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

DACIO SILVA BEZERRA

NODE-RED:
LOW-CODE PARA RESOLVER PROBLEMAS DE SISTEMAS ORIENTADOS A EVENTOS

CAMPINA GRANDE - PB

2022

DACIO SILVA BEZERRA

NODE-RED:
LOW-CODE PARA RESOLVER PROBLEMAS DE SISTEMAS ORIENTADOS A EVENTOS

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

Orientador : Professor Dr. Andrey Elísio Monteiro Brito

CAMPINA GRANDE - PB

2022

DACIO SILVA BEZERRA

NODE-RED:
LOW-CODE PARA RESOLVER PROBLEMAS DE SISTEMAS ORIENTADOS A EVENTOS

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

BANCA EXAMINADORA:

Professor Dr. Andrey Elísio Monteiro Brito
Orientador – UASC/CEEI/UFCG

Professor Dr. Reinaldo Cezar De Morais Gomes
Examinador – UASC/CEEI/UFCG

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

Trabalho aprovado em: 02 de Setembro de 2022.

CAMPINA GRANDE - PB

RESUMO

A crise resultante da pandemia de COVID-19 ainda não foi embora, mas já deixou várias lições. Países como Malásia e Coréia do Sul foram exemplos na contenção de diversos efeitos da crise, implementando sistemas de rastreamento de contato e distribuição de recursos. Essas duas atividades são exemplos de coordenação de dados em tempo real. A coordenação de recursos deveria ser mais fácil, mas cada fabricante, distribuidor e sistema hospitalar possuem sistemas que são diferentes e não podem compartilhar dados facilmente. De fato, nos próximos anos, tecnologias irão se consolidar de tal forma que haverão protocolos de comunicação assim como temos os nossos protocolos de rede para harmonização entre agentes. O presente artigo se importa em detalhar e exemplificar o uso da ferramenta Node-Red como uma forma de resolver tais problemas de harmonização e integração de dados em tempo real por meio do Low-Code, utilizando-se de um cenário acadêmico real. Além da exemplificação da construção de um plugin tendo em vista a ferramenta supracitada para suporte da solução de integração.

Palavras-chave: *Node-Red*, *Low-Code*, integração, eventos, mensagem, ferramenta, fluxo, *plugin*, *Kafka*, *Event Hubs*.

Node-RED: Low-Code para resolver problemas de sistemas orientados a eventos

Dacio Silva Bezerra
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
dacio.bezerra@ccc.ufcg.edu.br

Andrey Elísio Monteiro Brito
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
andrey@computacao.ufcg.edu.br

RESUMO

A crise resultante da pandemia de COVID-19 ainda não foi embora, mas já deixou várias lições. Países como Malásia e Coréia do Sul foram exemplos na contenção de diversos efeitos da crise, implementando sistemas de rastreamento de contato e distribuição de recursos. Essas duas atividades são exemplos de coordenação de dados em tempo real. A coordenação de recursos deveria ser mais fácil, mas cada fabricante, distribuidor e sistema hospitalar possuem sistemas que são diferentes e não podem compartilhar dados facilmente. De fato, nos próximos anos, tecnologias irão se consolidar de tal forma que haverão protocolos de comunicação assim como temos os nossos protocolos de rede para harmonização entre agentes. O presente artigo se importa em detalhar e exemplificar o uso da ferramenta *Node-Red* como uma forma de resolver tais problemas de harmonização e integração de dados em tempo real por meio do *Low-Code*, utilizando-se de um cenário acadêmico real. Além da exemplificação da construção de um plugin tendo em vista a ferramenta supracitada para suporte da solução de integração.

PALAVRAS-CHAVE

Node-Red, *Low-Code*, integração, eventos, mensagem, ferramenta, fluxo, *plugin, Kafka, EventHubs*.

REPOSITÓRIO

https://github.com/DacioSB/node-red-contrib-kafka-eventhub

1. INTRODUÇÃO

Antes de entendermos o problema sobre integração de fluxo de dados que temos hoje, precisamos entender o que é um *flow* ou fluxo[1]. O *flow* apenas encontra sentido em sua existência se houver serviços e aplicativos que preparam os dados para *streaming*, consomem o *flow* ou o manipulam de outra forma.

Um fluxo é composto por:

- Consumidores: Fazem requisições de streams para adquirirem dados a partir de interfaces auto gerenciadas;
- Produtores: Escolhem quais requisições aceitar ou rejeitar;
- Uma vez que a conexão se estabelece entre consumidores e produtores, consumidores não necessitam mais, de forma ativa, realizar requisições para produtores. É algo automaticamente disponibilizado para estes agentes;
- Produtores: mantêm controle da transmissão da informação, quando e quem deve fazer a transmissão;
- A informação é transmitida sobre protocolos padrões de rede.[1]

Atualmente, a integração entre organizações é um "faça você mesmo", ou "*do it yourself*". Os desenvolvedores precisam trabalhar bastante para entender (ou mesmo definir) o "encanamento" pelo qual trocam dados. E, se eles desejam uma troca de dados em tempo real, devem esperar que a fonte dos dados tenha criado uma API (*Application Programming Interface*) e um protocolo facilmente consumíveis[2].

Algumas soluções embrionárias[3] vêm surgindo nos últimos anos, como é o caso das funções *serverless* como as funções *lambda* da AWS (*Amazon Web Services*)[2] ou *Azure Functions*[3]. Basta escrever uma função, configurar algumas opções sobre dependências e conectividade e implantá-la. Posteriormente, a função é executada sempre que um evento aplicável é acionado.

Contudo, tudo isso exige mão de obra qualificada e tempo de trabalho na solução. O problema reside no fato de que nem sempre conseguimos isso de forma simples. Em uma pesquisa realizada em 2019 pelo *thenewstack.io*[4] com 300 desenvolvedores, foi constatado que estes passam 61% do seu tempo escrevendo novo código ou fazendo manutenção no já existente. O restante do tempo é utilizado para planejamento, testagem, resolução de problemas de segurança, dentre outras atividades importantes.

