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

DIEGO ALVES GAMA

DESAGREGAÇÃO DISTRIBUÍDA: EVOLUÇÃO ARQUITETURAL DO DESAGREGADOR NIALM DA LITEME

CAMPINA GRANDE - PB

2022

DIEGO ALVES GAMA

DESAGREGAÇÃO DISTRIBUÍDA: EVOLUÇÃO ARQUITETURAL DO DESAGREGADOR NIALM DA LITEME

Trabalho de Conclusão de 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 : Andrey Elísio Monteiro Brito

CAMPINA GRANDE - PB

2022

DIEGO ALVES GAMA

DESAGREGAÇÃO DISTRIBUÍDA: EVOLUÇÃO ARQUITETURAL DO DESAGREGADOR NIALM DA LITEME

Trabalho de Conclusão de 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:

Andrey Elísio Monteiro Brito
Orientador – UASC/CEEI/UFCG

Lívia Maria Rodrigues Sampaio Campos
Examinadora – 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 LiteMe é uma empresa de inteligência energética em ascensão que desagrega dados de consumo de energia de seus clientes via modelo NIALM, distinguindo o consumo de cada um dos aparelhos registrados, e os processa para oferecer seus serviços. Isso faz da desagregação um dos alicerces do negócio, e conforme a LiteMe se expande, dados de mais clientes precisam ser desagregados, o que leva à necessidade de replicar o desagregador NIALM. Atualmente o desagregador faz parte de uma arquitetura monolítica fortemente acoplada com o backend robusto da empresa, chamado de núcleo. Ele realiza as operações mais custosas da plataforma, que elevam o requisito de hardware para executá-la, e atua como servidor sempre disponível. Esse acoplamento prejudica a escalabilidade do NIALM, que não pode ser replicado sozinho. Uma arquitetura distribuída de microsserviço que permita separar o NIALM do núcleo, mantendo comunicação entre os dois, pode fornecer ao desagregador melhor escalabilidade, desacoplamento, menores requisitos de hardware e de disponibilidade. Este trabalho propõe uma arquitetura de desagregação distribuída para substituir a monolítica, de forma a executar em nuvem pública com uso de instâncias oportunistas (e.g. "Spots" na AWS). A arquitetura foi validada com apoio da empresa e testada em ambiente simulado. Após análise, foi possível alcançar os objetivos e reduzir custos de hospedagem em nuvem em até 72,59% em comparação à arquitetura monolítica.

Desagregação Distribuída: Evolução Arquitetural do Desagregador NIALM da LiteMe

Diego Alves Gama
Universidade Federal de Campina Grande
Rua Aprígio Veloso, 882, Bloco CO
CEP 58.429-900, Campina Grande, PB, Brasil
diego.gama@ccc.ufcg.edu.br

Andrey Brito
Universidade Federal de Campina Grande
Rua Aprígio Veloso, 882, Bloco CO
CEP 58.429-900, Campina Grande, PB, Brasil
andrey@computacao.ufcg.edu.br

RESUMO

A LiteMe é uma empresa de inteligência energética em ascensão que desagrega dados de consumo de energia de seus clientes via modelo NIALM, distinguindo o consumo de cada um dos aparelhos registrados, e os processa para oferecer seus serviços. Isso faz da desagregação um dos alicerces do negócio, e conforme a LiteMe se expande, dados de mais clientes precisam ser desagregados, o que leva à necessidade de replicar o desagregador NIALM. Atualmente o desagregador faz parte de uma arquitetura monolítica fortemente acoplada com o *backend* robusto da empresa, chamado de núcleo. Ele realiza as operações mais custosas da plataforma, que elevam o requisito de *hardware* para executá-la, e atua como servidor sempre disponível. Esse acoplamento prejudica a escalabilidade do NIALM, que não pode ser replicado sozinho. Uma arquitetura distribuída de microsserviço que permita separar o NIALM do núcleo, mantendo comunicação entre os dois, pode fornecer ao desagregador melhor escalabilidade, desacoplamento, menores requisitos de *hardware* e de disponibilidade. Este trabalho propõe uma arquitetura de desagregação distribuída para substituir a monolítica, de forma a executar em nuvem pública com uso de instâncias oportunistas (e.g. "Spots" na AWS). A arquitetura foi validada com apoio da empresa e testada em ambiente simulado. Após análise, foi possível alcançar os objetivos e reduzir custos de hospedagem em nuvem em até 72,59% em comparação à arquitetura monolítica.

Palavras Chave

Distribuído, escalabilidade, desagregação, baixa disponibilidade, instâncias oportunistas.

1. INTRODUÇÃO

A área de inteligência energética pode ser definida como uma combinação entre inteligência artificial e eficiência energética, com o objetivo de otimizar os recursos energéticos disponíveis e minimizar custos e gastos. Nesse contexto encontra-se a LiteMe, uma *startup* brasileira que tem o diferencial de utilizar medidores inteligentes próprios para coletar dados elétricos no nível de equipamentos individuais de seus clientes de maneira não intrusiva; isto é: sem instalá-los diretamente em cada dispositivo no interior de um edifício, mas sim apenas no quadros de distribuição [1]. Seus clientes são todos aqueles que possuem um edifício ou imóvel e que desejam monitorar seus consumos, sejam eles empresas com fábricas (i.e., escopo industrial), gerentes de lojas (i.e., escopo comercial) ou donos de uma residência (i.e., escopo residencial).

Uma vez que a empresa opta por meios não intrusivos de coleta, ela sabe apenas quais dispositivos estão no edifício, mas não qual é responsável por que consumo ou em qual instante. Por esse motivo, os medidores enviam os dados para o *backend* da empresa, os quais passam primeiro por um módulo de desagregação, que se trata de uma operação de extração de dados elétricos a nível de dispositivos a partir de sinais elétricos agregados (o consumo total daquele ambiente)[2]. Existem diferentes algoritmos para desagregação, mas a empresa utiliza uma Rede Neural Convolucional que implementa o modelo NIALM [1][3].

O desagregador NIALM então recebe os sinais agregados e produz os dados a nível de dispositivos (chamados de *appliances*), que são utilizados pelos outros componentes do *backend* para análises e previsões de consumo. Essas informações são acessíveis aos clientes através da aplicação web do *frontend* da empresa, que as consulta no *backend* em um relacionamento cliente-servidor e as apresenta em gráficos e relatórios detalhados.

Pode-se ver, portanto, que a desagregação com NIALM não apenas faz parte do diferencial competitivo da empresa, como também é o alicerce de seu negócio. Por essa razão ela está sempre melhorando o treinamento e topologia da rede, buscando melhores métricas (e.g. acurácia e precisão) para melhor desagregar os sinais de energia em informações mais detalhadas de consumo.

Cada novo cliente implica em um ou mais medidores implantados em seus edifícios, que trazem consigo um aumento na carga de dados a serem desagregados. Logo, ao passo que o alcance da LiteMe aumenta, a necessidade de escalabilidade da desagregação também cresce.

