# Desenvolvimento de um middleware IoT distribuído com foco em smart grids

Márcio da Silva Maciel, Alan Petrônio Pinheiro, Daniel de Oliveira Ferreira e Alailton José Alves Júnior

***Resumo*— Os sistemas elétricos de potência vem passado por um processo de modernização conhecido como *smart grids*. Ele está associado à digitalização da rede elétrica e monitoramento em larga escala. E a tecnologia de internet das coisas promete ajudar neste processo. Neste artigo é proposto um *middleware* distribuído para as redes elétricas com foco em tecnologias de comunicação heterogêneas e monitoramento em tempo real usando recursos de *Complex Event Processing* (CEP). A solução busca prover escalabilidade, redundância e padronização das mensagens para ajudar a prover interoperabilidade. Os resultados apresentados por meio de modelagem e simulação nos cenários avaliados demostram a viabilidade desta tecnologia para monitoramento em larga escala.**

***Palavras-Chave*— Middleware, Redes Elétricas, Internet das coisas, smart grids.**

***Abstract*— Electricial grids systems come underwent a process of modernization. It is associated with the digitization of the electrical grid and large-scale monitoring. The internet of things technology promises to help in this process. This article proposes a distributed middleware for electrical grids focusing on heterogeneous and real-time devices (name IEDs) using Complex Event Processing (CEP). The solution provide scalability, redundancy and standardization to provide interoperability. The results demonstrate that the solution is feasibility for large-scale monitoring.**

***Keywords*— Middleware, electrical grids, internet of the things, smart grids.**

## I. Introdução

Os sistemas de distribuição de eletricidade são cada vez mais complexos e vitais para a sociedade. Na grande maioria, a automação destes empregam sistemas supervisórios que centralizam o comando e são limitados quanto a quantidade de conexões. Em contrapartida, a quantidade de dados a ser monitorada é cada vez mais maior com o surgimento das *Smart Grid* (SG) [1] e a necessidade de sistemas que possam acompanhar este volume de dados em tempo real.

Corral-Plaza [2] propõem uma arquitetura para processar uma grande quantidade de dados em tempo real. Essa arquitetura foi avaliada em um cenário real que gerencia um sistema de distribuição de água. Entretanto não tratou a interoperabilidade com outros aplicativos. Akanbi [3] propôs um *middleware* de processamento de eventos e análise em tempo real distribuídos com dados heterogêneos para monitoramento ambiental. No entanto, esses dados podem ser consumidos e produzidos com agentes tradicionais. Shapsough [4] apresenta uma arquitetura que utiliza hardware, software e tecnologias de comunicação baseado em *Internet of Things* (IoT) para o monitoramento e gerenciamento em tempo real de um sistema de placas fotovoltaicas. A proposta utiliza *Message Queueing Telemetry Transport* (MQTT) para facilitar a comunicação em larga escala com baixo consumo de recursos. Porém, fica restrito somente a um tipo de protocolo, provendo pouca flexibilidade. Uma integração de tecnologias de IoT para *Smart Grid* (SG) foi proposta por Cavalieri [5]. Nela, ficou definida uma plataforma web expondo uma interface RESTful para servidores executando protocolos da IEC 61850 e permitindo um mapeamento das informações em mensagens MQTT. Apesar de definir mecanismos para introduzir a troca interoperável de informações, não foi apresentado recursos que permitam a escalabilidade, redundância e tolerância a falhas.

Neste contexto, o objetivo deste trabalho é propor uma arquitetura de *middleware* voltada a algumas características do setor elétrico, com foco em tecnologias de comunicação heterogêneas e com capacidade computacional de prover escalabilidade, redundância, uniformização (das mensagens), com recursos de *Complex Event Processing* (CEP) em tempo real e suporte à tecnologias de virtualização (*Digital Twin*) prevista em IoT.