[1] "Os autores retêm os direitos, ao abrigo de uma licença Creative Commons Atribuição CC BY, sobre todo o conteúdo deste artigo (incluindo todos os elementos que possam conter, tais como figuras, desenhos, tabelas), bem como sobre todos os materiais produzidos pelos autores que estejam relacionados ao trabalho relatado e que estejam referenciados no artigo (tais como códigos fonte e bases de dados). Essa licença permite que outros distribuam, adaptem e evoluam seu trabalho, mesmo comercialmente, desde que os autores sejam creditados pela criação original."

[2] https://aws.amazon.com/lambda/

[3] https://docs.microsoft.com/en-us/azure/azure-functions

1.1 O problema da flexibilidade no protocolo de mensageria

Por vezes queremos ter a flexibilidade de trabalhar com eventos em plataformas distintas, de forma a resolver problemas distintos. Por vezes um ambiente *on-premise*, como um servidor local, pode resolver uma gama de problemas comuns. Outras vezes o problema será resolvido apenas com um cluster *Kubernetes* robusto, equipado com máquinas virtuais para colocar fluxos de dados nesse local e não se preocupar com nada, tendo a certeza que o servidor não irá ser super utilizado com consumo de recursos absurdos, nem que é preciso se preocupar em fazer manutenção no mesmo.

Sabendo disso, precisamos compatibilizar dois mecanismos: *on-premise* e *cloud* (*Apache Kafka*[4] e *Azure Event Hubs*[5], por exemplo), de forma a trabalharmos da maneira mais confiável possível, tanto na nuvem, quanto localmente.

1.2 Exemplos do mundo real

Vamos exemplificar, para melhor esclarecimento do tópico, quais seriam potenciais integrações que exigiriam uma arquitetura como esta. Adiante, iremos apresentar dois âmbitos de aplicação.

Empresarial: Nesse ramo, temos várias integrações ocorrendo. Produtores de dados enviando dados de um lado e outros consumidores de dados recebendo eventos em outra ponta. Nem sempre os tipos de dados (produzidos e consumidos) são do mesmo tipo. Muitas vezes esse dado merece tratamento e processamento para sair de uma ponta para a outra. Vamos a um exemplo prático: sensores ou seres humanos coletores enviam dados em tempo real para uma base de dados. Esses eventos são enviados por esses sensores a cada 15 minutos ou a qualquer tempo por coletores humanos. Do outro lado eu possuo uma API (*Application Programming Interface*) que espera um modelo de dados que represente um ponto sinal de falha ou quase falha de algum componente da indústria em questão, um eixo, tambor, container, etc. De forma convencional, para se construir uma solução de integração como esta, deveria-se programar linhas e linhas de código para compreender todo o fluxo. Quanto mais parceiros obtêm-se como empresa, mais pontos de integração terão que ser feitos. Um parceiro produz eventos para bases de dados, outro em contrapartida, produz dados em tempo real e deseja enviá-los por interface REST e receber um *callback*, outro parceiro possui um dado em um data lake próprio e não consegue enviá-los, seu projeto deve consumí-los, processá-los e repassá-los. Quanto mais soluções, mais código gerado e mão de obra terei que obter para que tudo funcione bem.

Acadêmico: Pensando como acadêmico é fácil entender o problema sobre a produção, consumo e processamento de eventos. Para realizar pesquisas acadêmicas, o pesquisador necessita de dados. Os dados são cada vez mais necessários para obter-se um bom resultado em pesquisas desse teor. Sabe-se que precisa-se levar dados de um ponto A para um ponto B de forma programática. O acadêmico não necessariamente saberá programar algo que compreenda todo o caminho do dado. Toda essa solução pretende utilizar programação e todo um sistema intrincado de sistemas e funções. Se um engenheiro eletricista pretende formar uma visualização sobre informações coletadas a partir de circuitos e sensores em projetores em determinado bloco de uma universidade que estão on/off/problematic , como ele faria para atingir este objetivo? Normalmente o engenheiro teria que pedir auxílio de programadores com experiência.

1.3 Apresentação de uma possível solução viável para o cenário

Tendo em vista resolver problemas acima citados, precisamos de um mecanismo baseado em *Low-Code* que trabalhe bem com mensageria, criando plugins necessários para publicação e consumo de eventos a níveis industriais com *Apache Kafka/Event Hubs*, de forma confiável e escalável na nuvem, com mensagens trafegando de forma segura na plataforma e principalmente mostrando como a comunidade acadêmica, não adepta à programação, pode integrar seus projetos a plataformas de consumo de dados em tempo real.

O *Low-Code*[5] consiste em uma abordagem visual para o desenvolvimento de *software* que otimiza todo o processo de desenvolvimento para acelerar a entrega. Com isso, pode-se abstrair e automatizar cada etapa do ciclo de vida do aplicativo para otimizar a implantação de uma variedade de soluções.

A ferramenta explorada neste artigo chama-se *Node-Red*[6]. Ele se utiliza de *Low-Code* e um fluxo de dados passando entre componentes (canos por onde passam dados) onde conseguimos rapidamente plugar e desplugar elementos e ações.

Com essa ferramenta em mãos, podemos resolver um grande problema atualmente representado pela mão de obra de desenvolvedores. Sendo assim, qualquer pessoa sem conhecimento profundo em programação pode construir fluxos complexos integrando sistemas *source* e *sink*. Contudo, o presente artigo não se trata de uma propaganda da ferramenta, mas sim do potencial de integração de sistemas que ela apresenta e como podemos nos aproveitar disso para criar nossos próprios caminhos de solução facilmente implementáveis com o mínimo de esforço.

1.4 Contribuições para solução do cenário

Apesar de à primeira vista ser um lugar perfeito para experimentos como estes, temos que ter em mente que a plataforma *Node-Red* é totalmente *Open-Source*. Ou seja, todo o ecossistema em volta dele é mantido pela comunidade, inclusive alguns dos *plugins* produzidos para ele.

