_basic` recebe apenas um ponteiro para a estrutura `WDBusMessage` e pode ser utilizado para a obtenção de argumentos de uma mensagem recebida. Ao final do método, o iterador apontado por `top` é avançado com uma chamada a `wdbus_message_next_arg`, considerando que uma vez obtido, é mais barato replicar o dado na função chamadora do que realizar múltiplas chamadas a `wdbus_message_get_basic`.

Os métodos `wdbus_message_recurse` e `wdbus_container_open` fazem uso respectivamente de `dbus_message_iter_recurse` e `dbus_message_iter_open_container`, seguidos do procedimento descrito anteriormente para inserção de subiteradores na lista ligada de `WDBusContainer`.

O procedimento para remoção de elementos da lista é realizado com o método `wdbus_container_close`, com a verificação de que o iterador a ser fechado não é o inicial e a chamada a `dbus_message_iter_close_container` caso o tipo da mensagem seja `WDBUS_MESSAGE_OUT`. Caso o tipo seja `WDBUS_MESSAGE_IN`, é razoável assumir que o usuário espere que o iterador avance após o fechamento do contêiner e, para tanto, `wdbus_message_next_arg` é chamado. O método `wdbus_container_close_all` é definido para chamar `wdbus_container_close` até que todos os contêineres da mensagem sejam fechados.

Para facilitar o uso de `wdbus_container_open`, as macros `wdbus_array_open`, `wdbus_struct_open`, `wdbus_variant_open`, `wdbus_dict_open` e `wdbus_entry_open` são

definidas como chamas ao método com o devido tipo e sem assinatura nas quais ela não é necessária. No caso da macro para abertura de dicionário dois argumentos de assinatura são recebidos, um para chave e outro para o valor, e a assinatura final é montada usando a concatenação de *strings* literais adjacentes do preprocessador C.

Outro método para facilitar o uso de dicionários é `wdbus_dict_append`, que recebe a mensagem, tipo e ponteiro para chave e valor, e adiciona a entrada ao dicionário, com o segundo elemento dentro de um variante. Esse método é implementado dessa forma considerando que muitas APIs utilizam *strings* como chaves e variantes como valor para a obtenção de maior extensibilidade nessas interfaces (FREEDESKTOP, 2020b).

3.2.2 Módulo de Objetos

A `libdbus` não possui nenhuma estrutura específica para representar objetos em sua API pública. No lugar disso, a biblioteca expõe métodos para registro da função a ser utilizada para processar mensagens destinada a um determinado caminho. A estrutura `DBusObjectPathVTable` é definida com esta finalidade, composta por dois ponteiros de função `message_function` e `unregister_function`.

O primeiro, conforme descrito, será invocado para cada mensagem, recebendo como primeiro argumento a estrutura `DBusConnection`, que representa a conexão com o barramento, a mensagem e um ponteiro para dados do usuário, definido no momento do registro do objeto. Já o membro `unregister_function` será invocado quando o registro do objeto for desfeito. Isso pode ocorrer pelo fechamento da conexão com o barramento ou com o uso do método `dbus_connection_unregister_object_path`. Seus argumentos são a estrutura `DBusConnection` e ponteiro de dados do usuário.

Dois métodos são oferecidos para realizar o registro. O primeiro é `dbus_connection_register_object_path`, que recebe a estrutura `DBusConnection`, o caminho em que o registro será feito, a estrutura `DBusObjectPathVTable` e o ponteiro para dados do usuário. A documentação informa que este método não deve ser utilizado caso exista a possibilidade de outro objeto já estar registrado no caminho especificado, sendo aconselhado o uso do método `dbus_connection_try_register_object_path` caso exista essa possibilidade. Este método possui um argumento a mais: uma estrutura do tipo `DBusError` que conterá o motivo da falha.

Com base nesta API a WDBus constrói um conjunto de estruturas e métodos para abstrair objetos e interfaces. Adicionalmente, a biblioteca implementa duas das quatro interfaces padronizadas pela especificação do D-Bus (PENNINGTON et al., 2020), `org.freedesktop.DBus.Introspectable` e `org.freedesktop.DBus.Properties`. A Figura 10 apresenta um diagrama de classes com a relação entre as três principais estruturas deste módulo - `WDBusObject`, `WDBusInterface` e `WDBusMethod`.

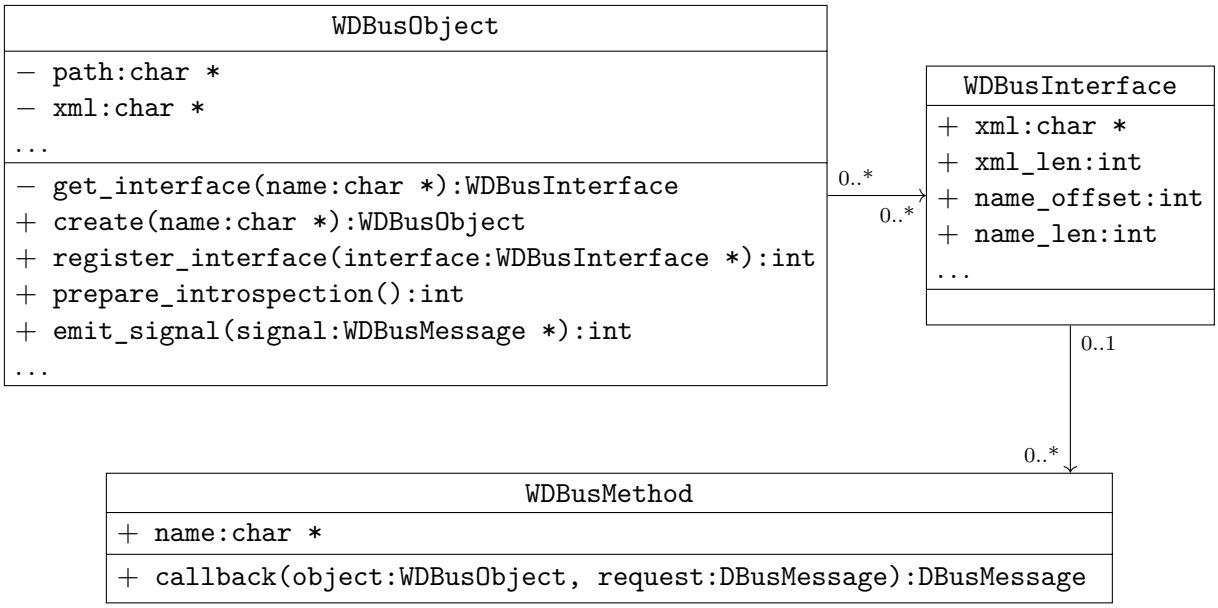


Figura 10 – .
Diagrama de classe das estruturas `WDBusObject`, `WDBusInterface` e `WDBusMethod`.

Cada uma dessas estruturas é responsável por uma abstração, descritas nos próximos tópicos. Em seguida, a implementação das interfaces `org.freedesktop.DBus.Introspectable` e `org.freedesktop.DBus.Properties` será detalhada.

Abstração de Objetos

A abstração de objetos é realizada com a estrutura `WDBusObject` e métodos associados. Sua definição é a seguinte.

```
1 struct _WDBusObject {
2   char *path, *xml;
3   WDBusContext *ctx;
4   WDBusInterfaceList *interfaces;
5   int xml_dirty;
6   WDBusFreeCallback user_free;
7   void *user_data;
8 };
```

A estrutura armazena dados como o caminho em que o objeto foi registrado, em `path`, o XML de introspecção, em `xml`, um ponteiro para dados do usuário e um ponteiro para a função que libera os recursos associados a esses dados. A lista das interfaces deste objeto é armazenada em uma lista ligada simples utilizando a estrutura `WDBusInterfaceList`, que gera a cardinalidade `0..*` do diagrama da Figura 10.

```
1 struct _WDBusInterfaceList {
2   WDBusInterface *interface;
3   struct _WDBusInterfaceList *next;
4 };
```

Ambas as estruturas são opacas, sendo `WDBusInterfaceList` usada apenas internamente a WDBus e `WDBusObject` alocável com o método `wdbus_object_create`, que recebe como argumento o caminho em que o objeto será registrado.

Conforme indicado pelo diagrama de classe, a relação de associação entre `WDBusObject` e `WDBusInterface` permite que múltiplas instâncias de `WDBusObject` registrem uma mesma instância de `WDBusInterface`. Este registro pode ser feito com o método `wdbus_object_register_interface`, que aloca uma nova instância de `WDBusInterfaceList`, adiciona esta instância a lista ligada e altera o valor do membro `xml_dirty` de `WDBusObject` para um valor verdadeiro, indicado a necessidade atualizar o membro `xml`.

Tal atualização pode ser realizada com `wdbus_object_prepare_introspection` em momento oportuno. O método percorre a lista de interfaces concatenando os membros `xml` de cada `WDBusInterface`, com o devido cabeçalho XML e abertura da tag `<node>` no início e fechamento `</node>` ao final. A separação entre registro de interface e preparação do XML é feita pois o processo é relativamente caro, envolvendo alocações de memória e cópias de *strings* e, em geral, o usuário irá invocar `wdbus_object_register_interface` repetidas vezes na criação do objeto, de forma que dados de introspecção construídos entre chamadas seriam descartados sem nenhum uso.

O método `wdbus_object_emit_signal` pode ser utilizado para emitir sinais a partir de um objeto. Seus argumentos são a estrutura `WDBusObject`, que deve ter sido registrada em um `WDBusContext`, e uma estrutura `WDBusMessage` com uma `DBusMessage` do tipo `DBUS_MESSAGE_TYPE_SIGNAL`.

O usuário da WDBus não pode definir ou recuperar diretamente o ponteiro de dados presente em `WDBusObject`, pois a estrutura é opaca. Para tanto, os métodos `wdbus_object_set_user_data` e `wdbus_object_get_user_data` são definidos. O primeiro tem como argumentos a estrutura `WDBusObject`, o ponteiro e a função a ser utilizada para liberar os recursos associados a este ponteiro. Já `wdbus_object_get_user_data` possui como argumento apenas a estrutura `WDBusObject`.

Por fim, `wdbus_object_free` pode ser utilizado para destruir uma instância de `WDBusObject`. Seu funcionamento consistem em usar o membro `user_free` em `user_data`, caminhar pela lista de interfaces liberando os nós formados pelas estrutura `WDBusInterfaceList` e liberar a memória alocada para `xml`, `path` e da própria estrutura `WDBusObject`.

Abstração de Interfaces

A estrutura `WDBusInterface` é definida como

```
1 struct _WDBusInterface {
2   WDBusPropertyCallback get, get_all, set;
3   int name_offset, name_len, xml_len, nmethod;
4   char *xml;
5   void *user_data;
6   WDBusFreeCallback user_free;
7   WDBusMethod methods[];
8 };
```

O membro `xml` armazena o XML da interface para introspecção, utilizado pelo método `wdbus_object_prepare_introspection` no processo descrito no tópico anterior.

Os membros `name_offset` e `name_len` indicam, respectivamente, o deslocamento e tamanho do nome da interface dentro de `xml`. O membro `user_data` armazena um ponteiro para dados do usuário e `user_free` aponta para o método a ser invocado para liberar recursos associados a este ponteiro.

Já `methods` é um *Flexible array member* (FAM), recurso padronizado por ISO (1999) que permite que o último membro da estrutura seja um *array* de tamanho não-especificado. Para que o acesso a este membro não resulte em comportamento indefinido, é preciso alocar memória adicional ao instanciar a estrutura. Assim sendo, se a interface a ser definida possuir ao menos um método, `WDBusInterface` deve ser alocada por meio de chamadas a `malloc` e funções similares, ou instanciada estaticamente, como uma estrutura global. O membro `nmethod` representa o número de elementos de `methods`.

Os membros `get`, `get_all` e `set` servem para a implementação de propriedades da interface. São ponteiros para métodos que serão invocados quando os métodos `Get`, `GetAll` e `Set` da interface `org.freedesktop.DBus.Properties` forem invocados em propriedades da interface.

Abstração de métodos

A abstração de métodos é feita pela estrutura `WDBusMethod`, que é definida como

```
1 struct _WDBusMethod {
2   char *name;
3   WDBusMethodCallback callback;
4 };
```

O membro `name` é o nome do método e `callback` é a função a ser chamada quando o método for invocado, cujo seu primeiro argumento é a estrutura `WDBusObject` e o segundo a `DBusMessage` recebida, do tipo `DBUS_MESSAGE_TYPE_METHOD_CALL`. O retorno

de `callback` deve ser uma `DBusMessage` do tipo `DBUS_MESSAGE_TYPE_METHOD_RETURN` ou `DBUS_MESSAGE_TYPE_ERROR`.

A invocação de `callback` é feita por `wdbus_message_function`, método da WDBus que preenche o membro `message_function` da estrutura `DBusObjectPathVTable` ao registrar objetos. Considerando que todas as informações referentes ao estado de um objeto estão armazenadas em `WDBusObject`, a `message_function` de todos os objetos registrados pode ser a mesma, mudando apenas o ponteiro de dados do usuário passado a `dbus_connection_try_register_object_path` para apontar para instância de `WDBusObject`.

Ao ser invocado, `wdbus_message_function` recupera a estrutura `WDBusObject` do ponteiro de dados do usuário e utiliza os métodos `dbus_message_get_interface` e `dbus_message_get_member` da `libdbus` para obter a interface e método que está sendo invocado. Em seguida, o método privado `wdbus_get_interface` é utilizado para recuperar a estrutura `WDBusInterface`. Este método percorre a lista de interfaces da seguinte forma:

```
1 for(WDBusInterfaceList *list = obj->interfaces; list; list = list->next)
2   if(len == list->interface->name_len && !strncmp(name,
3         &list->interface->xml[list->interface->name_offset],
4         list->interface->name_len)) {
5     interface = list->interface;
6     break;
7   }
```

Se a interface for encontrada, o membro `methods` de `WDBusInterface` é varrido

```
1 for(i = 0; i < interface->nmethod; i++) {
2   if(!strcmp(method, interface->methods[i].name)) {
3     reply = interface->methods[i].callback(obj, msg);
4     break;
5   }
6 }
```

Invocando `callback`, na linha 3, caso o método correspondente seja encontrado. Se a interface ou método não for encontrado, o retorno de `wdbus_message_function` indicará que o método não existe e a `libdbus` irá enviar uma mensagem de erro como resposta.

Para preencher o membro `unregister_function` de `DBusObjectPathVTable`, a WDBus utiliza o método `wdbus_unregister_function`, que apenas invoca o método `wdbus_object_free`. A definição de `wdbus_unregister_function` se faz necessária para compatibilizar a assinatura do método com a requerida por `unregister_function`.

Implementação das interfaces padronizadas

A WDBus implementa duas das quatro interfaces padronizadas pela especificação do D-Bus (PENNINGTON et al., 2020). A `org.freedesktop.DBus.Introspectable` é composta por apenas um método, `Introspect`, que não recebe argumentos e retorna uma *string* com o XML descrevendo o objeto. Já `org.freedesktop.DBus.Properties` introduz o conceito de propriedades aos objetos D-Bus, com três métodos e um sinal:

- O método `Get`, para obtenção do valor de uma propriedade, recebe como argumento uma *string* com o nome da interface que possui a propriedade desejada e uma *string* com o nome da propriedade. Seu retorno é um variante com o valor da propriedade.
- O método `GetAll` possibilita a obtenção de todas as propriedades de uma interface em uma única chamada. O método requer como argumento apenas o nome da interface e seu retorno é um dicionário que possui o nome das propriedades como chave.
- O método `Set` permite alterar o valor de uma propriedade. Seus argumentos são o nome da interface, da propriedade e um variante com o novo valor. O retorno de `Set` não possui argumentos, indicando apenas o sucesso ou falha na atualização do valor da propriedade que pode ocorrer, por exemplo, se a propriedade for somente leitura ou se o valor estiver fora de um limite aceitável.
- O sinal `PropertiesChanged` oferece um mecanismo opcional para notificar mudanças no valor das propriedades de uma interface. Seus argumentos são uma *string* com o nome da interface, um dicionário semelhante ao de `GetAll` com apenas as propriedades que mudaram de valor, e um vetor de *string* com as propriedades que mudaram, mas não tiveram o valor anexado ao sinal (por questões de desempenho, por exemplo).

O método `wdbus_introspect_method` implementa a primeira interface, recebendo como argumentos a estrutura `WDBusObject` e a mensagem de requisição em uma `DBusMessage`. O membro `xml_dirty` de `WDBusObject` é verificado e, caso definido, o método `wdbus_object_prepare_introspection` é invocado. Em seguida, a mensagem de resposta é construída com o conteúdo do membro `xml`. Uma instância global de `WDBusInterface` chamada `wdbus_introspectable_interface` é definida utilizando esse método.

Para a implementação de `org.freedesktop.DBus.Properties`, os métodos `wdbus_get_method`, `wdbus_get_all_method` e `wdbus_set_method` são definidos. Todos recebem como argumento a instância de `WDBusObject` e a requisição. A implementação das funcionalidades de `Get`, `GetAll` e `Set`, entretanto, é delegada à aplicação, que deve preencher os membros `get`, `get_all` e `set` de `WDBusInterface`. Os métodos da

aplicação devem receber como argumento o objeto em uma estrutura `WDBusObject`, a instância de `WDBusInterface`, uma *string* com o nome da propriedade e uma `WDBusMessage` da requisição. Para `get_all`, a *string* com o nome da propriedade será sempre um ponteiro nulo. Para `set`, o variante de `WDBusMessage` estará aberto.

A verificação dos argumentos e recuperação da instância de `WDBusInterface` são implementadas no método `wdbus_property_method`, que recebe os mesmos argumentos dos métodos anteriores, acrescido de um enumerador identificando qual operação está sendo realizada – `Get`, `GetAll` ou `Set`. Esta abordagem procura evitar a implementação em duplicidade das verificações, aumentando a manutenibilidade do código. Desta forma, os métodos `wdbus_get_method`, `wdbus_get_all_method` e `wdbus_set_method` apenas invocam `wdbus_property_method` com o enumerador apropriado.

Assim como a primeira interface, uma instância global de `WDBusInterface` nomeada `wdbus_properties_interface` é definida. A WDBus oferece ainda implementações padrão para os membros `get`, `get_all` e `set` de `WDBusInterface`. Os métodos `wdbus_properties_default_get` e `wdbus_properties_default_set` retornam uma mensagem de erro indicando que a propriedade não existe na interface, enquanto `wdbus_properties_default_get_all` retorna a mensagem de resposta com o dicionário de propriedades vazio.

3.2.3 Módulo de Contexto

A `libdbus` representa internamente a conexão com o barramento DBus por meio de duas filas, uma para recepção e outra para envio de mensagens, associadas a um mecanismo de transporte, como um soquete TCP ou de domínio Unix. Todo processamento realizado pela biblioteca pode ser descrito como leitura, escrita ou despacho de mensagens dessas filas.

Durante o processo de leitura, dados recebidos do mecanismo de transporte são delimitados e analisados para formar mensagens que são adicionadas a fila de recepção. O processo de escrita transforma as mensagens da fila de envio em um fluxo de dados que possa ser utilizado pelo mecanismo de transporte. Por fim, o processo de despacho consiste na invocação de métodos da aplicação que consumirão as mensagens da fila de recepção, possivelmente inserindo novas mensagens na fila de envio.

Ao utilizar a `libdbus`, a aplicação deve realizar estas três ações de maneira coordenada, esperando de forma eficiente pela prontidão do mecanismo de transporte para leitura e escrita, e notificando a biblioteca sobre esses eventos para que ocorra o processamento e despacho de mensagens. A seguir, os tópicos 3.2.3.1 e 3.2.3.2 apresentam a interface exposta pela `libdbus` para envio de mensagens e gerenciamento da conexão, e o tópico 3.2.3.3 detalha o uso desta API pela WDBus.

Interface da `libdbus` para conexão e envio de mensagens

A estrutura opaca `DBusConnection` representa uma conexão com outra aplicação. O principal método para obtenção desta estrutura é `dbus_bus_get`, que recebe como argumento um enumerador indicando se a conexão deve ser feita com o barramento do sistema ou da sessão, realizando a conexão e registro da aplicação no barramento. Caso a aplicação seja um serviço e deseje obter um nome específico, a requisição pode ser feita em seguida com o método `dbus_bus_request_name`.

