Pontifícia Universidade Católica
do Rio de Janeiro

Camila Antonaccio Wanous

Reengenharia do ContextNet utilizando Kafka

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática, do Departamento de Informática da PUC-Rio.

Orientador: Prof. Markus Endler

Rio de Janeiro
Maio de 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Camila Antonaccio Wanous

Reengenharia do ContextNet utilizando Kafka

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo:

Prof. Markus Endler
Orientador
Departamento de Informática – PUC-Rio

Profª. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. Sérgio Colcher
Departamento de Informática – PUC-Rio

Prof. Francisco José da Silva e Silva
UFMA

Rio de Janeiro, 28 de Maio de 2021

Camila Antonaccio Wanous

Graduação em Engenharia de Computação pelo IME.

Ficha Catalográfica

Antonaccio Wanous, Camila

Reengenharia do ContextNet utilizando Kafka / Camila Antonaccio Wanous; orientador: Markus Endler. – 2021.

57 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Comunicação Móvel – Teses. 2. Sistemas Distribuídos – Teses. 3. Internet das Coisas Móveis. 4. IoT. 5. Sistemas Distribuídos. 6. Rede de Computadores. 7. Comunicação Móvel. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Para minha avó, Anna
Pesce Antonaccio (in memorian)

Agradecimentos

A meu orientador Prof. Markus Endler por toda sabedoria, dedicação e carinho entregues durante todo o período que estivemos trabalhando nesta dissertação.

Ao DI da PUC-Rio por todo apoio e infra-estrutura.

A todos os meus amigos, em especial ao Vinícius que me encorajou ingressar no mestrado, ao Rodrigo e ao Matheus. A minha namorada, May, que tão pacientemente passou os últimos anos aprendendo junto comigo.

Ao LAC e a todos os seus membros pela paciência e pelas intensas trocas.

A minha mãe Ana, a meu pai Alfredo e a toda minha família.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Resumo

Antonaccio Wanous, Camila; Endler, Markus. **Reengenharia do ContextNet utilizando Kafka**. Rio de Janeiro, 2021. 57p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

ContextNet é um middleware de comunicação baseado no padrão OMG DDS, que suporta monitoramento em tempo real, unicast, groupcast e transmissão para milhares de nós móveis. Este trabalho substituiu a infraestrutura DDS (Data Distribution Service), utilizada na comunicação dos nós Core da ContextNet, pela plataforma Kafka; simplificou a construção de aplicativos que utilizam o middleware ContextNet; e adicionou novos recursos. Kafka é uma plataforma de streaming capaz de subscrever, publicar, armazenar e processar em fluxos em tempo real. A utilização do Kafka permitiu que as aplicações construídas sobre o ContextNet sejam paralelizáveis.

Palavras-chave

Internet das Coisas Móveis; IoT; Sistemas Distribuídos; Rede de Computadores; Comunicação Móvel.

Abstract

Antonaccio Wanous, Camila; Endler, Markus (Advisor). **Reengineering ContextNet using Kafka**. Rio de Janeiro, 2021. 57p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

ContextNet is a communication middleware based on the OMG DDS standard that supports real-time monitoring, unicast, groupcast, and broadcast to thousands of mobile nodes. This work replaced the DDS (Data Distribution Service) infrastructure used to communicate the ContextNet Core nodes with the Kafka platform; simplified the construction of applications that use the ContextNet middleware; and added new features. Kafka is a streaming platform capable of subscribing, publishing, storing, and processing in real-time streams. Using Kafka has enabled applications built on top of ContextNet to be parallelizable.

Keywords

Internet of Mobile Things; IoT; Distributed System; Computer Network.

Sumário

Lista de figuras

Lista de tabelas

Do mesmo modo
que te abriste à alegria
abre-te agora ao sofrimento
que é fruto dela
e seu avesso ardente.

Do mesmo modo
que da alegria foste
ao fundo
e te perdeste nela
e te achaste
nessa perda
deixa que a dor
se exerça agora
sem mentiras
nem desculpas
e em tua carne vaporize
toda ilusão

que a vida só consome
o que a alimenta.

Ferreira Gullar, *Aprendizagem.*

1
Introdução

Este trabalho consiste na reimplementação do ContextNet [1] substituindo o padrão de comunicação que o suporta por uma nova tecnologia; adicionando funcionalidades; mantendo seus princípios arquiteturais e seu propósito - suporte a aplicações IoT.

ContextNet é um *middleware* de comunicação baseado no padrão *Data Distribution Service* do *Object Management Group* (*OMG DDS*) [3], que suporta monitoramento em tempo real, unicast, groupcast e broadcast para milhares de nós móveis. Os elementos do ContextNet podem ser divididos entre estáticos e móveis. Os primeiros fazem parte do ContextNet Core e operam sobre a infraestrutura *OMG DDS* que o presente trabalho substituiu pela plataforma Kafka.

Acreditava-se que versão do ContextNet sobre Kafka seria mais escalável do que a versão seguindo o padrão OMG DDS, e para provar essa hipótese este trabalho foi desenvolvido. Esperava-se que com a substituição o novo ContextNet pudesse permitir a paralelização das aplicações operando sobre sua infraestrutura; que novas funcionalidades em linha com os princípios arquiteturais fossem adicionadas; e que fosse simplificada a construção de soluções.

O ContextNet foi desenvolvido para auxiliar a implementação de aplicações IoT que suportam tanto o rastreamento em tempo real quanto meios eficientes de interação entre milhares de dispositivos móveis, sensores e atuadores a elas conectados. Além disso, em muitas dessas aplicações, o conjunto de nós móveis participantes varia constantemente, pois os mesmos podem entrar e sair do sistema a qualquer momento e em qualquer ponto da rede, seja devido a circunstâncias específicas da aplicação ou a intermitência da conectividade sem fio. Tais aplicações requerem, portanto, uma infraestrutura de comunicação escalável, confiável e quase instantânea e a disseminação do contexto entre toda a rede. O ContextNet em ambas as versões provê essa infraestrutura.

Kafka [10] é uma plataforma de *streaming* distribuído capaz de subscrever; publicar; armazenar e processar em tempo real *streams*. Essa substituição veio acompanhada da adição de novas funcionalidades ao *middleware* descritas nos próximos capítulos; pela adoção do protocolo *Publish/Subscriber* para

comunicação entre os nós do Core; e pelo acréscimo de possibilidades de parametrizações por parte dos usuários.

A facilidade para a construção de soluções sobre o *middleware* ContextNet se une a possibilidade de paralelização componentes estáticos; a melhora no desempenho dos componentes estáticos; e a adição de funcionalidades como principais objetivos do trabalho. Também é objetivo deste trabalho verificar se com Kafka foi mantido o suporte a uma quantidade significativa de conexões móveis.

O ContextNet original segundo o artigo que o descreve [1] suporta uma quantidade significativa de conexões móveis por Gateway (até 12 mil nós conectados por Gateway foram testados com sucesso) e portanto satisfaz as atuais necessidade de seus usuários quanto ao número de conexões. Contudo, a versão original não permite que haja paralelização dos nós aplicação e trafega todos os tópicos por um único canal, obrigando todos os componentes estáticos a receberem todas as mensagens trafegadas, inclusive àqueles cujos tópicos não são de seu interesse, os sobrecarregando, e portanto, impossibilitando a construção de aplicações complexas ou escaláveis que sejam capazes de dar suporte a grandes quantidades de conexões móveis. A utilização do Kafka permitiu a nova versão do ContextNet paralelizar componentes estáticos e, portanto, permitirá o desenvolvimento de aplicações core complexas e/ou escaláveis.

O ContextNet original demanda uma configuração extensa e trabalhosa do ambiente o que onera os desenvolvedores. A relativa simplicidade da implementação com Kafka frente a implementação com DDS e a utilização de *Containers* tornaram mais fácil para os desenvolvedores a utilização do ContextNet. Bastará, em termos de infraestrutura, ter instalado em sua máquina o *Docker* para utilizar o ContextNet.

No novo middleware foram mantidos os princípios arquiteturais do ContextNet: escalabilidade quanto a quantidade de nós móveis; escalabilidade através do balanceamento do fluxo de dados; simplicidade e flexibilidade; suporte a mobilidade e desconexão temporária (*handovers*); suporte a comunicação de alta performance; capacidade para comunicação unicast, groupcast e broadcast e capacidade de monitoramento em tempo real grupos de nós móveis.

O ContextNet Core é composto por microsserviços independentes. Essa arquitetura reflete na organização deste texto que possui um capítulo exclusivamente dedicado a cada um desses microsserviços. O nó do Core que se comunica com os nós moveis e com estes microsserviços é chamado Gateway, e possui uma seção exclusiva para seu detalhamento. Dentro destas seções são discutidas as decisões de projeto e os aspectos de implementação; entre outros.

Este texto é composto por um capítulo de trabalhos relacionados; seguida por um capítulo sobre o Gateway e sobre os microsserviços; seguidos por um capítulo que trata sobre dos Testes; e por fim traz a conclusão e as perspectivas futuras.

2
Trabalhos Relacionados

A pesquisa por trabalhos relacionados se deu principalmente através da busca por middlewares para comunicação móvel real-time que utilizassem internamente o protocolo de comunicação *Publish/Subscriber*. A seguir são brevemente descritos alguns desses trabalhos encontrados, as referências teóricas sobre tais middlewares, e as diferenças e semelhanças entre eles e o novo ContextNet.

Em [2], Kai Waehner elenca os motivos pelos quais a indústria de telecomunicações está migrando para soluções que utilizem o Apache Kafka. O autor defende que telecomunicações hoje em dia é menos sobre voz e cada vez mais sobre texto (mensagens, *e-mail*) e imagens (por exemplo, streaming de vídeo); e que crescimento mais rápido vem dos serviços (de valor agregado) fornecidos através de redes móveis.

Dentre os motivos listados no artigo para o uso do Kafka em soluções da indústria de telecomunicações estão a necessidade de processamento em tempo real com escalabilidade; e de um sistema confiável com zero tempo de inatividade e sem perda de dados. Os processos tradicionais da telecomunicação são separados em OSS (*Operations Support System*, ou Sistemas de Apoio às Operações) - aos quais está conectada a rede móvel, BSS (*Business Support System*, ou Sistemas de Apoio às Empresas) e OSS-BSS-Integração; e segundo o artigo o Apache Kafka é utilizado em todas elas.

