Grupo de Estudo de Sistemas de Informação e Telecomunicação para Sistemas Elétricos-GTL

Utilizando mensageria distribuída, com Apache Kafka®, com garantia de entrega consistente de dados coletados a partir de dispositivos IoT, e sua aplicação na Indústria 4.0 do setor elétrico

FREDERICK NAZARIO MOSCHKOWICH(1); ROBERTO LUIZ KLEIN FILHO(1); FPTI(1);

RESUMO

O setor energético não pode mais se basear apenas na geração/distribuição de energia pois vivemos a quarta revolução industrial, ou Indústria 4.0, que pressupõe a automação de diversos dispositivos conectados entre si, chamados de Internet das Coisas. Isso gera um grande volume de dados circulando nas plantas, podendo a chegar na casa de milhões de mensagens por segundo. Baseado nesse cenário, uma solução é a utilização de algum intermediador entre as mensagens geradas e os consumidores desses dados, chamado de *middleware*. A disponibilização desses *middleware* de forma distribuída pode trazer ainda mais resultados na transmissão do volume de informação gerado.

PALAVRAS-CHAVE: Automação, Mensageria Distribuída, Internet das Coisas, Indústria 4.0, Middleware

1.0 - INTRODUÇÃO

Com o passar dos anos, as empresas do setor elétrico - tanto geradoras como distribuidoras - se veem com muitos dispositivos e sistemas interligados, gerando um grande volume de dados a ser processado. Por contar muitas vezes com limitações técnicas por conta de retrocompatibilidade, muitos sistemas legados acabam consumindo uma grande quantidade de recursos de infraestrutura para a sua continuidade de funcionamento.

Este artigo se propõe a realização de um estudo utilizando o Apache Kafka® na chamada indústria 4.0 para interligação entre os diversos sistemas e dispositivos cyber-físicos existentes nas plantas de energia.

Atualmente, vários tipos de protocolo são usados na indústria, podemos destacar o Goose, MQTT e Modbus. Para a interligação entre os dados recebidos através desses protocolos, *webservices* via utilizando os padrões REST ou SOAP são utilizados, porém esse tipo de comunicação, que utiliza conexões HTTP como base, se mostram ineficientes por conta do tamanho dos pacotes trafegados.

Os chamados *brokers* - categoria em que se enquadra o Kafka - podem beneficiar a análise dos dados trocados entre os vários agentes presentes nas plantas de energia, já que usam pacotes menores e trafegam apenas na camada TCP.

Existem vários outros tipos de serviços de mensageria, alguns até com velocidade maior de transmissão, no entanto, não provém garantia de entrega dos dados, muito menos trabalham de forma distribuída, possuindo uma limitação na quantidade de dados trafegados.

(*) Av. Tancredo Neves, n˚ 6731 – sala 08 - Edifício das Águas – CEP 85.867-900 Foz do Iguaçu, PR – Brasil
Tel: (+55 45) 3576-7406 – Cel: (+55 45) 99909-3276 – Email: frederick.nazario@pti.org.br

2.0 - MENSAGERIA DISTRIBUÍDA

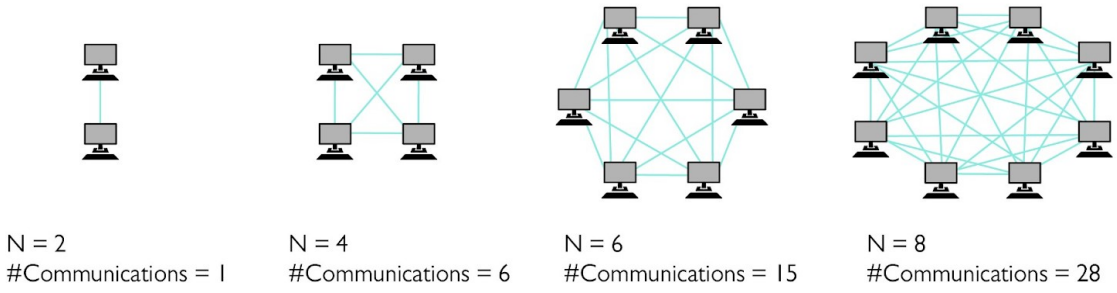

Figura 1- Exemplo de mensageria distribuída - Fonte: https://www.comsol.com/blogs/intro-distributed-memory-computing/