A `libdbus` oferece quatro métodos para envio de mensagens em uma conexão. O primeiro, `dbus_connection_send`, envia uma mensagem de maneira não bloqueante, recebendo como argumento um ponteiro para a estrutura `DBusConnection`, um ponteiro para a `DBusMessage` e um ponteiro para um inteiro que receberá o número serial da mensagem enviada, que pode ser nulo caso a aplicação não tenha interesse nessa informação. O segundo método, `dbus_connection_send_preallocated`, tem a mesma funcionalidade recebendo, porém, como argumento adicional uma estrutura do tipo opaco `DBusPreallocatedSend`, que representa uma mensagem pré-alocada pelo método `dbus_connection_preallocate_send`.

Os outros dois métodos da API, `dbus_connection_send_with_reply` e `dbus_connection_send_with_reply_and_block`, destinam-se ao envio de mensagem que invocam métodos e, portanto, esperam um retorno. Conforme indicado pelo nome, `dbus_connection_send_with_reply_and_block` realiza o envio de maneira bloqueante, recebendo como argumento a conexão, a mensagem a ser enviada e o tempo máximo de espera em milissegundos.

Já a versão não bloqueante recebe, além destes argumentos, um ponteiro de ponteiro para a estrutura opaca `DBusPendingCall`. Com esta estrutura é possível cancelar o envio da mensagem, com `dbus_pending_call_cancel`, verificar se uma resposta foi recebida, com `dbus_pending_call_get_completed`, e obter a resposta em uma estrutura `DBusMessage` com `dbus_pending_call_steal_reply`. Também é possível bloquear até que chamada pendente seja completada, isto é, até que a resposta seja recebida ou o tempo máximo de espera se esgote, com o método `dbus_pending_call_block`.

Por fim, para operar de maneira assíncrona, com um laço de eventos, o método `dbus_pending_call_set_notify` deve ser utilizado para definir o método a ser invocado ao receber a resposta. Seus argumentos são a estrutura `DBusPendingCall`, o método a ser invocado, um ponteiro de dados do usuário e o método para liberar recursos associados a este ponteiro. Ao receber a resposta ou quando o tempo de espera se esgotar, o método definido por `dbus_pending_call_set_notify` será invocado com a estrutura `DBusPendingCall` e o ponteiro de dados do usuário como argumentos.

Interface da `libdbus` para leitura, escrita e despacho

Uma das formas de realizar as operações de leitura, escrita e despacho consiste na utilização do método `dbus_connection_read_write_dispatch`, que realiza as três operações de maneira bloqueante. Se a aplicação faz mais do apenas reagir a mensagens recebidas, será necessário utilizar múltiplas *threads*. Seu uso ainda pode apresentar outras dificuldades, como o fato de que o método não pode ser interrompido, mesmo por outras *threads*, sem corromper o estado interno das estruturas utilizadas pela `libdbus`.

A alternativa a este método consiste no desenvolvimento de um laço de eventos que monitore a disponibilidade do mecanismo de transporte e notifique a `libdbus` na ocorrência de eventos. A biblioteca requisita dois tipos de monitoramento. O primeiro, representado pela estrutura `DBusWatch`, é associado a descritores de arquivos, notificando a `libdbus` na disponibilidade de escrita e leitura ou na ocorrência de erros. O segundo é o de temporização, representado pela estrutura `DBusTimeout`.

A aplicação que implementa um laço de eventos deve registrar os métodos para adicionar, remover e alterar o estado de ativação de instâncias de `DBusWatch` com o método `dbus_connection_set_watch_functions`, e `dbus_connection_set_timeout_functions` para instâncias de `DBusTimeout`. Ambos os métodos recebem como argumento o ponteiro para a estrutura `DBusConnection`, os três métodos a serem registrados, um ponteiro para dados do usuário e o método para liberar os recursos associados a este ponteiro. Os métodos registrados com `dbus_connection_set_watch_functions` receberão como argumentos, quando invocados, a estrutura `DBusWatch` e o ponteiro de dados do usuário. Analogamente, os métodos registrados com `dbus_connection_set_timeout_functions` receberão como argumentos a estrutura `DBusTimeout` e o ponteiro de dados do usuário.

Ligado a estrutura `DBusWatch`, a `libdbus` ainda expõem em sua API os métodos `dbus_watch_get_unix_fd`, que retorna o descritor de arquivos da instância `DBusWatch`, `dbus_watch_get_flags`, que informa se o descritor deve ser monitorado para leitura ou escrita, e `dbus_watch_get_enabled`, que indica se o monitoramento do descritor está ativo. Também é possível definir um ponteiro de dados do usuário com o método `dbus_watch_set_data`, que pode ser recuperado com o método `dbus_watch_get_data`. Quando o descritor de arquivo de uma instância de `DBusWatch` estiver pronto para o tipo de operação que está sendo monitorada, o método `dbus_watch_handle` deve ser invocado com essa estrutura como argumento, para que as operações de leitura e escrita sejam realizadas.

Relacionado a estrutura `DBusTimeout`, a biblioteca `libdbus` expõem os métodos `dbus_timeout_get_interval` e `dbus_timeout_get_enabled`, para obter a duração do temporizador e seu estado de ativação, respectivamente. A estrutura `DBusTimeout` também possui um ponteiro de dados do usuário que pode ser manipulado com os métodos

`dbus_timeout_set_data` e `dbus_timeout_get_data`. No esgotamento do temporizador, o método `dbus_timeout_handle` deve ser invocado na instância de `DBusTimeout`.

Uma analogia com orientação a objetos pode ser utilizada para facilitar o entendimento sobre essas duas estruturas. A Figura 11 apresenta os diagramas de classes para `DBusWatch` e `DBusTimeout`. Ambas as estruturas são representadas como classes abstratas, com `add`, `remove` e `toggled` não concretos, a fim de representar a necessidade de implementá-los e registrá-los com os métodos `dbus_connection_set_watch_functions` e `dbus_connection_set_timeout_functions`.

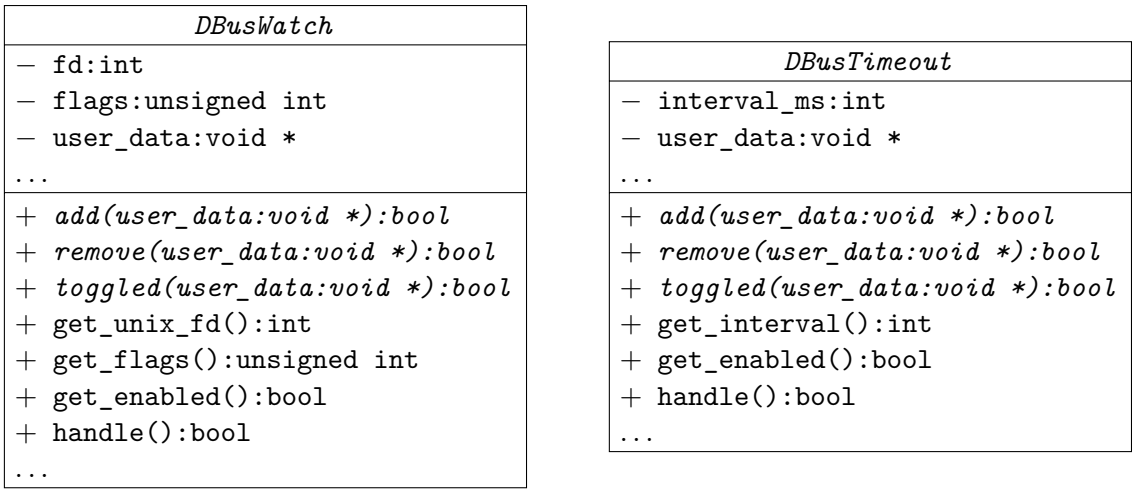


Figura 11 – Diagrama de classe das estruturas `DBusWatch` e `DBusTimeout`.

Por fim, o método `dbus_connection_set_wakeup_main_function` deve ser utilizado para registrar uma função que acorde o laço de eventos, e o método `dbus_connection_set_dispatch_status_function` pode ser utilizado para definir um método que notifique a existência de novas mensagens a serem despachadas. Também é possível verificar se existem mensagens a serem despachadas com o método `dbus_connection_get_dispatch_status`, tornando o uso de `dbus_connection_set_dispatch_status_function` opcional. Para realizar a operação de despacho, o método `dbus_connection_dispatch` deve ser utilizado. É importante notar que este método não deve ser utilizado dentro do método registrado com `dbus_connection_set_dispatch_status_function`, para que não ocorra um *deadlock*.

Implementação do laço de eventos e API da WDBus

A WDBus implementa seu laço de eventos com base na API *epoll* do kernel Linux. Outras APIs padronizadas por POSIX (2016), como *select* e *poll*, possuem capacidades semelhantes para monitorar descritores de arquivos, mas apresentam problemas de escalabilidade e desempenho (BANGA et al., 1999; LEMON, 2001; LOVE, 2013). Para temporização, a chamada de sistema `timerfd_create` é utilizada para criar temporiza-

dores baseados em descritores de arquivo, que podem ser monitorados por *epoll*. O sinal `SIGPOLL` é utilizado para acordar o laço de eventos, que o monitora por meio de um descritor de arquivos obtidos com a chamada de sistema `signalfd`. Tanto `timerfd_create` quanto `signalfd` são chamadas específicas de sistemas Linux.

Um objeto `epoll` é composto de duas listas de descritores de arquivos, uma com os descritores de interesse da aplicação e outra com os descritores prontos para operações de entrada e saída. A instância do objeto é interna ao kernel, e pode ser criada com a chamada de sistema `epoll_create1`, que retorna um descritor de arquivos que representa o `epoll`.

A chamada `epoll_ctl` permite manipular a lista de interesse, recebendo como argumentos o descritor de `epoll`, um inteiro indicando a operação a ser realizada (adicionar, modificar ou remover), o descritor de arquivos alvo da operação e uma estrutura `epoll_event`, que é definida como

```
1  typedef union epoll_data {
2      void        *ptr;
3      int          fd;
4      uint32_t     u32;
5      uint64_t     u64;
6  } epoll_data_t;
7
8  struct epoll_event {
9      uint32_t     events;      /* Epoll events */
10     epoll_data_t data;        /* User data variable */
11 };
```

O membro `events` é uma máscara de bits indicando os eventos de interesse, como `EPOLLIN` e `EPOLLOUT` para monitor a possibilidade de leitura e escrita sem bloqueio, respectivamente. Outras opções podem ser passadas neste membro, como `EPOLLONESHOT`, para que o descritor seja removido da lista de interesse assim que se torne pronto

Já a chamada de sistema `epoll_wait` pode ser utilizada para recuperar a lista de descritores prontos ou bloquear até que ao menos um descritor torne-se pronto. Seus argumentos são o descritor de `epoll`, um vetor de estruturas `epoll_event`, o tamanho máximo deste vetor e o tempo máximo de espera em milissegundos. Cada descritor pronto será representado por um elemento do vetor de `epoll_event`, com o retorno da chamada indicando o tamanho do vetor. O membro `events` indicará os eventos que tornaram o descritor pronto, como `EPOLLIN` e `EPOLLOUT` caso o descritor estivesse sendo monitorado para leitura ou escrita, ou `EPOLLERR` caso algum erro tenha ocorrido no recurso associado ao descritor. O campo `data` da estrutura será idêntico ao da `epoll_event` passada a `epoll_ctl`, podendo carregar um ponteiro de dados do usuário ou o próprio descritor de arquivos, por exemplo.

A WDBus define a estrutura `WDBusEventData` para representar eventos monitorados com *epoll*.

```
1  typedef enum _WDBusEventType {
2    WDBUS_WATCH_TYPE,
3    WDBUS_TIMEOUT_TYPE,
4    WDBUS_SIGNAL_TYPE,
5    WDBusEventTypeSize
6  } WDBusEventType;
7
8  typedef struct _WDBusEventData {
9    WDBusEventType type;
10   int fd;
11   union {
12     DBusWatch *watch;
13     DBusTimeout *timeout;
14     struct signalfd_siginfo *sfsi;
15   };
16 } WDBusEventData;
```

O membro `type` indica se o evento está associado a um descritor de arquivo monitorado por uma instância de `DBusWatch`, um temporizador de um `DBusTimeout`, ou ao sinal que acorda o laço de eventos. Uma `union` anônima carrega o ponteiro para a instância de `DBusWatch`, `DBusTimeout` ou `signalfd_siginfo` – estrutura em que as informações do sinal recebido podem ser examinadas.

Ao criar instâncias de `DBusWatch` e `DBusTimeout`, uma instância associada de `WDBusEventData` é criada e utilizada para preencher o ponteiro do usuário da estrutura `epoll_event`. Ao receber a lista de `epoll_event` com `epoll_wait`, a instância de `WDBusEventData` é recuperada. Com base no membro `type` da estrutura é possível determinar como processar esse evento. De maneira semelhante este membro da estrutura é utilizado no método `wdbus_event_data_free`, definido para liberar os recursos de uma instância de `WDBusEventData`.

Para armazenar as informações sobre a conexão e sobre o laço de eventos, a WDBus define a estrutura opaca `WDBusContext`:

```
1  struct _WDBusContext {
2    DBusConnection *conn;
3    int epfd, should_dispatch, nevent;
4    struct signalfd_siginfo siginfo;
5    WDBusEventData signal;
6    WDBusFreeCallback user_free;
7    void *user_data;
8    struct epoll_event event[];
9  };
```

Novamente um FAM é utilizado, desta vez para acomodar um vetor de estruturas do tipo `epoll_event` que é utilizado nas chamadas a `epoll_wait`. O tamanho deste vetor é fornecido como o único argumento do método `wdbus_context_create`, que instância `WDBusContext`. Uma chamada a `malloc` é utilizada para alocar a estrutura e o argumento informado é armazenado em `nevent`. O objeto `epoll` também é criado e seu descritor armazenado em `epfd`. A máscara de sinais do processo é preparada para o recebimento do sinal `SIGPOLL` por meio de um descritor de arquivos, que é obtido com uma chamada a `signalfd` e adicionado a `epoll`.

O membro `siginfo` é a instância de `signalfd_siginfo` utilizada para criar a estrutura `WDBusEventData` relacionada ao recebimento do sinal `SIGPOLL`, que é armazenada no membro `signal`. O membro `should_dispatch` é utilizado para indicar se é necessário despachar. Por fim, os membros `user_data` e `user_free` são respectivamente um ponteiro de dados do usuário e o método para liberar recursos relacionados a este ponteiro. Os métodos `wdbus_context_set_user_data` e `wdbus_context_get_user_data` podem ser utilizar para definir e recuperar este ponteiro.

Com a instância de `WDBusContext` é possível conectar-se ao barramento DBus invocando o método `wdbus_context_connect`, que é baseado no método `dbus_bus_get` da `libdbus`. Os argumentos recebidos são o ponteiro para a instância alocada por `wdbus_context_create`, o enumerado indicando em qual barramento pretende-se conectar, uma *string* com o nome de serviço a ser requisitado e um inteiro com *flags* adicionais a serem passadas ao requisitar o nome com `dbus_bus_request_name`. Se o ponteiro para a *string* com o nome for nulo, apenas a chamada a `dbus_bus_get` será realizada.

O método `wdbus_add_watch` é definido para adicionar uma instância de `DBusWatch` ao laço de eventos. O método `dbus_watch_get_data` é utilizado para verificar se o `DBusWatch` recebido já possui uma estrutura `WDBusEventData` e, caso contrário, alocá-la e defini-la como ponteiro do usuário de `DBusWatch` com `dbus_watch_set_data`, com `wdbus_event_data_free` como método para liberar os recursos associados. Os métodos `dbus_watch_get_unix_fd` e `dbus_watch_get_flags` são então utilizado para recuperar o descritor de arquivos e as *flags* de interesse. Em seguida, se o retorno de `dbus_watch_get_enabled` for positivo, o descritor é adicionado a `epoll` com uma chamada a `epoll_ctl`.

Para alternar o estado de ativação de um `DBusWatch`, a WDBus define o método `wdbus_watch_toggled`. Este método é invocado para alternar a ativação de uma instância de `DBusWatch` e, para tanto, utiliza o método `dbus_watch_get_enabled` para verificar o novo estado. Se o retorno for positivo, o método `dbus_watch_get_flags` é utilizado para recuperar as *flags* de interesse e `epoll_ctl` é invocado para adicionar o descritor a `epoll`. Se o retorno de `dbus_watch_get_enabled` for negativo, `epoll_ctl` é utilizado para remover o descritor de `epoll`.

O descritor de arquivos de uma estrutura `DBusWatch` pode ser removido de `epoll` com o método `wdbus_remove_watch`, que realiza tal função com uma chamada a `epoll_ctl`. Nenhum recurso é liberado por esta chamada, pois o método `wdbus_event_data_free`, informado na chamada a `dbus_watch_set_data` em `wdbus_add_watch`, será invocado pela `libdbus` ao desalocar a instância de `DBusWatch`.

Para adição de temporizadores a WDBus implementa `wdbus_add_timeout`. Semelhante a `wdbus_add_watch`, a existência de uma estrutura `WDBusEventData` associada é verificada, desta vez como uma chamada a `dbus_timeout_get_data`. Caso a instância de `WDBusEventData` precise ser criada, ou se o membro `fd` da estrutura existente for um valor inválido para um descritor de arquivos, uma chamada a `timerfd_create` é realizada para criar o temporizador. Em seguida, a ativação do temporizador é verificada com `dbus_timeout_get_enabled` e, caso o retorno seja positivo, o intervalo de temporização é recuperado com `dbus_timeout_get_interval` e utilizado em ama chamada a `timerfd_settime` para armar o temporizador. Por fim, o descritor de arquivos é adicionado à `epoll` com `epoll_ctl`.

O método `wdbus_timeout_toggled` implementa a mudança de estado de ativação de uma instância de `DBusTimeout`. O método `dbus_timeout_get_data` é utilizado para recuperar a estrutura `WDBusEventData`, e `dbus_timeout_get_enabled` é invocado para verificar a ativação do temporizador. Caso positivo, `dbus_timeout_get_interval` é utilizado para recuperar o período de temporização e defini-lo com `timerfd_settime`. Do contrário, uma chamada apropriada a `timerfd_settime` é realizada para desativar o temporizador.

Para remoção do descritor de um temporizador da `epoll`, o método `wdbus_remove_timeout` pode ser utilizado. A estrutura `WDBusEventData` com o descritor é recuperada com `dbus_timeout_get_data` e, caso o membro `fd` da estrutura seja um descritor válido, ele será fechado com uma chamada a `close`.

O método `wdbus_wakeup_main` acorda o laço eventos com uma chamada a `raise` com o sinal `SIGPOLL` como argumento. Sua definição é compatível com a assinatura requerida por `dbus_connection_set_wakeup_main_function`. Por fim, o método `wdbus_dispatch_status` é implementado para ser registrado com `dbus_connection_set_dispatch_status_function`, atualizando o membro `should_dispatch` de `WDBusContext` de acordo.

Esse registro, entretanto, assim como o registro dos métodos com `dbus_connection_set_watch_functions`, `dbus_connection_set_timeout_functions` e `dbus_connection_set_wakeup_main_function`, não é feito diretamente pelo usuário da WDBus, mas por meio do método `wdbus_context_setup`. Após o registro dos métodos do laço de eventos, o método ainda verifica a necessidade de despachar mensagens, com `dbus_connection_get_dispatch_status`, invocando `dbus_connection_dispatch` de acordo. Desta forma a conexão com o barramento pode

ser feita antes da configuração do laço de eventos, caso o usuário da WDBus assim prefira.

O último método relacionado ao laço de eventos é `wdbus_loop`, em que uma iteração do laço é executada. O método recebe o ponteiro para a instância de `WDBusContext` e um valor de limite de espera, que é utilizado na chamada de `epoll_wait`. Em seguida o vetor de `epoll_event` é percorrido, invocando `dbus_watch_handle` e `dbus_timeout_handle` conforme o tipo de evento e, em seguida, `dbus_connection_dispatch` de acordo com o valor de `should_dispatch` de `WDBusContext`. Caso o evento recebido seja o sinal `SIGPOLL`, `dbus_connection_get_dispatch_status` é utilizado para verificar a necessidade de despacho e `dbus_connection_dispatch` é invocado de acordo.

Sendo `epoll` representado por um descritor de arquivos, o próprio laço de eventos da WDBus pode ser adicionado a um laço de eventos da aplicação. Para tanto, o método `wdbus_context_get_epdf` deve ser utilizado para obter o descritor e, quando este se tornar pronto para leitura, `wdbus_loop` pode ser invocado com tempo limite igual a zero para uma iteração não bloqueante do laço de eventos da WDBus.