A arquitetura de um framework genérico para o processamento, armazenamento e análise de dados *IoT* em tempo real chamada *L-CoAP* e que utiliza Apache Kafka foi proposta em artigo de 2015 [6]. Diferentemente do ContextNet, o *middleware L-CoAP* não tem como objetivo permitir a comunicação real-time entre o seus componentes mas sim prover uma camada abstrata de acesso, consulta e controle de componentes *IoT*.

Segundo os autores [6] a *L-CoAP* é a combinação de recursos da *Cloud Computing* responsáveis pela abstração das limitações dos dispositivos *IoT*, com um middleware *IoT* adequado para dispositivos com restrições de recursos.

O *L-CoAP* é composto por três diferentes elementos: uma nuvem; os *Smart Gateways*, projetados para abstrair e proteger os dispositivos *IoT* de fontes externas; e os dispositivos finais; sendo um dos componentes da nuvem

uma instância Apache Kafka. Nesta arquitetura o Apache Kafka é responsável por distribuir em tempo real os fluxos de dados recebidos na Nuvem através dos *Smart Gateways*.

A arquitetura proposta por [6] não endereça os problemas da comunicação móvel em tempo real abarcadas pelo ContextNet pois está direcionado para a comunicação com dispositivos *IoT*. O trabalho apresenta uma arquitetura que poderia ser incorporada a camada Kafka do novo ContextNet ou adicionada como um novo microsserviços com diversas novas funcionalidades para componentes *IoT* inexistentes hoje.

Em [7] é descrita a *RADAR-base*, uma plataforma *mHealth* (*Mobile Health*, ou Saúde Móvel) de coleta de dados médicos construída sobre o Apache Kafka. A RADAR-base permite a coletas ativa e passiva de dados remota; fornece canais seguros de transmissão de dados; e uma solução escalável para armazenamento, gerenciamento e acesso a esses dados. A plataforma atualmente é utilizada no estudo *RADAR-CNS* para coletar dados de pacientes que sofrem de esclerose múltipla, depressão e epilepsia. Nessa plataforma deve-se a capacidade de processamento em tempo real ao *Kafka Streams*. Os nós móveis se conectam via *Kafka-Proxy* ao *cluster* Kafka.

[7] também não endereça os problemas da comunicação móvel como o ContextNet, se dedicando a proposição de uma forma de coleta e armazenamento de dados. A integração via conectores Kafka é mais restrita que a do ContextNet, e foi adicionada a nova versão como mais uma alternativa de conexão.

[8] discute critérios de seleção para arquiteturas de nuvens para suporte a aplicações *Smart City*. Propõe-se uma solução de nuvem móvel com base na abordagem *cloudlet*. Ela utiliza o *cluster Kafka* como *cloudlet* baseado em mensagens e o *Kafka Proxy REST* para conexão com clientes móveis. A solução proposta por [8] para *Smart City* em muito se assemelha a arquitetura do novo ContextNet, com os Gateways substituídos pelo *Kafka Proxy REST*.

3
Tecnologias Base

3.1
O ContextNet Original

O sistema de software ContextNet [1], desenvolvido desde 2012, é um *middleware mobile-cloud* para *IoT*, que essencialmente consiste de protocolos robustos de comunicação para enlaces sem fio, e para comunicação *intra-cloud*, que contemplam um numero arbitrário e dinâmico de nós, bem como conectividade intermitente.

Trata-se de uma arquitetura de microsserviços, onde serviços de processamento, visualização e análise de dados, bem como de atuação sobre atuadores, podem ser adicionados ou removidos de acordo com as necessidades específicas do monitorando ou do controle autônomo.

O ContextNet é composto por três grupos de nós: os estacionários que executam os serviços do ContextNet Core; os Mobile-Hub (M-Hub), móveis ou estacionários, que servem de intermediários entre o Core e os Mobile-Objects (M-OBJ) em sua vizinhança imediata; e os M-OBJ, smart things móveis ou estáticas. O presente trabalho visa substituir o ContextNet Core bem como o padrão de comunicação entre seus elementos, o *OMG DDS* [3]. Na implementação original do ContextNet as limitações do *OMG DDS* e a incapacidade de compatibilizar o caráter estático (definidos em tempo de projeto) dos tópicos DDS forçaram a implementação de um canal de comunicação *publisher/subscriber* com apenas um tópico.

Como pontuado na Introdução, o ContextNet Core é composto por diversos nós estáticos chamados Gateways projetados para atender milhares de conexões com mobiles; e por microsserviços independentes. São eles:

1. PoA Manager: responsável por realizar o balanceamento de carga sobre os Gateways. Entende-se por carga a quantidade de conexões móveis sob determinado Gateway;

2. Mobile Temporary Disconnect (MTD): responsável por armazenar e re-enviar mensagens para nós cuja conexão estava temporariamente indisponível;

3. Group Definer: responsável por classificar os nós móveis em grupos;

Em [9] o desempenho do ContextNet original foi documentado e será comparado ao da nova versão.

3.2 Kafka

Como exposto anteriormente Kafka é uma plataforma de *streaming* distribuído capaz de subscrever (transitividade); publicar; armazenar e processar streams em tempo real.

A primeira decisão importante tomada para a reengenharia do ContextNet Core foi a escolha da camada de comunicação que substituiria o *OMG DDS*. Buscava-se a manutenção da arquitetura de mini serviços originais; a adoção do protocolo *publish-subscriber* com múltiplos tópicos e a possibilidade de paralelização das aplicações que operavam sobre o Core. Também era desejado que a nova tecnologia fosse de fácil implementação para os desenvolvedores aumentando sua aderência.

Estudou-se as tecnologias *Kafka*, *Kinesis*, *GCP* e *Zeromq*. Escolheu-se o *Kafka* pois ele atendia os requisitos levantados: possui ampla documentação; é *open-source*; suporta diversas aplicações operando simultaneamente sobre um única instância; é paralelizável e escalável; adota estratégia de replicação o que o torna tolerante a falhas; é desenhado para aplicações em tempo real (*high-throughput*) e que movimentam um grande volume de dados; possui nativamente compressão de mensagens; e é flexível.

O Kafka permitiu que aplicações construídas sobre o ContextNet sejam paralelizáveis e escaláveis; que existam tópicos exclusivos para cada componente diminuindo a quantidade de mensagens processadas pelos nós estáticos ligados ao Core, tanto os de aplicação quanto os de serviços do ContextNet. O mecanismo nativo e personalizável de replicação do Kafka permite que o novo ContextNet Core seja tolerante a falhas, o original não é.

Além dos requisitos iniciais, o Kafka possibilita a criação dinâmica de tópicos o que permite conexão de novas aplicações ao Core durante sua operação; suporta protocolos amplamente utilizados em IoT como o $MQTT$ e permite a ordenação das mensagens - o que pode ser útil na comunicação entre algumas aplicações.

3.2.1 Comunicação *Publisher/Subscriber*

Com a substituição do DDS, pode-se adotar múltiplos tópicos e um verdadeiro protocolo de comunicação *Publisher/Subscriber*. Com isso optou-se por

adotar tópicos exclusivo para cada aplicação; para cada tipo de comunicação - unicast, groupcast e broadcast; para cada dado de controle; e para os dados de contexto. A quantidade de records consumidos pelos nós conectados ao Core diminuiu drasticamente, pois esses múltiplos tópicos e o modelo publish/subscriber permitem a subscrição exclusiva aos tópicos de interesse.

3.2.2 Kafka *Components*

Kafka [10] é um sistema distribuído que consiste em servidores e clientes que se comunicam através de um protocolo de rede TCP de alto desempenho. O Kafka é executado como um *cluster* de um ou mais servidores, altamente escalável e tolerante a falhas. Os nós desse *cluster* são chamados *Brokers*. Os clientes Kafka permitem a construção de aplicações distribuídas e microsserviços que leem, escrevem e processam fluxos de eventos em paralelo, em escala e de forma tolerante a falhas, mesmo no caso de problemas de rede ou falhas de máquina.

Segundo a terminologia Kafka, *Records* registram o fato de que "algo aconteceu"no mundo. Quando se ou escreve dados para Kafka, se faz sob a forma de *Records*. Conceitualmente, um *Records* tem uma chave, um valor, um carimbo de data/hora e cabeçalhos de metadados opcionais. Os *Records* são organizados e armazenados de forma durável em tópicos.

Os produtores são as aplicações clientes que publicam *Records* de um determinado tópico para o Kafka; e os consumidores são aplicações cliente que assinam (leem e processam) *Records* de um determinado tópico. Os tópicos são particionados, ou seja, estão espalhado por vários *"buckets"* localizados em diferentes pontos do *cluster* Kafka. Esta colocação distribuída de seus dados é muito importante para a escalabilidade, pois permite que as aplicações clientes leiam e escrevam os dados de/para muitos pontos ao mesmo tempo.

Dentro do *cluster* Kafka um tópico é distribuídos em n partições, sendo n no mínimo igual a um e no máximo igual ao número de *Brokers* do *cluster*. Quando um novo *Record* é publicado para um tópico, ele é na verdade anexado a uma das partições apenas do tópico. *Records* que possuem a mesma chave são anexados a mesma partição.

No Kafka os consumidores são agrupados em *"Consumer Groups"* Consumidores de um mesmo grupo são vinculados a partições distintas e portanto recebem *Records* diferentes para um mesmo tópico. A totalidade de *Records* enviados por um determinado tópico chega ao conjunto dos consumidores que pertencem ao mesmo grupo.

A distribuição dos tópicos em partições e a divisão lógica dos consu-

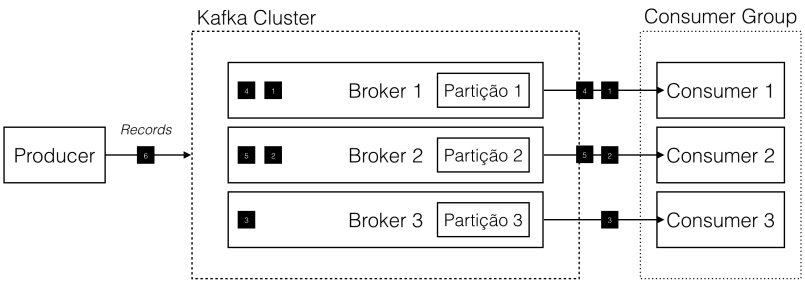

Figura 3.1: Kafka Cloud

midores em *"Consumer Groups"* permitem que aplicações consumidoras sejam paralelizáveis. Na figura 3.1 estão está um exemplo dessa paralelização. Um produtor (*Producer*) publica em um determinado tópico *Topic Records* no *Cluster Kafka* e eles são distribuídos por três partições (Partição 1, Partição 2, Partição 3). O *Cluster Kafka* possui três *Brokers* e o tópico *Topic* três partições. Uma aplicação está dividida em três nós consumidores que fazem parte de um mesmo *Consumer Group*. Eles estão ativos e consumindo os *Records* publicados. Cada um deles consome de uma das partições somente, e portanto, consome *Records* distintos. Dessa forma paraleliza-se a aplicação em três nós (*Consumer 1*, *Consumer 2*, *Consumer 3*).