## II. Metodologia e Arquitetura

A arquitetura do *middleware* aqui proposta foi desenvolvida com base tecnológica da plataforma denominada Apache Kafka. Ela é uma plataforma de código aberto distribuído, particionado e replicado baseado em um sistema de mensagens *publish-subscribe*, capaz de armazenar os dados com segurança, tolerância à falhas e escalabilidade [6]. Dentre os vários fatores citados, a vazão do sistema teve um diferencial na escolha da plataforma. Segundo Guo Fu et. al, a plataforma Kafka teve o melhor desempenho na vazão do sistema em comparação aos outros como RabbitMQ, RocketMQ, ActiveMQ e Pulsar e tem um papel importante na coleta e a análise de dados em larga escala [7]. As mensagens são categorizadas em tópicos. Estes são divididos em partições. As mensagens são gravadas nos tópicos e distribuídas entre as partições, provendo redundância e escalabilidade. Cada partição pode ser armazenada em um servidor diferente, de forma que um tópico pode ser escalonado horizontalmente entre múltiplos servidores além de fornecer um aumento de desempenho no processamento das mensagens[8].

Para conseguir atingir as funcionalidades desejadas, criou-se uma arquitetura de "agentes de middleware". Um agente é um sistema de computador capaz de realizar ações autônomas em um ambiente para atingir os objetivos que foram projetados [9]. Estes agentes são aplicações concorrentes que são executadas independentemente no middleware a cada vez

Márcio da Silva Maciel, e-mail: marciosmaciel@gmail.com; Alan Petrônio Pinheiro, e-mail: alanpetronio@ufu.br; Daniel de Oliveira Ferreira, e-mail: danieldeoliv.f@gmail.com; Alailton José Alves: alailton.junior@ufu.br; Universidade Federal de Uberlândia, Uberlândia-MG

que se chega uma mensagem a ele. Os principais agentes são designados de: (i) Serializador, (ii) Busca dinâmica, (iii) Analisador de Eventos e (iv) os *Enablers*. As aplicações desses módulos foram desenvolvidas utilizando o *framework* Apache Spark. O Apache Spark é um mecanismo de processamento de dados distribuído e paralelo em larga escala fornecendo armazenamento em memória para cálculos intermediários. Possui bibliotecas compostas de APIs para processamento de fluxo de dados em tempo real aplicando aprendizado de máquinas, processamento de grafos e consultas.[10] A Figura 1 descreve os módulos da arquitetura do *middleware* e as funcionalidades desempenhadas por cada um.