Para registro de objetos, o método `wdbus_context_register_object` é definido, utilizando internamente o método `dbus_connection_try_register_object_path` da `libdbus`. Para desfazer o registro o método `wdbus_context_unregister_object` pode ser utilizado, baseado no método `dbus_connection_unregister_object_path`. Ambos os métodos recebem como argumento as estruturas `WDBusContext` e `WDBusObject`.

Por fim, o método `wdbus_context_free` libera os recursos associados a uma instância de `WDBusContext`. No fechamento da conexão, a `libdbus` irá invocar a `unregister_function` utilizada no registro dos objetos, que no caso da WDBus é sempre o método `wdbus_unregister_function`, liberando os recursos associados a todos os objetos registrados. Desta forma, o método `wdbus_object_free` só deve ser invocado para liberar objetos não registrados.

3.3 SUNSPEC-DAEMON

O serviço `sunspec-daemon` agrega as funcionalidades relacionadas a comunicação com DERs. Tomando a arquitetura hierárquica, descrita em 2.3.1, a *daemon* é responsável pelas comunicações entre os Níveis 1 e 2. Pela arquitetura recursiva da Subseção 2.3.2, o serviço implementa o papel de VTN do DERMS.

Dentre as funcionalidades da `sunspec-daemon` estão armazenar a relação entre endereços IP dos equipamentos e nomes significativos para os operadores do sistema, gerenciar a conexão com esses dispositivos, realizar o *polling* de seus registradores Modbus, notificar mudanças nos valores dos pontos do perfil SunSpec dos dispositivos, converter valores SunSpec em registradores Modbus e escrevê-los nos dispositivos. Além disso, a

daemon também trata de aspectos de segurança, como o gerenciamento do protocolo TLS e dos certificados envolvidos na autenticação dos DERs.

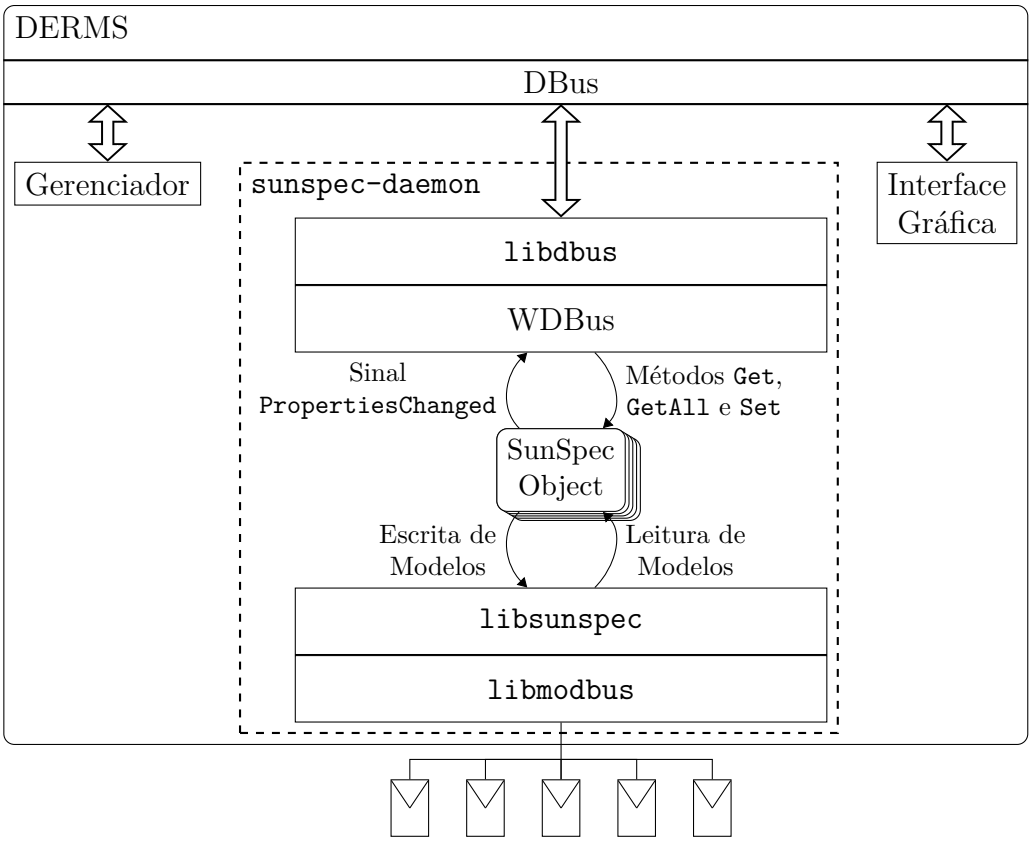


Figura 12 – Visão geral da `sunspec-daemon`.

A Figura 12 apresenta uma visão geral do funcionamento da `sunspec-daemon`. Cada dispositivo configurado no sistema é representado internamente por um objeto SunSpec, que interage com a `libsunspec` para implementar o modelo de dados SunSpec, que por sua vez baseia-se na `libmodbus` para utilizar o protocolo Modbus. Também interage com a WDBus para registrar-se no barramento DBus como um objeto implementado, além das interfaces `org.freedesktop.DBus.Introspectable` e `org.freedesktop.DBus.Properties`, uma interface para cada Modelo de Informação, com seus pontos mapeados em propriedades e sem nenhum método.

O gerenciamento dos objetos é baseado em um laço de eventos, que monitora os dispositivos e o barramento DBus. Tanto `libsunspec` quanto a `libmodbus` possuem APIs bloqueantes, sendo necessário o uso de *threads* para interagir com múltiplos dispositivos. De outra forma, um dispositivo com maior latência na comunicação poderia degradar o desempenho dos demais.

As interfaces dessas bibliotecas são detalhadas nas Subseções 3.3.1 e 3.3.2, bem como as modificações realizadas para melhorar o desempenho e segurança da aplicação. Em seguida, a operação da `sunspec-daemon` é descrita em 3.3.3 e os detalhes de implementação são apresentados em 3.3.4.

3.3.1 Biblioteca `libmodbus`

A `libmodbus` é uma biblioteca livre que implementa o envio e recebimento de dados seguindo as especificações do protocolo Modbus, tanto para operação como mestre quanto para operação como escravo. A biblioteca é escrita na linguagem C e suporta comunicações RTU e TCP. Comunicações RTU são implementadas por leituras e escritas em dispositivos de caracteres de sistemas *Unix* e derivados. Comunicações TCP utilizam a interface de soquetes definida por POSIX (2016).

Embora Modbus Organization (2018) especifique o uso de TLS para proteger o transporte do protocolo, a biblioteca ainda não suporta oficialmente tal funcionalidade. Para garantir a operação segura, portanto, foi necessário modificá-la, conforme descrito por Ferst (2018). Os tópicos a seguir apresentaram o API, com uma visão geral do funcionamento da `libmodbus`, e as modificações feitas para adição do trasporte sobre TLS.

API da `libmodbus`

Um mestre Modbus que se comunica com escravos via TCP pode ser implementado como

```
1  modbus_t *mb;
2  uint16_t tab_reg[32];
3
4  mb = modbus_new_tcp(MODBUS_SLAVE_ADDRESS, 502);
5  modbus_connect(mb);
6
7  modbus_read_registers(mb, 0, 5, tab_reg);
8
9  modbus_close(mb);
10 modbus_free(mb);
```

Nas linhas 4 e 5 o cliente cria um contexto Modbus, com o método `modbus_new_tcp`, e se conecta a um escravo com endereço definido pela macro `MODBUS_SLAVE_ADDRESS` por meio do método `modbus_connect`. Em seguida, cinco registradores são lidos na linha 7 com o método `modbus_read_registers`, iniciando pelo registrador zero. Por fim as linhas 9 e 10 encerram a conexão e liberam os recursos alocados, respectivamente, com os métodos `modbus_close` e `modbus_free`.

Além do método de leitura `modbus_read_registers` para registradores, a `libmodbus` oferece em sua interface os métodos `modbus_read_bits`, `modbus_read_input_bits` e `modbus_read_input_registers` para bobinas, entradas discretas e registradores de entrada, respectivamente. Para escrita, os métodos `modbus_write_bits` e `modbus_write_registers` são definidos para

bobinas e registradores, além do método para leitura e escrita em uma transação, `modbus_write_and_read_registers` específico para registradores.

Um servidor Modbus que atende uma única requisição TCP pode ser implementado como

```
1  modbus_t *mb;
2  modbus_mapping_t *map;
3  int s, query_size;
4
5  mb = modbus_new_tcp("0.0.0.0", 502);
6
7  mb_mapping = modbus_mapping_new_start_address(
8          BITS_ADDRESS, BITS_NB,
9          INPUT_BITS_ADDRESS, INPUT_BITS_NB,
10         REGISTERS_ADDRESS, REGISTERS_NB,
11         INPUT_REGISTERS_ADDRESS, INPUT_REGISTERS_NB);
12
13 s = modbus_tcp_listen(ctx, 1);
14 modbus_tcp_accept(ctx, &s);
15
16 query_size = modbus_receive(ctx, query);
17 modbus_reply(ctx, query, query_size, map);
18
19 modbus_close(mb);
20 modbus_mapping_free(map);
21 modbus_free(mb);
```

O método `modbus_new_tcp` é novamente usado, na linha 5, para criar o contexto Modbus. A chamada a `modbus_mapping_new_start_address`, entre as linhas 7 e 11, cria um mapeamento de `BITS_NB` bobinas, `INPUT_BITS_NB` entradas discretas, `REGISTERS_NB` registradores e `INPUT_REGISTERS_NB` registradores de entrada, iniciando respectivamente nos endereços `BITS_ADDRESS`, `INPUT_BITS_ADDRESS`, `REGISTER_ADDRESS` e `INPUT_REGISTERS_ADDRESS`. Caso deseje-se criar um mapeamento com todos os endereços iniciando em zero, o método `modbus_mapping_new` pode ser utilizado.

Em seguida, na linha 13, a chamada a `modbus_tcp_listen` retorna o descritor de arquivo para aceitar conexões na porta informada ao criar o contexto Modbus. Este descritor é então usado na linha 14 para invocar `modbus_tcp_accept`, que aceita a conexão de um cliente de forma bloqueante. O método `modbus_receive` é utilizado na linha 16 para receber uma requisição, que é respondida com o método `modbus_reply` na linha 17.

Por fim, `modbus_close` é utilizado na linha 19 para fechar a conexão, e os métodos `modbus_mapping_freee` e `modbus_free` são invocados para liberar os recursos associados ao mapeamento e contexto Modbus, nas linhas 20 e 21.

A estrutura que armazena o contexto Modbus, `modbus_t`, é definida como

```
struct _modbus {
  /* Slave address */
  int slave;
  /* Socket or file descriptor */
  int s;
  int debug;
  int error_recovery;
  struct timeval response_timeout;
  struct timeval byte_timeout;
  const modbus_backend_t *backend;
  void *backend_data;
};
...
typedef struct _modbus modbus_t;
```

Ao invocar os métodos que realizam algum tipo de comunicação, como `modbus_read_registers`, a biblioteca utiliza os métodos apontados pelo membro `backend` para manipular os dados armazenados na estrutura apontada pelo membro `backend_data`. A estrutura `modbus_backend_t` é definida como

```
typedef struct _modbus_backend {
  unsigned int backend_type;
  unsigned int header_length;
  unsigned int checksum_length;
  unsigned int max_adu_length;
  int (*set_slave) (modbus_t *ctx, int slave);
  int (*build_request_basis) (modbus_t *ctx, int function, int addr,
                              int nb, uint8_t *req);
  int (*build_response_basis) (sft_t *sft, uint8_t *rsp);
  int (*prepare_response_tid) (const uint8_t *req, int *req_length);
  int (*send_msg_pre) (uint8_t *req, int req_length);
  ssize_t (*send) (modbus_t *ctx, const uint8_t *req, int req_length);
  int (*receive) (modbus_t *ctx, uint8_t *req);
  ssize_t (*recv) (modbus_t *ctx, uint8_t *rsp, int rsp_length);
  int (*check_integrity) (modbus_t *ctx, uint8_t *msg,
                          const int msg_length);
  int (*pre_check_confirmation) (modbus_t *ctx, const uint8_t *req,
                                 const uint8_t *rsp, int rsp_length);
  int (*connect) (modbus_t *ctx);
  void (*close) (modbus_t *ctx);
  int (*flush) (modbus_t *ctx);
  int (*select) (modbus_t *ctx, fd_set *rset, struct timeval *tv,
                 int msg_length);
  void (*free) (modbus_t *ctx);
} modbus_backend_t;
```

Os métodos apontados por esta estrutura utilizarão as informações contidas em `backend_data` de `modbus_t` para realizar a comunicação. Este ponteiro não possui tipo definido porque o tipo de estrutura a ser apontada depende do tipo de comunicação do contexto Modbus. Para comunicações TCP uma estrutura do tipo `modbus_tcp_t` é utilizada e para comunicações RTU utiliza-se uma estrutura do tipo `modbus_rtu_t`. Suas definições são

```
1  typedef struct _modbus_tcp {
2    uint16_t t_id;
3    /* TCP port */
4    int port;
5    /* IP address */
6    char ip[16];
7  } modbus_tcp_t;
8
9  typedef struct _modbus_rtu {
10   /* Device: "/dev/ttyS0", "/dev/ttyUSB0" or "/dev/tty.USA19*" on Mac OS
       ↪ X. */
11   char *device;
12   /* Bauds: 9600, 19200, 57600, 115200, etc */
13   int baud;
14   /* Data bit */
15   uint8_t data_bit;
16   /* Stop bit */
17   uint8_t stop_bit;
18   /* Parity: 'N', 'O', 'E' */
19   char parity;
20 #if defined(_WIN32)
21   struct win32_ser w_ser;
22   DCB old_dcb;
23 #else
24   /* Save old termios settings */
25   struct termios old_tios;
26 #endif
27 #if HAVE_DECL_TIOCSRS485
28   int serial_mode;
29 #endif
30 #if HAVE_DECL_TIOCM_RTS
31   int rts;
32   int rts_delay;
33   int onebyte_time;
34   void (*set_rts) (modbus_t *ctx, int on);
35 #endif
36   /* To handle many slaves on the same link */
37   int confirmation_to_ignore;
38 } modbus_rtu_t;
```

Modificações realizadas

Um novo *backend* foi desenvolvido para suportar comunicações sobre TLS. Sua implementação consiste no conjunto de métodos requeridos pela estrutura `modbus_backend_t` e uma nova estrutura semelhante a `modbus_tcp_t` para armazenar o contexto da conexão TLS. A inclusão deste *backend* foi feita opcional por meio de opções a serem habilitadas no sistema de compilação da `libmodbus`. Adicionalmente, é possível escolher entre duas bibliotecas que implementam do protocolo TLS: `openssl` e `mbedtls`

A nova estrutura desenvolvida foi nomeada `modbus_tls_t` e sua definição é a seguinte:

```
1  typedef struct _modbus_tls {
2      /* Transaction ID */
3      uint16_t t_id;
4      /* TCP port */
5      int port;
6      /* IP address */
7      char ip[16];
8  #if defined(USE_OPENSSL)
9    SSL_CTX *ctx;
10   SSL *ssl;
11 #elif defined(USE_MBEDTLS)
12   mbedtls_entropy_context entropy;
13   mbedtls_ctr_drbg_context drbg;
14   mbedtls_x509_crt cert;
15   mbedtls_pk_context pk;
16   mbedtls_ssl_config cfg;
17   mbedtls_ssl_context ctx;
18 #endif
19 } modbus_tls_t;
```

Os primeiros campos da estrutura, entre as linha 3 e 7, são declarados na mesma ordem e tipos da estrutura `modbus_tcp_t`, possibilitando a conversão do tipo `modbus_tls_t` para `modbus_tcp_t` a fim de que alguns dos métodos do *backend* TCP possam ser reutilizados, evitando a duplicidade de códigos de mesma finalidade.

O restante da `modbus_tls_t` armazena as estruturas necessárias para uma sessão e uma conexão TLS, de acordo com a escolha da biblioteca que implementa este protocolo. Assim como `modbus_tcp_t` e `modbus_rtu_t`, a estrutura é feita opaca ao usuário da `libmodbus`, que deve manipulá-la apenas por meio dos métodos expostos pela API da biblioteca.

Para criar uma instância de `modbus_t` baseada em `modbus_tls_t`, o método `modbus_new_tls` deve ser utilizado, cujos argumentos são os mesmos de `modbus_new_tcp` acrescidos de *strings* com o caminho para os arquivos que contenham o certificado lo-

cal, a chave deste certificado e o certificado da autoridade certificadora. Análogo aos métodos públicos do *backend* TCP `modbus_tcp_listen` e `modbus_tcp_accept`, os métodos `modbus_tls_listen` e `modbus_tls_accept` são definidos para criação do descritor de arquivos do servidor e para aceitar conexões.

Por fim, a estrutura do tipo `modbus_backend_t` para uso com `modbus_tls_t` foi declarada como

```
const modbus_backend_t _modbus_tls_backend = {
    _MODBUS_BACKEND_TYPE_TCP,
    _MODBUS_TCP_HEADER_LENGTH,
    _MODBUS_TCP_CHECKSUM_LENGTH,
    MODBUS_TCP_MAX_ADU_LENGTH,
    _modbus_set_slave,
    _modbus_tcp_build_request_basis,
    _modbus_tcp_build_response_basis,
    _modbus_tcp_prepare_response_tid,
    _modbus_tcp_send_msg_pre,
    _modbus_tls_send,
    _modbus_tcp_receive,
    _modbus_tls_recv,
    _modbus_tcp_check_integrity,
    _modbus_tcp_pre_check_confirmation,
    _modbus_tls_connect,
    _modbus_tls_close,
    _modbus_tls_flush,
    _modbus_tls_select,
    _modbus_tls_free
};
```

Nota-se que os membros como `set_slave` e `build_request_basis` puderam ser reaproveitados do *backend* TCP, enquanto membros como `send` e `recv` receberam uma implementação própria para o *backend* TLS. Ferst (2018) detalha a implementação dos novos métodos e as modificações feitas ao sistema de compilação.

3.3.2 Biblioteca `libsunspec`

Com a finalidade de facilitar a adoção de seus padrões, a SunSpec Alliance provê um conjunto de bibliotecas e *software* de código aberto que servem tanto como implementações de referência de suas normas quanto como base para o desenvolvimento de novas aplicações na área. Dentre as bibliotecas fornecidas encontra-se a `libsunspec`, que implementa tanto a comunicação no padrão Modbus RTU quanto o modelo de dados descrito por (SunSpec Alliance, 2015) e (SunSpec Alliance, 2017).

Os Modelos de Informação são lidos de arquivos XML e representados por estruturas em C geradas dinamicamente. O procedimento de descoberta de modelos também

é implementado, utilizando a definição dos modelos lidos para montar uma lista ligada dos modelos de cada dispositivo.

Seu desenvolvimento, entretanto, parece estar paralisado desde 2015[1], e diversos *bugs* foram encontrados e corrigidos durante o desenvolvimento da `sunspec-daemon`, como vazamentos de memória, *buffer overflows* e acessos a dados não inicializados. Além disso, melhorias foram realizadas na API da biblioteca para viabilizar uma operação eficiente sobre seu modelo de dados.

A seguir, os Tópicos 3.3.2.1 e 3.3.2.2 detalham a estrutura de dados da biblioteca e apresentam sua API por meio de exemplos. Por fim, o Tópico 3.3.2.3 lista algumas das mudanças realizadas na `libsunspec`.

Estruturas de dados

Embora a biblioteca seja desenvolvida na linguagem C, seu modelo de dados pode ser interpretado pela perspectiva do paradigma de orientação a objetos para gerar um diagrama de classes que ajude a entender seu funcionamento. Este diagrama é apresentado de forma simplificada na Figura 13.