3.2.3 Estrutura hierárquica de Tópicos

A estrutura hierárquica de tópicos adotada no protocolo MQTT não é suportada pelo Kafka o que impede o novo ContextNet de também dar suporte a hierarquização dos tópicos, muito comum em dispositivos *IoT*.

3.3 Docker

O *Docker* é uma tecnologia para construção de *Containers* Sua utilização elimina problemas de compatibilidade entre sistemas operacionais e bibliotecas.

A nova versão do ContextNet foi implementada utilizando o *Docker* dado que este facilita a integração entre diferentes componentes e a construção da infraestrutura das soluções; e, portanto, o uso do middleware pelos desenvolvedores.

3.4 Protocolo MR-UDP

O MR-UDP [11] é uma versão estendida e otimizada do UDP confiável (R-UDP) comunicação móvel. Ele estende o protocolo R-UDP com característi-

cas tolerantes à mobilidade, tais como a capacidade de lidar com conectividade intermitente, travessia *Firewall/NAT* e robustez para comutação de endereços IP ou interfaces de rede (por exemplo, Celular para WiFi, e vice-versa). Ao usar UUID para endereçar a conexão e pacotes em vez de IP, o MR-UDP pode lidar com handovers.

4
Projeto do ContextNet Kafka Core

Nesta seção serão brevemente apresentados os principais componentes do ContextNet Kafka Core (CKC) e suas conexões. Estes componentes serão detalhados nas seções a seguir.

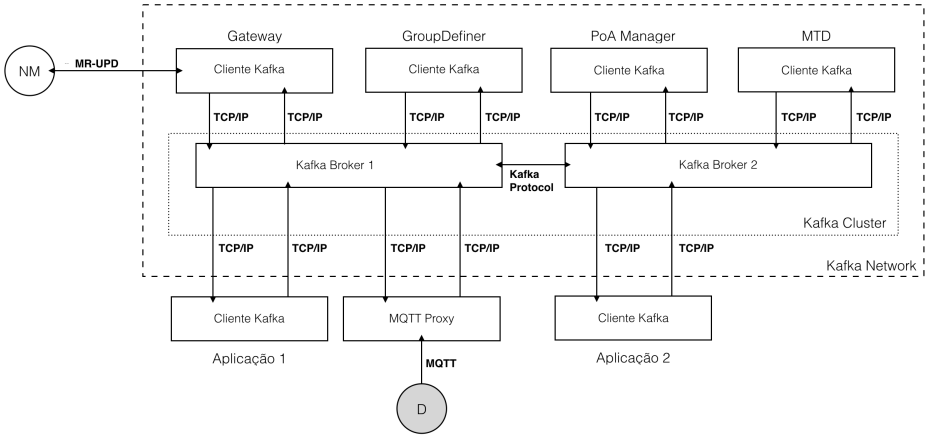

Figura 4.1: Diagrama ContextNet Kafka Core

Na figura 4.1 os nós móveis estão representados pelo elemento "MN"e se conectam ao CKC através do Gateway por uma conexão MR-UDP. O Gateway, que é um Cliente Kafka se comunica via TCP/IP com um dos servidores Kafka (*Brokers*). Os *Kafka Brokers* compõe o *Kafka Cluster*. O *Cluster* se comunica via TCP/IP com os demais micro-serviços do CKC: *PoA Manager*, *Group Definer* e *Mobile Temporary Disconnect* que são Clientes Kafka.

As aplicações ("Aplicação 1"e "Aplicação 2") desenvolvidas sobre o *middleware* também são Clientes Kafka e também se comunicam com o *Cluster* via TCP/IP.

Os *devices* que se comunicam exclusivamente via MQTT enviam dados para o *Cluster* por um *MQTT Proxy* que se comunica via TCI/IP com os *Brokers*.

4.1
Principais Funcionalidades

Nesta seção serão listadas as Principais funcionalidades ContextNet Kafka Core. Maiores detalhes serão dados nas próximas seções.

- *Unicast* em tempo real para nós móveis e nós aplicação
- *Groupcast* em tempo real para nós móveis (Grupos dinâmicos definidos a partir de informações de contexto)
- *Broadcast* em tempo real para nós móveis
- Coleta de informações sobre as conexões
- Balanceamento parametrizável de carga dos Gateways
- Armazenamento temporário de mensagens a nós móveis desconectados
- Distribuição frequente para os nós móveis de lista de endereços de Gateways ativos ordenada pela disponibilidade
- Coleta de dados via protocolo MQTT
- Possibilidade de paralelização das aplicações

4.2 Escalabilidade

O novo ContextNet tem como um dos seus principais objetivos tornar o ContextNet escalável no que tange as aplicações, e consequentemente aos nós móveis.

A escalabilidade quanto ao número de nós móveis por Gateway também é importante, contudo, ela é limitada pelo protocolo de comunicação móvel utilizado (no caso MR-UDP). Sendo assim, este trabalho focou em tornar escaláveis os nós estáticos que darão suporte a uma quantidade crescente de Gateways conectados ao mesmo Core, e portanto, quantidade crescentes de nós móveis.

A escalabilidade dos nós estáticos além de indiretamente permitir um aumento no número de nós móveis, torna o ContextNet mais apto para dar suporte a aplicações complexas que exijam muito processamento.

Ser escalável unicamente em relação ao número de nós móveis é limitante, uma vez que os nós interagem necessariamente com aplicações que necessitam ser capazes de suportá-los.

A versão atual do ContextNet não possui recursos suficientes para permitir que o fluxo de mensagens de um determinado canal de comunicação seja dividido entre diversos nós, recurso este que está presente na nova versão.

Com a paralelização espera-se que o novo ContextNet seja capaz de suportar aplicações que falhariam sem ela; e que as aplicações possam buscar por meio desta paralelização melhorar a performance da comunicação.

4.3 Projeto do Gateway para o ContextNet Kafka Core

O Gateway é o principal componente do ContextNet Kafka Core. É através dele que os nós móveis se comunicam com os nós estáticos. Ele é capaz de se comunicar utilizando tanto o protocolo MR-UDP quanto o protocolo do Kafka e de fazer tradução de um protocolo para o outro; enviar mensagens unicast dos nós móveis para os nós estáticos e vice e versa; enviar mensagens groupcast - sincronizadas ou não - dos nós móveis e dos nós estáticos para os nós móveis; enviar mensagens broadcast dos nós estáticos para os nós móveis; coletar e enviar dados sobre o estado do hardware e das conexões móveis; e identificar mensagens não entregues aos nós móveis destinatários e encaminha-las a um repositório (MTD).

O Gateway da nova versão do ContextNet Core além das funcionalidades acima listadas é capaz de publicar *Records* para diversos tópicos conforme o endereçamento dos nós; de coletar o tempo de resposta as mensagens *Ping* (utilizadas para medição de latência); de reconfigurar o tamanho da janelas de espera por respostas as mensagens *Ping* e manter um *mini-buffer* com as últimas mensagens enviadas por uma conexão móvel.

4.3.1 Decisões de projeto e justificativa

Nesta seção serão expostas decisões de projeto relativas ao Gateway bem como as justificativas para as mesmas.

4.3.2 Coleta de dados sobre as conexões móveis (*Ping*)

A coleta de dados sobre as conexões móveis é feita através do envio de mensagens *Ping*. O Gateway encaminha para os nós móveis a ele conectados tais mensagens e os nós móveis respondem a essa mensagem com uma mensagem *Pong*. Dessa forma mede-se a latência da comunicação.

A contabilização dessa latência poderia ser feita tanto pelo Gateway quanto pelo PoA Manager. Como o envio das mensagens necessariamente passa pelo Gateway é natural que se delegue a ele essa tarefa. Contudo o Gateway já é o elemento mais sobrecarregado em termos de processamento, delegar essa tarefa ao PoA Manager diluiria a carga sobre o Gateway. O aumento da quantidade de dados que trafegarão pelo Core nessa arquitetura, e que poderia ser visto como uma desvantagem não é significativo. Como não sobrecarrega os canais de comunicação do Core, e impacta positivamente na performance do Gateway delegou-se ao PoA Manager a contabilização.

No antigo ContextNet essa contabilização era realizada pelo Gateway, e consistia somente na contagem do número de conexões cuja latência estivesse acima de um limite pré-estabelecido. Essa informação só era consumida por nós aplicação conectados ao Core, e não influía no balanceamento de cargas. A nova arquitetura permitirá que a latência e outras métricas sobre as conexões sejam levadas em conta pelo algoritmo de balanceamento.

O processo de coleta dos *Pongs* ocorre dentro de janelas de tempo. Essas janelas se iniciam após o envio dos *Pings*, e durante sua vigência é registrado o momento em que mensagens *Pong* são recebidas. Mensagens *Pong* recebidas depois do fim da janela não são registradas. Esses registros são enviados para o PoA Manager – para que ele contabilize a latência.

Optou-se pelo uso de janelas e não pela coleta contínua de *Pongs* pois a segunda opção exigiria que o Gateway mantivesse uma estrutura atualizada em tempo real dos *Pings* enviados e dos *Pongs* recebidos por cada nó móvel, aumentando sua complexidade e seu o custo de processamento com um ganho de acurácia que não se considerou significativo.

Na nova versão a frequência de envio dos *Pings*, bem como o tempo das janelas para recebimento dos *Pongs* são definidos pelo PoA Manager e encaminhadas ao Gateway por um tópico de configuração "*PingConfig*". Ao receber uma mensagem "*PingConfig*" o Gateway reconfigura seu processo de envio de *Pings* e coleta de *Pongs*. A flexibilização da duração da janela permite que a mesma esteja coerente com as condições da rede; e que se avalie melhor essas condições aumentando a acurácia. Supondo que para uma determinada janela o percentual de nós móveis que respondem ao Ping em todos os Gateway seja muito baixo, aumentando-se a janela pode-se verificar se a baixa desse percentual se deve a desconexões ou a lentidão nas conexões. O mesmo também poderia ser verificado com a coleta contínua de *Pongs* só que a um custo de processamento muito maior.