Entretanto, o *backend* inteiro da LiteMe possui uma arquitetura monolítica, o que infelizmente impede o módulo de desagregação de ser replicado sozinho. O NIALM é acoplado ao *backend* para receber os dados de medição, e para acessar o banco de dados para armazenar seus resultados atuais e reaproveitar os anteriores. Isso não seria um problema se o restante do *backend*, chamado de núcleo, não fosse tão robusto, e com tantos componentes realizando tantas operações pesadas. Esses componentes são responsáveis por todo o gerenciamento de usuários, medidores, edifícios monitorados, geração de contas, análise energética e geração de relatórios. Tais operações elevam os requisitos de *hardware* do núcleo, o que prejudica o escalonamento em nuvens públicas, uma vez que a aquisição de máquinas individuais que satisfaçam esses requisitos é bastante custosa.

O escalonamento se mostra ainda mais difícil ao considerar a disponibilidade exigida pelo modelo de negócio. Os dados de medição não chegam ao NIALM em tempo real, o que o permite não estar disponível em determinados intervalos; além disso, ele é capaz de processar a carga rapidamente, e consegue assim mitigar acúmulo de carga atrasada. O núcleo, entretanto, deve manter-se

disponível, para que seu servidor sempre possa suprir as requisições do *frontend*, que deve sempre poder exibir os gráficos e relatórios gerados ao cliente. Em suma, os requisitos de ambos são diferentes: enquanto o núcleo precisa estar sempre disponível e consome mais recursos, o NIALM pode funcionar em lotes se necessário e consome apenas uma parcela deles.

Propõe-se então como solução uma nova arquitetura para a desagregação. Uma arquitetura distribuída capaz de desacoplar o NIALM do núcleo, que respeite sua necessidade de fácil manutenção e evolução, e que por fim permita tanto receber dados continuamente como se recuperar sem problemas em caso de baixa disponibilidade de máquinas.

O restante deste documento está organizado da seguinte forma: a Seção 2. Motivação, onde encontram-se detalhes da problemática que motivou a elaboração de uma nova arquitetura; a Seção 3. Metodologia, que mostra a abordagem utilizada para elaborar e desenvolver a arquitetura e os critérios para validá-la; a Seção 4. Solução com a descrição da arquitetura; a Seção 5. Resultados e Discussões que discute como a arquitetura se comporta em relação ao esperado; e finalmente, a Seção 6. Considerações Finais que conclui o documento com algumas considerações finais seguida da Seção 7. Reconhecimentos e Seção 8. Referências.

2. MOTIVAÇÃO

Nesta seção detalhamos os problemas de escalabilidade da arquitetura monolítica e seus requisitos; em seguida explicamos como a desagregação funciona atualmente; e por fim listamos requisitos que uma nova arquitetura precisa atender para solucionar os problemas.

2.1 Requisitos e Problemas de Escalabilidade

Para melhor nos aprofundarmos no problema, podemos observar o requisito de *hardware* inteiro do núcleo como existe agora, e depois separarmos o quanto o NIALM em si exige.

Atualmente a máquina virtual de produção da LiteMe oferece 16 GB de RAM, 8 vCPUs e 320 GB de armazenamento (com volumes persistentes). Essas configurações são necessárias para executar o núcleo inteiro sem problemas de desempenho. Destas configurações, entretanto, o NIALM precisa apenas de 1 vCPU, até 2 GB de RAM e nenhum armazenamento (uma vez que não utiliza nenhum banco de dados próprio).

Apesar desses requisitos serem suficientes no momento, conforme a quantidade de medidores implantados aumenta, o NIALM começará a processar uma maior quantidade de dados simultaneamente, exigindo mais processamento e mais memória. Em alguns casos, especialmente em testes de carga, a empresa já experienciou superlotação da máquina virtual, o que acarretava em problemas de travamento e indisponibilidade de todo o sistema até ser reiniciado.

No momento atual da escrita, o mundo ainda não se recuperou completamente da crise mundial de microcomponentes, causada pela COVID-19. A demanda por *hardware* cresceu, mas a oferta caiu, o que aumentou os preços. Graças à situação, o escalonamento vertical (i.e., evolução de configurações em uma única máquina) torna-se menos atrativo, ainda mais com a opção de escalonamento horizontal (i.e., maior quantidade de máquinas) oferecida por nuvens públicas.

Buscar replicar o *backend* inteiro (exceto o banco de dados, que pode ser um único servidor à parte), entretanto, implica em prover várias máquinas virtuais (chamadas de instâncias) com, ao menos, as mesmas especificações de memória e processamento. O tamanho necessário das instâncias a serem alocadas (conhecido como grão de alocação) é muito grande, e por esse mesmo motivo, o custo é alto para alocar múltiplas instâncias.

A dificuldade é intensificada com o requisito de alta disponibilidade, ou seja, a necessidade de manter as várias instâncias dedicadas. Em um plano de instâncias assim (i.e., sob demanda) o custo é mais elevado, visto que a nuvem precisa manter os recursos necessários dedicados para as instâncias enquanto forem necessárias. Se o requisito de disponibilidade for menor, não haverá necessidade de manter instâncias tão dedicadas. Nesse caso, um plano de instâncias que por vezes estão disponíveis e por vezes não (i.e., oportunistas) seria uma melhor escolha, dados os grandes descontos no custo original daquela mesma instância. Como referência, uma instância Spot (oportunista) na Amazon Web Services (AWS) Elastic Cloud Computing (EC2) pode ser até 90% mais barata que uma instância On Demand [4].

Como uma primeira estimativa de benefício, podemos comparar os ganhos utilizando as instâncias ofertadas, e seus preços atuais (até o momento de escrita), pela AWS EC2. O núcleo exige uma instância On Demand do tipo *a1.2xlarge*, de 16 GB de RAM e 8 vCPUs [5], que custa 0,204 USD por hora (ou 146,88 USD por mês), por instância [6]. A diminuição do grão de alocação permitirá a transição para o tipo *a1.medium*, de 2 GB de RAM e 1 vCPU [5], que custa 0,0255 USD por hora (ou 18,36 USD por mês) [6], por instância, que representa uma economia de 87,5%. Adicionalmente, a redução do requisito de disponibilidade permitiria o uso desse mesmo grão de alocação, porém em Spot, com o preço atual de 0,0049 USD por hora (ou 3.528 USD por mês) [7]. Isso leva a um desconto de cerca de 80,78% em relação ao uso de On Demand, e uma economia total de cerca de 97,60% em relação ao On Demand com o tipo de instância original.

Por fim, ao separar o módulo de desagregação NIALM do núcleo, é possível não só reduzir o grão de alocação como também o requisito de disponibilidade, o que permitiria o uso de instâncias oportunistas. Assim, os custos podem cair, uma vez que o NIALM poderá utilizar mais instâncias apenas temporariamente, recuperando o tempo perdido em caso de falha e processando a carga atrasada até ficar novamente ocioso.

2.2 A Desagregação Atualmente

No momento, o núcleo se comunica com o módulo de desagregação através de requisições HTTP a uma API exposta pelo módulo, com três rotas principais, como ilustrada pela Figura 1 abaixo.