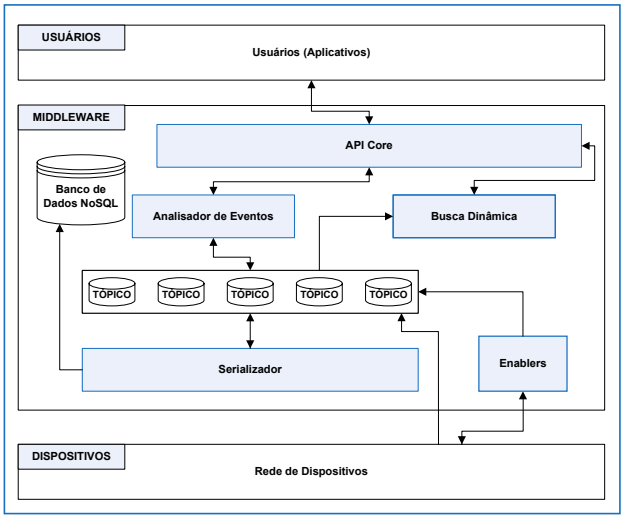


Fig. 1. Middleware - Arquitetura Proposta

- Serializador: esse módulo é responsável pela uniformização, serialização e armazenamento das mensagens que são enviadas ao *middleware*. Diferentes mensagens que chegam ao *middleware* são re-formatadas para um novo padrão que é igual para todos os dispositivos, independentemente do fabricante.
- Busca dinâmica: este agente é responsável por criar consultas dinâmicas (em padrão similar ao SQL) que são executadas em tempo quase real utilizando filtros com critérios definidos pelos usuários nas aplicações. Toda mensagem que chega ao *middleware* tem seus campos testados segundo as regras de consultas definidas pelos usuários. Aqueles pacotes que atendem a consulta, são separados em um tópico específico, onde a aplicação as consome.
- Analisador de Eventos: é responsável por identificar padrões comuns em diferentes pacotes. Através de técnicas de *clustering*, busca em um grupo de pacotes eventos raízes que associem tais pacotes. Alguns destes eventos são: queda de energia, subtensão, sobretensão. Isto quer dizer que quando falta energia em uma área, todos os dispositivos mandam - em curto prazo de tempo - mensagens individuais de queda de energia. Antes de repassar estas mensagens para a aplicação, o CEP ou 'Analisador de Eventos' identifica o que estes pacotes tem em comum. Ao perceber que eles têm o mesmo evento e pertencem a uma mesma área geográfica, desconsidera todos estes pacotes individuais e gera um novo e único pacote que caracteriza o evento na área.
- API Core: nesse módulo o *middleware* - através de um conjunto de APIs - disponibiliza funções para integração com outros aplicativos permitindo que outros softwares possam também receber dados do sistema aqui proposto.
- Enablers: através desse módulo o *middleware* se comunica com a rede de dispositivos, sendo responsável por integrar diversos protocolos e tecnologias. Por exemplo, a tecnologia de comunicação LoRa/LoRaWAN tem sua própria dinâmica e exige um servidor próprio conhecido como LNS (Lora Network Server). Neste caso, o LNS é um *enabler* que faz a ponte entre um dispositivo LoRa e o *middleware*. Outro exemplo e a comunicação com o protocolo DNP3 (típica do setor elétrico). Ele envolve o uso de um servidor DPN3 *master* como *enabler*. Este último trata as peculiaridades (do protocolo ou da tecnologia de comunicação) e despacha para o *middleware* - via MQTT - um pacote JSON com os dados recebidos.

Como observado na Figura 1, todas mensagens encaminhadas ao *middleware* são inicialmente interceptadas pelo agente serializador, que identifica o tipo da mensagem e o formato da mensagem (texto puro ou *JSON*). Esse agente uniformiza todas as mensagens, serializando no formato binário do Apache Avro, gravando-as nos respectivos tópicos de acordo com o tipo da mensagem. Apache Avro é um sistema de serialização de dados no formato binário. Fornece diversas estruturas de dados com a utilização de esquemas, tornando a serialização rápida e pequena [11]. Esse módulo também possui a funcionalidade de armazenar as mensagens em banco de dados do tipo NoSQL, de acordo com o tipo da mensagem, para futuras consultas de séries históricas. Seu uso visa otimização de desempenho tanto na gravação quanto na leitura de dados.

O *middleware* não deve ser capaz apenas de ler dados de diferentes fontes. Deve também propor canais de comunicação para prover a outros eventuais sistemas informações contidas nele. Para isto é também desenvolvida uma "API Core"(ver Figura 1) que permita a outras aplicações internas ou externas, consultar dados do *middleware*. Isto é feito através de um conjunto de métodos no formato API *Representational State Transfer* (REST) e apresenta um formato padronizado de requisições *Hypertext Transfer Protocol* (HTTP) para estas aplicações.

O agente de 'análise de eventos' usa em tempo real o algoritmo de clusterização K-means. Ele a avalia o agrupamento das mensagens no espaço n-dimensional (onde n é o número de campos de uma mensagem de evento). Aqueles *clusters* identificados são então convertidos em eventos de área. A limiarização de *clusters* é personalizada pelo usuário. Para garantir desempenho, a implementação do agente 'analisador de eventos' foi feita em Apache Spark, plataforma esta conhecida pelo seu alto paralelismo de processamento. O banco de dados usado nesta prova de conceito de *middleware* foi o MongoDB.

## III. Testes e validação

Para avaliar as funções propostas para o *middleware* e sua arquitetura, foi implantado um conjunto de cenários de testes.