4.3.3 Mensagens não enviadas

Quando há um problema na conexão entre o Gateway e um nó móvel que impeça que mensagens trafeguem por ela; e o Gateway, sem estar ciente do problema, tenta enviar uma mensagem por ela pode haver ou não falha no envio, falha essa gerada pelo protocolo de comunicação MR-UDP. No caso em que há falha no envio uma exceção é gerada, capturada e a mensagem encapsulada num *Record* e enviada ao MTD pelo Gateway através do tópico "*UnsentMessage*". No caso em que não há falha no envio e a mensagem não foi enviada, o protocolo de comunicação imediatamente notifica o Gateway sobre

o não envio. Essa notificação, contudo, não informa qual a mensagem não foi enviada.

Para possibilitar que a mensagem não enviada seja despachada para o MTD a nova versão mantém um *mini-buffer* de mensagens por conexão. Esse *mini-buffer* mantém em memória as duas últimas mensagem enviadas sem falha pelo Gateway por cada conexão (cada conexão conecta um único nó móvel ao Gateway). Caso a conexão notifique o Gateway acerca do não envio, as mensagens referentes a ela que estão no *mini-buffer* são despachadas para o *Mobile Temporary Disconnect*. Este *mini-buffer* garante que todas as mensagens serão enviadas pelo menos uma vez para seus nós destinatários.

Por meio de uma variável de ambiente pode-se determinar o uso ou não do *mini-buffer*. Seu uso garante que todas as mensagens serão entregues pelo menos uma vez aos destinatários contudo aumenta o tempo de execução da *thread* responsável pelo envio de mensagens. Está *thread* atualiza o *mini-buffer* ao fim de cada envio bem-sucedido, e essa atualização faz com que ela demore mais tempo para finalizar, e que, portanto, permaneça por mais tempo ocupando a *Thread Pool*. Além disso, atualizações do *mini-buffer* que dizem respeito a uma mesma conexão não podem ocorrer paralelamente o que faz com que o tempo de execução da atualização cresça muito caso diversas mensagens sejam enviadas por uma mesma conexão. A implementação do *mini-buffer* portanto pode introduzir lentidão no Gateway e delegou-se ao desenvolvedor a opção de adota-la ou não.

4.3.4 Possibilidade de Ordenação das Mensagens

O Kafka permitiria que a ordenação das mensagens fosse estendida para todo tipo de comunicação, contudo, isso demandaria muito processamento do Gateway, e diminuiria significativamente sua performance, e portanto, não foi adicionada a nova versão. Apenas a comunicação entre aplicações permite a ordenação das mensagens.

4.3.5 Padrão de Projeto: Thread Pool

O Gateway foi desenvolvido segundo o padrão de projeto *Thread Pool*. Durante os testes observou-se que o tamanho ideal da *Thread Pool* não pode ser estimado sem que antes se tenha conhecimento da quantidade de conexões móveis que o Gateway precisará suportar. Sendo assim optou-se por permitir que desenvolvedor defina o tamanho da *Thread Pool* segundo suas necessidades.

Há um tamanho padrão, mas caso haja necessidade o desenvolvedor pode declarar, via variável de ambiente, o tamanho da *Thread Pool*.

4.3.6 Aspectos de implementação

Os tópicos abaixo discutem aspectos importantes e detalhes da implementação do Gateway e trazem justificativas para as decisões de arquitetura interna do Gateway.

4.3.7 Gateway *Kafka-Subscriber*

Para a implementação do componente do Gateway responsável por coletar e filtrar os records que trafegam pelo Core (subscriber) havia duas tecnologias Kafka disponíveis: *Kafka Streams* e *Kafka Consumer*. Testou-se ambas para verificar se havia diferença significativa de performance entre elas. Nos testes foi comparado o tempo entre o envio e a recebimento por cada implementação de diversos *Records*. O processo de recebimento envolvia a coleta do *Record* e o filtro pelo valor, filtro este que possuía o mesmo critério em ambas as implementações.

Os testes foram realizados com as duas implementações operando simultaneamente sobre o Core, na mesma máquina, e recebendo simultaneamente os registros. Em cada teste foram enviados aproximadamente 10.000 *Records* sendo 5000 que atendiam ao critério do filtro, com 10 milissegundos de intervalo entre cada um. As implementações apresentaram desempenho semelhante. Dado que não havia diferenças no desempenho optou-se pelo *Kafka Consumer* pela maior simplicidade na implementação.

4.3.8 Simultaneidade no *Groupcast* e no *Broadcast*

Simultaneidade no envio das mensagens *groupcast* e *broadcast* pode ser desejada por muitas aplicações, simultaneidade que teoricamente só pode ser garantida até o Gateway dada a natureza instável das conexões móveis. Para chegar mais próximo possível dessa simultaneidade o Gateway deveria criar *threads* exclusivas para o envio de cada mensagem, *threads* essas que seriam teoricamente executadas em paralelo. O número de *threads* processadas simultaneamente é limitado e *threads* excedentes são armazenadas numa fila. Sendo assim, caso o limite de *threads* simultâneas seja atingido durante ou antes da criação dessas *threads* responsáveis pelo envio, parte delas teria que aguardar na fila de execução e não haveria paralelismo nem simultaneidade.

Optou-se pelo envio das mensagens groupcast e broadcast para os nós móveis de forma sequencial dentro de uma única *thread* ao invés do envio por meio de uma *thread* exclusiva para cada nó móvel para manter a *Thread Pool* menos sobrecarregada e mais livre para execução de outras tarefas. Caso fossem utilizadas *threads* exclusivas grande parte da *Thread Pool* e de sua fila de execução ficaria ocupada com elas dado o grande número de nós conectados aos Gateways, sobrando menos espaço para as demais tarefas.

Além disso, observou-se empiricamente durante o desenvolvimento que grande parte dos problemas que ocorrem com o Gateway tinham como fonte o manejo e a alocação das *threads* pela *Thread Pool*. Com uma única *thread* não há esse custo de alocação, manejo e criação de múltiplas *threads* para uma única tarefa, a *Thread Pool* fica menos sobrecarrega e a perda de simultaneidade como se mostrou acima é irrelevante dado que não há garantia da mesma na arquitetura concorrente.

4.4 Projeto do PoA Manager para o ContextNet Kafka Core

O PoA Manager é o elemento do Core responsável pelo balanceamento de carga dos Gateways; e pela produção de uma lista atualizada dos endereços dos Gateways ativos. Este balanceamento de carga é feito após o ranqueamento dos Gateways através do envio de mensagens de *handoff* para nós móveis.

Como o PoA Manager consome apenas relatórios periódicos dos Gateways o fluxo de dados que ele recebe é pequeno quando comparado ao dos demais componentes do Core. Sendo assim ele pode ocupar seu tempo com atividades que demandem processamento. Sendo assim o PoA estende as funcionalidades do antigo e agrega novas.

Na nova versão o algoritmo de ranqueamento pode ser implementado pelo desenvolvedor; os parâmetros da rotina de envio de *Pings* pelo Gateway são determinados pelo PoA Manager e as janelas de recebimento dos *Pongs* podem ser dinâmicas e parametrizadas pelo desenvolvedor; a contabilização da latência produz dados estatísticos sobre as conexões; a rotina de verificação de Gateway ativos foi remodelada e pode também ser parametrizada pelo desenvolvedor; e o algoritmo de balanceamento foi inteiramente alterado.

4.4.1 Decisões de projeto e justificativa

Os seguintes capítulos descrevem a decisões de projeto e o fluxo dos *Records* pelo PoA Manager. A figura a seguir trás uma visão geral dos processos

do PoA Manager e como eles se encadeiam. Nos tópicos a seguir cada um deles será descrito em mais detalhes.

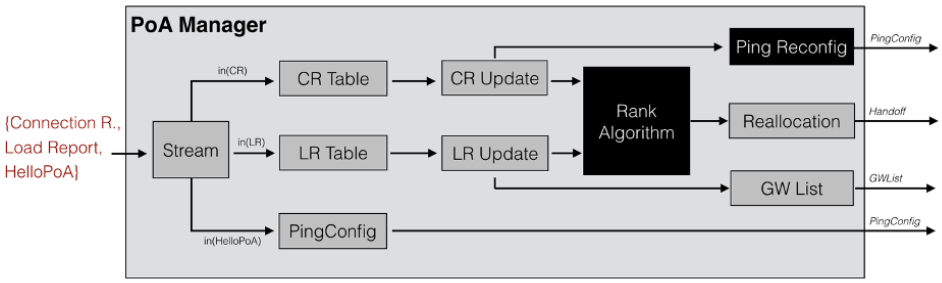

Figura 4.2: Visão Geral do PoA Manager

4.4.2 Configuração Inicial da rotina de *Pings*

Quando um Gateway se conecta ao Core ele informa ao PoA Manager sobre sua chegada através de um *record* "*HelloPoA*". Ao receber um "*HelloPoA*" o PoA Manager informa ao novo Gateway a frequência de envio das mensagens *Pings*, e as janelas de recebimento dos *Pongs* através de um *record* "*PingConfig*" endereçado ao novo Gateway. Essa configuração se encontra representada na 4.2 pelo fluxo que se inicia em "*Stream*" com o recebimento dos records, e que se desloca para o processo "*PingConfig*".

4.4.3 Algoritmo de Ranqueamento e Balanceamento de Carga

O balanceamento de carga dos Gateways, representado na 4.2 pelo processo "*Reallocation*" é realizado após o algoritmo de ranqueamento ranquear os Gateways e lhes atribuir notas. Esse algoritmo pode ser implementado pelo desenvolvedor.

O algoritmo de ranqueamento recebe como inputs os dados sobre as conexões – na 4.2 provenientes do processo "*CR Update*"; e os dados das cargas sobre os Gateways ativos – na 4.2 provenientes do processo "*LR Update*". Ele retorna como *output* um mapa contendo os Gateways ativos como chaves, e suas respectivas notas como valores. Quanto maior a nota de um Gateway mais apto este Gateway está para receber novas conexões.