Os *plugins* no Node-Red desempenham um papel importante para a aplicação desenvolvida na plataforma. Sem eles o sistema torna-se algo semi-inutilizado. Apesar de alguns desses *plugins* serem criados pelo *core* da organização *Node-Red*, 95% desses elementos plugáveis e desplugáveis são construídos pela comunidade. Tais *plugins* podem ser instalados na própria *runtime*.

Portanto, para alcançarmos o resultado desejado nesta pesquisa e construirmos um projeto piloto de integração entre plataformas utilizando *Node-Red*, precisamos superar outra limitação nos plugins de mensageria existentes na plataforma.

[4] https://kafka.apache.org/

[5] https://azure.microsoft.com/en-us/services/event-hubs/

[6] https://nodered.org/

Atualmente, na plataforma, existem três *plugins Kafka* (mais utilizados) para serem baixados. O primeiro chama-se *Kafka Manager*. O *Kafka Manager*[7] cumpre o que se pede. Possui produtores de dados, consumidores, um nó de conexão e várias opções de configuração. Contudo, ele peca no excesso de configurações, algo que para um usuário inexperiente, pode intimidar de início. Outro ponto negativo é a ausência de suporte à autenticação por meio de SASL(Simple Authentication and Security Layer) sha-256, suportando PLAINTEXT como forma única de se autenticar.

Os dois plugins *Kafka* baseados na biblioteca *kafkajs*[8] possuem suas devidas limitações. O primeiro deles chamado *kafka-client*[9] que por sua vez não possui suporte, mais uma vez, à autenticação via SASL. O segundo chama-se *kafkajs*[10], o mais completo e simples de todos eles, contudo, não há suporte a produção de mensagens em *batch*. Cada mensagem é enviada individualmente. E isso é um problema se esperamos utilizá-lo em produção. Por experiência, se estressado, esse plugin irá consumir bem mais recursos de memória e processamento do que deveria. De acordo com a pesquisa realizada por Stav Alfi em 2021[6], a produção de mensagens, levando em conta o *batch*, (*linger.ms* ou *size.batch*) saltou de 700/seg para 4.100/seg.

Outro recurso que utilizamos é o *Event Hubs*. Este é um homônimo do *Apache Kafka* para o ambiente *Azure*. As diferenças são poucas e irrelevantes para este artigo. Esse plugin não existe no ecossistema *Node-Red* e se torna necessária a criação de um *plugin* nesse ambiente para que possibilite o seu uso.

Além das limitações supracitadas, surgiu a necessidade de utilizar duas plataformas de mensageria diferentes. Portanto, criamos *plugins* para o *Node-Red* que atendem ambas as plataformas, de forma a suprimir as limitações dos *plugins* já existentes e criando novos *plugins*, como já citado anteriormente, para *Event Hubs*. Dessa forma, também resolvemos o problema da flexibilidade entre ambientes, onde o usuário pode querer escolher uma das alternativas, ambas as alternativas ou simplesmente deixar essa escolha com o engenheiro de integração.

2. ARQUITETURA E PROJETO DA SOLUÇÃO

Para demonstrarmos a solução em um cenário real, precisamos entender como o problema foi desenhado e pensado. Iremos expor isso em mais detalhes com imagens e diagramas, mas antes devemos entender o porquê dessa demonstração e não outra solução qualquer.

O problema se dá no consumo de energia em salas de aula diversas. Uma cena muito comum é a do professor ou funcionário saindo do recinto e esquecendo o aparelho ligado. Hoje nós temos como controlar dispositivos de forma remota graças ao projeto *Lite.me*[11] que proporciona a visualização de diversos consumos e controle destes dispositivos. O fato é que sempre que há ação direta ou indireta humana, há a possibilidade de ocorrer erros, e esses erros, na conta de energia elétrica, custam caro. Uma solução para esse caso seria a automação dessa atividade básica que é ligar e desligar um botão, seja remotamente, seja presencialmente, na classe de aula.

Contudo, as informações colhidas para a tomada de decisão não são de fácil integração. Temos informações vindas de sensores diversos e de filas *Kafka*. Pensando nesse cenário de fonte de dados, e em um cenário além onde precisa-se fazer uma integração desses dados com filas *Kafka* e *Event Hubs* proprietários, bancos de dados e APIs para diversos fins, foi que tomou-se a decisão de implementar um fluxo *Node-Red*.

2.1 Entendendo a integração

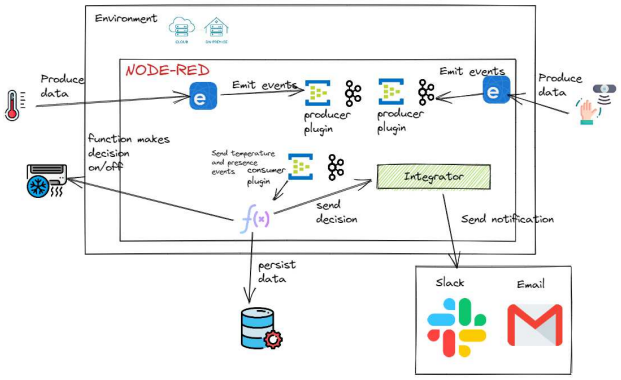


Figura 1 - Arquitetura da solução

A figura 1 revela o projeto de integração utilizado como mote deste artigo. Nela podemos ver quem são os produtores de dados. Sensores *Sonoff*[12] estão posicionados em seus respectivos lugares enviando dados para *ewelink*. *Ewelink*[13] é uma plataforma de aplicativos que podem ser encontrados na *play store* suportando várias marcas de dispositivos inteligentes, incluindo *Sonoff*.

Utilizando-se dos *plugins* produzidos para suporte *ewelink* disponíveis para *Node-Red*, nós podemos recuperar muitas informações sobre os dispositivos monitorados pelo *ewelink* (aplicativo) e fazer uso dessas informações no nosso fluxo. Temos em uso, então, sensores de temperatura e sensores de presença nas salas de testes.