Cada um deles avalia um (ou mais) recursos funcionais da solução proposta. A Tabela I mostra a distribuição dos serviços utilizando 4 servidores (Dell PowerEdge M630), 128 GB RAM, cada um com as mesmas configurações de hardware.

TABELA I
DISTRIBUIÇÃO DOS SERVIÇOS ENTRE OS SERVIDORES

| Serviço | Servidor | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| Kafka Cluster | X | X | X | |
| Spark Cluster | X | X | X | |
| Database NoSQL | | | | X |
| RESTful Services | | | | X |

Foi implementado um programa de computador para automatização de testes que gera mensagens sintéticas de dados de telemedição, designado como "gerador de mensagens **MOCK**". Um conjunto de mensagens em diferentes formatos foi produzido pelo MOCK para simular o monitoramento de uma planta elétrica de distribuição fictícia. Foi escolhida a planta elétrica de distribuição da cidade de Brasília para simular seu comportamento. Isto foi feito usando a base de dados da ANEEL (Agência Nacional Energia Elétrica) conhecida como BDGD [12] que anualmente é atualizada com o cadastro de todos os ativos da rede de distribuição. Associados a estes ativos, foram geradas três tipos de mensagens (ou pacotes) sintéticas de telemedição a saber:

- Pacote de monitoramento e telemedição (**P10 e P30**): contém medidas síncronas (periódicas) de telemedições de ativos. Quando o pacote é marcado em seu cabeçalho com a flag P10, indica que está monitorando um importante ativo da rede elétrica (um transformador, religador, etc). Por isto tem nível de prioridade médio para telemedição. Quando a flag é a P30, indica que monitora uma unidade consumidora (casas, comércios, indústrias, etc) de baixa, média e alta tensão. Tem geralmente nível de prioridade baixo.
- Pacote de eventos (**P13**): mensagens assíncronas geradas quando o dispositivo de hardware do monitoramento identifica algum cenário de alerta ou alarme. Este pacote é usado para informar as aplicações sobre a ocorrência de eventos (sobrecorrentes, variações tensões, etc) em tempo real. Tem geralmente o maior nível de prioridade.

Quanto a caracterização das funcionalidades básicas do *middleware*, foram definidos os cenários de testes elencados na sequência. Os testes de categoria 1 avaliaram o desempenho computacional do *middleware*. Já os de categoria 2 e 3 avaliaram os agentes. Nestes testes, o MOCK foi configurado para ler as tabelas UCBT, UCMT e UCAT (contem cadastro das unidades consumidoras) do BDGD para gerar pacotes P30. Ao todo são cerca de 1,2 milhão de registros de clientes. A tabela UNTRD do BDGD também foi usada para gerar pacotes P10 de monitoramento de ativos. Todos estes pacotes eram enviados pelo MOCK repetidamente ao *middleware* para estes testes.