O processo de balanceamento de carga recebe – como pode ser visto na 4.2 – o mapa de notas dos Gateways bem como os dados de conexão dos mesmos. A partir dessas informações ele calcula quantos nós idealmente deveriam estar em conectados a cada Gateway. Esse cálculo está representado na fórmula a seguir sendo g o número total de Gateways; $W(X)$ o peso do Gateway X; n

o número total de nós; e $In(X)$ o número ideal de nós móveis conectados ao Gateway X. $In(X) = nW(X)/(\sum_{i=1}^{g} W(i))$

Após calcular o número ideal de nós conectados a cada Gateway a realocação é iniciada e são geradas mensagens de reconexão para diversos nós móveis; reconexões essas que visam fazer com que os Gateways atinjam esse número previamente calculado.

4.4.4 Processamento dos *Load Reports*

Os Gateways enviam periodicamente os dados acerca da situação do hardware para o PoA Manager através do tópico "*LoadReport*". Os *LoadReport* carregam os seguintes dados: taxa de utilização e dimensão da *Thread Pool*; taxa de carga da CPU; memória total e fração disponível; memória da máquina virtual Java total e fração disponível.

O período de envio é fixo; igual para todos os Gateways e conhecido pelo PoA Manager. Na 4.2 os "*LoadReport*" chegam ao PoA Manager através do processo "*Stream*", e são armazenados em uma *Table* – representada pelo processo "*LR Table*". Também são registrados os momentos de chegada desses tópicos.

Periodicamente o PoA Manager inicia o processo de "*LR Update*" que é responsável por desserializar o conteúdo dos tópicos e por checar através dos registros de chegada quais Gateways estão ativos. Um Gateway é considerado inativo caso não nenhum "*Load Report*" seu tenha sido recebido nas últimas X janelas; sendo X informado pelo desenvolvedor e sendo o tamanho da janela igual ao período de envio dos "*Load Reports*" pelo Gateway.

Após verificar quais Gateways estão ativos o PoA Manager notifica os demais elementos conectados ao core através do tópico "*GWList*" – processo "*GW List*" da 4.2 - que Gateways são esses e qual o endereço dos mesmos. Esse tópico é recebido pelo Gateways e encaminhado a todos os nós móveis. O processo de "*LR Update*" ao final alimenta também o algoritmo de ranqueamento fornecendo os dados das cargas sobre os Gateways ativos como podemos ver representado na 4.2.

4.4.5 Processamento dos *Connection Report*

Os "*ConnectionReport*" responsáveis pelo transporte dos dados sobre as conexões móveis de cada Gateway chegam ao PoA Manager através do processo "*Stream*" representado na 4.2 e são armazenados em uma *Table* – representada pelo processo "*CR Table*". O processamento desses tópicos se

da no "*CR Update*", processo periódico responsável pela desserialização dos tópicos que se encontram na "*CR Table*"; e pela contabilização da latência. No "*ConnectionReport*" trafegam os dados acerca das conexões e são eles: horário que a mensagem *Ping* foi enviada aos nós móveis; lista com os tempos de resposta dos nós móveis; e lista com todos os nós móveis.

A partir dos dados enviados através do "*ConnectionReport*" (horário que a mensagem *Ping* foi enviada aos nós móveis; lista com os horários de chegada das respostas dos nós móveis; e lista com todos os nós móveis) contabilizam-se métricas chamadas "*Connection Statistics*" referentes a latência na comunicação de cada Gateway. Por latência entende-se o tempo do *round-trip time* entre o envio do *Ping* e o recebimento do *Pong*. Dentre as "*Connection Statistics*" estão a média, o desvio padrão, a variância, a mediana, o valor máximo e o valor mínimo da latência; e o percentual de nós cujo *Pong* foi recebido dentro da janela – "*Ping Rate*". Essas informações sobre as conexões são inputs geradas no "*CR Update*" alimentam – como pode-se ver na 4.2 - o algoritmo de ranqueamento.

Tais informações também são utilizadas no processo de "*Ping Reconfig*" responsável por avaliar se há necessidade de alteração na duração das janelas de Ping, e caso haja que alteração deveria ser. Este processo pode ser desativado e parametrizado pelo desenvolvedor. A partir do "*Ping Rate*" de cada Gateway o processo calcula o "*Ping Rate*" total da rede métrica utilizada no cálculo da alteração. O desenvolvedor informa para cada intervalo de "*Ping Rate*" por qual fator deve-se multiplicar a duração da janela. Caso esse fator seja um, a janela permanece a mesma.

4.5 Projeto do Mobile Temporary Disconnect para ContextNet Kafka Core

O novo MTD possui todas as funcionalidades da versão anterior acrescidas do descarte automático das mensagens. Ele armazena as mensagens que não puderam ser enviadas aos nós móveis destinatários por alguma interrupção na conexão entre eles e um dos Gateways. Como foi exposto na seção que descreve o projeto do Gateway; essas mensagens não enviadas são reinseridas no Core no tópico "*UnsentMessage*". O MTD coleta todos esses *Records* e os armazena até que note, através da leitura de um *"ConnectionReport"* que o no destinatário de se reconectou. Ao notar a reconexão o MTD recoloca o *Record* no Core endereçado ao destinatário.

4.5.1 Política de descarte das mensagens após “X"segundos no MTD

No antigo MTD as mensagens não enviadas ficavam indefinidamente armazenadas no MTD. No novo MTD as mensagens são descartadas após “X” segundos, sendo “X” informado pelo remetente da mensagem ou um valor *default* definido pelo desenvolvedor. O desenvolvedor também pode optar por padronizar “X” para todas as mensagens.

Essa política de descarte das mensagens além de diminuir a quantidade de memória que o MTD necessita para operar, impede que mensagens cuja entrega não faz mais sentido devido ao “atraso” trafeguem e ocupem desnecessariamente toda a estrutura do ContextNet e dos nós móveis. A implementação desta política torna o MTD mais complexo, e faz com que ele necessite de mais processamento para operar. Avaliou-se que as funcionalidade ganhas compensariam o aumento no demanda por processamento.

4.5.2 Aspectos de implementação

4.5.2.1 Arquitetura Padrão: *Half-Sync/Half-Async*

O MTD recebe continuamente mensagens não enviadas (através do tópico "*UnsentMessage*") e informações sobre os nós móveis conectados aos Gateways (através do tópico "*ConnectionReport*"). Um MTD que subsequente ao recebimento de uma mensagem não enviada realizasse o armazenamento da mesma em uma estrutura; e que subsequente ao recebimento de informações sobre nós móveis conectados ao um Gateway despachasse as mensagens armazenadas desses nós necessitaria utilizar *Locks* na manutenção da coerência dessa estrutura. O uso de uso de *Locks* implicaria em perda de performance, e portanto arquitetura do MTD foi projetada sem o uso dele. Sendo assim o padrão de projeto escolhido para implementação do MTD foi o "*Half-Sync/Half-Async*” que permite o armazenamento e o despacho paralelos ao processamento da chegada de mensagens e de informações, sem o uso de *Lock*.

Seguindo esse padrão, parte do processamento é síncrono a chegada dos *Records*; parte é assíncrono; e as partes se comunicam através de filas. A parte assíncrona consiste no armazenamento das mensagens não enviadas em uma estrutura (*unsentTable*); no descarte da *unsentTable* das mensagens cujo tempo de permanência já estourou; e no envio das mensagens da *unsentTable* cujos destinatários se re-conectaram. Essas três tarefas são executadas em sequência continuamente. A parte síncrona consiste na colocação das "*UnsentMes-*

sage"recebidas em uma fila de mensagens; e na colocação das informações sobre as conexões recebidas no "*ConnectionReport*"em uma fila de destinatários. Nesta fila de destinatários são enfileirados conjuntos de nós que se re-conectaram, cada conjunto é composto pela intercessão do conjunto nós móveis recebido pelo MTD com o conjunto de nós destinatários que possuam mensagens armazenadas no MTD; e diz respeito a um único Gateway.

A fila de mensagens é consumida durante a tarefa de armazenamento das mensagens não enviadas; e a fila de destinatários é consumida durante a tarefa de envio das mensagens cujos destinatários se re-conectaram.

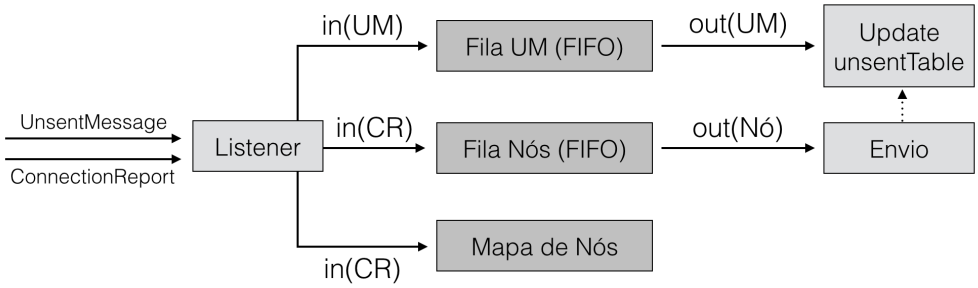

Figura 4.3: Filas MTD

As mensagens não enviadas são armazenadas na *unsentTable* e indexadas tanto pelo horário máximo de permanência quanto pela identificação do nó destinatário. Indexar ao horário máximo de permanência facilita a detecção daquelas mensagens que devem ser descartadas; e indexar a identificação do nó destinatário facilita o envio aos nós que se re-conectaram.

A figura 4.3 mostra a distribuição dos dados pelas filas no MTD. Os tópicos "*UnsentMessage*", "*ConnectionReport*"são continuamente consumidos pelo *Listener* MTD. Os *Records* "*UnsentMessage*"são colocados na "Fila UM"; a partir dos *Records* "*ConnectionReport*"são identificados e colocados na "Fila Nós"os nós destinatários que se re-conectaram ao *middleware*.

Continuamente as seguintes tarefas são executadas "*Update unsentTable*"e "Envio". A "*Update unsentTable*"consiste no consumo de "*UnsentMessages*"da "Fila UM"; inserção das mesmas na *unsentTable*; e descarte da *unsentTable* daquelas "*UnsentMessage*"cujo tempo de permanência se esgotou. O "Envio"consiste no consumo dos nós destinatários re-conectados da "Fila Nós"; obtenção junto a *unsentTable* das mensagens a serem enviadas para esses nós; e o envio dessas mensagens.

4.6 Projeto do Group Definer para o ContextNet Kafka Core

O Group Definer (GD) tem como principal função classificar em diferentes grupos os nós móveis baseando-se em informações de contexto enviadas pelos mesmos, e portanto, permitir que haja groupcast para nós móveis.