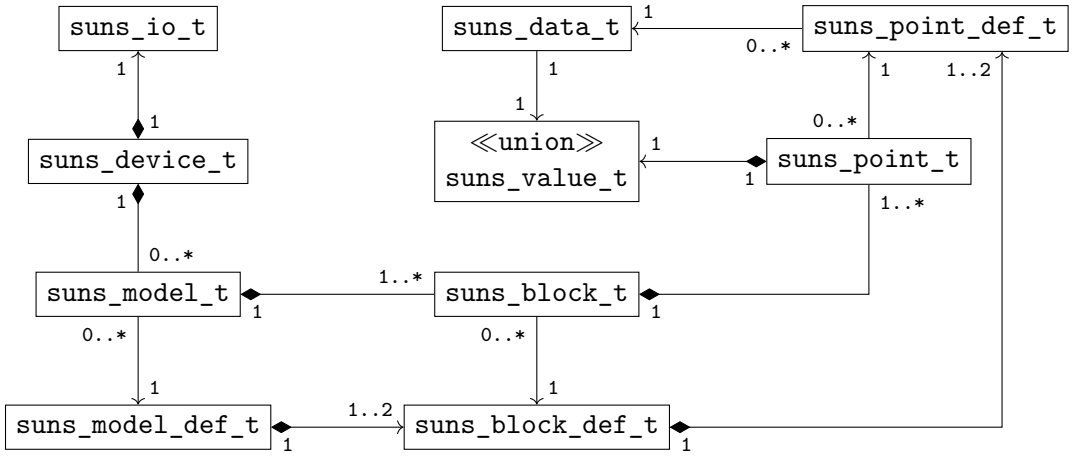


Figura 13 – Diagrama de classes simplificado da biblioteca `libsunspec`.

A estrutura `suns_model_def_t` armazena a definição de um modelos SunSpec lido de um arquivo XML, e é definida da seguinte forma:

```
1 typedef struct _suns_model_def_t {
2     uint16_t id, len;
3     char id_str[SUNS_MODEL_ID_LEN];
4     char name[SUNS_MODEL_NAME_LEN];
5     suns_block_def_t *blocks[SUNS_BLOCK_TYPE_COUNT];
6     struct _suns_model_def_t *next;
7 } suns_model_def_t;
```

[1] <https://github.com/sunspec/libsunspec/commits/master>

Os membros `id` e `id_str` armazenam a ID do modelo como um inteiro e como uma *string*. Os membros `len` e `name` armazenam, respectivamente, o tamanho e nome do modelo conforme a especificação. O membro `next` permite formar uma lista ligada simples com a estrutura `suns_model_def_t`. Quando o procedimento de descoberta de um dispositivo requisita um determinado modelo, a `libsunspec` verifica se este está em uma lista global de modelos interna à biblioteca e, do contrário, carrega sua definição do arquivo XML.

O membro `blocks` armazena dois ponteiros para do tipo `suns_block_def_t`, sendo `blocks[0]` para blocos fixos e `blocks[1]` para blocos de repetição, se o modelo possuir. A estrutura `suns_block_def_t` é definida como

```
1  typedef struct _suns_block_def_t {
2      uint16_t len;
3      uint8_t repeating;
4      suns_point_def_t *points;
5      struct _suns_block_def_t *next;
6  } suns_block_def_t;
```

Novamente, o membro `next` do mesmo tipo da estrutura possibilita a construção de uma lista ligada simples com instâncias de `suns_block_def_t`. Os membros `len` e `repeating` armazenam o tamanho do bloco e seu tipo – fixo ou repetição. O membro `points` contém a lista de pontos do bloco, representados por estruturas do tipo `suns_point_def_t`.

A definição de `suns_point_def_t` é a seguinte

```
1  typedef struct _suns_point_def_t {
2      char *id;
3      uint16_t offset;
4      suns_data_t *type;
5      uint16_t len;
6      uint8_t required;
7      uint8_t access;
8      char *sf_name;
9      uint16_t sf_offset;
10     int16_t sf_value;
11     char *units;
12     struct _suns_point_def_t *next;
13 } suns_point_def_t;
```

O membro `id` é uma *string* com o nome do ponto. Os membros `offset`, `len`, `required` e `access` armazenam respectivamente o deslocamento do ponto em relação ao início do bloco, o tamanho do ponto, se o ponto é obrigatório ou opcional, e se o ponto é apenas leitura ou pode ser escrito.

Se o ponto possuir um fator de escala, o nome do ponto `sunssf` com o fator estará armazena no membro `sf_name`, e o deslocamento deste ponto em relação ao início do bloco estará em `sf_offset`. A unidade do ponto é armazenada como uma *string* em `unit` e `sf_value` não é utilizado.

O ponteiro `type` de `suns_point_def_t` aponta para uma estrutura do tipo `suns_data_t`, que possui a seguinte definição

```
1  typedef struct _suns_data_t {
2      char *id;
3      int16_t type;
4      int16_t base_type;
5      int16_t len;
6      int16_t has_unimpl_value;
7      suns_value_t unimpl_value;
8      void (*to_float)(suns_value_t value, int16_t sf, float *f32);
9      void (*from_float)(suns_value_t *value, int16_t sf, float f32);
10     void (*to_str)(suns_value_t value, int16_t sf, char *str, uint16_t
       ↪ len);
11     uint16_t (*is_implemented)(suns_value_t value);
12     uint16_t (*modbus_to_value)(unsigned char *buf, suns_value_t *value,
       ↪  uint16_t len);
13     void (*modbus_from_value)(unsigned char *buf, suns_value_t value,
       ↪ uint16_t len);
14 } suns_data_t;
```

Diferente das demais estruturas apresentadas, as instâncias de `suns_data_t` são definidas estaticamente, em tempo de compilação, e representam os tipos SunSpec descritos na Seção 2.4.2. Os membros `id` e `type` indicam o tipo SunSpec representado por essa instância de `suns_data_t`, respectivamente como uma *string* e com um enumerador. O membro `base_type` é um enumerador do mesmo tipo de `type` e indica uma forma alternativa de tratar o tipo. Acumuladores e enumeradores possuem como `base_type` o inteiro sem sinal de mesmo tamanho, por exemplo, enquanto `sunssf` tem como `base_type` `int16`.

O membro `len` indica o tamanho do tipo, em número de registradores, enquanto `unimpl_value` e `has_unimpl_value` indicam o valor de NaN do tipo e se este valor indica erro, respectivamente. Os métodos apontados por `to_float` e `from_float` convertem o tipo representado por essa instância de `suns_data_t` para ponto flutuante e de ponto flutuante para o tipo SunSpec, respectivamente. O método `to_str` gera uma *string* que representa o valor do ponto. O método `is_implemented` verifica se o valor do ponto é igual ao valor armazenado em `unimpl_value`. Por fim, os métodos `modbus_from_value` e `modbus_to_value` convertem o ponto de e para registradores Modbus, respectivamente.

Retomando o diagrama de classes da Figura 13, observa-se que a relação entre `suns_model_def_t` e `suns_block_def_t` são de composição. Isso se deve ao fato de que as instâncias de `suns_block_def_t` são alocadas na criação de um `suns_model_def_t`, e não são compartilhadas entre instâncias de `suns_model_def_t`. Além disso, ao destruir a definição de um modelo, seus blocos são automaticamente desalocados pela `libsunspec`.

De maneira semelhante, as estruturas `suns_point_def_t` são criadas ao instanciar um `suns_block_def_t`, e outras definições de bloco não compartilham um mesmo

ponto. O tempo de vida de um objeto `suns_point_def_t` também é associado ao do objeto `suns_block_def_t` que o ponto pertence. Assim sendo, a relação entre as estruturas `suns_block_def_t` e `suns_point_def_t` também é de composição, conforme ilustrado na Figura 13. Por outro lado, o mesmo diagrama mostra apenas uma associação entre `suns_point_def_t` e `suns_data_t`, pois a alocação estática de `suns_data_t` e o compartilhamento das mesmas estruturas em múltiplas instâncias de `suns_point_def_t` fazem com que a relação entre as duas estruturas não constitua posse.

Além das estruturas para implementar as definições dos Modelos de Informação, a `libsunspec` também oferece estruturas para representar dispositivos SunSpec e armazenar o valor dos pontos de seus modelos. Um dispositivo é instanciado com a estrutura `suns_device_t`, definida como

```
1 typedef struct _suns_device_t {
2     uint16_t base_addr;
3     suns_modbus_io_t modbus_io;
4     suns_model_t *models;
5 } suns_device_t;
```

O membro `base_addr` armazena o endereço base do mapeamento de modelos, conforme encontrado pelo procedimento de descoberta dos modelos. O membro `modbus_io` possui o tipo `suns_io_t`, que é definido como

```
1 typedef struct suns_io_t {
2     suns_io_close_func_t close;
3     suns_io_connect_func_t connect;
4     suns_io_disconnect_func_t disconnect;
5     suns_io_read_func_t read;
6     suns_io_write_func_t write;
7     suns_io_flush_func_t flush;
8     void *prot;
9 } suns_io_t;
```

O funcionamento dessa estrutura é semelhante ao de `modbus_backend_t` da `libmodbus`, com a notável diferença de que o ponteiro de dados específicos do *backend* está armazenado na estrutura, no membro `prot`. Na `libmodbus` esse ponteiro está na estrutura de contexto `modbus_t`, permitindo que diferentes dispositivos compartilhem uma mesma instância de `modbus_backend_t`. Por outro lado, cada instância de `suns_device_t` deve possuir uma instância própria de `suns_io_t`, formando a relação de composição expressa no diagrama de classes da Figura 13.

Originalmente, a `libsunspec` oferece implementações dos métodos da estrutura `suns_io_t` para simulação, com escritas em memória, e para comunicações Modbus RTU sobre a interface ANSI/CEA-2045 (EPRI, 2014). Métodos adicionais foram desenvolvidos para utilização da biblioteca `libmodbus` para comunicações Modbus RTU, Modbus TCP e Modbus TCP protegido por TLS.

O membro `models` de `suns_device_t` aponta para uma instância de `suns_model_t`, cujo tipo é definido como

```
1  typedef struct _suns_model_t {
2      struct _suns_device_t *device;
3      uint16_t id;
4      uint16_t len;
5      uint16_t index;
6      uint16_t addr;
7      uint16_t block_count;
8      suns_model_def_t *model_def;
9      struct _suns_model_t *next;
10     suns_block_t *blocks[1];/* array is sized during model allocation */
11 } suns_model_t;
```

A estrutura constrói uma lista ligada, por meio do membro `next`, formando a cardinalidade `0..*` do diagrama da Figura 13. O membro `device`, que aponta para a estrutura `suns_device_t` a que esta instância de `suns_model_t` pertence, evidencia a posse de `suns_device_t` sobre `suns_model_t`, gerando a relação de composição entre as duas estruturas e tornando a navegabilidade bidirecional.

O membro `model_def` aponta para a definição do Modelo de Informação em uma estrutura do tipo `suns_model_def_t`. Os membros `id` e `len` armazenam, respectivamente a ID e tamanho do modelo, conforme lido do mapeamento. No caso de `id`, o dado é armazenado em redundância, pois também está presente no membro de mesmo nome de `suns_model_def_t`. Para `len`, entretanto, o valor pode diferir da definição do modelo, casos existam registradores de *padding* ao final do modelo.

Os membros `addr` e `block_count` armazenam, respectivamente, o endereço inicial do mapeamento do modelo e o número de blocos. O membro `index` possibilita que um mesmo dispositivo possua múltiplas instâncias de um mesmo modelo. O membro `blocks` é um FAM[2] que armazena estruturas do tipo `suns_block_t`, que são definidas como:

[2]Segundo ISO (1999), um FAM deve ser declarado sem tamanho. Entretanto, antes da padronização do recurso ou da implementação pelos compiladores, alguns programadores já utilizavam FAMs independentemente do suporte oferecido pelas ferramentas. Em alguns casos, o compilador permite a declaração de um *array* de tamanho zero e, em outros, o membro é declarado com tamanho um. A definição de `suns_model_t` utiliza esta última forma para declarar `suns_block_t`, provavelmente buscando a compatibilidade com o maior número possível de compiladores. Essa forma modifica o valor retornado por `sizeof` e, portanto, cuidados devem ser tomados ao utilizar o operador nessas estruturas.

```
typedef struct _suns_block_t {
    struct _suns_model_t *model;
    suns_block_def_t *block_def;
    uint16_t addr;
    uint16_t type;
    uint16_t index;
    suns_point_t *points;
    suns_point_t *points_sf;
} suns_block_t;
```

O membro `model` aponta para o `suns_model_t` a que este `suns_block_t` pertence, novamente tornado a navegabilidade entre as estruturas bidirecional e evidenciando a relação de composição. O membro `block_def` aponta para a definição do bloco no Modelo de Informações. O membro `addr` armazena o endereço inicial do bloco no mapeamento Modbus. Embora a informação já esteja presente na estrutura `suns_block_def_t`, o membro `type` indica se o bloco é fixo ou de repetição. Caso seja de repetição, o membro `index` terá o índice do bloco.

O membro `points_sf` não é utilizado. Já `points` armazena a lista de pontos do bloco, cujo tipo `suns_point_t` é definido como

```
typedef struct _suns_point_t {
    struct _suns_block_t *block;
    suns_point_def_t *point_def;
    uint16_t addr;
    struct _suns_point_t *sf_point;
    uint8_t impl;
    suns_value_t value_base;
    uint16_t value_sf;
    char *value_ptr;
    uint8_t dirty;
    struct _suns_point_t *next;
} suns_point_t;
```

O membro `block` aponta para o bloco a que o ponto pertence, formando novamente uma navegabilidade bidirecional. A relação de posse entre `suns_block_t` e `suns_point_t` também torna a ligação entre as estruturas uma composição. O membro `point_def` aponta para a uma estrutura do tipo `suns_point_def_t`, pela qual a aplicação poderá obter o nome do ponto e os métodos para manipulá-lo, por meio da estrutura `suns_data_t`.

O membro `addr` indica o endereço em que o ponto inicia no mapeamento Modbus e o membro `sf_point` aponta para o ponto `sunssf` com o fator de escala desse ponto, caso ele possua. O valor de `dirt` indica se a aplicação modificou o ponto desde a última leitura do dispositivo. Ao requisitar a escrita dos modelos no dispositivo, a `libsunspec` apenas escreverá os pontos cujo valor de `dirt` seja verdadeiro. Os membros `impl` e `value_sf` não são utilizados.

O membro `value_base` é uma união do tipo `suns_value_t`, definida como

```
typedef union {
    int16_t s16;
    uint16_t u16;
    int32_t s32;
    uint32_t u32;
    int64_t s64;
    uint64_t u64;
    float f32;
    double f64;
    char *str;
} suns_value_t;
```

Esse é o membro que armazena de fato os valores lidos do dispositivo. A aplicação pode definir qual campo de `suns_value_t` deve ser utilizado conforme as informações da estrutura `suns_point_def_t` do ponto. Finalmente, o membro `value_ptr` aponta para `value_base.str` caso o ponto seja uma *string*. Caso o ponto seja de outro tipo, `value_ptr` é nulo.

Exemplos de utilização

O seguinte código pode ser utilizado para ler o ponto `POINT_NAME` do modelo `MODEL_ID`:

```
suns_device_t *device;
suns_model_t *model;
suns_point_t *point;
char value_str[VALUE_STR_SIZE];

device = suns_device_alloc();
suns_device_libmodbus_tcp(device, SLAVE_ADDR, SLAVE_PORT);
suns_device_connect(device, 0);
suns_device_scan(device);

model = suns_device_get_model(device, MODEL_ID, NULL, 1);
suns_model_read(model, NULL, NULL);
point = suns_model_get_point(model, POINT_NAME, 0);
point->point_def->type->to_str(point->value_base,
        point->sf_point?point->sf->value_base.s16:0,
        value_str, VALUE_STR_SIZE);

printf("%s: %s\n", point->point_def->id, value_str);
suns_device_disconnect(device);
suns_device_close(device);
suns_device_free(device);
```

Primeiramente, uma instância de `suns_device_t` é alocada na linha 6 com uma chamada a `suns_device_alloc`, que não recebe nenhum argumento. Em seguida, o método `suns_device_libmodbus_tcp` é utilizado na linha 7 para popular a estrutura `suns_io_t` com os métodos da `libmodbus` e alocar o contexto Modbus `modbus_t`.

Na linha 8, o método `suns_device_connect` realiza a conexão com o dispositivo e, em seguida, `suns_device_scan` é invocado para realizar o procedimento de descoberta de modelos. A estrutura `suns_model_t` do modelo `MODEL_ID` é então recuperada com método `suns_device_get_model`, na linha 11. Esse método recebe quatro argumentos: a estrutura `suns_device_t`, o ID do modelo como um inteiro, o nome do modelo como uma *string* e o índice do modelo, iniciando em um. O segundo argumento só é considerado se o terceiro for nulo.

Os registradores do modelo são então lidos do dispositivo com `suns_model_read`, na linha 12. Além da estrutura `suns_model_t`, o método recebe como argumento um ponteiro de função e um ponteiro de dados do usuário. Ao receber os registradores atualizados do dispositivo, `suns_model_read` invoca o método apontado para cada ponto cujo valor mudou, passando como argumentos a estrutura `suns_point_t` e o ponteiro de dados do usuário.

A estrutura `suns_point_t` é obtida da lista de pontos do modelo com o método `suns_model_get_point`, na linha 13. O método recebe a estrutura `suns_model_t`, o nome do ponto como uma *string* e o índice do ponto, para o caso de pontos em blocos de repetição. Diferente de `suns_device_get_model`, o índice informado a `suns_model_get_point` inicia em zero.

Entre as linhas 14 e 16, o método `to_str` da instância de `suns_data_t` associada ao ponto é invocado para preencher a *string* `value_str` com o valor do ponto, que é então exibido com a chamada a `printf` na linha 18. Finalmente, os métodos `suns_device_disconnect`, `suns_device_close` e `suns_device_free` são invocados, entre as linhas 19 e 21, para desconectar do dispositivo, frechar a conexão, e liberar os recursos associados.

Para escrever o valor `POINT_VALUE` nesse mesmo ponto sem alterar o fator de escala, o seguinte código pode ser utilizado:

```
1  suns_device_t *device;
2  suns_model_t *model;
3  suns_point_t *point;
4  suns_value_t value = {.u16 = POINT_VALUE};
5
6  device = suns_device_alloc();
7  suns_device_libmodbus_tcp(device, SLAVE_ADDR, SLAVE_PORT);
8  suns_device_connect(device, 0);
9  suns_device_scan(device);
10
11 model = suns_device_get_model(device, MODEL_ID, NULL, 1);
```

```
12
13 point = suns_model_get_point(model, POINT_NAME, 0);
14 suns_point_set_uint16(point, value.u16,
15         point->sf_point?point->sf_point->value_base.s16:0);
16
17 suns_model_write(model);
18
19 suns_device_disconnect(device);
20 suns_device_close(device);
21 suns_device_free(device);
```

Até a linha 11, o mesmo procedimento do exemplo anterior é realizado para alocar a estrutura `suns_device_t`, conectar ao dispositivo, descobrir seus modelos e recuperar a estrutura `suns_model_t`. Em seguida, a estrutura `suns_point_t` é recuperada com método `suns_model_get_point`, na linha 13.

Nas linhas 14 e 15, o método `suns_point_set_uint16` é utilizado para definir o novo valor do ponto, assumindo que seu tipo é `uint16` e mantendo o fator de escala, caso o ponto possua um. Em seguida, na linha 17, o método `suns_model_write` é utilizado para escrever os pontos atualizados do modelo no dispositivo. Por fim, o mesmo procedimento de desconexão e liberação de recursos é feito entre as linhas 19 e 21.

A interface da `libsunspec` também disponibiliza os métodos `suns_point_set_uint32` e `suns_point_set_uint64` para definir pontos inteiros sem sinal de 32 e 64 bits, `suns_point_set_int16`, `suns_point_set_int32` e `suns_point_set_int64` para definir pontos inteiros com sinal de 16, 32 e 64 bits, `suns_point_set_float32` para definir pontos *float* e `suns_point_set_str` para definir pontos do tipo `string`. Outros tipos, como `enum16` e `ipaddr`, devem utilizar o tipo base, conforme definido no membro `base_type` da instância de `suns_data_t` associada ao ponto.

Finalmente, a biblioteca fornece os métodos `suns_point_get_uint16`, `suns_point_get_uint32` e `suns_point_get_uint64` para obter o valor de pontos inteiros sem sinal de 16, 32 e 64 bits, `suns_point_get_int16`, `suns_point_get_int32` e `suns_point_get_int64` para obter o valor de pontos inteiros com sinal de 16, 32 e 64 bits, `suns_point_get_float32` para obter o valor de pontos *float* e `suns_point_get_str` para obter a *string* de pontos do tipo `string`.