Ao se agregar um grande número de sistemas e/ou dispositivos se comunicando, temos um limite máximo de conexões entre eles, limitando, assim, o número máximo de dispositivos interconectados. Esse número máximo pode ser calculado através da fórmula:

$$\frac{1}{2}n(n-1)$$

Figura 2 - Onde n é o número de nós no sistema

Na tentativa de resolução deste problema, uma camada intermediária, *middleware*, pode ser inserida na tentativa de organizar melhor o fluxo entre produtores e consumidores. Conhecido como o padrão de projeto observador (*observing* em inglês), esses *middlewares* são uma camada lógica e de persistência que recebem, através de tópicos, as mensagens geradas pelos produtores e, por sua vez, os consumidores recebem essas mensagens de acordo com suas necessidades.

Existem diversos tipos de *middlewares* para o serviço de mensageria, tais como: *Mosquitto*, que utiliza o protocolo MQTT e *RabitMQ* que utiliza o protocolo AMQP. No entanto, em sua grande maioria, esse tipo de serviço não age de forma distribuída, fazendo com que não se possa garantir, de fato, a entrega das mensagens aos consumidores interessados.

A vantagem de utilização de *middleware* para a distribuição das mensagens entre sistemas/dispositivos reduz, de forma clara, a complexidade da escalabilidade horizontal. Com isso, também se obtêm vantagem no funcionamento dos dispositivos/sistemas, visto que, eles devem apenas processar o que for de sua responsabilidade, passando a informação ao *middleware*.

Neste cenário, caso o serviço de mensageria pudesse se utilizar de paralelismo e funcionar de modo distribuído em um *cluster*, pode-se obter resultados muito mais satisfatórios. Entre os resultados podemos destacar: maior escalabilidade, tolerância a faltas e garantia de entrega das mensagens. Com esta arquitetura distribuída, ao se adicionar mais nós de publicadores/assinante, podemos também adicionar outros nós no cluster de mensageria.

2.1 Apache Kafka

O Apache Kafka® é um desses *middlewares* que funciona como um *broker* de mensageria, funcionando também de forma distribuída de forma nativa. Foi desenvolvido pela equipe do *LinkedIn* para gerenciar os trilhões de transações diárias da rede social. Kafka utiliza o Apache Zookeeper® para manipular o processo de

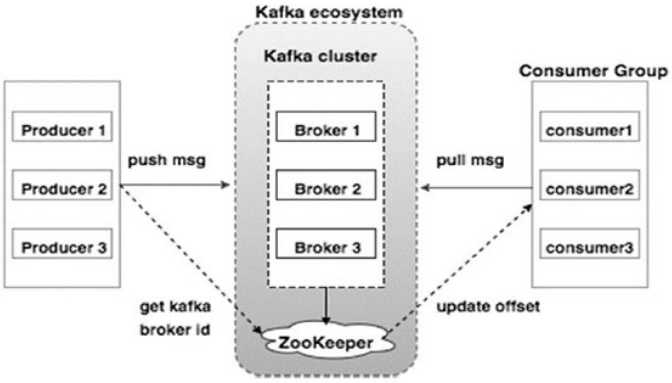

Figura 3 - Arquitetura do Apache Kafka

paralelismo e distribuição dos serviços, assim como suas configurações básicas e regras de acesso.

Com todo um ecossistema de serviços em sua volta, o Apache Kafka® se mostra poderoso para os mais variados casos em que é necessário um sistema publicador/assinante. Através da API *Stream*, o Kafka pode persistir dados que chegam através dos publicadores, processá-los e alterá-los para que sejam novamente inseridos na fila e, finalmente consumidos pelos assinantes.

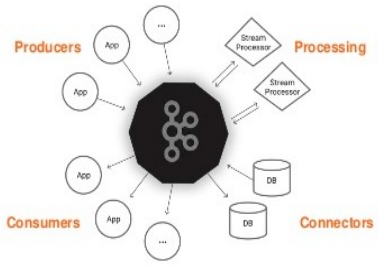

Figura 4 - Kafka Ecosistema

O sistema de tópicos do Kafka é dividido em partições onde os tópicos são gravados em diferentes nós do *cluster* fazendo com que, caso um dos nós do *broker* fique fora do ar, todo o restante do sistema continue funcionando de maneira transparente para os usuários. Os tópicos recebem diversas mensagens, estas por sua vez, contém um *payload* que pode conter desde um texto simples até um estrutura de dados tipo JSON, por exemplo.