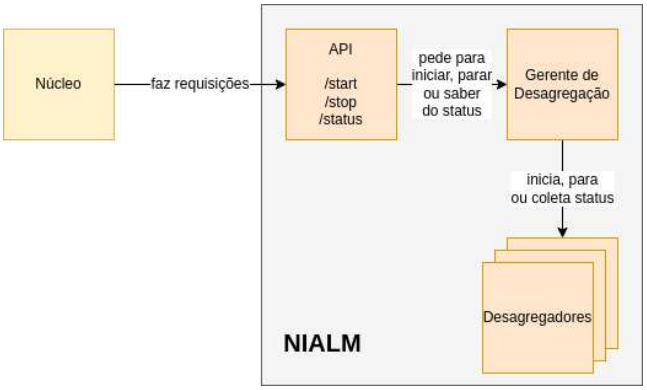

Figura 1 - Componentes da desagregação

- **/start**, que pede ao NIALM que um desagregador seja executado para um medidor específico, caso já não exista um executando.
- **/stop**, que pede ao NIALM que interrompa a desagregação para um medidor específico, caso ele esteja executando.
- **/status**, que pede ao NIALM o status de desagregação de um determinado medidor.

Uma vez que o núcleo pede ao NIALM que a desagregação de um determinado medidor comece, o gerente de desagregação do NIALM constrói um desagregador utilizando as informações do medidor e decidindo o modelo da rede neural que irá usar (e.g. rede de escopo industrial). As informações dos modelos se encontram em arquivos JSON. Esses são os arquivos alterados conforme as redes evoluem em treinamento e topografia.

O desagregador é então executado em uma *thread* dedicada, começando a desagregar a partir da última medição não desagregada. Ele inicializa um histórico vazio no formato de fila que será preenchido com dados coletados conforme executa, e servirá de entrada para a rede. Após essa inicialização, um laço de desagregação com os seguintes passos é executado, ilustrado pela Figura 2 abaixo.

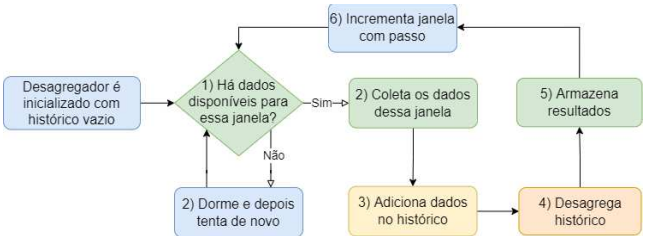

Figura 2 - Laço de desagregação

1. Checa se há dados ainda não desagregados para a janela de tempo atual. A janela é calculada com a última data usada e o passo de desagregação, que é uma quantidade de segundos que depende do modelo (geralmente de 60 segundos). Por exemplo, com uma data de 16:00 e passo de 60 segundos, a janela usada é de 16:00-16:01.
2. Se houver dados, coleta os dados contidos na janela. Se não, dorme por um tempo e tenta novamente o passo 1).
3. Adiciona os dados de medição da janela atual ao histórico. O tamanho do histórico depende das configurações do modelo da rede. Se o histórico já estiver cheio, os dados mais antigos são descartados para abrir espaço para os mais novos.
4. Executa a desagregação utilizando o histórico como entrada.
5. Armazena os resultados no banco de dados.
6. Atualiza a data usando o passo de desagregação, de forma a obter uma janela mais recente.

Esse laço é repetido indefinidamente, e é interrompido apenas caso o núcleo faça a requisição */stop*. Nesse caso, o gerente de desagregadores termina a *thread* que executa o laço de desagregação.

É graças a essa lógica de gerência de *multithreading* e ao laço de desagregação que o módulo consegue recuperar carga atrasada, uma vez que sua carga é adquirida no banco de dados.

Adicionalmente, o módulo do NIALM usa também o banco de dados do *backend* para operações de leitura e escrita. Ele utiliza três coleções distintas do banco de dados MongoDB do *backend* da LiteMe: uma para recuperar dados dos medidores (usada para configurar desagregadores), outra para recuperar dados de medição, e a última para persistir os resultados da desagregação.

Dessa forma, podemos isolar quase completamente a desagregação do *backend*, visto que da perspectiva do núcleo o que importa é que as requisições sejam respondidas conforme esperado e que os resultados estejam persistidos no banco de dados. Em outras palavras, que a desagregação não só ocorra, mas que seja executada apenas para medidores permitidos.

Nesse ponto é necessário chamar a atenção para o núcleo. O motivo do NIALM conseguir recuperar os dados dos medidores diretamente do banco de dados se dá pelo fato de que o núcleo é quem os persiste lá. Os dados enviados pelos medidores são publicados em uma fila de mensagens no Kafka, da qual o núcleo é consumidor. O núcleo consome os dados de medição e os persiste no banco de dados. Logo, o núcleo é responsável por preparar o conjunto de dados de entrada para o NIALM.

Essas estratégias funcionam bem enquanto o módulo é único e individual, porém se múltiplas réplicas forem criadas, o núcleo terá que escolher se comunicar apenas com uma. Para se comunicar com todas as réplicas ao mesmo tempo, ele precisaria difundir as requisições de */start* e */stop* entre as réplicas com a ajuda de algum *proxy*, fazendo com que haja retrabalho e duplicação de resultados de desagregação. Se o *proxy* for capaz de balancear cargas, ele ainda teria de conhecer qual réplica está desagregando dados de qual medidor, para poder responder corretamente os comandos de */stop* e */status*. Logo, é necessário que haja algum controle de trabalho e orquestração, de forma a minimizar alterações no núcleo, bem como que não haja desperdício ou duplicação de dados.

Como uma observação à parte, deixando de lado as comunicações com o núcleo, o NIALM utiliza diretórios de configuração. Nele estão arquivos que possuem as configurações para cada modelo de rede neural utilizado. Esse diretório precisa estar disponível durante o *startup* do NIALM, e consistentemente entre todas as réplicas. Isso é um problema facilmente solucionado através de volumes, mas como é integral ao funcionamento do NIALM e requer algum manejo à parte, ele precisa ser mencionado.

2.3 Requisitos da Desagregação Distribuída

Para alcançar os benefícios de executar em uma nuvem pública, é necessário que a nova arquitetura seja preparada para as incertezas e requisitos da infraestrutura.

Adicionalmente, uma vez que é desejável que a desagregação consiga ocorrer em um modelo de instâncias oportunistas, ela deve estar pronta para baixa disponibilidade; isto é, para interrupção repentina por escassez de instâncias e por falhas de infraestrutura.

Para nortear a elaboração da arquitetura, bem como validá-la, citamos abaixo os requisitos aos quais ela deve obedecer.

- **Desacoplamento**: a desagregação deve ser fracamente acoplada em relação ao núcleo, não dependendo de ser executada na mesma máquina (física ou virtual).
- **Tolerância à falhas**: ela deve ser tolerante à falhas de parada (ou travamento) e à baixa disponibilidade de máquinas, para suportar uso de instâncias oportunistas.
- **Configurabilidade**: ela deve ser facilmente customizável, para que permaneça relevante conforme a rede neural do desagregador evolua e seja de fácil manutenção.
- **Corretude**: a desagregação deve continuar correta e funcional na nova arquitetura, produzindo os mesmos resultados.

3. METODOLOGIA

Guiados pelos requisitos, projetamos a nova arquitetura. A desagregação foi adaptada e componentes auxiliares antes não existentes foram implementados, obedecendo ao projeto da solução.