Mudanças realizadas

A biblioteca disponibilizada nos repositórios da SunSpec Alliance tem suas últimas modificações no ano de 2015. Em seu estado original, inúmeros problemas foram encontrados e corrigidos. Alguns deles são:

- A lista global de `suns_model_def_t` não era atualizada corretamente. Com isso, todo modelo encontrado por `suns_device_scan` era recarregado do arquivo XML e, após liberar os recursos de `suns_device_t`, a referência para o modelo carregado era perdida, gerando um vazamento de memória.

- Outros vazamentos de memória do leitor de arquivos XML foram detectados e corrigidos com o auxílio da ferramenta Valgrind (Valgrind Developers, 2020).

- Com o auxílio das ferramentas Valgrind e Cppcheck(Cppcheck Developers, 2020), diversos uso de variáveis e regiões de memória não inicializados foram detectado e corrigidos. Também foram detectados e removidos pontos inatingíveis de código.

Além disso, algumas melhorias foram feitas na biblioteca, como

- Adição dos métodos da `suns_io_t` para utilizar a `libmodbus` como implementação do protocolo Modbus RTU e Modbus TCP, com suporte a comunicação sobre TLS.

- Embora a estrutura `suns_point_def_t` já possuísse o membro `access`, seu valor não era lido do modelo e nem considerado na operação da biblioteca. O código para obter essa informação do arquivo XML foi desenvolvido e os métodos de leitura e escrita foram modificados para considerar esse valor.

- O suporte a comunicação com ANSI/CEA-2045 depende de bibliotecas proprietárias que não estão presentes nos repositórios da SunSpec. Para viabilizar a compilação da biblioteca, o suporte a esse tipo de comunicação foi feito opcional por meio de macros do preprocessador C.

- Originalmente, o método `suns_model_read` só recebia a estrutura `suns_model_t` como argumento, e não verificava quais pontos haviam sido atualizados. Esse comportamento dificultava a detecção de mudanças nos pontos, sendo necessário manter uma cópia dos pontos antes de invocar o método e, em seguida, comparar ponto a ponto. A adição do ponteiro de função otimiza esse processo e facilita o uso da biblioteca.

3.3.3 Operação da `sunspec-daemon`

Ao iniciar, `sunspec-daemon` assume o nome `org.sunspec.Daemon` no barramento DBus e carrega do arquivo `/etc/sunspec/sunspec_daemon.conf` os dispositivos a serem monitorados. A biblioteca `inih` é utilizada para analisar o arquivo, que é escrito no formato INI. Um exemplo de configuração seria:

```
[NomeDoDER]
address = 10.13.37.100
connection_reties=5
connection_interval=5000
read_retries=8
read_interval=2500

[NomeOutroDER]
address = 10.13.37.101
port = 1502
tls = true
cert = /caminho/para/certificado.pem
key = /caminho/para/certificado.key
ca = /caminho/para/certificado_ca.pem
poll_interval=900
```

Dois dispositivos são configurados. O primeiro, nomeado `NomeDoDER`, possui o endereço IP `10.13.37.100` e porta 502, padrão do protocolo Modbus. O número de tentativas de conexão e leituras são definidos pelas opções `connection_retries` e `read_retries` como cinco e oito. Os intervalos entre essas tentativas são especificados, respectivamente, por `connection_interval` e `read_interval` como 5 segundos e 2,5 segundos.

O segundo recebe o nome `NomeOutroDER`, com endereço IP `10.13.37.101` e porta 1502. A utilização de TLS é habilitada para este dispositivo, com o certificado e chave local configurados com `cert` e `key`, e o certificado da Autoridade Certificadora (CA, do inglês *Certificate Authority*) definido por `ca`. O intervalo padrão de *polling* é alterado de 1 segundo para 900 ms. As demais configurações não presentes utilizam o valor padrão, conforme a Tabela 8.

Os valores padrão podem ser alterados com a definição de opções antes de qualquer dispositivo. Por exemplo:

```
tls=true
ca = /caminho/para/certificado_ca.pem

[NomeDoDER]
address = 10.13.37.100
tls = false
connection_reties=5
connection_interval=5000
read_retries=8
read_interval=2500

[NomeOutroDER]
address = 10.13.37.101
port = 1502
cert = /caminho/para/certificado.pem
```

```
16 key = /caminho/para/certificado.key
17 poll_interval=900
```

Configura os dispositivos com as mesmas opções anteriores, porém, novos dispositivos a serem configurados utilizarão TLS por padrão, com o certificado da CA em `/caminho/para/certificado_ca.pem`.

Chave	Valor padrão
`address`	`127.0.0.1`
`port`	502
`tls`	`false`
`connection_retries`	3
`connection_interval`	5000
`read_retries`	5
`read_interval`	3000
`poll_interval`	1000
`cert`	`/etc/sunspec/master.pem`
`key`	`/etc/sunspec/master.key`
`ca`	`/etc/ca.pem`

Tabela 8 – Valores padrão das configurações de um dispositivo da `sunspec-daemon`.

Cada dispositivo configurado é representado internamente por um objeto SunSpec. Ao ser criado, esse objeto é registrado no barramento DBus com o método `wdbus_context_register_object` no caminho `/org/sunspec/Devices/NomeDoDER`. Inicialmente o objeto possui, além das interfaces padrão fornecidas pela WDBus, apenas a interface `org.sunspec.Connection`, que expõem os métodos `Connect`, para requisitar a conexão de dispositivos desconectados, `Disconnect`, para requisitar a desconexão, e `Reconnect`, para requisitar a desconexão e reconexão do dispositivo. A interface também possui propriedades como `Address`, `Type` e `Status`, para acessar o endereço configurado do dispositivo, se comunicação é feita por Modbus TCP ou TLS, e o estado da conexão.

Após a criação, o objeto segue o comportamento descrito pela máquina de estados da Figura 14, cujos símbolos estão descritos nas Tabelas 9a e 9b. Inicialmente no estado “Conectando”, o método `suns_device_connect` é invocado por até N vezes para realizar a conexão, sendo N obtido da opção `connection_retries`. Quando a conexão é estabelecida, o objeto irá para o estado “Escaneando o perfil”, em que o método `suns_device_scan` é utilizado para a descoberta dos modelos implementados.

Para cada modelo encontrado, o método `wdbus_object_register_interface` será invocado para registrar uma interface com o nome `org.sunspec.Model.N`, composta apenas por propriedades que refletem os pontos do modelo `N`. As instâncias de `WDBusInterface` são geradas dinamicamente a partir das estruturas `suns_model_def_t` construídas pela `libsunspec` dos arquivos XML de cada modelo, mantendo uma lista

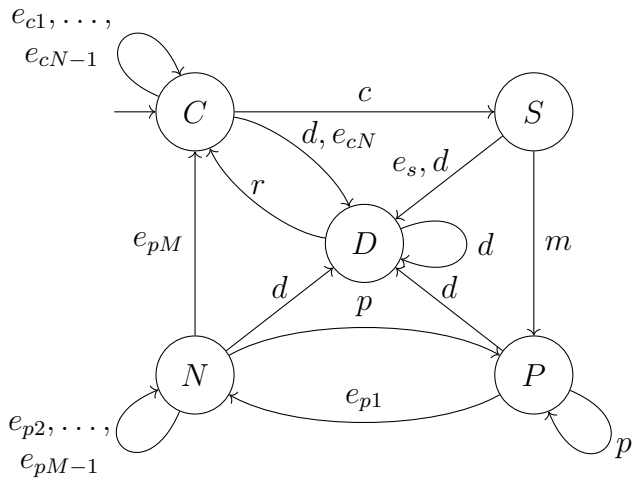


Figura 14 – Máquina de estados de um objeto SunSpec.

Estado	Símbolo
Conectando	C
Escaneando o perfil	S
Polling	P
Não está respondendo	N
Desconectado	D

(a) Estados

Transição	Símbolo
Conectado	c
n-ésimo erro ao conectar	e_{cn}
Perfil mapeado	m
Erro ao mapear perfil	e_s
poll	p
n-ésimo erro de *poll*	e_{pn}
Desconexão requisitada	d
Reconexão requisitada	r

(b) Transições

Tabela 9 – Símbolos da máquina de estados de um objeto SunSpec.

global de `WDBusInterface` já instanciadas. A Tabela 10 apresenta a relação entre tipos SunSpec e DBus.

Após o mapeamento dos modelos, o objeto irá para o estado "*Polling*", em que os métodos `suns_model_write` e `suns_model_read` serão invocados para cada modelo no intervalo de *polling* configurado. A interface de *callback* de `suns_model_read` é utilizada para montar o sinal `PropertiesChanged` com os pontos que mudaram de valor e, após o retorno do método, `wdbus_object_emit_signal` é invocado para emitir o sinal se ao menos um ponto tiver sido atualizado.

Caso `suns_model_write` ou `suns_model_read` falhe, o objeto irá para o estado "Não está respondendo", onde outras $M - 1$ tentativas de realizar a escrita ou leitura do modelo serão feitas, com M obtido da opção `read_retries`. Se uma delas obtiver êxito, o objeto voltará para o estado de "*Polling*". Do contrário, o dispositivo será desconectado, com uma chamada a `suns_device_disconnect` e o objeto irá para o estado "Conectando".

Se todas as tentativas de conexão falhem, `suns_device_scan` não encontrar um mapeamento válido dos modelos ou o método `Disconnect` da interface

D-Bus	SunSpec	D-Bus	SunSpec
int16	int16	int32	int32
	sunssf	uint32	uint32
uint16	uint16		acc32
	count		enum32
	acc16		bitfield32
	enum16	double	float
	bitfield16	int64	int64
string	string	uint64	uint64
	ipaddr		acc64
	ipv6addr		

Tabela 10 – Relação entre os tipos definidos pelo padrão SunSpec e pelo protocolo D-Bus.

`org.sunspec.Connection` for invocado, o objeto irá para o estado "Desconectado", onde ficará até que uma reconexão seja requisitada pelos métodos `Connect` ou `Reconnect` da mesma interface. Enquanto desconectado, outros serviços ainda podem acessar as propriedades das demais interfaces, mas os valores dos pontos não serão atualizados e nem escritos no dispositivo.

Uma aplicação que deseja interagir com um DER, como o serviço do gerenciador ou a interface gráfica, pode descobrir os modelos implementados pelo dispositivo invocando o método `Introspect` da interface `org.freedesktop.DBus.Introspectable`. Para obter e atualizar o valor de um ponto de um modelo, os método `Get` e `Set` de `org.freedesktop.DBus.Properties` podem ser utilizados, e a evolução de um ponto pode ser acompanhada pelo sinal `PropertiesChanged`. Esse sinal também é emitido para as propriedades da interface `org.sunspec.Connection` e pode ser utilizado, portanto, para monitorar o estado da conexão.

3.3.4 Implementação da `sunspec-daemon`

A implementação da `sunspec-daemon` é baseada em um laço de eventos que monitora os dispositivos e o barramento DBus. Como as bibliotecas `libsunspec` e `libmodbus` possuem apenas APIs bloqueantes, faz-se necessária a utilização de *threads* para que dispositivos mais lentos não degradem o desempenho geral do sistema. A solução empregada é ilustrada pela Figura 15.

Uma *thread* principal monitora, por meio de uma *epoll*, os eventos do barramento DBus e dos objetos SunSpec. Eventos do barramento são obtidos monitorando o descritor de arquivo da *epoll* de WDBus, retornado da chamada a `wdbus_context_get_epdf`. Os eventos de temporização de um dispositivo, como o intervalo de *polling* ou o tempo de espera entre as tentativas reconexão, são gerados com um temporizador associado ao

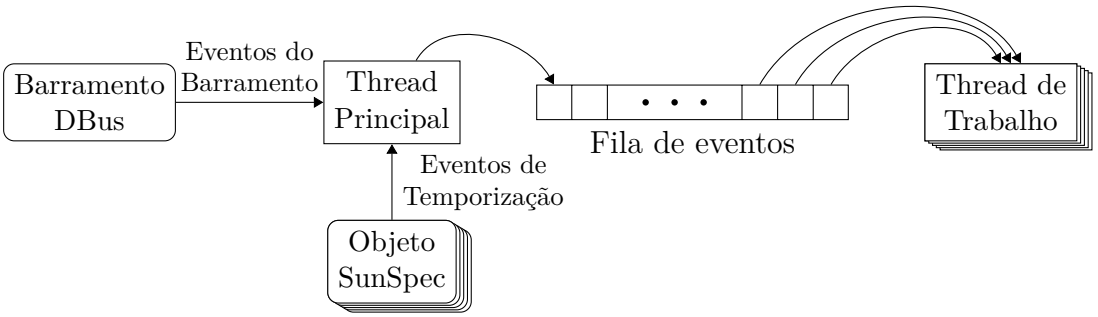


Figura 15 – Fila de eventos da `sunspec-daemon`.

objeto, criado com `timerfd_create`. Para eventos não temporizados, como o sucesso na conexão com o dispositivo ou fim do mapeamento dos modelos, uma chamada a `sigqueue` com um ponteiro do objeto como valor é utilizada para notificar o laço de eventos.

Os eventos detectados são inseridos em uma fila, que é consumida por um grupo de *threads* de trabalho que executam as ações associadas ao evento. Para eventos relacionados ao barramento DBus, a ação se limita a invocar `wdbus_loop`. Para eventos dos objetos, entretanto, essas ações envolvem chamadas potencialmente bloqueantes, como `suns_device_connect` ou `suns_device_scan`. O número de *threads* de trabalho e o tamanho da fila de eventos podem ser configurados em tempo de compilação. Atualmente o sistema utiliza quatro *thread*, igual ao número de núcleos máquina utilizada, e uma fila de tamanho definido empiricamente como 16 posições.

3.4 `DER-DAEMON`

A `der-daemon` é responsável pelas comunicações com a concessionária ou qualquer outra entidade que controle o grupo de DERs. Seguindo a arquitetura descrita em 2.3.1, o serviço implementa as comunicações entre o nível 2 e os níveis 3 e 4. Pela arquitetura recursiva, descrita em 2.3.2, o serviço desempenha o papel de VEN do FDEMS.

A *daemon* baseia-se na implementação de referência de EPRI (2019) da Nota de Aplicação 2018-001 (DNP Users Group, 2019), que utiliza a biblioteca OpenDNP3 (Automatak, 2019) para comunicações com o protocolo DNP3. A biblioteca WDBus é utilizada para interagir com o barramento DBus, no qual o serviço assume o nome `com.epri.DNP3` e expõem um único objeto no caminho `/com/epri/DNP3`.

Além das interfaces padrão providas pela WDBus, o objeto implementa a interface `org.dnp.Points`, em que os pontos do perfil descrito pela Nota de Aplicação são mapeados em dicionários de tuplas, e a interface `com.epri.CSIP`, composta apenas por sinais que indicam a ativação das funções descritas por EPRI (2016b). Uma visão geral do sistema é apresentada na Figura 16.

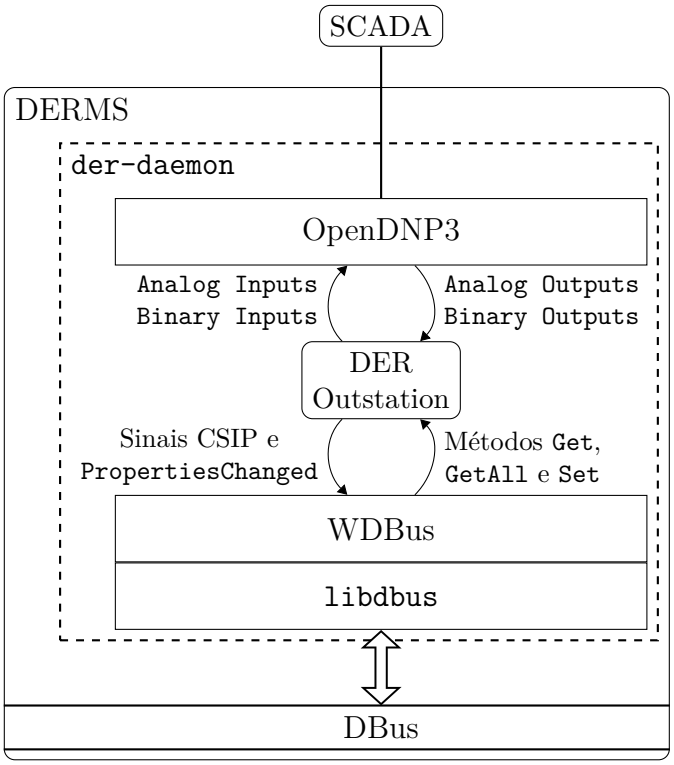


Figura 16 – Visão geral da `der-daemon`.

A seguir, a API da biblioteca OpenDNP3 é apresentada, em 3.4.1, e a implementação da Nota de Aplicação 2018-001 é descrita em 3.4.2.

3.4.1 Biblioteca OpenDNP3

A OpenDNP3 é uma biblioteca de código aberto que implementa o protocolo DNP3 com foco portabilidade, desempenho e segurança. Escrita em C++, a biblioteca não realiza cópias para analisar as mensagens recebidas, e possui uma API não bloqueante. A implementação atende completamente aos dois primeiros níveis de implementação do protocolo, e grande parte do terceiro.

Uma suíte de testes automatizados é aplicada a cada alteração submetida ao repositório do projeto, incluindo os testes de conformidade descritos por DNP Users Group (2020) e testes unitários que cobrem cerca de 80% do código-fonte. Além disso, técnicas de *fuzzing* são aplicadas em testes de integração a fim de encontrar falhas de segurança na implementação.

Nos tópicos a seguir, exemplos de implementação de uma *outstation* e um mestre DNP3 são utilizados para apresentar a API da biblioteca.

Exemplo de implementação de uma *Outstation*

Para utilizar a OpenDNP3, todas as aplicações precisam primeiramente instanciar um objeto da classe `DNP3Manager`. Entre outras coisas, esse objeto é responsável pelo conjunto de *threads* de trabalho que possibilitam a interface assíncrona da biblioteca. Seu construtor recebe o número de *threads* a serem criadas e um objeto do tipo `ILogHandler`, que é utilizado para escrever mensagens de *log*. Para os casos mais simples, a biblioteca oferece a classe `ConsoleLogger`, que escreve tais mensagens na saída padrão do processo.

```
7 DNP3Manager manager(std::thread::hardware_concurrency(),
8       ConsoleLogger::Create());
```

Em seguida, a aplicação deverá abrir um canal por meio de um dos métodos da classe `DNP3Manager`, como `AddSerial`, `AddTCPClient` e `AddTLSServer`. Esses métodos recebem como primeiro parâmetro uma *string* para identificar o canal em mensagens de *log*, seguido por um inteiro indicando o nível de *log* e outros parâmetros específicos a cada tipo de canal. Por exemplo, para a criação de um servidor TCP, o método `AddTCPServer` pode ser utilizado como:

```
10 auto channel = manager.AddTCPServer("OutstationServer",
11     flags::WARN|flags::ERR|flags::EVENT,
12     ServerAcceptMode::CloseExisting, "0.0.0.0", 20000,
13     PrintingChannelListener::Create());
```

Uma *outstation* é representada por um objeto de uma classe que herde de `IOutstation`. A criação desse objeto deve ser feita por meio do método `AddOutstation` da instância de `IChannel`. Além de uma *string* para nomear a *outstation* em mensagens de *log*, o método recebe como argumentos:

- Um objeto de uma classe, a ser definida pelo usuário, que herde de `ICommandHandler`. Os métodos `Select` e `Operate` desse objeto serão invocados pela OpenDNP3 para que a aplicação reaja aos comandos de `select`, `operate` e `direct operate`.
- Um objeto de uma classe que herde de `IOutstationApplication`, cujos métodos serão invocados para os demais comandos, como `cold restart` e `assign class`. Para os casos mais simples, a biblioteca oferece uma implementação padrão da classe, chamada `DefaultOutstationApplication`, que possui o método estático `Create` para criar instâncias da classe.
- Uma instância de `OutstationStackConfig` com as configurações da *outstation*, como o endereço da *outstation*, do mestre, número de pontos de cada tipo, grupo e variação de cada ponto, etc.

No código a seguir, é instanciado um `OutstationStackConfig` com endereços da *outstation* e do mestre iguais a 10 e 1, e habilita-se o uso de resposta não solicitadas.