As mensagens pode conter uma chave utilizada para fragmentar e compactar os tópicos. Com isso, se consegue garantir a ordem das mensagens, visto que as mensagens com a mesma chave são direcionadas para uma única partição. Além disso, existe no Kafka o chamado *replication factor* que funciona como parâmetro para copiar a partição para outros nós do *cluster*, garantindo, assim, a tolerância a falhas. Ou seja,

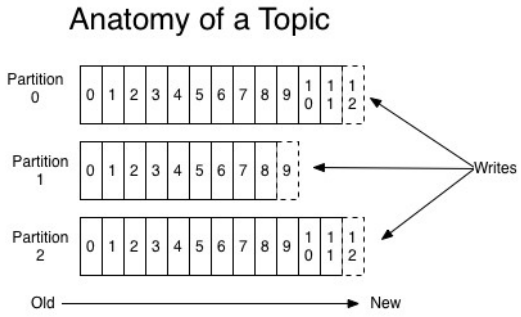

Figura 5 - Anatomia de um tópico

pode-se garantir tanto a ordem de chegada das mensagens como sua consistência.

Caso o sistema consumidor das mensagens tenha que lidar com um volume de dados muito grande, o Kafka possibilita a criação de um grupo de consumidores (*consumer groups*) onde cada um pode consumir uma partição de um tópico, balanceando a carga de leitura e provendo escalabilidade horizontal para os assinantes dos publicadores.

O Apache Kafka® conta também com um *framework*, chamado de Kafka *Connect*, que permite a interconexão direta entre o *broker* e sistemas externos, tais como banco de dados, sistemas de arquivos, entre outros. Os *connectors* são componentes prontos para uso que auxiliam da importação de dados externos em tópicos Kafka e também podem exportar esses tópicos diretamente a sistemas externos. Caso um *connector* não esteja disponível, é possível a criação de outros através de aplicações escritas na linguagem Java. Dentre os conectores já existentes, podemos destacar os compatíveis com o sistema de armazenamento da *Amazon*, o S3, um sem número de conectores para bancos de dados relacionais ou não, tais como: MySQL, PostgreSQL, Cassandra e InfluxDB.

2.2 Sistema de aquisição de dados

Um sistema de aquisição de dados, tem como principal objetivo, receber informações de processos físicos, a partir da leitura de sensores integrados a equipamentos, transformar esses dados em grandezas de engenharia e armazená-los em bases de dados para que, posteriormente, seja possível realizar a análise desses dados, a fim de obter informações pertinentes ao desempenho do equipamento, realizar manutenção preditiva ou utilizá-los para melhor gerir a produção.

Sendo assim, sistemas de aquisição de dados devem estar sempre disponíveis, garantindo que toda informação gerada, seja coletada, processada e armazenada. Devem também, ser escaláveis, ou seja, capazes de acompanhar o crescimento do número de equipamentos monitorados, o que acarreta em um aumento no volume de dados a serem aquisitados.

Para que isso seja possível, o *software* deve ser implementado utilizando tecnologias que possibilitem esse crescimento, garantindo a escalabilidade, disponibilidade e integridade das informações coletadas.

O sistema computacional proposto para este artigo, tem a finalidade de ser utilizado para realizar o monitoramento de equipamentos que, possuam sensores ligados a eles e os repassem a um servidor central a fim de armazenamento temporal dos dados.

2.3 Implementação

O sistema demonstrado neste artigo utiliza a utilização de microsserviços, pois, assim, é possível simular de maneira escalável, o funcionamento como se estivesse implementado em qualquer tipo de indústria do setor energético.

Microsserviços são *softwares* pequenos, com responsabilidade única e autônomos que funcionam em conjunto, mas que, caso um deles não esteja operacional, não afete a atividade de outro microsserviço. Como benefício na utilização de microsserviços podemos destacar: a liberdade para substituição e composição de serviços sem afetar todo o ecossistema; facilidade de entrega, entrega contínua, escalabilidade, entre outros benefícios

2.3.1. Serviço de coleta

Responsável em receber dados brutos dos simulando os dispositivos e gravar em uma base dados - no caso utilizando o InfluxDB.