- Teste 1A : neste teste, o MOCK envia um fluxo de pacotes ao *middleware*. Estes pacotes são então formatados pelo *middleware* para o padrão JSON ou Apache Avro de acordo com os parâmetros definidos. O *middleware* encaminha cada mensagem para seu respectivo tópico. O objetivo é avaliar o processamento de eventos de diferentes protocolos de medição uniformizando em um único padrão de mensagens verificando o tempo gasto.
- Teste 1B : este teste foca na capacidade do *middleware* em armazenar diretamente no BD os dados recebidos (desonerando as aplicações desta tarefa) com relevante volume de dados/pacotes. O MOCK gera seu fluxo de pacotes que ao chegarem no *middleware*, faz o processo de serialização, seja em JSON ou Apache Avro. Em seguida, são armazenados (pelo *middleware*) no banco de dados de acordo com o tipo do pacote.
- Teste 1C : neste teste verifica-se a capacidade de funcionamento do agente "busca dinâmica". Para isto, os pacotes do MOCK são preenchidos com valores randômicos dentro de um intervalo bem identificado. São eles: 100<Corrente<300; 210<Tensão<230; 1000<Pot. Aparente<2000; 0.9<Fator potência <1. Os outros campos os valores são constantes e indiferentes. Foram cadastradas por uma eventual aplicação as seguintes regras de consultas a serem executadas pelo agente 'analisador de eventos' do *middleware*. Para cada uma destas regras criadas, é também criado um "tópico"no *middleware* que receberá uma cópia dos pacotes que atenderam a condição encontrada pelo agente. Neste tópico, a aplicação pode consultar os pacotes selecionados. Foram gerados alguns pacotes especiais para serem selecionados e com as condições: (i) condição 1: Corrente > 300;(ii) condição 2: Corrente > 300 e Tensão > 230 e Fator de Potência < 0.9. O objetivo deste teste é caracterizar o funcionamento do sistema de filtragem de mensagens em tempo real.
- Teste 2A: nesse teste foi feita uma filtragem simples identificando os pacotes com a condição Corrente > 200A e gravados em banco de dados. Teve como objetivo verificar a capacidade computacional em relação ao número de partições utilizadas, mensagens utilizando ou não a sinalização de confirmação de mensagens (ACK) avaliando a latência do pacote filtrado na condição colocada durante a gravação em banco de dados.
- Teste 3A: este teste foca na capacidade do evento 'analisdor de eventos'. O MOCK foi configurado como nos testes anteriores, gerando cerca de 1,2 milhões de pacotes. Aleatoriamente foram selecionados 4 transformadores (da tabela UNTRD) e 60% das unidades consumidoras ligadas a ele foram configuradas para gerar alarmes (pacote P13) de queda de energia (pacote P33). O módulo 'Analisador de Eventos' deverá fazer o cluster dos pacotes e em havendo algum, identificar se o número de mensagens obedece ao percentual estipulado de 60% de unidades consumidoras para gerar uma mensagem de evento raiz. Esses pacotes tem as coordenadas geográficas do ativo e o código da UNTRD relacionada. Baseado no algoritmo K-means, onde serão analisados os eixos específicos e a formação de *clusters* com base nos seguintes eixos: coordenadas geográficas e o circuito alimentador.

## IV. Resultados e Discussões

A Figura 2 apresenta o tempo gasto em função do número de partições Kafka no *cluster* computacional. O *cluster* foi configurado com 3 *brokers* no mesmo servidor (*single node*). Nesse cenário foram enviados 1,2 milhões de mensagens no formato P10 e P30. A configuração '*ACK-0*' indica que não é aguardada a confirmação da mensagem gravada no *broker*. Já para a configuração '*ACK-1*', o *broker* aguarda que a mensagem seja gravada somente no *broker* líder, enquanto para *ACK-all* o *broker* líder aguardar que todas as réplicas sejam efetivamente gravadas. Os resultados mostram que essa variação no número de partições influencia no tempo gasto para processar as mensagens. Nesse cenário o tempo médio gasto ficou próximo de 21 segundos com 10 partições e configuração de *ack-all*. Comparando com o mesmo número de partições e a configuração de confirmação de mensagem *ack-all*, este cenário apresentado na Figura 3 teve um tempo médio 14% maior utilizando os *brokers* em servidores separados. Entretanto, nesse cenário, arquitetura garante um serviço tolerante a falhas, mesmo que um dos servidores fiquem *off-line*.