Também é configurado um `Analog Input` e um `Analog Output`, com grupo e variação para ponto flutuante de 32-bits com qualidade, e um `Binary Input` e um `Binary Output`, também com qualidade.

```
15 OutstationStackConfig cfg(DatabaseSizes(1, 0, 1, 0, 0, 1, 1, 0, 0));
16
17 cfg.outstation.eventBufferConfig =
18     EventBufferConfig(10, 0, 10, 0, 0, 10, 10, 0);
19
20 cfg.outstation.params.allowUnsolicited = true;
21
22 cfg.link.LocalAddr = 10;
23 cfg.link.RemoteAddr = 1;
24
25 auto ai = &cfg.dbConfig.analog[0];
26 std::tie(ai->clazz, ai->svariation, ai->evariation) =
27   std::make_tuple(PointClass::Class1,
28       StaticAnalogVariation::Group30Var5,
29       EventAnalogVariation::Group32Var5);
30
31 auto bi = &cfg.dbConfig.binary[0];
32 std::tie(bi->clazz, bi->svariation, bi->evariation) =
33   std::make_tuple(PointClass::Class1,
34       StaticBinaryVariation::Group1Var2,
35       EventBinaryVariation::Group2Var2);
36
37 auto ao = &cfg.dbConfig.aoStatus[0];
38 std::tie(ao->clazz, ao->svariation, ao->evariation) =
39   std::make_tuple(PointClass::Class1,
40       StaticAnalogOutputStatusVariation::Group40Var3,
41       EventAnalogOutputStatusVariation::Group42Var5);
42
43 auto bo = &cfg.dbConfig.boStatus[0];
44 std::tie(bo->clazz, bo->svariation, bo->evariation) =
45   std::make_tuple(PointClass::Class1,
46       StaticBinaryOutputStatusVariation::Group10Var2,
47       EventBinaryOutputStatusVariation::Group11Var2);
```

Um exemplo de classe que herda de `ICommandHandler` e processa os comandos para esses pontos pode ser declarada como:

```
1 class ExampleCommandHandler : public ICommandHandler
2 {
3   bool bi, bo;
4   int ai, ao;
5   UpdateBuilder builder;
6   std::shared_ptr<IOutstation> outstation;
7
```

```
8   public:
9     void setOutstation(std::shared_ptr<IOutstation> &outstation) {
10      this->outstation = outstation;
11    }
12
13    void show();
14    void incAnalog();
15    void toggleBinary();
16
17    void Start() {};
18    void End() {};
19
20    CommandStatus Select(const ControlRelayOutputBlock& command,
21        uint16_t index);
22    CommandStatus Operate(const ControlRelayOutputBlock& command,
23        uint16_t index, OperateType opType);
24    // ...
25 };
```

Com as seguintes implementações dos métodos `Select` e `Operate` para `Binary Outputs` e `Analog Outputs`:

```
1  CommandStatus
2  Select(const ControlRelayOutputBlock& command, uint16_t index)
3  {
4    return index==0?CommandStatus::SUCCESS:CommandStatus::OUT_OF_RANGE;
5  };
6
7  CommandStatus
8  Operate(const ControlRelayOutputBlock& command, uint16_t index,
     ↪ OperateType opType)
9  {
10   if(index != 0)
11     return CommandStatus::OUT_OF_RANGE;
12
13   bo = command.functionCode == ControlCode::LATCH_ON;
14
15   builder.Update(BinaryOutputStatus(bo, 0), index);
16   outstation->Apply(builder.Build());
17
18   return CommandStatus::SUCCESS;
19 }
```

```
1  CommandStatus
2  Select(const AnalogOutputInt32& command, uint16_t index)
3  {
4    return index==0?CommandStatus::SUCCESS:CommandStatus::OUT_OF_RANGE;
5  };
```

```
6
7  CommandStatus
8  Operate(const AnalogOutputInt32& command, uint16_t index, OperateType
       ↪ opType)
9  {
10   if(index != 0)
11     return CommandStatus::OUT_OF_RANGE;
12
13   ao = (int)command.value;
14
15   builder.Update(AnalogOutputStatus(ao, 0), index);
16   outstation->Apply(builder.Build());
17
18   return CommandStatus::SUCCESS;
19 }
```

Para criar uma *outstation* com essa classe de exemplo e a implementação padrão de `IOutstationApplication`, o seguinte código pode ser utilizado.

```
49 commandHandler = std::make_shared<ExampleCommandHandler>();
50
51 outstation = channel->AddOutstation("outstation", commandHandler,
52     DefaultOutstationApplication::Create(), cfg);
53
54 commandHandler->setOutstation(outstation);
```

Para dar início a operação da *outstation*, o método `Enable` deve ser invocado. O código a seguir exemplifica o seu uso e cria um laço para ler comandos da entrada padrão para modificar o valor do `Analog Input` e `Binary Input` da *outstation*.

```
56 outstation->Enable();
57
58 cont = true;
59 while(cont) {
60   std::cout << "(a)nalog␣(b)inary␣(s)tate␣(q)uit" << std::endl <<">␣";
61   std::cin >> input;
62   switch(input[0]) {
63     case 'a':
64       commandHandler->incAnalog();
65       break;
66     case 'b':
67       commandHandler->toggleBinary();
68       break;
69     case 's':
70       commandHandler->show();
71       break;
72     case 'q':
73       cont = false;
```

```
74        break;
75      default:
76        break;
77    }
78 }
79
80 return 0;
```

Os métodos `incAnalog` e `toggleBinary` de `ExampleCommandHandler` são implementados como:

```
1 void incAnalog() {
2   ai++;
3   builder.Update(Analog(ai), 0);
4   outstation->Apply(builder.Build());
5 }
```

```
1 void toggleBinary() {
2   bi = !bi;
3   builder.Update(Binary(bi), 0);
4   outstation->Apply(builder.Build());
5 }
```

O método `Apply` de `IOutstation` é utilizado nesses métodos e nas implementações de `Select` e `Operate` para atualizar o valor dos pontos, gerando os eventos e respostas não solicitadas necessárias. Por fim, o método `show` é implementado como:

```
1 void show() {
2   std::cout << "Analog Input: " << ai << std::endl;
3   std::cout << "Analog Output: " << ao << std::endl;
4   std::cout << "Binary Input: " << bi << std::endl;
5   std::cout << "Binary Output: " << bo << std::endl;
6 }
```

Exemplo de Implementação de um Mestre

A implementação de um mestre DNP3 com a biblioteca também deve instanciar um objeto `DNP3Manager` e um canal. O código a seguir exemplifica o uso do método `AddTCPClient` para tal.

```
10 DNP3Manager manager(std::thread::hardware_concurrency(),
11     ConsoleLogger::Create());
12
13 auto channel = manager.AddTCPClient("MasterClient",
14     flags::WARN|flags::ERR|flags::EVENT, ChannelRetry::Default(),
15     "127.0.0.1", "0.0.0.0", 20000, PrintingChannelListener::Create());
```

Um mestre DNP3 é representado por uma instância de uma classe que herde de `IMaster`, que deve ser obtida pelo método `AddMaster` de `IChannel`. Semelhante ao método para criação da *outstation*, este método recebe uma *string* para identificar o mestre nos *logs* e mais três parâmetros:

- Um objeto de uma classe que herde de `ISOEHandler`, a ser definida pelo usuário da biblioteca. O método `Process` desse objeto será invocado para cada resposta, solicitada ou não, recebida.
- Um objeto de uma classe que herde de `IMasterApplication`. Os métodos desse objeto serão invocados para outros eventos na comunicação com a *outstation*, como a abertura ou fechamento da camada de aplicação.
- Uma instância de `MasterStackConfig` com as configurações do mestre, como seu endereço e o endereço da *outstation*.

Um exemplo de classe que herda de `ISOEHandler` e apenas escreve os valores recebidos na saída padrão seria:

```
1  class ExampleSOEHandler : public ISOEHandler
2  {
3    public:
4      void Start() {};
5      void End() {};
6
7      void Process(const HeaderInfo& info, const ICollection<Indexed<
     ↪ Binary>>& values) {
8        values.ForeachItem(
9          [] (const Indexed<Binary> &pair) {
10           std::cout << "Binary␣Input␣" << pair.index
11                     << ":␣" << (pair.value.value?"True":"False")
12                     << std::endl;
13         });
14     }
15
16     void Process(const HeaderInfo& info, const ICollection<Indexed<
     ↪ Analog>>& values) {
17       values.ForeachItem(
18         [] (const Indexed<Analog> &pair) {
19           std::cout << "Analog␣Input␣"
20                     << pair.index << ":␣" << pair.value.value
21                     << std::endl;
22         });
23     }
24
25     // ...
26 }
```

O trecho de código a seguir instancia um objeto da classe `MasterStackConfig` e configura o endereço local e da *outstation*. Em seguida, a instância da classe de exemplo `ExampleSOEHandler` é criada e o método `AddMaster` do canal é invocado para criar o mestre. Por fim, utiliza-se um laço para ler comandos da entrada padrão.

```
17 MasterStackConfig config;
18
19 config.link.LocalAddr = 1;
20 config.link.RemoteAddr = 10;
21
22 soeHandler = std::make_shared<ExampleSOEHandler>();
23
24 master = channel->AddMaster("master", soeHandler,
25     asiodnp3::DefaultMasterApplication::Create(), config);
26
27 master->Enable();
28
29 cont = true;
30 while(cont) {
31   std::cout
32     << "(a)nalog, (b)inary, (s)can, (t)oggle unsolicited, (q)uit"
33     << std::endl << "> ";
34   std::cin >> input;
35   switch(input[0]) {
36     case 'a':
37       master->DirectOperate(AnalogOutputInt32(++ao), 0, print_result);
38       break;
39     case 'b':
40       master->DirectOperate(
41           ControlRelayOutputBlock(
42             bo?ControlCode::LATCH_OFF:ControlCode::LATCH_ON),
43            0, print_result);
44       break;
45     case 's':
46       master->ScanRange(GroupVariationID(30, 5), 0, 1);
47       master->ScanRange(GroupVariationID(1, 2), 0, 1);
48       break;
49     case 't':
50       if(unsol) {
51         master->PerformFunction("Disable Unsolicited Requests",
52             FunctionCode::DISABLE_UNSOLICITED, {
53               Header::AllObjects(60, 2),
54               Header::AllObjects(60, 3),
55               Header::AllObjects(60, 4)
56             }
57           );
58       } else {
```

```
59          master->PerformFunction("Enable␣Unsolicited␣Requests",
60              FunctionCode::ENABLE_UNSOLICITED, {
61                Header::AllObjects(60, 2),
62                Header::AllObjects(60, 3),
63                Header::AllObjects(60, 4)
64              }
65            );
66        }
67        unsol = !unsol;
68        break;
69      case 'q':
70        cont = false;
71        break;
72      case '\n':
73      default:
74       break;
75    }
76 }
```

3.4.2 Implementação de Referência do EPRI

EPRI (2019) apresenta uma implementação de referência da Nota de Aplicação 2018-001 utilizando a biblioteca OpenDNP3. Originalmente, o sistema realiza *Inter Process Communication* (IPC) por meio de um arquivo no formato CSV para receber o valor dos pontos. Em um sistema real, entretanto, esse mecanismo de atualização deve ser implementado dentro da aplicação ou utilizando outras formas de IPC.

A aplicação define a quatro `structs` para descrever as configurações dos pontos do perfil, como nome, grupo, variação e vamos máximos e mínimos.

```
1  typedef struct {
2      u_int16_t   PointIndex;
3      std::string NameStr;
4      int32_t     MinValue;
5      int32_t     MaxValue;
6      PointClass  DefEvtClass;
7  } AnalogInputPointDefinition;
8
9  typedef struct {
10     u_int16_t   PointIndex;
11     std::string NameStr;
12     int32_t     MinValue;
13     int32_t     MaxValue;
14     bool        Supported;
15     u_int16_t   MappedInput;
16 } AnalogOutputPointDefinition;
```

```
17
18 typedef struct {
19     u_int16_t   PointIndex;
20     std::string NameStr;
21     PointClass  DefEvtClass;
22 } BinaryInputPointDefinition;
23
24 typedef struct {
25     u_int16_t   PointIndex;
26     std::string NameStr;
27     PointClass  DefEvtClass;
28     bool        Supported;
29     u_int16_t   MappedInput;
30 } BinaryOutputPointDefinition;
```

Um vetor de cada `struct` é instanciado e inicializado com os dados da Nota de Aplicação.

```
1  const AnalogOutputPointDefinition AOPoints[MAX_AOPOINTS] =
2  {
3      {0,"RefVolt",0,2147483647,true,29},
4      {1,"RefVoltOffset",-2147483648,2147483647,true,30},
5      // ...
6      {669,"SelectedSchedulePoint100Value",-2147483647,2147483647,true
       ↪ ,780},
7  };
8
9  const BinaryOutputPointDefinition BOPoints[MAX_BOPOINTS] =
10 {
11     {0,"System␣Set␣Lockout␣State",PointClass::Class2,false,65535},
12     {1,"System␣Initiate␣Start-up␣Sequence␣(Return␣to␣Service)",
       ↪ PointClass::Class2,false,65535},
13     // ...
14     {49,"Set␣Selected␣Schedule␣Repeat␣Weekly␣Saturday",PointClass::
       ↪ Class2,true,116}
15 };
16
17 const AnalogInputPointDefinition AIPoints[MAX_AIPOINTS] =
18 {
19     {0,"DER␣Profile␣Version␣Number",100,100,PointClass::Class3},
20     {1,"DER␣Profile␣Implementation␣Level",1,3,PointClass::Class3},
21     // ...
22     {1008,"Battery␣Bank␣#b␣Internal␣Voltage␣␣Low␣Threshold"
       ↪ ,0,2147483647,PointClass::Class2}
23 };
24
25 const BinaryInputPointDefinition BIPoints[MAX_BIPOINTS] =
26 {
```

```
27      {0,"System␣Communication␣Error",PointClass::Class1},
28      {1,"System␣Has␣Priority␣1␣Alarms",PointClass::Class1},
29      // ...
30      {328,"Battery␣Bank␣#b␣Electrical␣Warning",PointClass::Class1}
31  };
```

Com base nesses vetores a aplicação instância e configura o objeto da classe `OutstationStackConfig` a ser utilizado na chamada a `AddOutstation`.

A interface de `ICommandHandler` é implementada na classe `DERCommandHandler`, que possui uma série de atributos privados que armazenam as referências a serem seguidas pelo DER, como limites de potência ativa, tensão e frequência. Seus métodos `Select` e `Operate` utilizam os vetores que descrevem o perfil para validar os comandos recebidos, atualizando os atributos em caso positivo, e retornando o erro adequado para o mestre caso contrário.

O construtor de `DERCommandHandler` recebe como argumento um ponteiro para um objeto que herde da classe `IDERCommandHandlerCallback`, que define uma interface de métodos a serem chamados ao ativar e desativar as funções descritas por EPRI (2016b). `DERCommandHandlerCallbackDefault` é definida como um exemplo de classe que implementa essa interface, cujos métodos apenas imprimem na saída padrão uma mensagem informando qual função foi ativada ou desativada.

3.4.3 Integração com D-Bus

A biblioteca WDBus foi utilizada para transformar a implementação de referência em um serviço no barramento D-Bus, com o nome `com.epri.DNP3`. Um único objeto no caminho `/com/epri/DNP3` é exposto, com três interfaces.

A primeira, `org.dnp.Points`, apresenta os dicionários de tuplas `AnalogOutput`, `BinaryOutput`, `AnalogInput` e `BinaryInput`. As chaves desse dicionário representam o índice de cada ponto, enquanto os elementos da tupla armazenam o valor do ponto e sua qualidade. A fim de facilitar a manipulação de múltiplos pontos com a mesma qualidade, a interface apresenta os métodos `SetAnalogOutput`, `SetBinaryOutput`, `SetAnalogInput` e `SetBinaryInput`, que recebem como primeiro argumento a qualidade e como segundo argumento um dicionário indicando os pontos a serem atualizados.

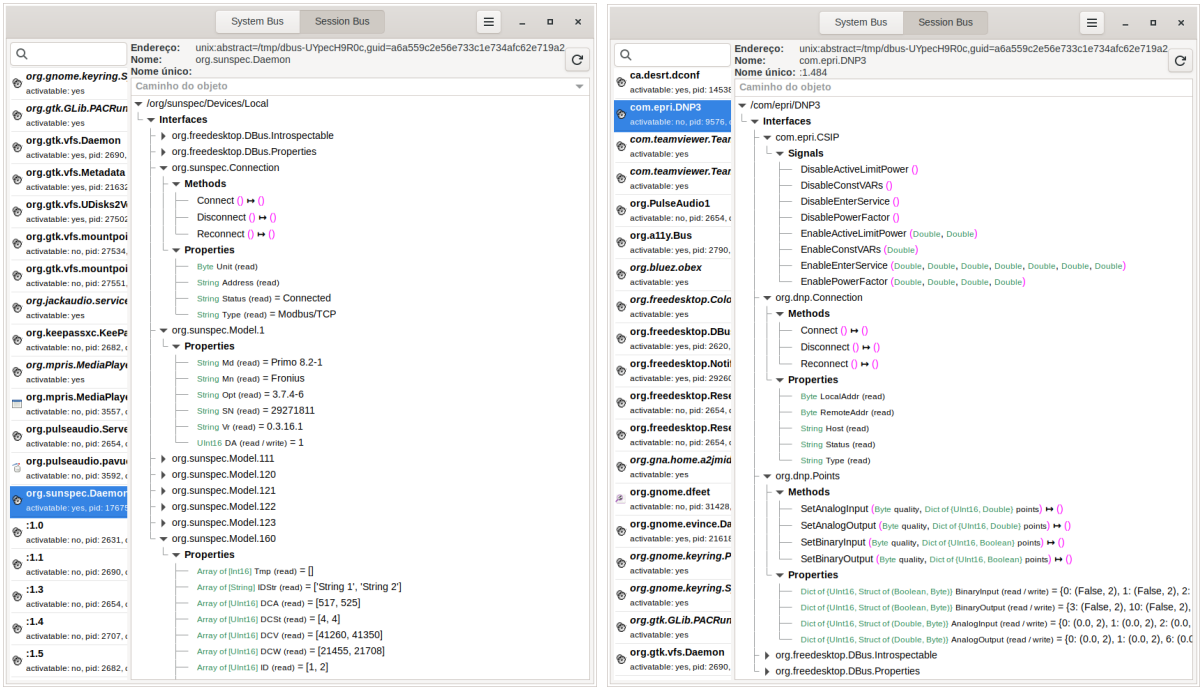
A segunda interface, `org.dnp.Connection`, apresenta as propriedades `Type`, que indica o tipo da conexão (DNP3 sobre TCP ou TLS), `Host` que indica o endereço IP do Master, `LocalAddr` que apresenta o endereço local da Outstation para o protocolo DNP3, `RemoteAddr`, que apresenta o endereço DNP3 do Master e `Status`, que indica o estado da conexão. A interface também apresenta os métodos `Connect`, `Disconnect` e `Reconnect`, para tentar uma nova conexão, desconectar-se e reconectar-se ao Mestre.

Por fim, a interface `com.epri.CSIP` é composta apenas por sinais, correspondentes aos *callbacks* de `IDERCommandHandlerCallback`, como `EnableEnterService` e `DisableEnterService`, que indicam a autorização para iniciar a operação dos DERs e a requisição para interrompê-la.

3.5 OPERAÇÃO DO SISTEMA

As Figuras 17 apresentam o barramento DBus com as duas *daemons* desenvolvidas em execução. Na Figura 17a a `sunspec-daemon` está conectada a um inversor comercial. O nome do objeto no barramento é "Local", e a conexão não utiliza *Transport Layer Security* (TLS). Cada modelo implementado pelo dispositivo é apresentado como uma interface do objeto, e as propriedades das interfaces representam os pontos de cada modelo.

Na Figura 17b a `der-daemon` está aguardando a conexão do mestre DNP3. O único objeto exposto no barramento é `/com/epri/DNP3`, que possui a interface `org.dnp.Points`. O valor dos pontos pode ser manipulado tanto por meio das propriedades quanto pelos métodos dessa interface, como `SetAnalogInput` e `SetBinaryOutput`.



(a) `sunspec-daemon` **(b)** `der-daemon`

Figura 17 – Captura de tela do programa d-feet exibindo as interfaces de objeto de cada *daemon* desenvolvida.