Para utiliza-la basta ao desenvolvedor implementar a interface *GroupSelection* componente responsável pela lógica de agrupamento.

No novo ContextNet a quantidade de dados que trafegam pelo Group Definer diminuiu drasticamente quando comparada com a versão anterior uma vez que que na nova versão o GD apenas consome tópicos *"ContextReport"*. A nova versão passou a permitiu que haja paralelização de nós GD com a mesma lógica caso haja necessidade; e que nós GD com lógicas diferentes coexistissem em um mesmo Core.

No novo ContextNet todas as mensagens recebidas pelo Gateway de nós móveis poderão trazer informações de Contexto. Estas informações são colocadas pelo Gateways no *Record "ContextReport"* e consumidas pelo Group Definer. Por meio dessa informações o algoritmo implementado no Group Definer decide a que grupos o nó móvel pertence, e coloca essa informação no Core através de um *Record "GroupAdvertisement"*. Os Gateways consomem os *"GroupAdvertisement"* e através de suas informações endereçam as mensagens de grupo.

4.6.1 Decisões de Projeto

O novo ContextNet Core permite que duas implementações com lógicas distintas do Group Definer coexistam. Na versão antiga a cada *"GroupReport"* referente a um determinado nó móvel que chegasse ao Gateway o mesmo desconsiderava as informações dos *"GroupReports"* antigos. Na nova versão os Group Definer possuem identificadores que trafegam como chave dos *"GroupReports Records"*. Por meio desses identificadores o Gateway consegue saber a origem de cada *"GroupReport"* recebido e manter em memória as informações do último enviado por cada Group Definer.

Essa possibilidade de coexistência de duas implementações distintas permite que duas aplicações com lógicas de agrupamento dos nós distintas também coexistam. A figura 4.4 ilustra esta situação. Nela há duas implementações do Group Definer (*"Group Definer 1"* e *"Group Definer 2"*) capazes de classificar os nós móveis em quatro grupos: *grupo11*, *grupo12*, *grupo21* e *grupo22*. A implementação *"Group Definer 1"* atende a *"Aplicação 1"* e classifica os nós nos grupos *grupo11* e *grupo12*; e a implementação *"Group Definer 2"* atende a

"Aplicação 2" e classifica os nós nos grupos *grupo11*, *grupo12*. Nesta configuração poderão existir existir nós como o "MN"da 4.4 que pertencem ao *grupo11* e *grupo22* e que recebem mensagens *groupcast* direcionadas a ambos os grupos.

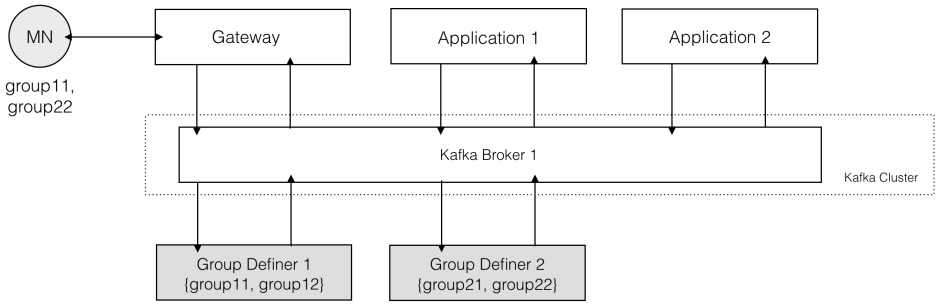

Figura 4.4: Filas GD

4.6.2 Aspectos de implementação

Assim como na antiga versão o desenvolvedor deve implementar uma Interface para fazer uso do Group Definer. Tal interface é responsável por receber os tópicos de contexto dos nós e a partir deles aplicar alguma lógica para definir a que grupo ou grupos um determinado nó móvel pertence.

5
Testes

Os testes projetados para o ContextNet Kafka Core são divididos em dois grupos: os que visam avaliar performance e os que visam avaliar a efetividade de alguns componentes. Entende-se por performance a capacidade dos componentes do ContextNet de dar suporte a uma comunicação efetivamente real-time em cenários com alto volume de mensagens trocadas e grande quantidade de nós móveis conectados. Ou seja, os testes de performance avaliam o quão escalável é a comunicação no ContextNet Core.

Dentre os testes de performance há aqueles que visam avaliar o impacto da paralelização no desempenho dos nós estáticos e aqueles que visam avaliar o desempenho do Gateway.

Na seção a seguir está descrita a infraestrutura dos testes e nas demais estão descritos os testes projetados, seus objetivos, o setup e as métricas coletadas.

5.1
Infraestrutura de Testes

Todos os testes foram realizados em máquinas virtuais da *Cloud-DI*. Ao todo foram utilizadas 8 máquinas virtuais com 2 processadores, sendo 5 com 16 GB de RAM e 3 com 24 GB de RAM. Todas as máquinas virtuais estão conectadas em topologia de barramento.

Os Gateways foram alocados nas máquinas com 24GB de RAM por serem os componentes com maior necessidade de memória RAM devido ao alto número de conexões MR-UDP.

Empiricamente notou-se que essas conexões ocupam muito espaço de memória, principalmente no momento de conexão. Portanto o quanto mais memória RAM disponível há para o Gateway mais conexões ele consegue estabelecer.

Em quase todos os testes há a seguinte distribuição: três máquinas virtuais de 16GB são utilizadas para a simulação dos nós móveis; uma máquina de 16GB abriga a estrutura do Kafka; e uma máquina de 16GB abriga o processamento da aplicação e os demais componentes do ContextNet Kafka Core (Group Definer, MTD e PoA Manager). Nos testes em que há uma

distribuição de máquinas virtuais diferente desta mencionada, ela será descrita especificamente na subseção.

Todos os componentes do ContextNet Kafka Core foram "dockerizados"para facilitar a implementação no ambiente de testes e a repetição dos experimentos por qualquer desenvolvedor. Todas as imagens utilizadas nos testes estão publicadas no repositório do LAC e serão identificadas nos capítulos seguintes.

Para todos os testes descritos nas seções abaixo foram realizados 5 experimentos.

5.2 Configuração da Simulação dos Nós Móveis

Para realização dos testes a seguir descritos foram simuladas várias quantidades de nós móveis. A forma como essa simulação é feita interfere diretamente nos resultados encontrados e, portanto, antes de dar início aos testes foram realizados experimentos com diversas configurações diferentes de nós móveis para que se pudesse avaliar qual seria a mais adequada para realização dos mesmos.

Por configurações entende-se a disposição dos containers que simulam os nós móveis, bem como o próprio nó móvel e o intervalo de tempo entre pedidos de conexão subsequentes dos nós móveis quaisquer.

Para a simulação os nós móveis foram divididos em grupos de tamanhos iguais. Cada grupo é simulado através de um executável Java. Foi criada e publicada uma imagem desse executável, imagem que é executada em um container durante a simulação.

Nos experimentos feitos foi utilizado como métrica o *round-trip time* das mensagens enviadas dos nós para um nó aplicação no Core. Buscava-se uma configuração que suportasse o comportamento do MR-UDP, e que influenciasse o mínimo nas medidas do *round-trip time*.

Observou-se que quanto maior a quantidade de containers menor o *round-trip time*. O número de containers é limitado pela memória RAM da máquina. O tamanho dos containers não aumenta proporcionalmente com o aumento do número de nós executados dentro dele, pois cada container executa uma máquina virtual Java cujo tamanho não está unicamente relacionado a quantidade de nós móveis simulados que ela executa. O maior número de containers suportados por uma máquina virtual foi 40 e, portanto, foram utilizados 40 containers em todas as simulações.

Na configuração adotada para simulação cada nó móvel possui n threads em uma *Thread Pool* a sua disposição. Como o número total de threads

simultâneas em uma máquina é limitado, n também é limitado. Observou-se que quanto maior n menor o *round-trip time*. O maior n suportado por uma máquina foi 4 e, portanto, cada nó simulado tinha à sua disposição 4 threads.

Observou-se também que o Gateway tem um limite no número de conexões que é capaz de suportar simultaneamente por tempo. As conexões MR-UDP consomem um espaço significativo de memória para serem estabelecidas, e então, quando muitas são estabelecidas simultaneamente a máquina virtual Java fica sem memória e começa a derrubar as conexões já existentes. Para que o processo de conexão dos nós móveis com o Gateway fosse possível fixou-se um intervalo mínimo entre o estabelecimento de duas conexões subsequentes.

Foram realizados experimentos para descobrir o menor intervalo possível entre as conexões, e chegou-se no valor de 1000 milissegundos. Como nas simulações temos 40 containers independentes se conectando simultaneamente com o Gateway, para que o intervalo médio entre duas conexões se mantivesse em 1000 milissegundos, o intervalo entre duas conexões solicitadas por nós móveis simulados de um mesmo container teria que ser de 40000 milissegundos.

5.3 Testes de Escalabilidade da comunicação básica (Inbound)

Os testes de comunicação básica são testes de performance e possuem um cenário. Neles os nós móveis iniciam a comunicação com o Gateway enviando mensagens para uma aplicação a ele conectado, e permanecem conectados durante todo o experimento enviando mensagens para a aplicação a uma taxa constante. Essa aplicação responde a todas as mensagens dos nós móveis. Conectado ao Core também está um nó MTD.

Estes testes têm como métricas o *round-trip time* (RTT) das mensagens enviadas pelos nós móveis e a taxa de entrega (TE) das mesmas. Tanto o RTT quanto o TE são medidos pelo processo que controla os nós móveis.

5.3.1 Cenário 1

O Cenário 1 é o cenário base para todos os demais testes, a partir dos resultados dele alguns dos cenários serão avaliados. Os testes nesse cenário têm como objetivo verificar o quão veloz (RTT) e confiável (TE) é a comunicação nó móvel – Core – nó móvel a uma determinada taxas de envio de mensagens (TM) para diferentes quantidades de nós (QTD); quantidade de brokers Kafka (BK); e quantidade de Gateways (GW).

Nos testes desse cenário variamos a quantidade de nós (1000, 5000); a quantidade de brokers Kafka (1,3) e consequentemente a quantidade de nós

aplicação respondendo as mensagens dos nós; e a quantidade de Gateways (1, 3).

A taxa de envio de mensagens escolhida foi de 4 por minuto por nó, taxas também utilizada em [1] para avaliar o desempenho da ContextNet original. Além de ser uma taxa realística para aplicações IoT; sua utilização permite uma comparação mais justa entre os resultados do ContextNet novo e do original.