3.1 Desenvolvimento

O processo de design da solução foi feito de forma iterativa, onde um diálogo contínuo com a equipe do NIALM da LiteMe foi parte essencial para validação e controle de qualidade.

Em seguida, a solução foi desenvolvida em ciclos iterativos (*sprints*) de uma semana, inspirados pela metodologia SCRUM. A cada ciclo uma das funcionalidades foi implementada, testada e validada com o orientador deste trabalho. Após todos os componentes auxiliares da solução terem sido desenvolvidos, a implementação foi avaliada novamente com a equipe do NIALM, que proveu feedback utilizado para refinar o resultado final.

3.2 Teste de Corretude

Em seguida, para avaliar a corretude da solução implementada, um subconjunto de dados equivalentes a um dia de medições foi exportado do banco de dados de produção. Um outro banco de dados foi então populado com esses dados, o que permitiu checar se as saídas produzidas pela solução estariam de acordo com as saídas produzidas pela versão original.

3.3 Teste de Tolerância a Falhas

De forma similar, foi possível testar como a solução se recupera perante falhas de infraestrutura. Foram consideradas apenas falhas de interrupção, uma vez que almejamos uma infraestrutura mais barata, porém instável, e foram as seguintes, ilustradas pela Figura 3 abaixo.

Figura 3 - Falhas na desagregação

- **Falha 1**: réplica do NIALM é interrompida antes de executar um passo de desagregação, para um dado medidor.
- **Falha 2**: réplica do NIALM é interrompida antes de finalizar um passo de desagregação para um dado medidor.
- **Falha 3**: réplica do NIALM é interrompida após finalizar um passo de desagregação, mas antes que possa entregar os resultados.

Em cada uma dessas falhas é esperado que outra réplica (ou a mesma ao inicializar novamente) seja capaz de retomar o trabalho, de maneira a garantir que não haja perda de trabalho ou retrabalho.

Para testar o comportamento da solução mediante às falhas, foram utilizadas três réplicas simultâneas, e as três falhas foram induzidas manualmente e em diferentes réplicas. A injeção de falhas manual acontecia após observar as mensagens de log que indicavam que o sistema estava naquela etapa. Para permitir essa interação, foram inseridos atrasos nos estados de modo que houvesse tempo hábil para agir. Esse teste foi repetido 10 vezes.

3.4 Avaliação de Configurabilidade e Desacoplamento

A configurabilidade da solução foi avaliada de forma simples, ao observar se a nova arquitetura permite a mesma facilidade de troca dos arquivos de configuração. Isto é, se permite a troca por modelos de rede neural mais evoluídos com facilidade.

Já o desacoplamento observando se era necessário executar o núcleo e o NIALM em uma mesma máquina, e ao mesmo tempo.

A apresentação da arquitetura encontra-se na seção Solução, e a validação dos critérios em Resultados e Discussão

4. SOLUÇÃO

A solução abaixo é apresentada primeiro com uma visão geral mostrando as decisões principais, e depois em uma visão de componentes mais detalhista. Após isso, são feitas algumas considerações de orquestração em nuvem.

4.1 Visão Geral e Decisões Principais

4.1.1 Gerente de Trabalho

Ao analisar a desagregação atualmente, pode-se ver que um dos problemas que surge com a replicação do NIALM é a duplicação de dados. Isso se dá porque o NIALM mantém estado, e esse estado determina quem está sendo desagregado.

Uma abordagem para lidar com esse problema seria usar um serviço externo que gerenciaria o estado de cada réplica, de forma que o conjunto de medidores sendo desagregados por uma réplica seja disjunto (i.e., não possua interseção) em relação aos conjuntos de medidores das outras réplicas. Isso também significaria que sempre que uma réplica fosse terminada (e.g. por falha de infraestrutura de nuvem), esse serviço precisaria decidir qual outra réplica deveria assumir a carga da anterior. Essa

abordagem faz com que esse novo serviço seja complexo e que tenha que gerir o estado de outros e balancear carga.

Ao invés disso, optamos por remover completamente o estado interno do NIALM, e com ele o laço de desagregação por *thread* usado. Essa responsabilidade foi extraída para um serviço externo, ao invés de parcialmente. Esse serviço pode gerar as entradas necessárias para o NIALM, fazendo ele mesmo o laço de desagregação, com uma diferença. Ao invés de desagregar, ele prepara a entrada e hiperparâmetros que o NIALM precisa, com exceção do histórico de dados (para não roubar responsabilidades). Esses hiperparâmetros são pegos a partir dos arquivos de configuração JSON do NIALM. Em outras palavras, ao invés de efetuar o trabalho de desagregação, ele gera trabalhos individuais que réplicas podem exercer. Por isso chamaremos esse componente de *gerente de trabalho*.

Com essa abordagem, as réplicas trabalhadoras apenas executam um trabalho. Elas recebem a entrada, recuperam os dados a partir do MongoDB para preencher o histórico, efetuam a desagregação e persistem o resultado. Ao terminá-lo buscam outro, sem precisar registrar quais medidores estavam desagregando, uma vez que essa informação é passada no trabalho. Dessa forma as réplicas podem desagregar qualquer medidor.

Entretanto, essa abordagem apresenta algumas perguntas sobre acoplamento: Com quem o núcleo interage? Como o gerente de trabalho se comunicará com as réplicas? Como deixar essa comunicação assíncrona, já que as réplicas podem ser terminadas por interrupções na infraestrutura?

4.1.1.1 Com quem o núcleo interage?

No que diz respeito ao núcleo, isso seria facilmente solucionado ao extrair não apenas o estado do NIALM, mas também sua API, uma vez que ela serve para gerenciar o estado interno. Porém isso não é sequer necessário, e é um dos exageros de responsabilidade do núcleo.

O núcleo utiliza atualmente essa API apenas para que consiga garantir que o NIALM siga desagregando quem deve, mas para saber quem deve ou não, ele armazena essa informação no MongoDB. O exagero de responsabilidade se dá no fato de que o núcleo também está se preocupando com desagregação, algo que deveria ser responsabilidade apenas do NIALM.

Já que a desagregação com NIALM já interage com o MongoDB, basta que o gerente de trabalho busque periodicamente essa informação no MongoDB, e ele mesmo a utilize para terminar ou retomar a desagregação para um medidor

4.1.1.2 Como o gerente de trabalho se comunicará com as réplicas?

Para não causar novamente o problema que estamos evitando, que é acoplamento forte, é preciso desacoplar no espaço - de forma que as réplicas trabalhadoras do NIALM executem em instâncias distintas - e no tempo - para permitir assincronia dada a baixa disponibilidade que almejamos. Isso pode ser conseguido com o uso de fila de mensagens.

Com uso de fila de mensagens, o gerente de trabalho pode apenas produzir itens de trabalho e os inserir na fila, e as réplicas podem consumir esses itens de trabalho quando estiverem disponíveis. Assim, tanto o gerente produtor quanto as réplicas consumidoras só precisam se comunicar com a fila de trabalho (desacoplado no espaço), e na própria conveniência (desacoplado no tempo).

4.1.1.3 Como deixar essa comunicação assíncrona, já que as réplicas podem ser terminadas por interrupções na infraestrutura?