A Figura 18 apresenta um diagrama de sequência ilustrando a operação do sistema. Ao fim do intervalo de *polling*, a `sunspec-daemon` inicia a leitura dos pontos de cada modelo dos DERs conectados. A cada resposta recebida, um sinal `PropertiesChanged` é emitido com os Pontos que mudaram de valor. O gerenciador do sistema escuta por esse sinal e atualiza seu estado interno a cada nova mensagem.

Ao enviar o sinal do Modelo 111 (monitoramento de inversor monofásico), o gerenciador utiliza o valor do Ponto SunSpec `PhVphA` para atualizar o Ponto DNP3

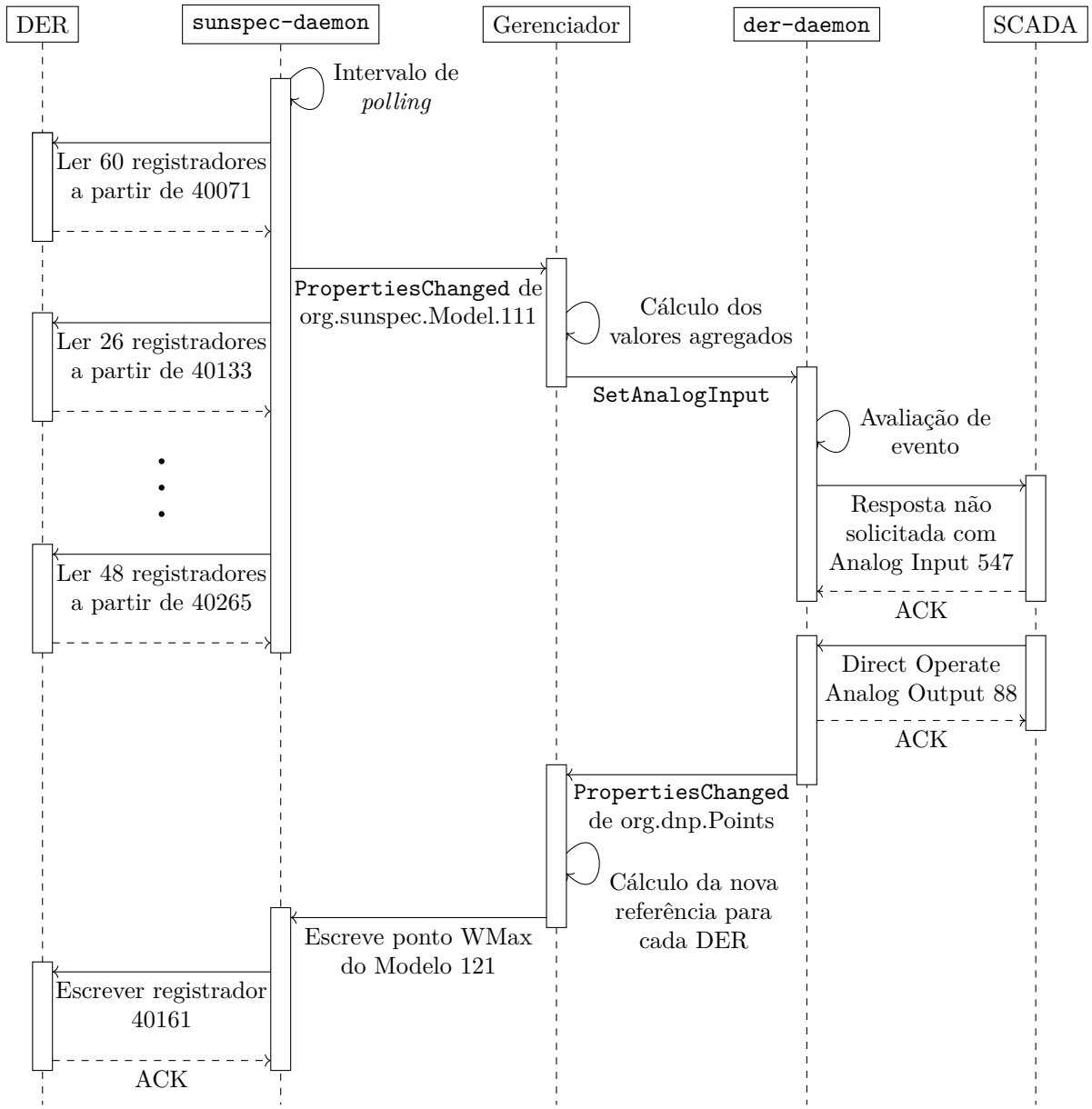


Figura 18 – Exemplo de operação completa do sistema.

`PhV.phsA.mag`, mapeado na Analog Input 547. A `der-daemon` identifica que o novo valor gerou um Evento DNP3, que é então transmitido para o sistema SCADA por meio de uma Resposta Não Solicitada. Em seguida, o operador da rede decide de alterar o limite de geração de potência ativa, escrevendo a nova referência na Analog Output 88. A `der-daemon` emite um sinal `PropertiesChanged` com o novo valor, que é utilizado pelo gerenciador para calcular a nova referência de cada DER conectado ao sistema.

O gerenciador então escreve o novo valor na propriedade `WMax` da interface `org.sunspec.Model.121` de um dos objetos da `sunspec-daemon`. Essa interface representa o modelo de controle de configurações básica de inversores, que neste inversor está mapeado a partir do registrador Modbus 40161. O primeiro Ponto desse modelo é `WMax`, que possui o tipo SunSpec `uint16`, e portanto ocupa um único registrador.

Finalmente, a `sunspec-daemon` envia uma requisição com o Código de Função 0x10, "escrever em múltiplos registradores", para atualizar o registrador Modbus do DER.

3.5.1 Testes

Dois testes são propostos para verificar o desempenho do sistema implementado e o atendimento dos requisitos levantados na Seção 2.2. O primeiro deles verificará o comportamento da `sunspec-daemon` quando utilizada para monitorar uma grande quantidade de DERs. O segundo verificará o tempo de resposta da `der-daemon` para reportar mudanças nos valores de pontos específicos.

Teste de escalabilidade da `sunspec-daemon`

O principal serviço de comunicação a ser testado para verificar a escalabilidade do sistema é a `sunspec-daemon`, uma vez que esta pode estar conectada a um grande número de DERs, enquanto a `der-daemon` estará conectada a um único mestre.

O teste proposto concentra-se no estado P da máquina de estados apresentada na Figura 14, e irá aumentar gradativamente o número de conexões até o limite permitido pela `libmodbus`[3]

O tempo será medido entre o disparo do evento p e o início da leitura dos modelos do dispositivo com o auxílio de filtros eBPF, utilizando uma `kprobe` no método `timerfd_triggered`, interno ao *kernel*, que é invocado quando o temporizador de um dispositivo expira, e uma `uprobe` no método `device_poll`, da `sunspec-daemon`, que representa o início da leitura dos modelos do DER.

A título de comparação, uma versão alternativa da `sunspec-daemon` foi implementada, onde uma *thread* é criada para cada conexão, eliminando a fila de eventos apresenta na Figura 15. Essa mudança reduz o número de *syscalls* necessárias para iniciar a leitura dos modelos, mas aumenta consideravelmente o uso de memória por conexão, uma vez que a pilha mínima de uma *thread* é de 16 kB[4]. A implementação alternativa será chamada de `sunspec-daemon-tpc` na apresentação dos resultados no Capítulo 4

[3] A biblioteca `libmodbus` utiliza internamente a *syscall* `select` no soquete TCP. A documentação desse método indica que o valor do *file descriptor* não pode ser maior que 1024. Levando em conta que outros descritores são abertos (`stdin`, `stdout`, `stderr`, *epoll*, `signalfd`, etc.), o número máximo de conexões Modbus simultâneas com a biblioteca é de aproximadamente 1000 dispositivos.

[4] Assumindo que a *stack* utilizada por cada conexão não ultrapasse 4 kB, é razoável assumir que apenas a primeira página será de fato mapeada para cada *thread*. Ainda assim, o uso de memória por conexão é 19,7 vezes maior do que a versão com fila de eventos, que utiliza apenas 208 bytes por dispositivo.

Teste de latência da `der-daemon`

Utilizando a OpenDNP3, um mestre DNP3 foi desenvolvido para medir o tempo de resposta do sistema. Ao estabelecer a conexão com a *outstation*, o mestre habilitará respostas não solicitadas e escreverá repetidamente na `BinaryOutput` 16, que está mapeada na `BinaryInput` 3 (DNP Users Group, 2019). A mudança no estado da `BinaryInput` gerara um evento que será transmitido como uma resposta não solicitada. O tempo entre a escrita e o recebimento do evento será medido para obter o tempo de resposta do sistema.

3.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a arquitetura do sistema desenvolvido e descreveu a implementação das duas *daemons* que implementam os serviços de comunicação do sistema. A primeira, `sunspec-daemon`, comunica-se com os DERs utilizando o protocolo SunSpec Modbus, enquanto a `der-daemon` utiliza o protocolo DNP3 para comunicar-se com a concessionária.

Para evitar a dependência de grandes bibliotecas, como GTK, ou de um sistema de inicialização, como o systemd, optou-se pela implementação de um *binding* DBus próprio, chamado WDBus. Por ter seu uso compartilhado por diferentes partes do sistema, WDBus foi implementado como uma biblioteca.

Por fim, foi apresentado um diagrama de sequência para descrever a operação completa do sistema, e foram apresentadas algumas imagens do programa d-feet, utilizado para inspecionar o barramento DBus durante a operação do sistema.

4 RESULTADOS

Os resultados dos testes propostos na Seção 3.5.1 são apresentados nas Seções 4.1 e 4.2. A Seção 4.3 apresenta as contribuições realizadas a projetos de código aberto durante o desenvolvimento deste trabalho.

4.1 ESCALABILIDADE DO SISTEMA

Para a execução do teste proposto na Seção 3.5.1.1, o sistema foi conectado a uma máquina comum[1] executando um escravo Modbus escutando a porta 1024. O perfil SunSpec Modbus desse escravo é semelhante a um inversor comercial, com um mapa de 315 registradores implementado os seguintes modelos:

- 1: Modelo comum implementado por todos os dispositivos compatíveis com SunSpec Modbus;
- 111: Modelo de monitoramento de variáveis de ponto flutuante de um inversor monofásico;
- 120: Capacidades nominais do DER;
- 122: Medidas e *status* adicionais de controle do inversor;
- 123: Controle imediato/direto do inversor;
- 160: Modelo de extensão para Rastreador de Ponto Máximo de Potência (MPPT, do inglês *Maximum Power Point Tracking*) de múltiplos inversores

O início do mapa Modbus, os cabeçalhos dos modelos e os pontos de *padding* não são lidos após o reconhecimento do perfil SunSpec Modbus, de forma que no processo de *polling* serão lidos apenas 297 registradores de 16-bits.

O teste foi repetido com 200, 400, 600, 800 e 1000 dispositivos, gerando os resultados apresentados na Figura 19. Observa-se que até 200 conexões ambas as implementações apresentam um comportamento muito semelhante. Com 400 conexões a variabilidade da latência na implementação alternativa é um pouco maior, mas ambas as soluções ainda se mantém abaixo do período de *polling*, e portanto não devem perder eventos.

Com 600 conexões, a implementação original apresenta uma latência média maior do que a `sunspec-daemon-tpc`, e também uma maior variação nessa latência. Nesse ponto a `sunspec-daemon` já pode perder eventos de *polling* caso o dispositivo demore a responder à leitura dos modelos.

[1] Um *laptop* com processador Intel® Core™ i7-8750H (6 núcleos com *clock* base de 2.20 GHz e *boots* de até 4.10 GHz) e 32 GBs de memória RAM.

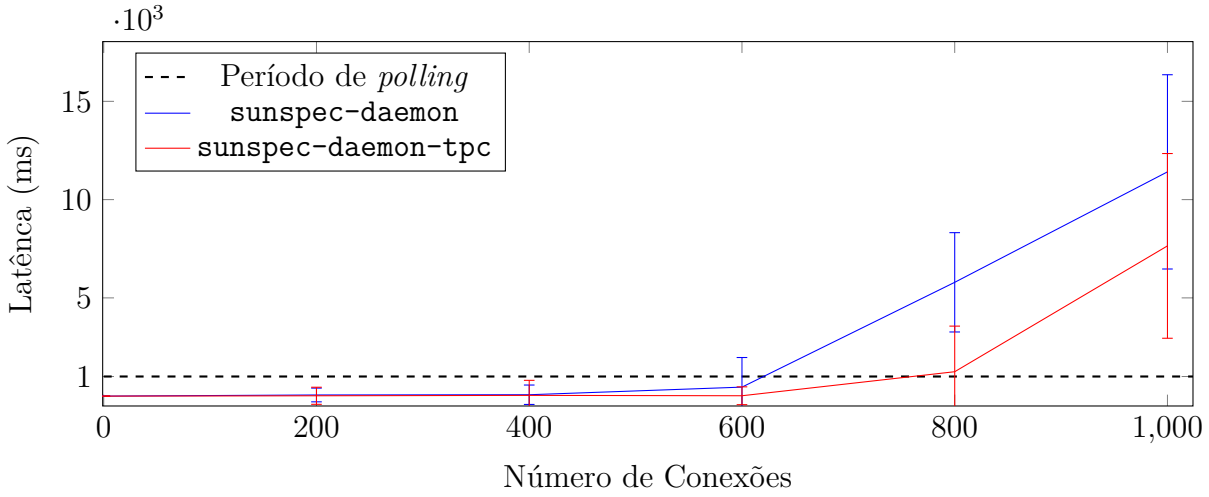


Figura 19 – Teste de escalabilidade da `sunspec-daemon`.

Com o auxílio do depurador GDB, pode-se verificar que esse número de conexões foi suficiente para encher a fila de eventos, causando os atrasos observados. Havendo memória disponível para o sistema de comunicação, é possível aliviar o problema aumentando o tamanho da fila ou criando mais *threads* de trabalho para atender as conexões.

Uma solução mais elaborada poderia combinar dinamicamente o redimensionamento da fila de eventos e a criação de novas *threads* de trabalho conforme a necessidade do sistema. Entretanto, para os fins desse trabalho, entende-se que o número de DERs em uma instalação não deve mudar frequentemente e, como uma solução mais simples, foi adicionado uma opção em tempo de compilação para redimensionamento da fila de eventos.

Com 800 conexões, ambas as implementações apresentam uma latência média acima do intervalo de *polling*, embora a `sunspec-daemon-tpc` ainda apresente uma maior estabilidade. Por fim, com 1000 conexões as duas *daemon* apresenta uma latência média muito maior que o período de *polling*, o que certamente inviabiliza a operação do sistema com esse número de dispositivos e intervalo de atualização.

4.2 TESTE DE LATÊNCIA

O mestre DNP3 descrito na Seção 3.5.1.2 foi executado em uma máquina comum conectada ao sistema, gerando os resultados da Figura 20. Embora a latência média obtida de 30,45 ms seja ordens de magnitude abaixo do requisito de 30 s levantando na Seção 2.2, podemos notar uma grande variabilidade nas medidas, resultando em um desvio padrão de $\pm 19,12$ ms. A Figura 21 apresenta o histograma dos mesmos dados, onde observa-se que as medidas concentram-se nas classes abaixo 10 ms ou acima 45 ms, sem nenhum amostra com valores intermediários.

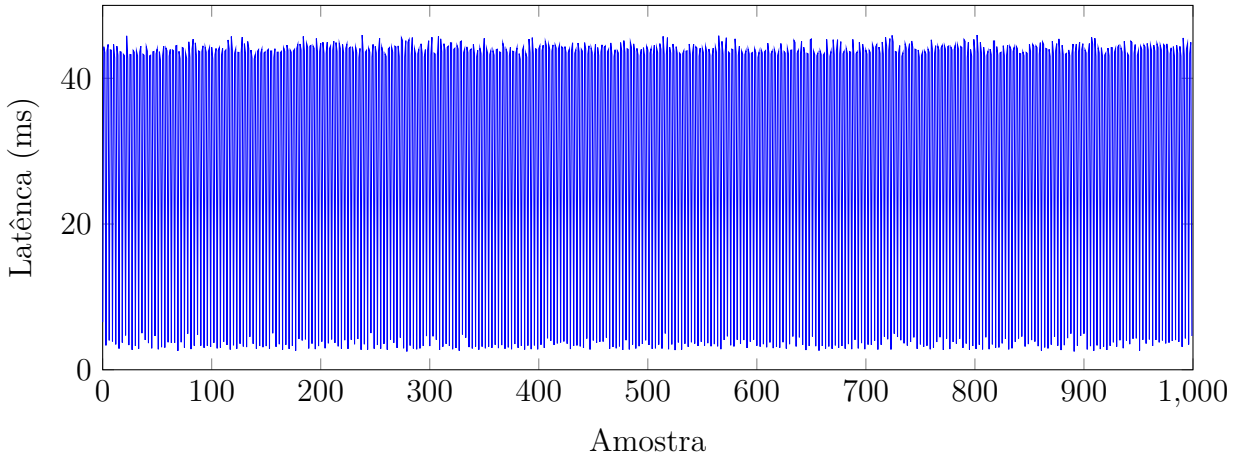


Figura 20 – Teste de latência da `der-daemon`.

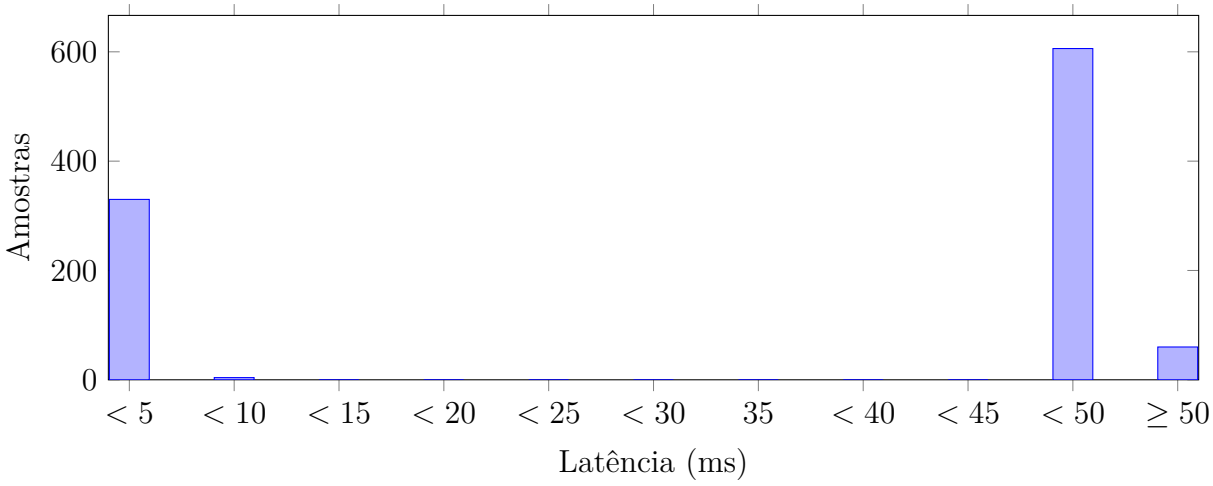


Figura 21 – Histograma de latência da `der-daemon`.

Um atraso de 40 ms é um valor notório em comunicações TCP como resultado da interação entre duas otimizações do protocolo: *Delayed ACK* (BRADEN, 1989) e o algoritmo de Nagle (NAGLE, 1984). A primeira otimização consiste em aguardar até 40 ms[2] para enviar um ACK a fim de que múltiplos segmentos possam ser reconhecidos em um único pacote. Já o algoritmo de Nagle consiste em aguardar até 200 ms antes de enviar os dados de um *buffer* TCP menor que o Tamanho Máximo do Segmento (MSS, do inglês *Maximum Segment Size*), a não ser que todos os segmentos enviados anteriormente tenham sido reconhecidos.

Dado que os pacotes DNP3 sendo trocados entre mestre e *outstation* são muito menores que o MSS de uma rede Ethernet, a maioria dos ACKs serão retidos por 40 ms antes de serem enviados, atrasando o envio da resposta não solicitada que reporta o novo valor da `BinaryInput`. Modificando o código da biblioteca OpenDNP3 para desabilitar

[2]Na *stack* TCP/IP do Linux, esse valor vem da macro `TCP_DELACK_MIN`, que é definida como `HZ/25`. Conforme a documentação da configuração `HZ` (*Timer frequency*), o valor dessa macro é geralmente 1000Hz, exceto para alguns casos em servidores onde o valor 100Hz pode ser mais benéfico

o algoritmo de Nagle por meio da *syscall* `setsockopt` com a opção `TCP_NODELAY`, foram obtidos os dados apresentados na Figura 22, em que a latência média observada foi de $1,75 \pm 0,39$ ms.