Essas informações são injetadas em filas *Kafka* ou *EventHubs* a depender da escolha do usuário do fluxo. Essa escolha pode ser feita, se é importante para ele, no painel que será mostrado posteriormente.

É importante manter essas filas proprietárias internas, pois a partir delas podemos rotear as mensagens dentro de nossa aplicação de forma fácil, bem como utilizá-las além da nossa integração, implementando micro serviços orientados a eventos.

Fazendo uso de *plugins*, também na integração, sendo estes responsáveis por se subscreverem nas filas específicas de produção de eventos e com isso possibilitando a produção e

[7] https://flows.nodered.org/node/node-red-contrib-kafka-manager

[8] https://kafka.js.org/

[9] https://flows.nodered.org/node/node-red-contrib-kafka-client

[10] https://flows.nodered.org/node/node-red-contrib-kafkajs

[11] https://liteme.com.br/dashboards

[12] https://sonoff.tech/

[13] https://ewelink.cc/

consumo de eventos dentro de nossa integração. Contudo, com o poder desses protocolos, nada nos impede de produzir em uma aplicação e consumir em outra.

2.2 Criação do plugin de *streaming* de dados

Figura 2 - Nós do *Plugin* implementado para a solução

O *plugin* criado para esse projeto (mostrado na figura 2) chama-se ***node-red-contrib-streamplugin***. O prefixo "***node-red-contrib***" é uma convenção, e as demais sentenças precisam fazer sentido para a funcionalidade do *plugin*. No nosso caso, *kafka-eventhub* faria sentido para um nome, tendo em vista que ainda trabalhamos com apenas duas plataformas de *streaming* de dados. Contudo, como detalharemos posteriormente, não podemos depender de nomes específicos das plataformas visando o caráter generalista do *plugin* em questão.

Os nós são criados em TypeScript. A *runtime* do *Node-Red* é o *NodeJs*[14]. Portanto, plugins precisam, necessariamente, ser escritos em *TypeScript* ou *JavaScript*. Um *plugin Node-Red* é uma aplicação *NodeJs* comum com um *package.json* indicando as importações, nomes dos nós do *plugin* e as localizações dos arquivos .js a serem processados pelo *Node-Red*. Também possui um */src* onde geralmente ficam arquivos de código utilizando métodos específicos do *Node-Red* para identificar o que deve ser feito em cada estágio do ciclo de vida de um nó, bem como a possibilidade de uso de bibliotecas de terceiros. A utilização de arquivos HTML(*HyperText Markup Languag*e) também é imprescindível para o *plugin*, tendo em vista que essa é a única forma do usuário preencher valores importantes para este, como caracteres de conexão de banco, identificadores, etc. São esses valores que serão usados nos arquivos de código na execução de comandos durante o ciclo de vida do nó do *plugin*.

2.3 Visão geral do fluxo da solução

Ainda neste artigo iremos detalhar cada um dos passos para alcançarmos nosso objetivo dentro do fluxo. Nesta seção teremos uma visão superficial de como o fluxo foi pensado para exemplificar e utilizar um problema do mundo real.

Dentro do fluxo possuímos um módulo de tomada de decisão. O módulo possui as informações de temperatura do ambiente onde está o dispositivo e acesso a informações de movimento no recinto do dispositivo. A partir de então, o módulo consegue inserir essas informações em um banco de dados *SQL Server*, que servirá tanto para *auditing*, quanto para processamento do próprio módulo.

Resumindo a função do módulo atuador: recebe informações em tempo real de temperatura e movimento, insere no banco de dados ao mesmo tempo que pega nesse mesmo banco de dados as informações dos últimos X minutos sobre temperatura ambiente e movimento, tomando decisões de desligar ou ligar o ar condicionado.

O módulo lógico também é responsável por ativar uma área de integração de alertas. Esses alertas são enviados tanto para o *slack* [15], caso um grupo esteja monitorando por lá, quanto para um *email* ou vários *emails*. Esses alertas avisam sobre o desligamento ou ativação do ar condicionado, com o nome oficial do dispositivo no corpo da mensagem.

Devemos também entender como funciona nossa *runtime* e como se utilizar do máximo que o *Node-Red* pode proporcionar para permitir a existência de um sistema orientado a eventos tolerante a falha.

2.4 Forma de implantação

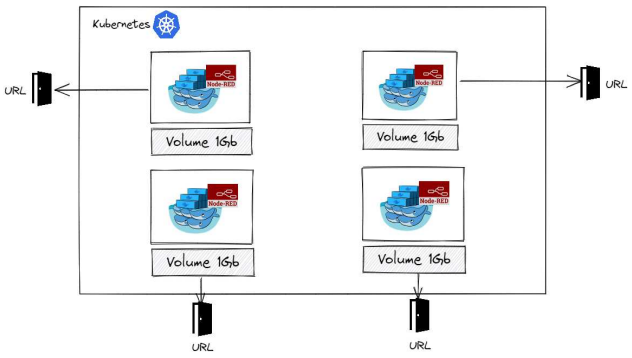


Figura 3 - Implantação do integrador

Como mostrado na figura 3, o *Node-Red* pode ser encontrado como *container docker*. Para facilitar, pode-se utilizar o *Helm*[16] para a implantação do serviço. Lá podemos configurar o volume que virá conjuntamente a *runtime* para que esta, quando desativada, consiga retomar o contexto de seus fluxos e operações.

Permitindo a utilização de uma integração *Node-Red* por *pod*, temos a confiança de que se uma integração cair, não irá afetar outra. Além disso, tendo um *ip* externo, conseguimos acessar o editor de fora do *cluster Kubernetes*, permitindo a edição do fluxo de qualquer lugar da *internet*.