A fila de trabalho ainda não resolve todos os casos. Caso uma réplica consuma um trabalho e seja terminada na metade, o que acontece (falha número 2)? Neste momento a garantia de entrega é importante. A réplica só confirma que consumiu o item quando o trabalho é terminado e o resultado é persistido no banco de dados. Dessa forma, caso ela demore demais para responder, a fila pode disparar um timeout e permitir que outra réplica consuma esse item. Assim, garante-se que todos os trabalhos serão terminados em algum momento.

Na rara ocasião onde o trabalho é terminado e resultados são persistidos no MongoDB, mas a réplica é interrompida logo antes de confirmar o consumo do item, ou se demorou demais para confirmar disparando timeout (falha número 3), ocorrerá retrabalho e duplicação de dados quando outra réplica o consumir. Aqui ainda há uma última contingência. Como parte de um trabalho, o NIALM já consulta o banco de dados algumas vezes. Uma consulta simples pode ser adicionada para checar se o intervalo a ser desagregado já está na coleção de resultados. Se os resultados já constarem, basta que pule a desagregação e confirme o consumo sem retrabalho.

4.1.2 Auxiliar de Agregação

Poderíamos parar por aqui, pois já permitimos aqui que a desagregação escale sem duplicação de resultados e retrabalho, além de que funcione com baixa disponibilidade e tolerância a falhas. Entretanto, como discutido anteriormente, conforme mais medidores são implantados, mais dados são produzidos, e o núcleo é ainda o responsável por pré-processá-los antes de persisti-los no MongoDB. Para permitir que a desagregação inteira e tudo que esteja ligado ao crescimento de medidores escale, é necessário remover isso também do núcleo.

O módulo de pré-processamento pode ser reutilizado, uma vez que já está testado pela LiteMe, e apenas extraído para um serviço auxiliar. Chamaremos esse serviço de auxiliar de agregação. O que o auxiliar de agregação faz resume-se a consumir medições de energia do Kafka, pré-processar, e persistir os dados processados no MongoDB. Como o Kafka é um serviço de tópicos de mensagens para modelo publish-subscribe, ele distribui carga entre consumidores de um mesmo grupo, e já poupa esforços de lidar com retrabalho. Com isso, o módulo já é também escalável, e completamente desacoplado do núcleo, das réplicas e do gerente de trabalho.

4.1.3 Auxiliar de Agregação

Ao extrair a desagregação para a nuvem, surge um problema com a comunicação com o banco de dados: a latência. O MongoDB é acessado múltiplas vezes na versão original do NIALM, e na nova arquitetura isso não muda. Na realidade, a quantidade de acessos aumenta. Na versão original, o NIALM o acessa apenas para buscar os dados a serem desagregados, e mantém o histórico em memória ao passo que executa. Na nova arquitetura, não só o gerente efetua esses mesmos passos, como o histórico não é mais mantido, e sim preenchido pelas réplicas para cada item.

A ausência de um MongoDB local faz com que esse aumento de acessos seja problemático, pois a latência causada pela comunicação entre redes aumenta bastante o tempo de processamento de um único item. Apesar da desagregação não ser em tempo real, quanto mais tempo um item demorar a ser processado, maior o desperdício de recursos em instâncias oportunistas.

Para mitigar isso, optamos por adicionar um MongoDB local, independente do original mantido pelo LiteMe. Esse MongoDB contém apenas as informações necessárias para a desagregação, e é acessado apenas pelos serviços da nuvem, enquanto o original é acessado pelo núcleo. A única exceção para a comunicação estritamente local é que o gerente de trabalho precisa checar quais medidores devem ser desagregados. Esta informação precisa ser recuperada no banco original, pois é mantida pelo núcleo. Entretanto, como estas mudanças de configuração nos medidores não são frequentes, não há um impacto no desempenho da desagregação.

Para garantir que o MongoDB local possui os dados, basta que haja duas réplicas de auxiliar de agregação: uma que popula o MongoDB original na infraestrutura da LiteMe, e outra que popula o MongoDB local na nuvem. Dessa forma, reduzimos a latência onde há mais consultas ao banco de dados.

4.1.4 Fila de Mensagens usando Kafka

Para a fila de mensagens mencionada, apesar de boas opções como RabbitMQ [8] que oferecem as funcionalidades desejadas, a LiteMe já utiliza o Kafka com modelo de publisher-subscribe. Podemos reutilizar o Kafka para eliminar a dependência com um componente a mais, e em especial, reduzir os custos de operação para replicação de fila, visto que o Kafka já funciona de forma distribuída por padrão utilizando partições de tópicos e consumer groups [9].

Por fim, a Figura 4 abaixo ilustra a nova arquitetura apresentada.

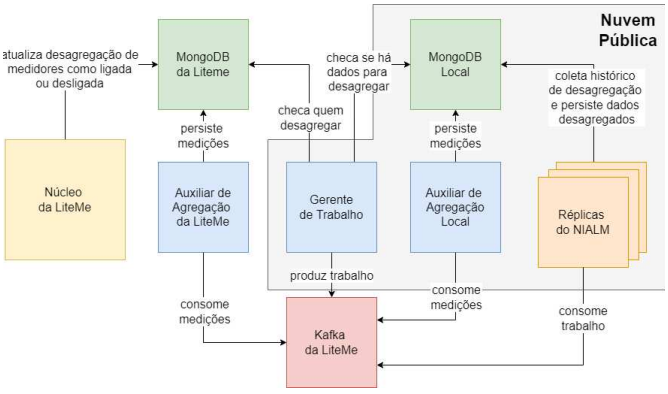

Figura 4 - Diagrama de contexto

É importante ressaltar que para que as réplicas possam operar com baixa disponibilidade, o gerente de trabalho e o auxiliar de agregação precisam funcionar com alta disponibilidade.

Se o tempo de indisponibilidade do auxiliar de agregação ultrapassar o período de retenção de dados do Kafka, alguns dados de medição serão perdidos. Os dados de medição são úteis para o LiteMe mesmo antes de serem desagregados, e como o núcleo requer alta disponibilidade, esses dados também precisam estar disponíveis.

Já se o gerente de trabalho ficar indisponível e nesse período a desagregação para um medidor for habilitada e depois desabilitada novamente, a janela de dados que deveria ter sido desagregada será desconsiderada. Além disso, se as réplicas do NIALM estiverem disponíveis e o gerente não, não haverá trabalho para elas, e desperdiçarão recursos.

4.2 Visão de Componentes

A arquitetura não é intercambiável, há algumas mudanças que são necessárias em código existente. O núcleo precisa desconsiderar comunicação com NIALM via API, e precisa parar de consumir dados do Kafka. As réplicas do NIALM não podem mais iniciar seus laços de desagregação e precisam, ao invés disso, receber um consumidor Kafka que invocará a desagregação. Apesar disso, essas alterações de código são simples. Os novos serviços, entretanto, são complexos de visualizar, e portanto é necessário compreendê-los mais de perto. A Figura 5 abaixo ilustra o diagrama de componentes da arquitetura.