O período de observação se iniciou quando todos os nós estavam conectados ao Gateway e terminou quando todos os nós já haviam enviados 500 mensagens.

Ao final deste teste avaliamos o quanto cada uma das variáveis influencia no RTT e no TE; avaliamos se o desempenho do novo ContextNet para comunicação Inbound é satisfatório; se a comunicação Inbound é confiável; se o desempenho se mantém constante ao longo do tempo; e compararemos as duas versões do ContextNet. Consideramos satisfatório RTT inferiores a 1000 milissegundos.

5.3.2 Resultados Cenário 1

Para o Cenário 1 foram selecionadas 8 configurações diferentes variando-se o número de nós (NM), gateways (GW), aplicações e brokers Kafka (BK/APP). Uma dessas configurações (5000 MN, 3 GW, 1 BK/APP) não conseguiu ser executada, pois falhas nas conexões MR-UDP ocorriam durante suas tentativas de execução.

ID	NM	BK/APP	GW	Média RTT	Des Padrão RTT	TE
1	1000	1	1	14,307	7,355	100,00%
2	1000	3	1	12,016	5,288	100,00%
3	5000	1	1	49,455	122,506	100,00%
4	5000	3	1	11,683	15,933	100,00%
5	1000	1	3	16,270	116,187	99,97%
6	1000	3	3	15,226	17,437	100,00%
8	5000	3	3	21,345	94,302	93,22%

Figura 5.1: Resultados Testes Cenário 1

A Figura 5.1 traz um resumo dos resultados dos testes do Cenário 1. Cada uma das configurações está identificada por um ID (primeira coluna da Figura 5.1) que será utilizado ao longo do capítulo para identificá-las. Para cada uma das configurações a Figura 5.1 apresenta a média (Média RTT) e o desvio padrão (Des Padrão RTT) do round-trip-time de todas as mensagens trafegadas; e a taxa de entrega das mesmas (TE).

Analisando a Figura 5.1 conclui-se que o desempenho do novo ContextNet para comunicação Inbound é satisfatório uma vez que em seu pior desempenho o RTT médio é 49,45 milissegundos, consideravelmente inferior a 1000 milissegundos; e que a média do RTT para todas as configurações é 20,04 milissegundos.

Conclui-se também a partir da Figura 5.1 que o ContextNet é extremamente confiável para cenários com até 5000 nós móveis por Gateway. A TE foi de 100% em 5 das 7 configurações testadas e ficou acima de 99,9% em outra. Na configuração 8, única com 15000 nós móveis conectados no sistema, a TE diminuiu para 93,22%, o que pode indicar que em configurações com um número maior de nós móveis diminua a confiança no ContextNet Kafka Core. Pode-se observar que apenas o número de nós móveis influencia diretamente na TE.

Pelo exposto na Figura 5.1 comparando os RTT das configurações 3 e 4 pode-se concluir que a paralelização da aplicação (ocorrida em 4) pode contribuir para uma melhora no desempenho. A média do RTT em 3 é de 49,45 milissegundos e o desvio padrão 122,50 enquanto em 4, com o mesmo número de nós móveis (5000), a média do RTT é 11,68 milissegundos, e o desvio padrão 15,93.

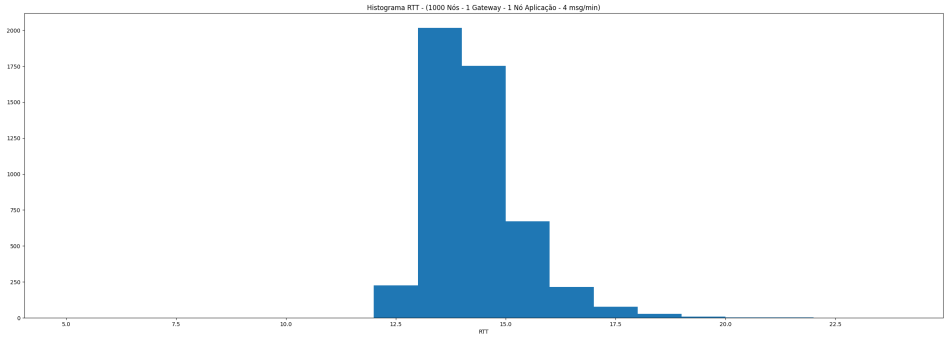

Figura 5.2: Histograma Configuração 1

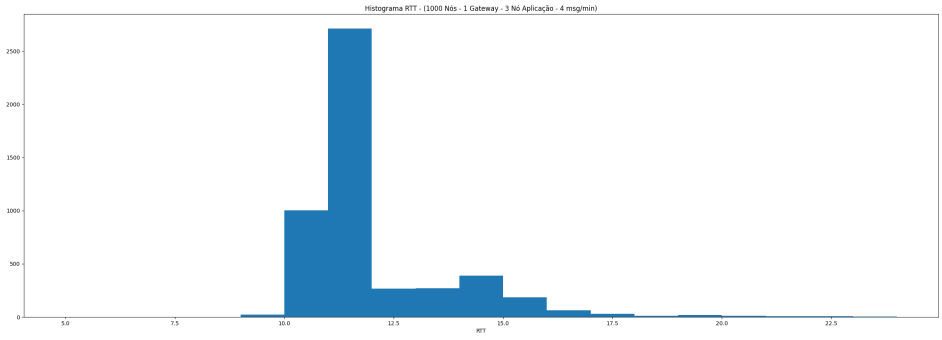

Figura 5.3: Histograma Configuração 2

Dado que tanto a média quanto o desvio padrão do RTT das configurações 1 e 2 estão muito próximos, para compararmos o desempenho das configurações 1 e 2 também observamos os histogramas das Figuras 5.2 e 5.3 e analisando a Figura 5.4.

Os histogramas das Figuras 5.2 e 5.3 trazem a distribuição do RTT médio de cada nó. Por 5.2 sabemos que o RTT médio da grande maioria dos nós da configuração 1 está entre 12,5 e 15 milissegundos; enquanto por 5.3 sabemos que a grande maioria dos nós da configuração 2 está entre 10 e 12,5 milissegundos. Pode-se concluir que a comunicação é mais veloz para os nós da configuração 2 do que para os nós da configuração 1.

RTT/Exp ID	1	2
<= 10	15,20%	31,39%
<= 12,5	13,03%	28,73%
<= 15	47,81%	31,64%
<= 17,5	10,85%	4,31%
<= 20	8,77%	2,24%
<= 22,5	1,64%	0,58%
> 22,5	2,69%	1,10%

Figura 5.4: Comparação Configurações 1 e 2

A Figura 5.4 traz a distribuição de todos os RTT das configurações 1 e 2 em faixas de 2,5 milissegundos cada. Pela Figura 5.4 sabe-se que 15,2% dos RTT da configuração 1 foram menores que 10 milissegundos; e 31,39% dos RTT da configuração 2 foram menores que 10 milissegundos. Enquanto na configuração 2 60,12% dos RTT são menores que 12,5 milissegundos, apenas 28,23% são na configuração 1. Apenas 8,24% dos RTT da configuração 2 são maiores que 17,5 milissegundos; já na configuração 2 23,96% dos RTT são.

Por estas observações pode-se concluir que o desempenho da configuração 2 é superior ao da configuração 1, e dadas as características das configurações analisadas, pode-se atribuir essa superioridade a paralelização das aplicações.

Da impossibilidade da execução da configuração 7 com 5000 MN, 3 GW, 1 BK/APP exposta acima; e do sucesso da execução da configuração 8 pode-se concluir que a paralelização da aplicação (única diferença entre as configurações 7 e 8) pode ser determinante na melhora de desempenho do ContextNet; e até na viabilidade de alguns cenários.

As conclusões acerca das comparações acima feitas entre as configurações 1 e 2; e 7 e 8 corroboram o que havia sido concluindo na comparação entre as configurações 3 e 4.

Com base nas observações das Figuras 5.1 e 5.5 podemos comparar as configurações 5 e 6. A média do RTT delas é próxima, contudo, o desvio padrão

RTT/Exp ID	1	2	3	4	5	6	8
<= 10 ms	15,20%	31,39%	19,56%	60,46%	16,50%	24,21%	14,29%
<= 20 ms	80,47%	66,92%	56,32%	35,19%	77,59%	65,73%	64,28%
<= 30 ms	3,03%	1,16%	6,07%	2,34%	3,63%	5,99%	13,20%
<= 40 ms	0,47%	0,19%	2,41%	1,00%	0,78%	1,69%	3,55%
<= 50 ms	0,34%	0,13%	1,80%	0,54%	0,50%	0,97%	1,86%
<= 100 ms	0,42%	0,17%	4,23%	0,37%	0,78%	1,11%	1,86%
> 100 ms	0,07%	0,03%	9,60%	0,10%	0,22%	0,31%	0,96%

Figura 5.5: Comparação Configurações Inboud

da configuração 5 é significativamente maior, ou seja, nela há uma dispersão maior dos RTT. A Figura 5.5 traz a distribuição de todos os RTT de todas as configurações em faixas de 10 e 50 milissegundos. Por 5.5 notamos que não há divergências relevantes nas distribuições das configurações 5 e 6.

Para verificar se o aumento do número de Gateways influencia no desempenho comparamos os pares de configurações 1 e 5; e 2 e 6. Por 5.1 sabemos que 1 e 5; e 2 e 6 possuem médias de RTT muito próximas, com as configurações com mais de 1 Gateway com médias de RTT ligeiramente maiores. Em 5.5 notamos que as distribuições de RTT em faixas são muito semelhantes entre os pares 1 e 5; e 2 e 6. Por estas observações pode-se afirmar que o aumento do número de Gateways por si só não piora significativamente o desempenho do ContextNet.

Para verificar se o desempenho do ContextNet se mantém constante ao longo do tempo, as Figuras 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 foram plotadas. Nelas está plotado o RTT médio por ordem de envio. A ordem varia de 0 a 500, pois foram enviadas 500 mensagens por nó.