[14] https://nodejs.org/

[15] https://slack.com/

[16] https://helm.sh/

3. SISTEMA EM USO EM CENÁRIO REAL

Trataremos o funcionamento do fluxo de dados principal para resolução da lacuna apresentada anteriormente, apresentando detalhes de implementação e organização e publicação em ambiente *Open-Source*.

3.1 Fluxo principal

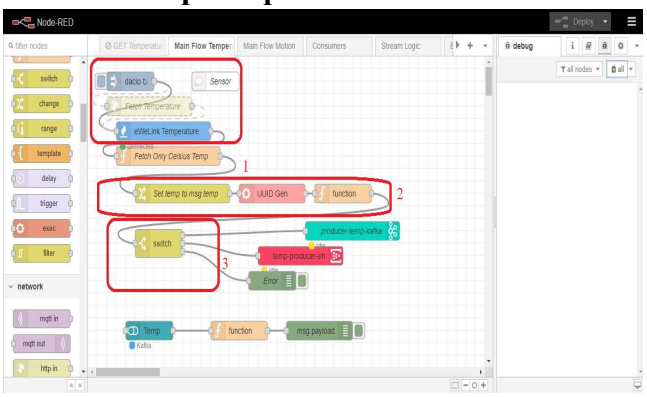


Figura 4 - Fluxo do sensor de temperatura

No *subflow* (aba do *Node-Red*) mostrado na figura 4 vemos a primeira fonte de dados em ação. (1)Um nó *trigger* recorrente (de um em um minuto) se conecta ao *ewelink* via nó do *Node-Red* (nó azul) que por sua vez se conecta ao aplicativo do *ewelink* para recolher informações de temperatura de um dispositivo do auditório do LSD(Laboratório de Sistemas Distribuídos). (2)Há lá um processamento da mensagem para adicionar um *producer token*, id da mensagem e todo o metadado necessário e que uma mensagem deveria ter ao ser enviada. (3)Após isso, temos um *switch* indicando qual das duas plataformas foi escolhida para acolher o dado. Tendo em vista que até o momento nosso *plugin* de *streaming* apenas possui suporte para *Apache Kafka* e *Event Hubs*, este *switch* possui apenas duas saídas. Nesse caso, vemos nosso *plugin* em ação pela primeira vez.

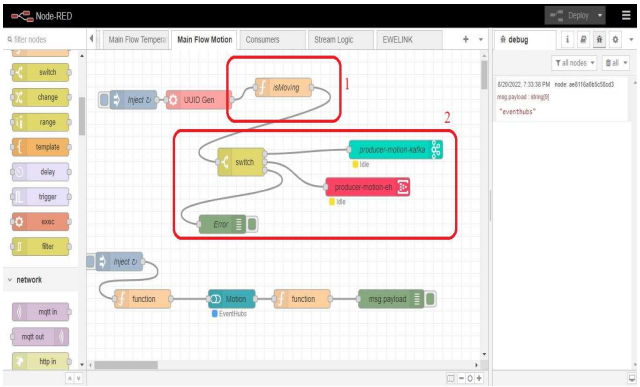


Figura 5 - Fluxo do sensor de movimento

Na segunda aba, apresentada na figura 5, (1)recebemos o dado por fila *Kafka*, (2)processamos a mensagem, colocamos metadados nela e por fim despachamos para um dos dois produtores, seguindo a mesma ideia do *subflow* anterior.

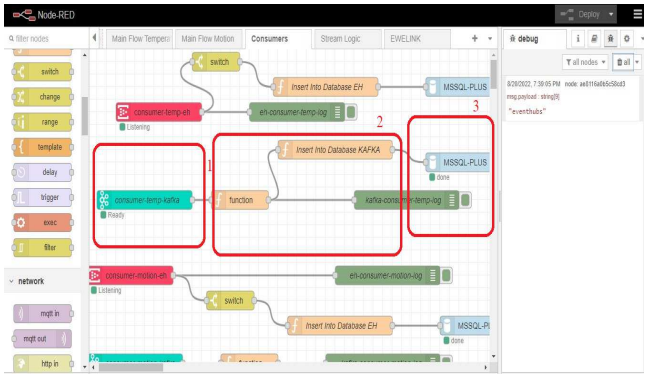


Figura 6 - Fluxo de consumidores

No terceiro *subflow*, representado pela figura 6, nós possuímos vários consumidores que irão (1)consumir essas mensagens de forma assíncrona, (2)fazer a conversão dessas mensagens em linhas de tabelas e por fim (3)inserimos na tabela destino para fins de auditoria. Será a partir dessa inserção que poderemos tomar decisões a partir de informações como *timestamp* da mensagem, temperatura do sensor, movimento recolhido pelo sensor de movimento, etc. Nesse exemplo, o banco de dados estava em outro lugar, na nuvem da *Azure*, custando ao mês apenas R$13,50, na cotação atual.

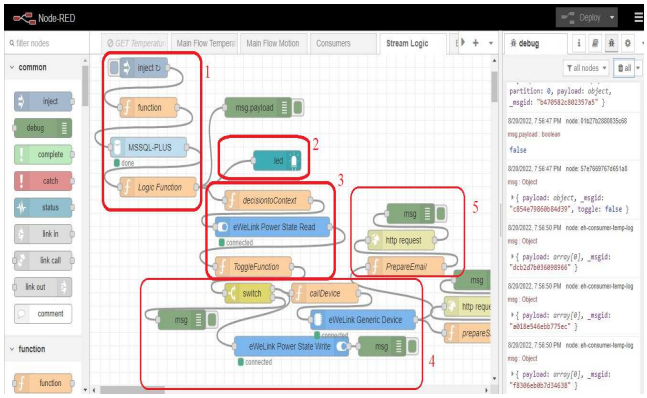


Figura 7 - Fluxo de lógica de decisão

No subflow de nome *StreamLogic* temos, de fato, o módulo lógico de tomada de decisão. (1) A cada *x* minutos, perguntamos ao banco de dados se nos últimos *y* minutos houve algum movimento no recinto do ar condicionado e se a temperatura atual está abaixo do estipulado. Se houver movimento, a temperatura está abaixo e o último estado do ar condicionado é "*off*", então liga-se o dispositivo. O caminho de volta é bem simples de entender, basta fazer o caminho inverso na lógica apresentada junto com seus parâmetros.