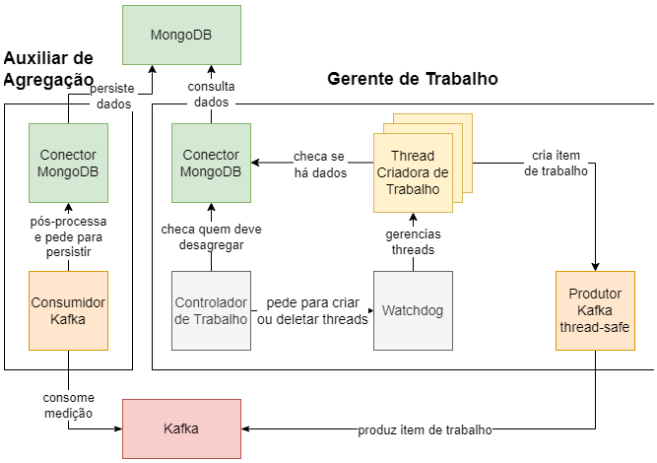

Figura 5 - Diagrama de componentes

O auxiliar de agregação é simples, e possui apenas o conector MongoDB e o consumidor Kafka, que aplicará o pré-processamento para cada medição.

O gerente de trabalho, por sua vez, possui mais componentes internos. Além do conector MongoDB ele possui um controlador de trabalho responsável por gerenciar quem deve desagregar ou não e um *watchdog* que aplicará as decisões do controlador, bem como é responsável por checar a saúde das *threads*. *Threads* essas que são análogas às do NIALM, como dito anteriormente, e que criam trabalhos e os publicam na fila do Kafka através de um produtor *thread-safe*

Para melhor ilustrar as diferenças entre as arquiteturas, a Figura 5 compara a versão original acoplada ao núcleo monolítico, com a solução.

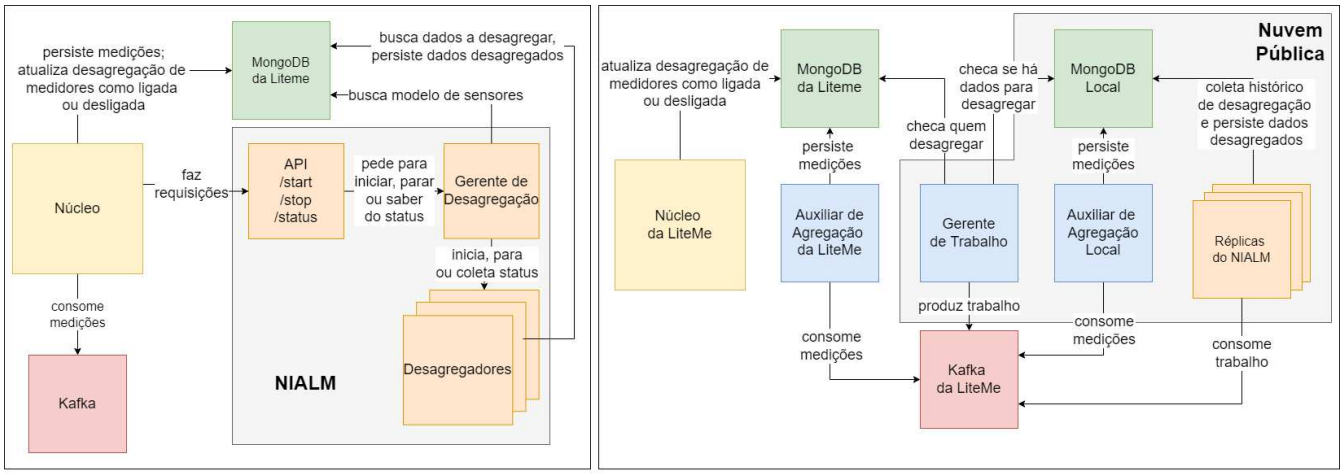

Figura 5 - NIALM Original e NIALM Distribuído

4.3 Preocupações de Orquestração

Para aplicar a nova arquitetura de forma escalável e prática, é interessante o uso de um orquestrador para monitorar e gerenciar os microsserviços que precisam de alta disponibilidade. Utilizamos para isso Kubernetes [10], que é bem consolidado no mercado [11] e possui uma linguagem declarativa útil principalmente para manter um estado ideal do sistema, que é o que queremos.

4.3.1 Arquivos de Configuração

Como dito anteriormente, o gerente de trabalho requer os mesmos arquivos de modelos que o NIALM. Caso esses serviços não fossem conteinerizados, isso poderia oferecer um problema. Mas tanto com o uso de Docker, como com o uso de Kubernetes, isso é facilmente resolvível com um volume compartilhado, onde ambos podem acessar os dados. Como não pode haver discrepância entre os arquivos, para configurar os modelos da rede, basta alterar os arquivos JSON no volume e reiniciar as aplicações.

4.3.2 Saúde dos Microsserviços

Além do *deploy* e configurações necessárias, o Kubernetes permite o uso de *probes* que fiscalizam as réplicas dos microsserviços (chamadas de *pods*) em busca de uma dada métrica que o permite decidir se a situação está ideal, e senão como agir perante isso. Os *probes* são:

- *Liveness probe*: que avalia se o *pod* está saudável o suficiente para continuar executando. Se não, o *pod* é terminado e reiniciado [12]. Utilizando esse *probe*, uma tolerância de falhas antes do *restart* pode ser configurada.
- *Readiness probe*: que avalia se o *pod* está pronto o suficiente para receber tráfego de rede. Se não, o *pod* é isolado de quaisquer camadas de Serviço, que são responsáveis por rotear tráfego para o *pod* [12]. Isso não é útil no nosso caso, pois nenhum de nossos *pods* expõe portas ou recebem tráfego por elas. Eles se comunicam apenas via Kafka e MongoDB.

Com essas features em mente, as seguintes métricas de saúde foram decididas para cada um dos dois, conforme descreve a Tabela 1 abaixo.

Microsserviço	***Liveness probe***
Gerente de trabalho	Checa se a *thread* principal está executando.
Auxiliar de agregação	Checa se a *thread* consumidora está executando.

Tabela 1 - *Liveness probe* por microsserviço

O MongoDB foi omitido da tabela pois seu *deploy* em Kubernetes pode ser feito utilizando um Operador Kubernetes MongoDB [13] que regula os *liveness probes* e *readiness probes* de réplicas MongoDB automaticamente.

Ainda sobre o MongoDB, uma de suas preocupações com desempenho em nuvem é o limite de recursos, pois quanto maior o uso de memória permitida para o banco mais rápido ele fica, mas ao mesmo tempo maiores são os requisitos para executá-lo. Usando as recomendações para produção de limites de CPU e memória para um *pod* [14], podemos restringir o uso de CPU para 25% de um único núcleo e a memória para 512 MB, o que nos ajuda a manter baixo custo.

5. RESULTADOS E DISCUSSÕES

Com a solução elaborada e desenvolvida, podemos discutir seu comportamento. Esta seção analisa primeiramente os recursos necessários para uso da solução e seu desempenho. Em seguida, compara o comportamento com os critérios de validação citados anteriormente. Por fim, compara os requisitos anteriores e os atuais, trazendo algumas observações extras.

5.1 Consumo de Recursos e Desempenho

A solução foi executada em um cluster minikube [15], de maneira a monitorar mais facilmente o desempenho e consumo de cada *pod* de réplicas NIALM, auxiliar de agregação e gerente de trabalho. O MongoDB não foi monitorado, pois foi configurado

para consumir apenas os 25% de CPU e 512 MB de memória permitidos.