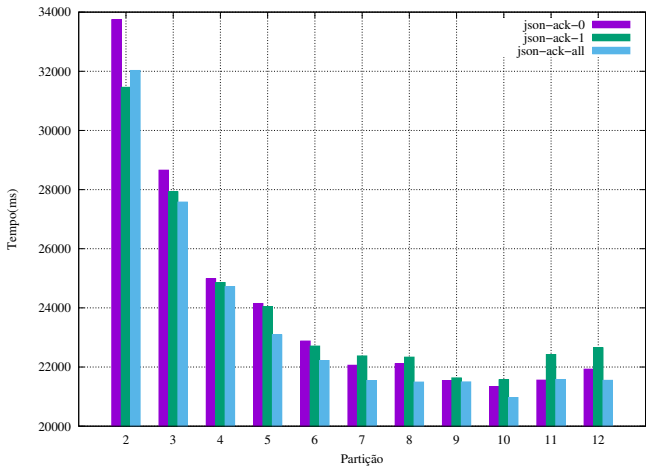


Fig. 2. Tempo gasto em função do número de partições no cluster - Single Node

Nas simulações apresentadas na Figura 3, o *cluster* também foi configurado com 3 *brokers*. Contudo, cada *broker* está sendo executado em servidor separado (*multi node*). A Figura mostra o tempo gasto em função do número de partições. Nesse cenário foram enviados 1,2 milhões de mensagens no formato P10 e P30. As simulações foram executadas para configurações *ACK-0*, *ACK1* e *ACK-all*. Todas as mensagens são serializadas no formato JSON e gravadas nos seus respectivos tópicos de acordo com o tipo da mensagem.

A Figura 4 e 5 apresenta o tempo de processamento de serialização das mensagens em função do número de partições utilizando os formatos *JSON* e o formato Apache Avro Binário [11], gravadas nos seus respectivos tópicos ou em tabelas no banco de dados. Nesse cenário foram enviadas 1,2 milhões de mensagens no formato P10 e P30 para serem serializadas.

Já a Figura 6 apresenta o atraso em função do número de partições no *cluster* para filtrar um conjunto de 20 mensagens

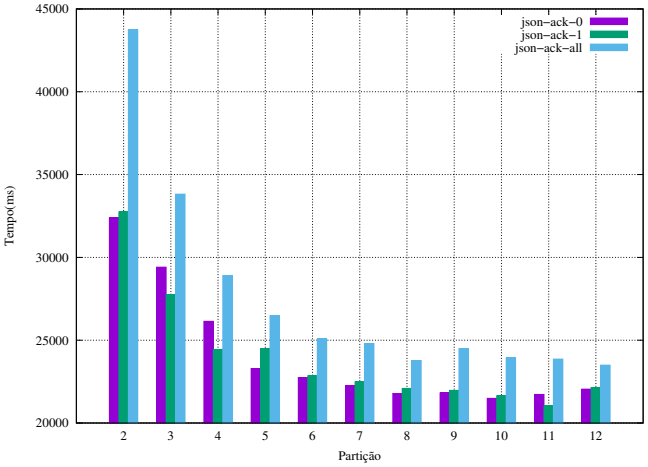


Fig. 3. Tempo gasto em função do número de partições no cluster - Multi Node

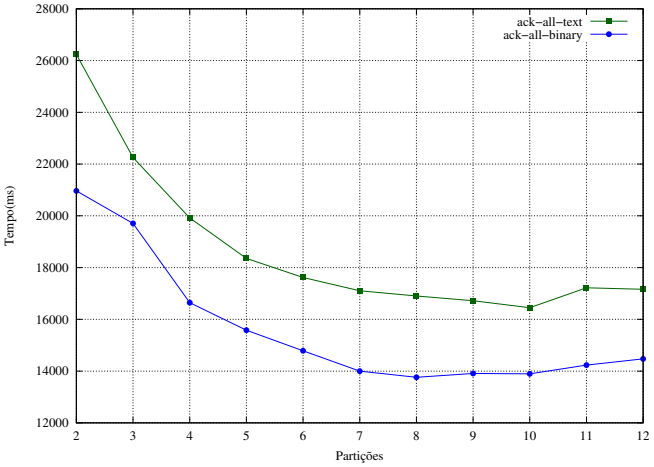


Fig. 4. Tempo processamento de serialização em função do número de partições

seguindo um critério. O *cluster* foi configurado no modo distribuído com 3 *brokers* em servidores distintos, com cada servidor com 1 *broker* em execução. Nesse cenário foram enviadas 1,2 milhões de mensagens no formato P30. Dentre essas mensagens, 20 unidades consumidoras foram selecionadas e configurados os respectivos campos para serem utilizados nos critérios de filtragem: **Condição 1:** Corrente > 300; **Condição 2:** Corrente > 300 AND Tensão > 230 AND Fator Potência < 0.9;

A Figura 7 e 8 apresenta o tratamento de eventos de CEP, mais especificamente o de "queda de energia", habitualmente o mais comum em algumas redes de distribuição. O alinhamento dos eventos no gráfico demonstra o padrão formado (ou seja, o *cluster*). Os eixos indicam a latitude e longitude destas unidades que emitiram mensagens de alarme (P13). Tudo é feito em tempo real e a visualização de tais gráficos é meramente ilustrativa. O próprio *middleware* faz tudo automaticamente, com base nas configurações indicadas.