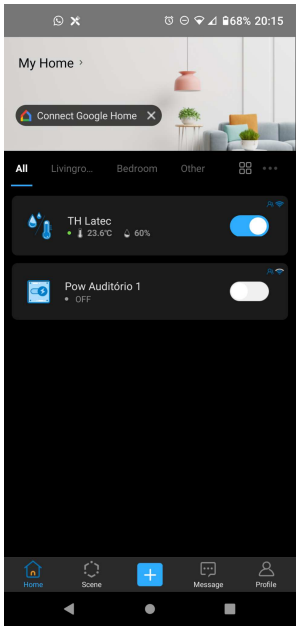


Figura 8 - Dispositivo Ar Condicionado desligado no aplicativo *ewelink*

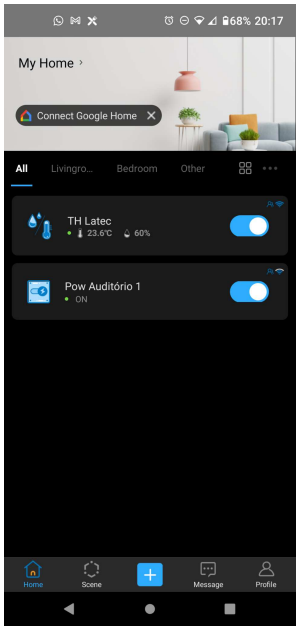


Figura 9 - Dispositivo Ar Condicionado ligado no aplicativo *ewelink*

Com essa decisão, ligamos um LED(*Light-Emitting Diode*) (2) no *dashboard*, (3) lemos o estado atual do dispositivo, se a decisão de ligar/desligar for diferente do estado atual, então (4)modificamos o estado do ar condicionado, mostrado na tela do *ewelink* nas imagens 8 e 9, e ativamos o integrador de emails e slack (5). As imagens 10 e 11, por sua vez, mostram a integração de alertas em funcionamento utilizando-se de APIs para envio de alertas/mensagens.

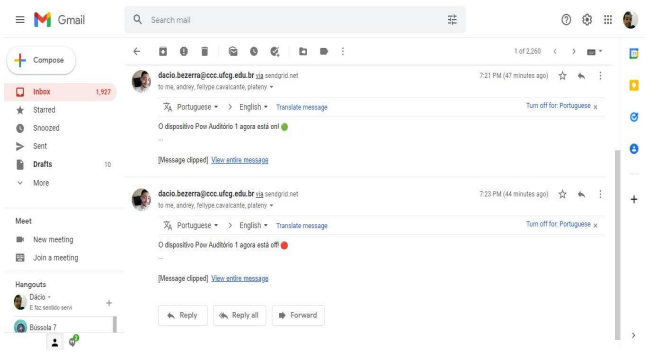


Figura 10 - Alerta por *email* (on/off)

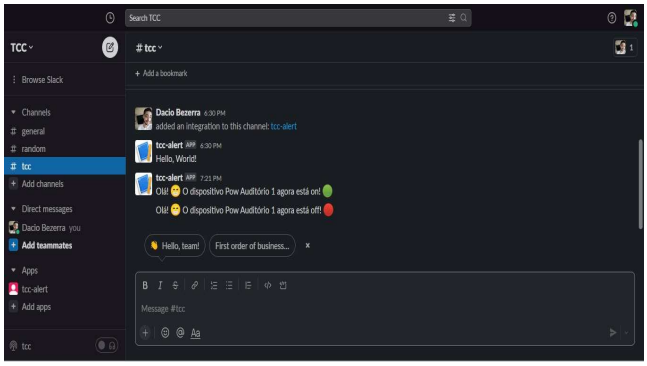


Figura 11 - Alerta por *slack* (on/off)

3.2 UI para usuário *Node-Red*

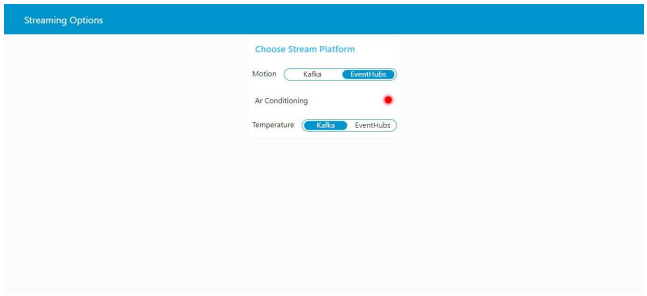


Figura 12 - UI para usuário do fluxo (ar condicionado desligado)

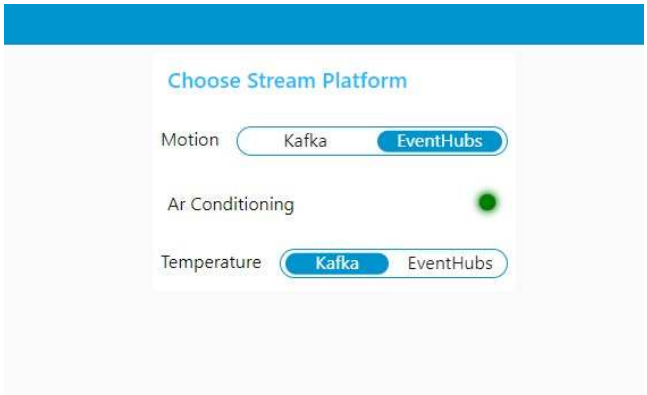


Figura 13 - UI para usuário do fluxo (ar condicionado ligado)

Durante o fluxo possuímos nós de UI(*User Interface*). O *dashboard*/UI pode ser acessado via *url-do-flow/ui*. A UI é configurada para permitir ao usuário a tomada de decisão sobre o fluxo ou a visualização de dados. No nosso caso, a UI está servindo para ambos os propósitos, como pode-se perceber a partir das imagens 12 e 13. Nessa tela podemos verificar o estado do ar condicionado por meio de um componente LED. Nela também podemos escolher para qual plataforma nossos dados de movimento serão enviados/recebidos e também podemos escolher, de forma intercambiável, para qual plataforma nossos dados de temperatura serão enviados/recebidos. Isso só é possível graças à flexibilidade do nosso *plugin* criado para esse cenário e aplicável em tantos outros.