O consumo de um *pod* NIALM alcançou até 85% do tempo de CPU (ou 850 mCPU) e 650 MB de memória RAM. O uso de CPU e memória variam drasticamente conforme há mais ou menos medições no histórico, e também com a complexidade dos diferentes modelos de rede. Com esses recursos, um *pod* consegue desagregar e publicar resultados de em média 1,5 itens de trabalho por segundo no MongoDB. O tempo também varia conforme a complexidade dos modelos, por vezes disparando timeout no Kafka por demorar muito a confirmar o recebimento do item.

O consumo de um *pod* de gerente de trabalho é de apenas 0,5% de tempo de CPU e 40 MB de memória. Como esperado, o gerente é bastante simples; seus recursos são distribuídos entre operações com o MongoDB e gerenciamento de *threads*. Como tudo o que as *threads* fazem é computar janelas de tempo para produzir itens para o Kafka, não há muito consumo de recursos. O gerente de trabalho é capaz de produzir itens de trabalho de um dia inteiro de medição em cerca de 5 minutos, ou 288 itens por segundo, considerando um passo de desagregação de 60 segundos por item. Esse tempo não cai com mais medidores para desagregar, uma vez que as produções de trabalho para cada medidor ocorrem em paralelo.

O auxiliar de agregação tem um consumo maior, alcançando 20% de CPU e 300 MB de memória por *pod*. Como o módulo de agregação foi extraído do núcleo e reutilizado neste microsserviço, todo o consumo de recursos vem do próprio núcleo. Foi tratado como uma caixa preta e não foi modificado.

A alta vazão do gerente de trabalho e baixa vazão de um *pod* NIALM faz com que muitos itens fiquem retidos por bastante tempo no Kafka. Para aumentar a vazão do NIALM, basta replicá-lo com mais *pods*. Mas mesmo se isso for possível, como não há certeza de quando um *pod* NIALM estará disponível, é necessário que o Kafka consiga reter bastante carga por um bom espaço de tempo. Caso o NIALM não retenha itens por tempo suficiente antes de serem processados, isso causará perda de dados.

Além disso, como por vezes itens mais complexos demoram e disparam timeout do Kafka, conforme as redes neurais que o NIALM usa fiquem mais e mais complexas, é necessário que haja um ajuste fino da tolerância de timeout do Kafka. Pouco tempo causará várias réplicas perdendo trabalho, e muito tempo reduzirá a eficiência do sistema, com réplicas problemáticas segurando o item por mais tempo do que deveriam.

5.2 Critérios de Validação

Ao avaliar a arquitetura seguindo os critérios previamente estabelecidos, podemos notar que o desacoplamento foi alcançado com sucesso, uma vez que os microsserviços não interagem diretamente entre si. Na verdade, seguem desacoplados no tempo e no espaço, graças ao intermédio do Kafka e do MongoDB.

A configurabilidade também foi alcançada, uma vez que nada precisa ser alterado nos microsserviços para que a rede suporte mais modelos, como também para que novos medidores possam ter desagregação ligada ou desligada. O núcleo decide quais medidores serão desagregados, ao passo que os modelos usados pelo NIALM podem ser mudados ao trocar os arquivos do volume usado pelos pods do gerente de trabalho e do NIALM.

Quanto à corretude, os testes foram bem sucedidos. Foi possível comparar os resultados produzidos pela nova arquitetura com os anteriores existentes nos dados exportados do MongoDB de produção. Os resultados de uma desagregação são especificamente listas de dados de consumo por aparelho identificado. Dessa forma, foram checados se para desagregações iguais (i.e., usando os mesmos intervalos de tempo, medidores, e modelos), os dados de consumos e aparelhos são iguais aos originais. O teste foi um sucesso.

Por fim, a arquitetura foi avaliada como de fato sendo tolerante às falhas propostas. O esperado era que outras réplicas pudessem retomar o trabalho de uma réplica interrompida, e isso de fato acontece pois o contador (offset) do consumer group não é avançado; isto é, o Kafka não considera que o consumer group do NIALM consumiu aquele item, e esperará outro *pod* consumi-lo para avançar o contador. O mesmo acontece quando um *pod* dispara timeout. Dessa forma, é garantida a desagregação de um item mesmo sob falhas, permitindo que a desagregação possa executar em instâncias oportunistas

5.3 Comparação de Requisitos: antes e depois

Antes de extrair a desagregação do núcleo monolítico, replicar o NIALM significava replicar o núcleo inteiro. Em outras palavras, exigir mais 16 GB de RAM e 8 vCPUs de alguma outra instância, descartando os problemas apontados de replicação.

Com a nova solução isso não é mais o caso. Uma vez que o NIALM está desacoplado, é possível replicá-lo sem replicar o núcleo. Visto que uma única instância de gerente de trabalho pode produzir muitos trabalhos, é possível ter quantas réplicas trabalhadoras NIALM conforme necessário e desejado. Adicionalmente, caso fosse necessário replicar o gerente de

Os requisitos mínimos para a nova arquitetura funcionar são de 45,5% de tempo de CPU e 852 MB de memória de instâncias de alta demanda. Estes requisitos compreendem um *pod* MongoDB, um *pod* de gerente de trabalho e um *pod* de auxiliar de agregação. Já de instâncias oportunistas, temos um requisito base de 1 CPU (arredondando os 85% de CPU para 100%) e 650 MB de memória por *pod* de NIALM. Se ambos fossem um único tipo de instância, o requisito base para execução da arquitetura seria de 2 vCPU e 2 GB de RAM.

Esse valor é um pouco superior aos 1 vCPU e 2 GB de RAM requeridos pelo NIALM na versão atual (descartando os requisitos do núcleo). Mas é importante ressaltar que não seriam esses os recursos necessários para escalar o NIALM na versão atual, mas sim os recursos requeridos pelo núcleo, visto que o NIALM na versão atual não é escalável.

Usando novamente como comparação os valores da AWS, se todos os recursos fossem concentrados em uma instância On Demand, a instância necessária seria do tipo a1.large, ao invés de a1.medium [5]. O custo seria de 0,051 USD por hora ao invés de 0,0255 USD por hora.

Entretanto, considerando a separação entre recursos de alta disponibilidade e de baixa disponibilidade, uma a1.medium On Demand comporta os recursos de alta, enquanto uma a1.medium Spot comporta os de baixa (*pods* NIALM). Isso leva a um custo base de 0,051 USD [6] + 0,0049 USD [7] por hora, totalizando

0,0559 USD por hora. Basta adicionar 0,0049 USD por hora, a cada nova réplica do NIALM.

Comparando com o custo de uma instância a1.2xlarge [5] - que comportaria o núcleo - e que custa 0,204 USD por hora, seriam necessárias 31 réplicas de NIALM (que custam somadas ao preço base 0.2029 USD) para alcançar o custo de replicar o NIALM como está hoje.

5.4 Considerações Extras

Duas breves, mas importantes considerações devem ser feitas ao implantar essa arquitetura em uma nuvem pública.

Tanto o Kafka como o MongoDB têm papel crucial no funcionamento dos microsserviços e ambos contém informações sensíveis. A comunicação dos serviços com o Kafka e o MongoDB precisam ser asseguradas por protocolos de segurança, como TLS (Transport Layer Security), para mitigar ameaças contra integridade e confidencialidade da informação.