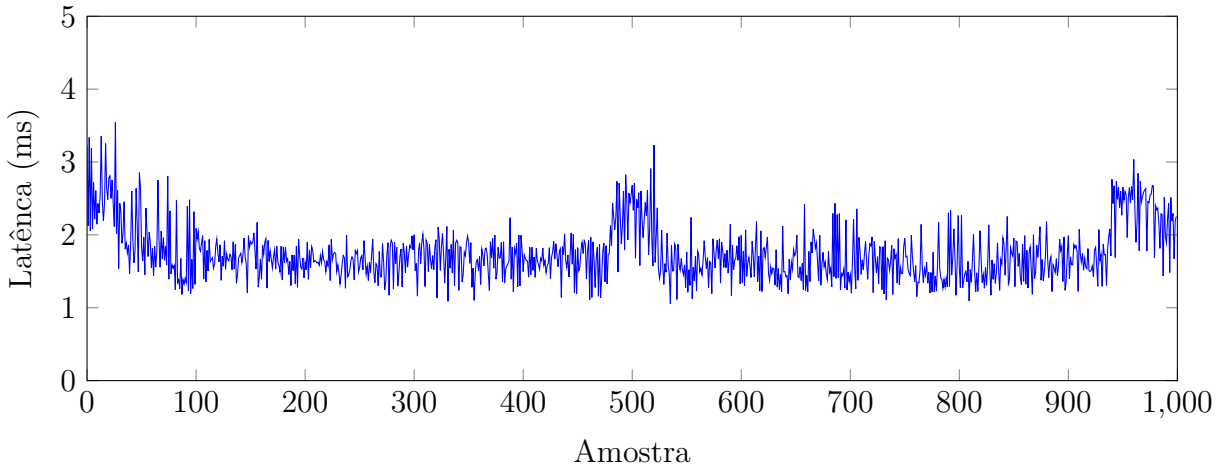


Figura 22 – Teste de latência da `der-daemon` com `TCP_NODELAY`.

Repetindo os testes com DNP3/TLS, obtiveram-se os dados apresentados na Figura 23, onde a latência média foi de $1,98 \pm 0,39$ ms. Embora a média tenha aumentado, possivelmente devido ao tamanho maior do pacote TLS e os custos adicionais para criptografia, o desvio padrão se manteve estável. Em ambos os casos a latência do sistema ainda se mostra ordens de magnitude menor do que o requisito levantado inicialmente na Seção 2.2.

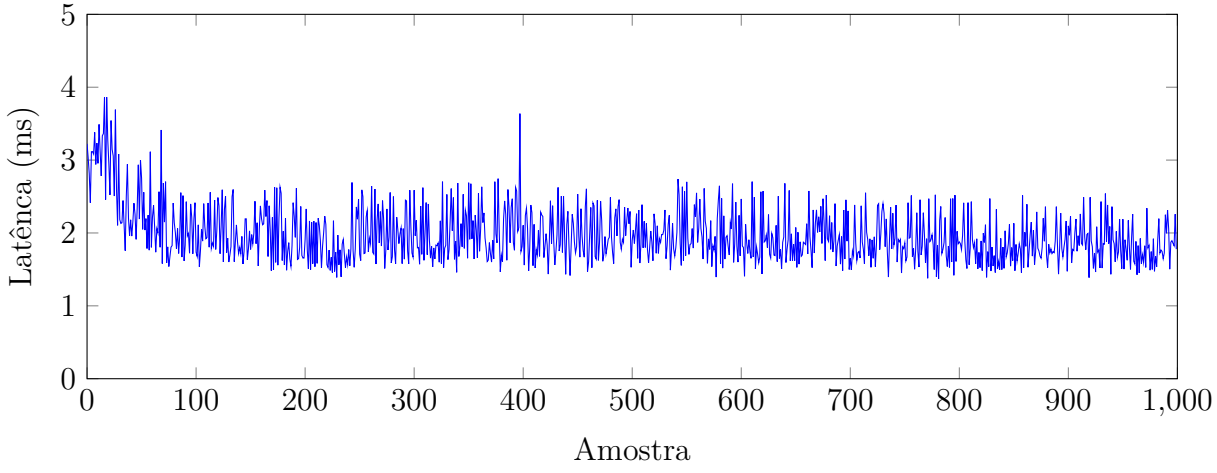


Figura 23 – Teste de latência da `der-daemon` com TLS.

4.3 CONTRIBUIÇÕES A PROJETOS DO CÓDIGO ABERTO

As seguintes contribuições foram submetidas a projetos de código aberto utilizados no desenvolvimento deste trabalho:

- Suporte a Modbus/TLS na `libmodbus`: Repositório com o código desenvolvido está disponível publicamente e uma *pull request* foi aberta no repositório oficial do projeto;
- Utilização da `libmodbus` na `libsunspec`: Repositório com o código desenvolvido está disponível publicamente;
- Suporte a decodificação de Modbus/TLS e DNP3/TLS no Wireshark: Os *commits* `8037be0ef6e5` e `da80daaf831b` foram propostos para possibilitar a decodificação dos protocolos Modbus e DNP3 sobre TLS. As modificações foram revisadas e aceitas pela comunidade e estão disponíveis a partir da versão 3.5.0;
- WDBus: o *binding* DBus desenvolvido está disponível publicamente com um licença permissiva.

5 CONCLUSÃO

O presente trabalho implementou os sistemas de comunicação de DERMS utilizando os protocolos DNP3 e SunSpec Modbus para comunicações com o DMS e DERs, respectivamente. A implementação buscou atender da melhor forma possível as normas existentes, apesar de algumas dessas normas ainda estarem em desenvolvimento. Requisitos descritos pela IEEE 1547 para outros IEDs foram adaptados a DERMSs para possibilitar a avaliação do sistema.

O sistema implementado utilizou uma arquitetura orientada a serviços, em que *daemons* que implementam os protocolos se conectaram a um barramento de serviços, o D-Bus, para possibilitar que os demais serviços do sistema se comuniquem com o DMS e DER e reajam a eventos gerados por esses.

Dois testes foram propostos para avaliar a implementação. O primeiro teste verificou a escalabilidade do sistema, conectando uma grande quantidade de DERs a `sunspec-daemon`, serviço responsável pelas comunicações utilizando o protocolo SunSpec Modbus. Os resultados obtidos demonstram que o sistema é capaz de monitor mais de quatro centenas de conexões sem atrasos significativos no período de *polling*. Apesar de satisfatório, observou-se que o sistema pode ser aprimorado para gerenciar um número ainda maior de dispositivos, caso surja tal necessidade.

O segundo teste, para verificar o tempo de resposta do serviço responsável pelas comunicações DNP3, a `der-daemon`, apresentou uma latência muito inferior ao requisito atual das normas, que é de 30 s. Mesmo com a utilização de criptografia e do protocolo TLS, a latência média do sistema ficou abaixo de 2 ms.

REFERÊNCIAS

ABNT. **Sistemas fotovoltaicos (FV) Características da interface de conexão com a rede elétrica de distribuição**. 2013.

_____. **Sistemas fotovoltaicos conectados à rede Requisitos mínimos para documentação, ensaios de comissionamento, inspeção e avaliação de desempenho**. 2014.

_____. **Redes e sistemas de comunicação para automação de sistemas de potência. Parte 10: Ensaios de conformidade**. 2018.

AMULYA et al. Experimenting with iec 61850 and goose messaging. In: **2017 4th International Conference on Power, Control Embedded Systems (ICPCES)**. [S.l.: s.n.], 2017. p. 1–6.

ANEEL. **Resolução Normativa N 482**. 2012. Disponível em: <http://www2.aneel.gov.br/cedoc/ren2012482.pdf>.

_____. **Resolução Normativa N 687**. 2015. Disponível em: <http://www2.aneel.gov.br/cedoc/ren2015687.pdf>.

_____. **Procedimentos de Distribuição de Energia Elétrica no Sistema Nacional - PRODIST**. 2018. Revisão 10. Disponível em: <http://www.aneel.gov.br/prodist>.

Automatak. **OpenDNP3**. 2019. Disponível em: <https://dnp3.github.io/>.

Aziz, O. et al. Research trends in enterprise service bus (epapazoglou2007servicosb) applications: A systematic mapping study. **IEEE Access**, v. 8, p. 31180–31197, 2020.

BANGA, G. et al. A scalable and explicit event delivery mechanism for unix. In: **USENIX Annual Technical Conference, General Track**. [S.l.: s.n.], 1999. p. 253–265.

BASSO, T. S.; DEBLASIO, R. Ieee 1547 series of standards: interconnection issues. **IEEE Transactions on Power Electronics**, IEEE, v. 19, n. 5, p. 1159–1162, 2004.

Bezerra, B. et al. Expansion pressure: Energy challenges in brazil and chile. **IEEE Power and Energy Magazine**, v. 10, n. 3, p. 48–58, May 2012. ISSN 1540-7977.

Blaabjerg, F. et al. Distributed power-generation systems and protection. **Proceedings of the IEEE**, v. 105, n. 7, p. 1311–1331, July 2017. ISSN 0018-9219.

BRADEN, R. **Requirements for Internet Hosts - Communication Layers**. [S.l.], 1989. <http://www.rfc-editor.org/rfc/rfc1122.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc1122.txt>.

California Public Utilities Commission. **Order Instituting Rulemaking to Consider Streamlining Interconnection of Distributed Energy Resources and Improvements to Rule 21**. 2017. Disponível em: <http://docs.cpuc.ca.gov/PublishedDocs/Published/G000/M192/K079/192079467.docx>.

CLEVELAND, F. et al. **Distributed Energy Resources (DER): Hierarchical Classification of Use Cases and the Process for Developing Information Exchange Requirements and Object Models**. [S.l.], 2014.

Cppcheck Developers. **Cppcheck**. 2020. Disponível em: <http://cppcheck.sourceforge.net/>.

Demirok, E. et al. Local reactive power control methods for overvoltage prevention of distributed solar inverters in low-voltage grids. **IEEE Journal of Photovoltaics**, v. 1, n. 2, p. 174–182, Oct 2011. ISSN 2156-3381.

DNP Users Group. **DNP3 Profile for Communications with Distributed Energy Resources (DER)**. [S.l.], 2019.

_____. **DNP3 Intelligent Electronic Device (IED) Certification Procedure**. [S.l.], 2020.

Eid, B. M. et al. Control methods and objectives for electronically coupled distributed energy resources in microgrids: A review. **IEEE Systems Journal**, v. 10, n. 2, p. 446–458, June 2016. ISSN 1932-8184.

EPRI. Enterprise integration functions for distributed energy resources: Phase 1. **Power Delivery & Utilization**, Palo Alto, CA, n. 3002001249, Dec 2013.

_____. Introduction to the ansi/cea-2045 standard. **Power Delivery & Utilization**, Palo Alto, CA, n. 3002004020, Jun 2014.

_____. Grid interactive microgrid controllers and the management of aggregated distributed energy resources (DER): Relationship of microgrid controller with distributed energy resource management system (DERMS) and utility distributed management system (DMS). **Power Delivery & Utilization**, Palo Alto, CA, n. 3002007067, Nov 2015.

_____. Grid interactive microgrid controllers and the management of aggregated distributed energy resources (der): Relationship of microgrid controller with distributed energy resource management system (derms) and utility distributed management system (dms). **Power Delivery & Utilization**, Palo Alto, CA, n. 3002007067, Nov 2015.

_____. Common functions for der group management: Third edition. **Power Delivery & Utilization**, Palo Alto, CA, n. 3002008215, Nov 2016.

_____. Common functions for smart inverters: 4th edition. **Power Delivery & Utilization**, Palo Alto, CA, n. 3002008217, Dec 2016.

_____. Open source der outstation for dnp application note an2018-001: Reference implementation of dnp application note an2018-001 "dnp3 profile for communications with distributed energy resources". **Power Delivery & Utilization**, Palo Alto, CA, n. 3002015355, Feb 2019.

Feng, J. et al. Evaluating demand response impacts on capacity credit of renewable distributed generation in smart distribution systems. **IEEE Access**, v. 6, p. 14307–14317, 2018. ISSN 2169-3536.

FERNANDEZ, E. B.; YOSHIOKA, N. Two patterns for distributed systems: enterprise service bus (esb) and distributed publish/subscribe. In: **Proceedings of the 18th Conference on Pattern Languages of Programs**. [S.l.: s.n.], 2011. p. 1–10.

Ferst, M. K. **Implementação de Comunicação Segura com Modbus e TLS**. 136 p. Monografia (Trabalho de Conclusão de Curso) — Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Pato Branco, 2018.

FREDERICK, G. S. Open communication standards for energy storage and distributed energy resources. **Current Sustainable/Renewable Energy Reports**, Springer, p. 1–6, 2017.

FREEDESKTOP. **D-Bus**. 2020. Disponível em: <https://www.freedesktop.org/wiki/Software/dbus/>.

_____. **D-Bus API Design Guidelines**. 2020. Disponível em: <https://dbus.freedesktop.org/doc/dbus-api-design.html>.

_____. **D-Bus Documentation**. 2020. Disponível em: <https://dbus.freedesktop.org/doc/api/html/index.html>.

GNOME Foundation. **GLib**. 2020. Disponível em: <https://gitlab.gnome.org/GNOME/glib>.

GREER, C. et al. **Nist framework and roadmap for smart grid interoperability standards, release 3.0**. [S.l.], 2014.

GUNGOR, V. C. et al. Smart grid technologies: Communication technologies and standards. **IEEE Transactions on Industrial Informatics**, v. 7, n. 4, p. 529–539, Nov 2011. ISSN 1551-3203.

_____. Smart grid and smart homes: Key players and pilot projects. **IEEE Industrial Electronics Magazine**, v. 6, n. 4, p. 18–34, Dec 2012. ISSN 1932-4529.

Hatziargyriou, N. et al. Microgrids. **IEEE Power and Energy Magazine**, v. 5, n. 4, p. 78–94, July 2007. ISSN 1540-7977.

HEFNER, A. et al. **IEC 61850 Information Model Concepts and Updates for Distributed Energy Resources (DER) Use Cases and Functions**. [S.l.], 2015.

IEA-PVPS, P. PVPS report snapshot of global pv 1992-2013. **Report IEA-PVPS T1-24**, 2014.

IEC. **61850-1: Communication networks and systems Part 1: Introduction and overview**. [S.l.], 2013.

IEC. **IEC TC57 - WG10/17/18**. 2019. Acesso em: 17 nov. 2019. Disponível em: <http://www.tc57wg10.info/>.

IEEE. Ieee guide for the interoperability of energy storage systems integrated with the electric power infrastructure. **IEEE Std 2030.2-2015**, p. 1–138, June 2015.

IEEE Standard Conformance Test Procedures for Equipment Interconnecting Distributed Energy Resources with Electric Power Systems and Associated Interfaces. **IEEE Std 1547.1-2020**, p. 1–282, 2020.

IEEE Standard for Electric Power Systems Communications-Distributed Network Protocol (DNP3). **IEEE Std 1815-2012 (Revision of IEEE Std 1815-2010)**, p. 1–821, Oct 2012.

IEEE Standard for Exchanging Information Between Networks Implementing IEC 61850 and IEEE Std 1815(TM) (DNP3). **IEEE Std 1815.1-2015 (Incorporates IEEE Std 1815.1-2015/Cor 1-2016)**, p. 1–358, Dec 2016.

IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2019 (Revision of IEEE 754-2008)**, p. 1–84, 2019.

IEEE Standard for Interconnection and Interoperability of Distributed Energy Resources with Associated Electric Power Systems Interfaces. **IEEE Std 1547-2018**, p. 1–138, April 2018.

IEEE Standard for Smart Energy Profile Application Protocol. **IEEE Std 2030.5-2018**, 2018.

ISO, C. Programming languagesc. **American National Standards Institute, ISO/IEC**, v. 9899, 1999.

LEMON, J. Kqueue-a generic and scalable event notification facility. In: **USENIX Annual Technical Conference, FREENIX Track**. [S.l.: s.n.], 2001. p. 141–153.

LOVE, R. **Linux system programming: talking directly to the kernel and C library**. [S.l.]: "O'Reilly Media, Inc.", 2013.

MACKENZIE, C. M. et al. Reference model for service oriented architecture 1.0. **OASIS standard**, v. 12, n. S 18, 2006.

MARK, E. A. **A Planning and Implementation Guide for Business and Technology**. [S.l.]: Wiley Online Library, 2006.

MARKS, E. A.; BELL, M. **Service-oriented architecture: a planning and implementation guide for business and technology**. [S.l.]: John Wiley & Sons, 2008.

MESA. **Open Standards for Energy Storage**. [S.l.], 2016.

Modbus FAQ. About the modbus organization. **FAQ. Modbus Organization inc.**, 2017.

_____. About the protocol. **FAQ. Modbus Organization inc.**, 2017.

Modbus Org. **MODBUS over serial line specification & implementation guide V1.02**.

Modbus Organization. **Modbus/TCP Security Protocol Specification**. [S.l.], 2018. 27 p.

NAGLE, J. **Congestion Control in IP/TCP Internetworks**. [S.l.], 1984.

NIKNEJAD, N. et al. Understanding service-oriented architecture (SOA): A systematic literature review and directions for further investigation. **Information Systems**, Elsevier BV, v. 91, p. 101491, jul. 2020. Disponível em: <https://doi.org/10.1016/j.is.2020.101491>.

NIST. **Smart Grid Interoperability Panel**. 2019. Acesso em: 20 nov. 2019. Disponível em: <https://www.nist.gov/programs-projects/smart-grid-national-coordination/smart-grid-interoperability-panel-sgip>.

OPENADR. **Open ADR Alliance - Overview**. 2017. Acesso em: 18 out. 2017. Disponível em: <http://www.openadr.org/about-us>.

PAPAZOGLOU, M. P.; HEUVEL, W.-J. V. D. Service oriented architectures: approaches, technologies and research issues. **The VLDB journal**, Springer, v. 16, n. 3, p. 389–415, 2007.

PENNINGTON, H. et al. **D-Bus Specification**. 2020. Disponível em: <https://dbus.freedesktop.org/doc/dbus-specification.html>.

POSIX. Standard for information technology–portable operating system interface (POSIX(R)) base specifications, issue 7. **IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)**, p. 1–3957, 2016.

RENJIT, A. et al. Derms reference control methods for der group management. In: . [S.l.]: AIM, 2019.

Santos, L. N. et al. A distributed generation manager with support for distributed network operator commands. In: **2020 IEEE International Conference on Industrial Technology (ICIT)**. [S.l.: s.n.], 2020. p. 810–815.

Schneider Eletric. Modicon is now schneider electric. **Our Brand History**, 2019.

SCHWARZ, D.-I. K.; EICHBAEUMLE, I. Iso 9506 (mms).

SEAL, B. **SunShot Initiative**. [S.l.], 2011.

SEPA. **DERMS Requirements**. [S.l.], 2019.

SGIP. **Basic Application Profile for Distribution Feeder Measurement based on the IEC 61850 Standard**. [S.l.], 2015.

SMA. **SunSpec Modbus Interface for SUNNY BOY / SUNNY TRIPOWER**. [S.l.], 2015.

Smart Inverter Working Group. Recommendations for utility communications with distributed energy resources (DER) systems with smart inverters. **California Public Utilities Commission**, Feb 2015.

SunSpec Alliance. **SunSpec Information Model Specifications**. [S.l.], 2015.

_____. Sunspec alliance: Information standards for distributed energy: Corporate backgrounder. **SunSpec Alliance**, 2016.

_____. **SunSpec Information Model Reference**. [S.l.], 2017. V1.5.

SYSTEMD. **sd-bus**. 2020. Disponível em: <https://www.freedesktop.org/software/systemd/man/sd-bus.html>.

_____. **The systemd System and Service Manager**. 2020. Disponível em: <https://systemd.io/>.

The Enlightenment Project. **Eldbus**. 2020. Disponível em: <https://docs.enlightenment.org/efl/current/eldbus_main.html>.

_____. **Enlightenment Foundation Libraries**. 2020. Disponível em: <http://www.enlightenment.org/about-efl>.

Valgrind Developers. **Valgrind**. 2020. Disponível em: <https://www.valgrind.org/>.