2.3.2. Serviço de manipulação

Responsável por receber o *payload* de saída do serviço de coleta e transformar cada canal do equipamento em um objeto novo a ser enviado para o próximo serviço.

2.3.3. Serviço de verificação

Responsável por receber o *payload* de saída do serviço de manipulação, consultar uma base dados com informações sobre os equipamentos e seus canais. Com isso é possível identificar qual processo físico está associado ao canal do equipamento, criando um novo *payload* do objeto tratado para envio ao serviço de persistência.

2.3.4. Serviço de persistência

Responsável por receber o *payload* de saída do serviço de verificação e gravar esses dados no banco de dados tratados.

2.4 Estudo de Caso

Para este estudo de caso, foi utilizado uma ferramenta de simulação de carga, pois o teste utilizando centenas de equipamentos reais ficaria inviável. A ideia é verificar o comportamento de diversos dispositivos gerando, coletando e transmitindo dados ao sistema de aquisição.

Foram definidos dois cenários onde o teste irá abrir processos (*threads*) paralelos por dois minutos utilizando um conjunto de carga útil a fim de se aproximar o máximo possível do ambiente real em campo.

Para a geração de tantos processos, foi utilizado um computador industrial com 50 núcleos, 128 *GigaBytes* de memória *RAM*, este contendo uma instância da ferramenta de simulação de carga e outros 7 computadores com processador i7 de 6ª geração com 16 *GigaBytes* de memória *RAM*, um para cada microsserviço e um para cada par do kafka e Zookeeper, sendo três no total.

Sendo assim, utilizando a arquitetura distribuída proposta, foi possível simular um ambiente industrial real, afim de validar a utilização do Apache Kafka para a troca de informações entre microsserviços.

Para isso, dois testes foram realizados, ambos com a mesma duração de 2 minutos, um intervalo de envio de 500ms (duas requisições por segundo) e com o mesmo *payload* de entrada, conforme figura a seguir:

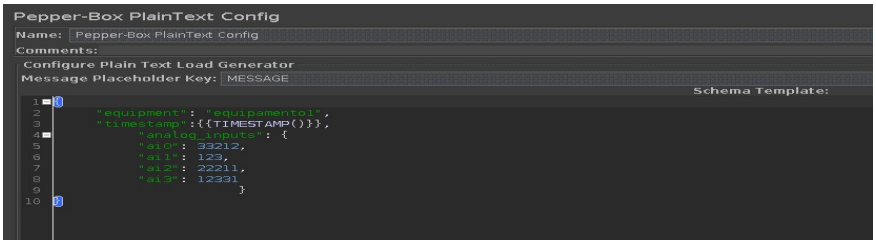

Figura 6 - Payload de entrada

Os dois testes diferenciam-se apenas pela quantidade de *threads* (equipamentos simulados) simultâneas.

No primeiro cenário, a ferramenta de teste carga foi configurada para simular a utilização de 100 equipamentos simultâneos e no segundo teste, a concorrência foi aumentada para 500 equipamentos.

2.5 Resultados Obtidos

No primeiro cenário, o simulador de carga foi capaz de enviar 22800 requisições no tempo determinado, conforme figura abaixo:

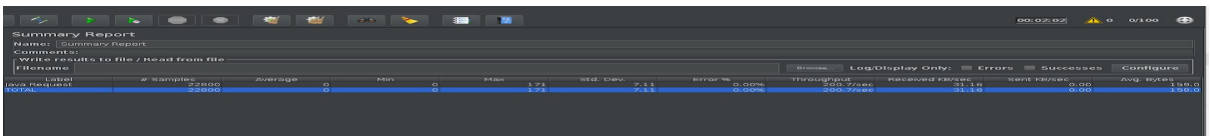

Figura 7- Teste com 100 threads

O microsserviço de coleta, foi capaz de receber, processar e armazenar todas as requisições, com 100% de eficácia, ou seja, sem apresentar nenhum erro ou inconsistência nos dados, o que pode ser visto na imagem a seguir:

```
Connected to http://localhost:8086 version 1.3.9
InfluxDB shell version: 1.3.9
> use dadosBrutos
Using database dadosBrutos
> select count(*) from dadosBrutos
name: dadosBrutos
time count_ai0 count_ai1 count_ai2 count_ai3 count_timestampEnvio count_timestampServicoColeta
---- --------- --------- --------- --------- -------------------- ----------------------------
0    22800     22800     22800     22800     22800                22800
>
```