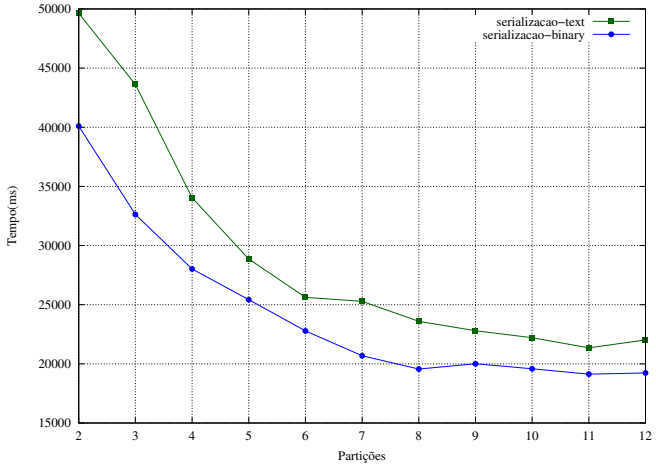


Fig. 5. Tempo de processamento – Armazenamento em Banco de Dados

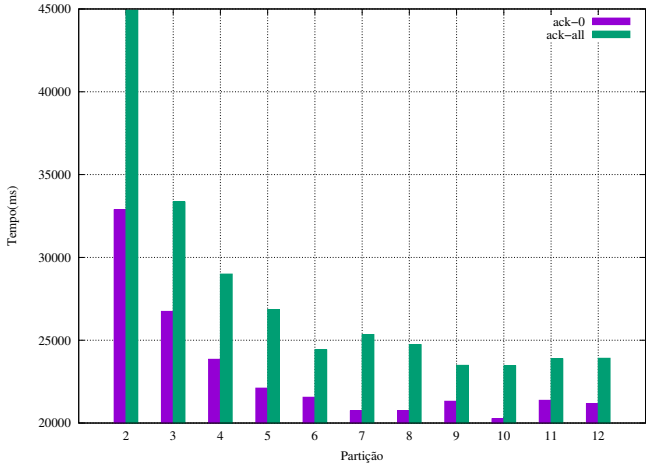


Fig. 6. Atraso - Filtragem de mensagens em tempo real

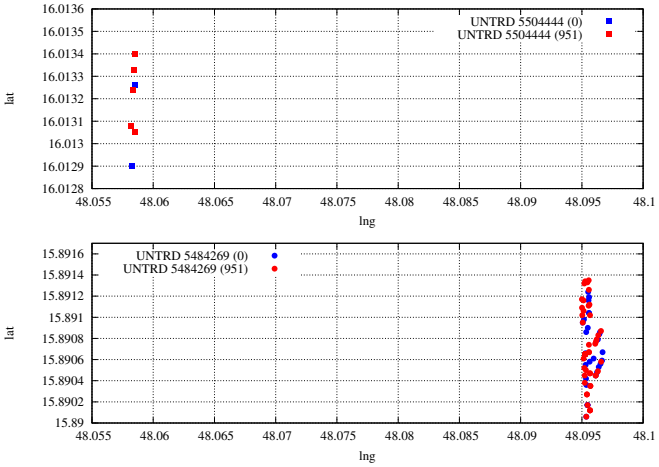


Fig. 7. Clusterização dos eventos - Queda de energia

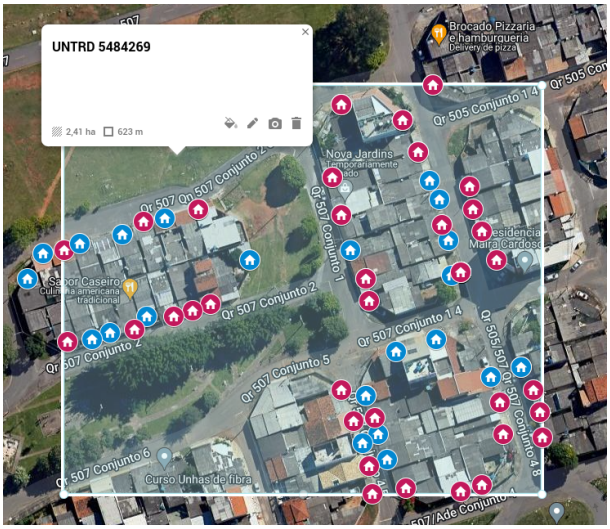


Fig. 8. Monitoramento CEP - Queda de energia - Visão Mapa

## V. Conclusões

Neste artigo foi proposto o desenvolvimento de *middleware* IoT distribuído para redes elétricas de distribuição. Por meio da arquitetura proposta foram implementadas funções para integrar dispositivos de comunicação heterogêneas, integração de dispositivos com processamento de eventos em larga escala com monitoramento em tempo quase real, provendo escalabilidade, redundância e uniformização de mensagens. Para trabalhos futuros, inclusão de recursos de inteligência artificial nas funcionalidades de CEP. Considerando as métricas e os cenários de testes utilizados, os resultados apresentados por meio de simulações mostraram que o *middleware* atende aos requerimentos entendidos como básicos para o setor elétrico, indicando a viabilidade desta tecnologia para o setor elétrico em alternativa aos clássicos sistemas supervisórios.

## Referências

[1] P. Zhang, F. Li, and N. Bhatt, "Next-generation monitoring, analysis, and control for the future smart control center," *IEEE Transactions on Smart Grid*, vol. 1, no. 2, pp. 186–192, 2010.