3.3 Contribuição para a comunidade

Esse fluxo mostrado nesse projeto pode ser encontrado em formato JSON(*JavaScript Object Notation*) e importado no *Node-Red* facilmente usando o JSON na página principal do projeto no *github*.

Sabendo que esse *plugin* também é uma contribuição para a comunidade *Node-Red* e *npm*, podemos encontrá-lo na página oficial do *Node-Red*[17] apontada pela figura 15, ou como *plugin* importável na própria *runtime*, como podemos ver na figura 14.

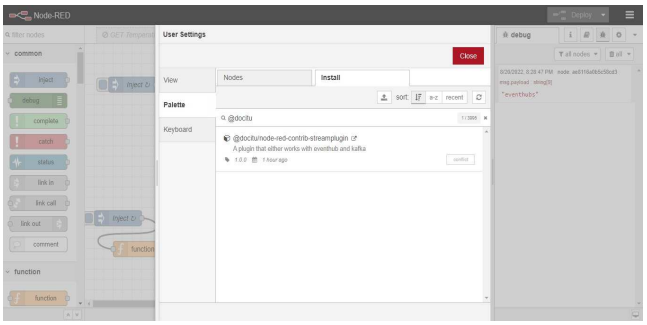
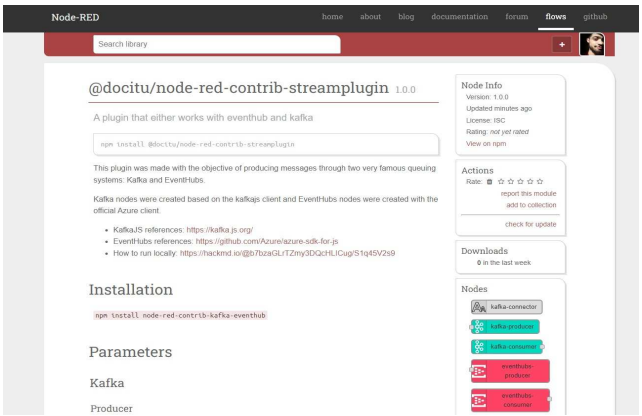


Figura 14 - Aba *Palette* com a opção para instalar nosso *plugin*



17 https://flows.nodered.org/node/@docitu/node-red-contrib-streamplugin

Figura 15 - Página do *Node-Red* disponibilizando o *plugin* criado

4. EXPERIÊNCIA E LIÇÕES APRENDIDAS

O processo de desenvolvimento se deu tanto em máquinas locais do autor desse artigo como em máquinas na nuvem e em testes no LSD (Laboratório de Sistemas Distribuídos) na UFCG.

Os principais desafios se concentraram no próprio processo de desenvolvimento de algo desconhecido (*plugin*) e na experimentação em ambiente controlado.

4.1 Processo de desenvolvimento

O desenvolvimento se dividiu em duas partes: criação do fluxo e desenvolvimento do plugin. A criação do fluxo é a parte moderada do processo. Os nós são colocados no *grid* na forma "puxa e solta" e não precisa ser um *expert* em integração em *Node-Red* para entender como funciona. Dentro de poucas horas consegue-se construir uma integração completa. Um bom exemplo foi a criação de mais uma opção de alerta por *e-mail*, adicionamos mais uma "perninha" no final do fluxo, configuramos a requisição e já possuíamos mais uma forma de alerta em mãos.

A mesma facilidade não podemos encontrar na segunda parte do desenvolvimento referente ao desenvolvimento do *plugin*. Produzimos o código principal durante dois meses e mais alguns dias para a adição da possibilidade de inserção de certificado *ca.crt* para autenticação em *clusters Kafka* que requerem esse tipo de autenticação.

Antes da criação do *plugin* em si, precisa-se entender sobre o ciclo de vida de um nó no Node-Red e como funciona a sintaxe para a produção de um plugin. Apesar de haver uma documentação rica no site deles[7], dificulta-se um pouco o processo ao introduzir o *TypeScript* em um projeto onde a maioria das pessoas usa apenas o *JavaScript*. Apesar de poucos exemplos com o transpilador mais famoso do *JavaScript* e uma curva de aprendizado maior que o comum, foi prazeroso construir o produto final em *TypeScript*, devido à robustez da linguagem ao nos ajudar a entender erros em tempo de compilação.

A criação de cinco nós para o plugin (conector *Kafka*, produtor *Kafka*, consumidor *Kafka*, produtor *Event Hubs*, consumidor *Event Hubs*) significa criar dez arquivos que o *Node-Red* irá interpretar, ou seja, um arquivo *JavaScript* e um arquivo HTML para cada nó.

O primeiro arquivo a ser produzido foi o arquivo HTML de opções de conexão do *plugin Kafka* que se refere ao que o usuário irá ver ao clicar no nó. Lá, ele pode inserir valores de conexões, dar nome ao nó, marcar ou desmarcar opções, etc. Precisamos buscar alguns exemplos da estrutura de um arquivo similar na *internet* para utilizarmos em nosso projeto.

A criação da conexão é bem simples. Pegamos as opções escolhidas pelo usuário no HTML, criamos a conexão no arquivo *kafka-connector.ts* e passamos essas opções de configuração dadas pelo usuário como parâmetros da conexão.