Figura 8 - Total de registros gravados na base de]dados

O microsserviço de persistência, recebe uma quantidade de requisições diretamente proporcional a quantidade de canais do *payload* original, e que neste caso, foi capaz de processar e armazenar 91.200 registros, ou seja, 22.800 registros para cada grandeza, conforme podemos observar na figura abaixo:

```
> select count(*) from luminosidade
name: luminosidade
time count_timestampEnvio count_timestampServicoColeta count_timestampServicoManipulacao count_timestampServicoSalvaTratados count_timestampServicoVerificacao count_value
---- -------------------- ---------------------------- ------------------------------- ------------------------------- ------------------------------- -----------
0    22800                22800                        22800                           22800                           22800                           22800
> select count(*) from pressao
name: pressao
time count_timestampEnvio count_timestampServicoColeta count_timestampServicoManipulacao count_timestampServicoSalvaTratados count_timestampServicoVerificacao count_value
---- -------------------- ---------------------------- ------------------------------- ------------------------------- ------------------------------- -----------
0    22800                22800                        22800                           22800                           22800                           22800
> select count(*) from umidade
name: umidade
time count_timestampEnvio count_timestampServicoColeta count_timestampServicoManipulacao count_timestampServicoSalvaTratados count_timestampServicoVerificacao count_value
---- -------------------- ---------------------------- ------------------------------- ------------------------------- ------------------------------- -----------
0    22800                22800                        22800                           22800                           22800                           22800
> select count(*) from temperatura
name: temperatura
time count_timestampEnvio count_timestampServicoColeta count_timestampServicoManipulacao count_timestampServicoSalvaTratados count_timestampServicoVerificacao count_value
---- -------------------- ---------------------------- ------------------------------- ------------------------------- ------------------------------- -----------
0    22800                22800                        22800                           22800                           22800                           22800
>
```

Figura 9 - Dados tratados gravados com suas grandezas

No segundo cenário o simulador foi capaz de produzir 110.963 requisições, conforme figura a seguir:

Figura 10 - 500 threads no segundo cenário

E da mesma forma que o primeiro teste, o serviço de coleta foi capaz de receber, processar e armazenar todas as requisições, sem apresentar erros ou inconsistência, como demonstrado a seguir:

```
Connected to http://localhost:8086 version 1.3.9
InfluxDB shell version: 1.3.9
> use dadosBrutos
Using database dadosBrutos
> select count(*) from dadosBrutos
name: dadosBrutos
time count_ai0 count_ai1 count_ai2 count_ai3 count_timestampEnvio count_timestampServicoColeta
---- --------- --------- --------- --------- -------------------- ----------------------------
0    110963    110963    110963    110963    110963               110963
>
```

Figura 11- Número total de registros gravados na base de dados de coleta

O microsserviço de persistência, apresentou uma pequena inconsistência devido a limitação de memória que ocorreu no decorrer do teste, o que causou uma pequena perda de dados, podendo ser observada na figura a seguir:

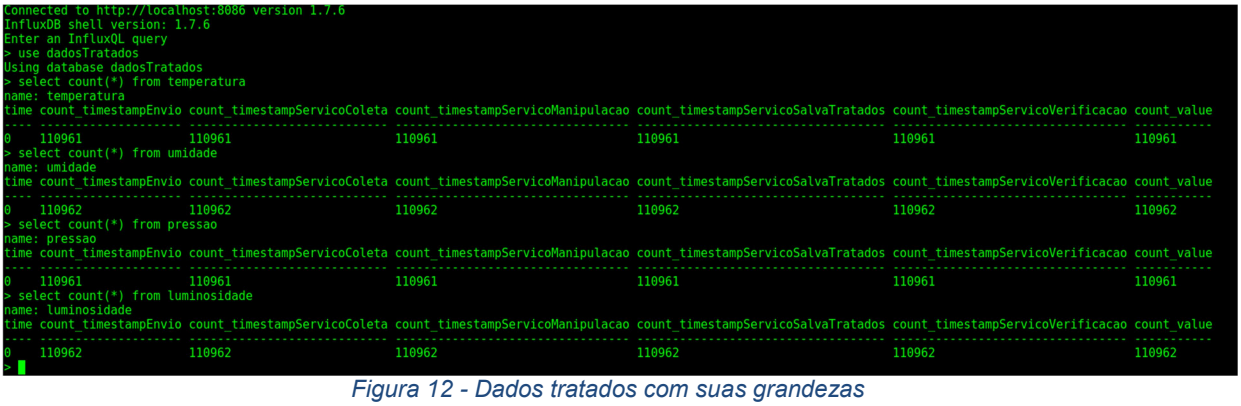

Figura 12 - Dados tratados com suas grandezas

3.0 - CONCLUSÃO

Com o estudo proposto neste artigo, pode-se observar que a solução utilizando microsserviços com mensageria distribuída é altamente viável para utilização juntamente com diversos sistemas cyber-físicos que porventura existam em plantas de energia, sejam elas de geração ou distribuição de energia.
Pode-se observar que houve uma pequena perda de informações quando existem acima de 500 requisições simultâneas devido a limitação de *hardware* utilizados no teste.
Porém, para que isso não ocorra, pode-se acrescentar outras aplicações (*consumer group)* para um melhor balanceamento de carga nas aplicações, considerando que o Apache Kafka foi capaz de processar todos os recebimentos de *payloads*. Além disso, a utilização da API do Kafka *Streams* pode obter maior desempenho no tratamento da transformação das informações que são tratadas pelo microsserviço Manipula.

4.0 - REFERÊNCIAS BIBLIOGRÁFICAS

(1) ANDERL, R. Industrie 4.0 - advanced engineering of smart products and smart production 09 october 2014. International Seminar on High Technology, 01 2015.

(2) BRETTEL, M. et al. How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective. International journal of mechanical, industrial science and engineering, v. 8, n. 1, p. 37–44, 2014.

(3) FOWLER, M.; LEWIS, J. Microservices a definition of this new architectural term. URL: http://martinfowler. com/articles/microservices. html, 2014.

(4) GARG, N. Apache Kafka. [S.l.]: Packt Publishing Ltd, 2013.

(5) NEWMAN, S. Building microservices: designing fine-grained systems. [S.l.]: ”O’Reilly Media, Inc.”, 2015.

(6) THEIN, K. Apache kafka: Next generation distributed messaging system. International Journal of Scientific Engineering and Technology Research, v. 3, n. 47, p. 9478–9483, 2014.

(7) WOLFF, E. Microservices: flexible software architecture. [S.l.]: Addison-Wesley Professional, 2016.

5.0 - DADOS BIOGRÁFICOS

Frederick Nazario Moschkowich
Natural de Rio de Janeiro - RJ

Possui graduação em Jornalismo pelo Centro Universitário do Rio de Janeiro (UniverCidade, 2005) e graduação como Tecnólogo em Análise e Desenvolvimento de Sistemas pelo Instituto Federal do Paraná (IFPR, 2017). Cursando pós-graduação em Arquitetura de Software Distribuído pela Pontifícia Universidade Católica de Minas Gerais (PUC-MG, 2019) e aluno especial do mestrado em Engenharia Elétrica e Computação pela Universidade do Oeste Paranaense (Unioeste, 2019). Atua como analista de sistemas e desenvolvimento de aplicações para o sistema elétrico de potência desde 2017 no Laboratório de Automação e Simulação de Sistemas Elétricos (LASSE) na Fundação Parque Tecnológico Itaipu (PTI).

Roberto Luiz Klein Filho
Natural de Curitiba - PR
Possui graduação como Tecnólogo em Análise e Desenvolvimento de Sistemas pelo Instituto Federal do Paraná (IFPR, 2017). Cursando pós-graduação em Tecnologia Java pela Universidade Tecnológica Federal do Paraná (UTFPR, 2019). Atua como analista de sistemas e desenvolvimento de aplicações para o sistema elétrico de potência desde 2017 no Laboratório de Automação e Simulação de Sistemas Elétricos (LASSE) na Fundação Parque Tecnológico Itaipu (PTI).