[2] D. Corral-Plaza, I. Medina-Bulo, G. Ortiz, and J. Boubeta-Puig, "A stream processing architecture for heterogeneous data sources in the Internet of Things," *Computer Standards and Interfaces*, vol. 70, no. January 2019, p. 103426, 2020.

[3] A. Akanbi and M. Masinde, "A distributed stream processing middleware framework for real-time analysis of heterogeneous data on big data platform: Case of environmental monitoring," *Sensors (Switzerland)*, vol. 20, no. 11, pp. 1–25, 2020.

[4] S. Shapsough, M. Takrouri, R. Dhaouadi, and I. A. Zualkernan, "Using IoT and smart monitoring devices to optimize the efficiency of large-scale distributed solar farms," *Wireless Networks*, vol. 27, no. 6, pp. 4313–4329, 2021.

[5] S. Cavalieri, G. Cantali, and A. Susinna, "Integration of IoT Technologies into the Smart Grid," *Sensors*, vol. 22, no. 7, 2022.

[6] Apache Kafka, "Apache Kafka." https://kafka.apache.org/, 2022.

[7] G. Fu, Y. Zhang, and G. E. Yu, "A Fair Comparison of Message Queuing Systems," *IEEE Access*, pp. 421–432, 2020.

[8] G. S. &. T. P. Neha Narkhede, *Kafka: The Definitive Guide*. 2017.

[9] M. Wooldridge, *An Introduction to MultiAgent Systems - 2nd Edition*, vol. 41. 2009.

[10] Apache Spark, "Apache Spark™ - Unified Engine for large-scale data analytics." https://spark.apache.org/, 2022.

[11] Apache Avro, "Apache Avro." https://avro.apache.org/, 2022.

[12] ANEEL, "Regras e Procedimentos de Distribuição (Prodist) — Português (Brasil)." https://www.gov.br/aneel/pt-br/centrais-de-conteudos/procedimentos-regulatorios/prodist, 2022.