Vale ressaltar que o plugin de conexão não aparece listado como um nó pois este é um nó *embedded*. Ou seja, outros nós (como o de produtor) irão utilizar o retorno desse nó que foi

configurado, contudo, ele não aparece na listagem de nós. Com isso, não é preciso configurar uma conexão para cada produtor e consumidor. Simplesmente indica-se a conexão representada por este nó *embedded* e o nó de produtor/consumidor está pronto.

Algo desafiante no desenvolvimento foi descobrir como impedir a exportação de credenciais junto do *.json* que representa o fluxo. A exportação do fluxo inteiro pode se dar com alguns cliques de botão e o resultado é um JSON que por sua vez pode ser importado. Esse JSON vem com todas as propriedades e conexões, inclusive senhas e credenciais utilizadas em nós de conexão, que por sua vez são indesejadas para compartilhamento. Descobrimos que o *Node-Red* possui uma alternativa para esse problema chamado *credentials*. Quando alguém coloca suas credenciais no HTML e esse campo de preenchimento estiver marcado como credentials, ele não é exportado junto como JSON do fluxo.

A criação dos nós EventHubs foi um pouco mais simples devido à limitação de opções do próprio SDK (*Software Development Kit*) da *Azure* para *Event Hubs*. O único desafio foi entender como o *Event Hubs* pode salvar seus *offsets* para possíveis reprocessamentos. O *offset* no mundo *Kafka* se refere a um apontador como índice de uma lista. A partir desse índice o *Kafka* tem noção de onde deverá consumir, permitindo o *reset* deste *offset* ou o reprocessamento de certas mensagens a partir de determinado *offset*/índice. Hoje o *Kafka* possui o *Apache Zookeeper*[18] para manter o histórico desses *offsets*, contudo, o *Event Hubs* não possui essa funcionalidade de forma nativa. Portanto, seguindo as boas práticas indicadas pela própria *Azure*, criamos uma opção de gerenciamento de *offsets* a partir do *BlobContainer* (*storage* não estruturado da *Azure*) para escrita e consulta de histórico de *offsets*.

4.2 Limitações e desafios

Uma limitação que se conecta diretamente com os nós do *Event Hubs* se refere à conexão do *plugin*. Como os nós EventHubs foram criados após os nós Kafka, o tempo curto não permitiu a criação de um nó *embeddable* para o EventHubs, ou seja, para cada produtor e consumidor *Event Hubs* ainda é preciso passar *connection strings* para a conexão.

Outra situação limítrofe tem relação com *mocks*. Até a última semana da produção desse projeto, estávamos utilizando valores *fake* para ingestão e produção de dados. Ou seja, para temperatura estávamos utilizando uma API de temperatura, para o movimento estávamos utilizando uma função "burra" e para o dispositivo que deveria ser ligado ou desligado utilizamos o nó de LED para mostrarmos na UI do *Node-Red*. Essa limitação impôs a insegurança de que a solução poderia funcionar em um cenário real conectando-se a dispositivos reais por meio do *ewelink*. Contudo, em três dias essas conexões foram realizadas e as integrações se deram de forma descomplicada.

Outra situação se refere a mudanças de prioridade. Nosso objetivo, a *priori*, seria além do fluxo e *plugin*, construir um módulo de segurança apartado da *runtime* onde poderíamos pedir para esse módulo criptografar o tráfego e apenas ele saberia descriptografá-lo. Contudo, a opção foi vista como opcional e depois descartada.

Por fim, a proposta inicial foi mostrar o potencial da solução para o *Lite.me campus* inteligente. A integração possibilitaria soluções tangíveis para resolver muitos dos problemas de integração de dados que a plataforma possui. Contudo, o encontro com os componentes para apresentar a ideia demorou a acontecer e isso impossibilitou uma interação mais a fundo com o produto deles. Mesmo assim, conseguimos ter acesso suficiente para realizar este projeto piloto.

5. TRABALHOS FUTUROS

Algumas ideias de trabalhos futuros foram exploradas, listando-se a seguir as principais delas:

- Passar mais tempo entendendo as necessidades do *Lite.Me* e apresentar algo útil para eles tendo em vista suas necessidades e usando a solução desse projeto como piloto para demais integrações via *Low-Code*;
- Produção de mais nós para o *plugin* abrangendo outras plataformas como MQTT(*MQ Telemetry Transport*), *RabitMQ*, IBM MQ, *Azure Event Grid*, *Amazon SQS*, etc. Dessa forma, os usuários do *Node-Red*, quando pensarem em *streaming* e as opções que podem ter, escolherão o *plugin* produzido por nós;
- O fluxo pode ser ainda mais complexo. Podemos introduzir um canal de eficiência energética no fluxo e utilizar essa informação como entrada para nossa função de tomada de decisão. Contudo, esse tópico relaciona-se diretamente com com o primeiro tópico desta seção. Uma maior interação com o *Lite.Me* irá proporcionar inputs e outputs diversos.

6. REFERÊNCIAS

[1] URQUHART, J. Flow Architectures The Future of Streaming and Event-Driven Integration. 1. ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2021. p. 4.

[2] URQUHART, J. Flow Architectures The Future of Streaming and Event-Driven Integration. 1. ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2021. p. 7.

[3] URQUHART, J. Flow Architectures The Future of Streaming and Event-Driven Integration. 1. ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2021. p. 14.

[4] [n. d.]. How Much Time Do Developers Spend Actually Writing Code? Retrieved Aug 15, 2022 from https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code

[5] [n. d.]. Low-Code. Retrieved Aug 11, 2022 from https://www.ibm.com/cloud/learn/low-code

[6] [n. d.]. Producing & Consuming (Much) More With Kafka. Retrieved Aug 21, 2022 from https://medium.com/octopol-engineering/producing-consuming-much-more-with-kafka-eb0e499daec5

[7] [n. d.]. Documentation. Retrieved Aug 21, 2022 from https://nodered.org/docs/

[18] https://zookeeper.apache.org/