Ao exportar dados do MongoDB de produção, foi observado que o tamanho de um conjunto de dados equivalente a um único dia de medições de 6 medidores tem tamanho razoável de 600 MB de armazenamento. Isso é aproximadamente 100 MB por dia, por medidor. Conforme mais medidores são implantados, é interessante que haja uma limpeza esporádica de dados muito antigos, pois caso contrário o custo com armazenamento na nuvem pública pode ser um empecilho para uso da arquitetura.

6. CONSIDERAÇÕES FINAIS

Observando que os critérios de tolerância a falhas, configurabilidade, corretude e desacoplamento previamente citados foram alcançados, podemos concluir que a arquitetura distribuída proposta é capaz de substituir a atual monolítica. Em especial, graças à tolerância às falhas propostas, a solução pode com sucesso ser executada em nuvem pública, um formato popular para escalabilidade horizontal, e ainda aproveitar-se de instâncias oportunistas.

Adicionalmente, ao realizar a extração de responsabilidades em microsserviços, os requisitos de *hardware* da arquitetura puderam ser reduzidos de maneira a minimizar bastante os custos em nuvem. Com o resultado final foi possível alcançar 72,59% de economia de custos, quando comparado o custo mínimo de hospedar o núcleo inteiro com o custo mínimo de hospedar a nova solução. Essa economia é inferior à de 97,60% estimada anteriormente na Seção 2.1, visto que a estimativa considera apenas o consumo do NIALM original, e não uma versão adaptada com apoio de outros microsserviços e um banco de dados local. Apesar disso, ainda é uma economia.

A arquitetura ainda possui espaço para melhorar, em especial no que diz respeito à segurança. Trazer a solução para um ambiente de nuvem pode aumentar as chances de ataques, e um modelo de ameaças não estava no escopo deste trabalho. Além da adição de protocolos seguros de comunicação (como TLS), permitir conexões da arquitetura na nuvem apenas com o local exato onde o núcleo executa pode minimizar a exposição. Indo além, considerar um modelo de ameaças com apoio de frameworks como STRIDE [16], e o uso de ferramentas de computação confidencial (e.g. SCONE [17]), pode trazer ainda mais confiança no uso da desagregação em nuvem.

O desenvolvimento e avaliação da arquitetura proposta foi baseada em NIALM, mas pode ser naturalmente utilizada para quaisquer algoritmos de análise em que o processamento de dados dos mais diversos tipos de sensores de IoT (como séries temporais, imagem, áudio, etc.) requiser esforço computacional considerável (por exemplo, como uma rede neural ou processo iterativo). Os princípios usados para alcançar desacoplamento temporal e espacial com tolerância a falhas podem ser usados em outros casos de uso.

7. RECONHECIMENTOS

Agradeço a Deus pelas coisas fora de meu controle não terem me atrapalhado, e por todos os obstáculos terem sido superados.

Agradeço com todo amor à minha família pelo apoio constante, que me permitiu ter foco profissional e acadêmico, e por todo o conforto do meu lar para me auxiliar.

Agradeço com todo o meu amor à minha incrível noiva, cuja soma dos pequenos gestos amontoou em um enorme impacto. Seus encorajamentos me permitiram chegar até aqui, e seus conselhos me auxiliaram em escolhas difíceis. Minha estrela guia.

Agradeço muito a meu orientador, que me proporcionou um excelente trabalho com impacto real e direto. Graças à sua experiência e boa vontade, pude em nossas conversas descobrir o gosto pela área e superar todos os problemas técnicos que surgiram.

Agradeço também à equipe do LiteMe pela oportunidade e pela generosa compreensão nos momentos de dificuldades, especialmente pela constante disposição de meu Scrum Master Fellype.

Por fim, agradeço ao Laboratório de Sistemas Distribuídos, pela oportunidade de trabalho e estudos que me proporcionaram a experiência necessária para este trabalho, além da infraestrutura sempre disponível.

8. REFERÊNCIAS

[1] LiteMe. LiteMe | Inteligência Energética. Quem Somos. Retirado 11 de Agosto de 2022 em https://www.liteme.com.br/about

[2] ARMEL, K. Carrie et al. Is disaggregation the holy grail of energy efficiency? The case of electricity, p. 2. Energy policy, v. 52, p. 213-234, 2013.

[3] HART, George William. Nonintrusive appliance load monitoring. Proceedings of the IEEE, v. 80, n. 12, p. 1870-1891, 1992.

[4] Amazon. 2022. Instâncias spot do Amazon EC2 - Economize em até 90% dos preços sob demanda. Retirado 11 de Agosto de 2022 em: https://aws.amazon.com/pt/ec2/spot/

[5] Amazon. 2022. Tipos de instâncias do Amazon EC2 - AWS. Retirado 11 de Agosto de 2022 em: https://aws.amazon.com/pt/ec2/instance-types/

[6] Amazon. 2022. Amazon EC2 on demand - preço sob demanda - AWS. Retirado 11 de Agosto de 2022 em: https://aws.amazon.com/pt/ec2/pricing/on-demand/

[7] Amazon. 2022. Definição de preço de instâncias spot do Amazon EC2. Retirado 11 de Agosto de 2022 em: https://aws.amazon.com/pt/ec2/spot/pricing/

[8] RabbitMQ. Reliability Guide - RabbitMQ. Retirado 11 de Agosto de 2022 em: https://www.rabbitmq.com/reliability.html

[9] Apache Software Foundation. 2022. Apache Kafka. Retirado 11 de Agosto de 2022 em: https://kafka.apache.org/documentation/

[10] Kubernetes. 2022. Kubernetes. Retirado 11 de Agosto de 2022 em: https://kubernetes.io/

[11] CNCF. 2022. CNCF Annual Survey 2021 | Cloud Native Computing Foundation. Retirado 11 de Agosto de 2022 em: https://www.cncf.io/reports/cncf-annual-survey-2021/

[12] Kubernetes. Configure Liveness, Readiness and Startup Probes | Kubernetes. Retirado 11 de Agosto de 2022 em: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

[13] MongoDB. Quick Start - MongoDB Kubernetes Operator 1.16. Retirado 11 de Agosto de 2022 em: https://www.mongodb.com/docs/kubernetes-operator/stable/kind-quick-start/

[14] MongoDB. MongoDB Enterprise Kubernetes Operator Production Notes - MongoDB Kubernetes Operator 1.16, Set CPU and Memory Utilization Bounds for MongoDB Pods. Retirado 11 de Agosto de 2022 em: https://www.mongodb.com/docs/kubernetes-operator/master/reference/production-notes/#set-cpu-and-memory-utilization-bounds-for-mongodb-pods

[15] Kubernetes. 2022. Welcome! | minikube. Retirado 11 de Agosto de 2022 em: https://minikube.sigs.k8s.io/docs/

[16] Microsoft. 2009. The STRIDE Threat Model | Microsoft Docs. Retirado 11 de Agosto de 2022 em: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)

[17] Scontain. 2022. Overview - Confidential Computing. Retirado 11 de Agosto de 2022 em: https://sconedocs.github.io/workflows/