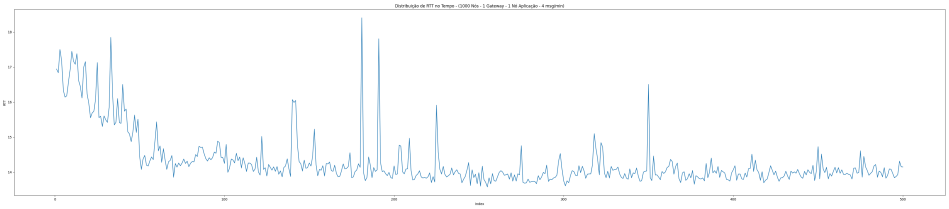

Figura 5.6: Distribuição no Tempo - Configuração 1

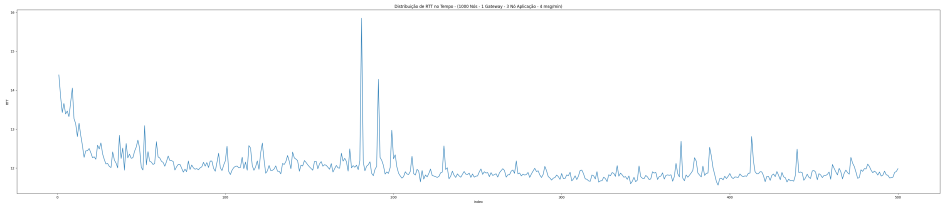

Figura 5.7: Distribuição no Tempo - Configuração 2

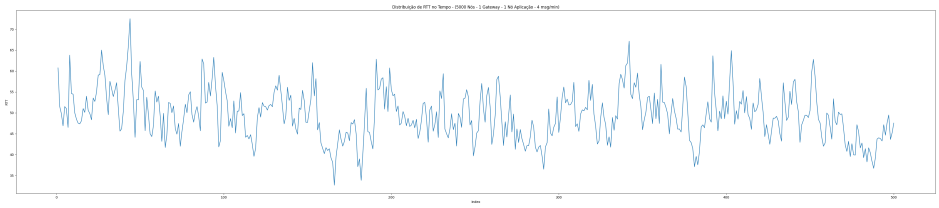

Figura 5.8: Distribuição no Tempo - Configuração 3

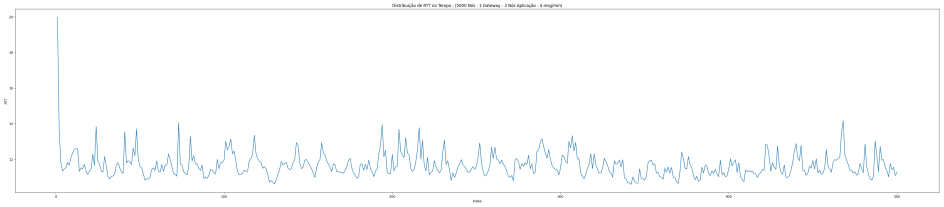

Figura 5.9: Distribuição no Tempo - Configuração 4

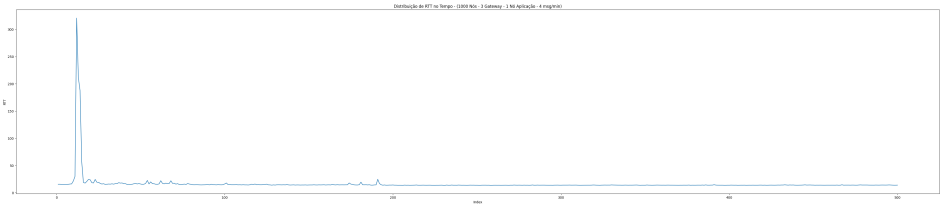

Figura 5.10: Distribuição no Tempo - Configuração 5

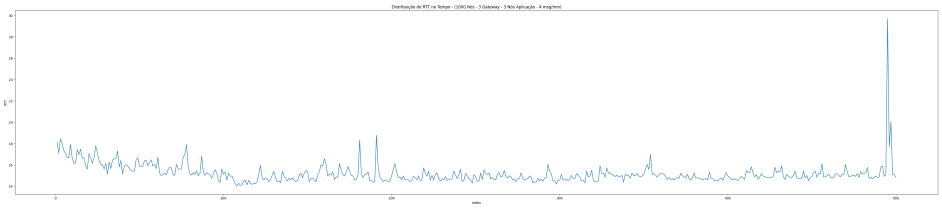

Figura 5.11: Distribuição no Tempo - Configuração 6

Observando as Figuras 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 nota-se que não há piora no desempenho do ContextNet ao longo do tempo. O desempenho permanece constante.

Em [9] o desempenho do antigo ContextNet está documentado. O RTT variava entre 20 e 45 milissegundos, e foi tido como satisfatório. Sendo assim, pode-se concluir pela Figura 5.5 que o desempenho do novo ContextNet frente ao antigo é satisfatório. Os cenários afins de ambos os ContextNet não foram diretamente comparados pois não há como fazer essa comparação direta entre simulações totalmente distintas.

Dado o exposto acima, conclui-se que a comunicação Inbound do novo ContextNet é confiável; que seu desempenho é satisfatório, se mantém constante ao longo do tempo, e é comparável ao do antigo ContextNet. Também concluímos que a paralelização das aplicações pode melhorar o desempenho

em determinadas situações; que o aumento do número de nós móveis pode prejudicar a TE, e consequentemente na confiabilidade; e que o aumento do número de Gateways por si só não piora significativamente o desempenho do ContextNet.

5.4 Testes de Escalabilidade da comunicação e balanceamento de carga (Outbound)

Os testes de comunicação básica são testes de performance e funcionalidade, e foram divididos em três cenários: o primeiro (Cenário 2) com nós móveis estáticos e sem o MTD; o segundo (Cenário 3) com nós móveis que se desconectavam e se reconectavam com uma determinada frequência e sem o MTD; o terceiro (Cenário 4) com nós móveis que se desconectam e se reconectam com uma determinada frequência e com MTD.

Em todos os cenários os nós móveis iniciam a comunicação com o Gateway enviando uma mensagem para uma aplicação a ele conectado. Após receber essa comunicação inicial a aplicação passa a enviar com determinada taxa (TM) mensagem ao nós móvel. Conectado ao Core também pode estar um nó MTD.

Estes testes têm como métricas o round-trip time (RTT) das mensagens enviadas pela aplicação aos nós móveis e a taxa de entrega (TE) das mesmas. Tanto o RTT quanto o TE são medidos pelos processos que controlam a aplicação.

Ao final deste teste avaliamos se o desempenho do novo ContextNet para comunicação Outbound é satisfatório; comparamos o desempenho das comunicações Inbound e Outbound; avaliamos o quanto o MTD torna a comunicação mais confiável e o quanto ele influência no desempenho.

Para todos os cenários a taxa de envio de mensagens escolhida foi de 4 por minuto por nó, mesma do Cenário 1 para facilitar comparações.

O período de observação se iniciou quando todos os nós estavam conectados ao Gateway e terminou quando a aplicação havia enviado 500 mensagens.

5.4.1 Cenário 2

O Cenário 2 é o cenário base para todos os demais testes, a partir dos resultados dele alguns dos cenários listados neste capítulo serão avaliados.

Os testes nesse cenário têm como objetivo verificar o quão veloz (RTT) e confiável (TE) é a comunicação Core – nó móvel – Core a uma determinada

taxas de envio de mensagens (TM) para uma determinada quantidades de nós (QTD); quantidade de brokers Kafka (BK); quantidade de Gateways (GW).

Nos testes desse cenário utilizamos 5000 nós; um broker Kafka e consequentemente um nó aplicação enviando mensagens aos nós; e um Gateway.

O Cenário 2 também será comparado ao Cenário 1. Ambos possuem as mesmas métricas e as mesmas variáveis, contudo, tratam de tipos de comunicação distintas. Essa comparação visa observar se há e qual é a diferença de performance entre a comunicação Inbound e Outbound no ContextNet Kafka Core.

5.4.2 Cenário 3

Os resultados do Cenário 3 serão comparados com os resultados do Cenário 2. Ambos possuem as mesmas métricas e as mesmas variáveis.

Os testes neste cenário têm como objetivo verificar o quão veloz (RTT) e confiável (TE) é a comunicação Core - nó móvel – Core em um ambiente onde há falhas na rede que conecta os nós móveis ao Gateway. Para simular essas falhas parte dos nós móveis (DES) se desconecta e se reconecta com uma determinada frequência.

Nos testes desse cenário utilizamos 5000 nós; um broker Kafka e consequentemente um nó aplicação enviando mensagens aos nós; e variamos DES (30% e 70%). A cada desconexão os nós aguardam 15000 milissegundos para se reconectar.

Ao final deste teste avaliaremos o impacto dessas reconexões no RTT e na TE comparando seus resultados com os do Cenário 2.

5.4.3 Cenário 4

Os resultados do Cenário 4 serão comparados com os resultados dos cenários 3 e 4. Eles possuem as mesmas métricas e as mesmas variáveis. O MTD está ativo neste cenário.

Comparando os resultados do Cenário 4 com os do Cenário 3 verificamos o quão efetivo é o MTD; e qual o impacto dele na RTT e na TE.

5.5 Resultados Cenário 1, Cenário 2

Neste subcapítulo será avaliado o desempenho da comunicação Outbound (Cenário 2, configuração mtd_1); e serão comparados os Cenários 1 e 2.

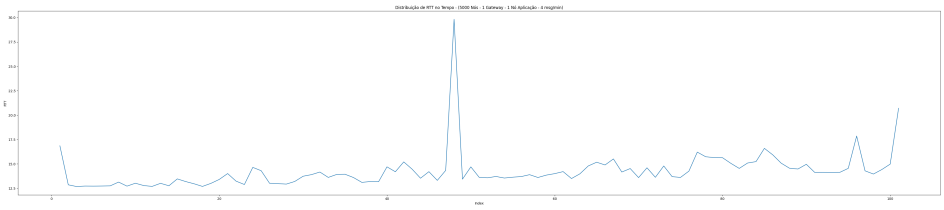

Figura 5.12: Distribuição no Tempo - Configuração mtd_1

Para verificar se o desempenho do ContextNet se mantém constante para a comunicação Outbound ao longo do tempo, a Figuras 5.12 foi plotada. Nela está plotado o RTT médio por ordem de envio. A ordem varia de 0 a 100, pois foram enviadas 100 mensagens por nó. Observando a Figuras 5.12 nota-se que não há piora no desempenho da comunicação Outbound do ContextNet ao longo do tempo. O desempenho permanece constante.

Pelo exposto em 5.17 sabemos que TE para mtd_1 é superior a 99,9%, ou seja, a comunicação Outboud do ContextNet é confiável. Ainda pelo exposto em 5.17 sabemos que o RTT de mtd_1 é de 14,25 milissegundos, e que o desvio padrão 13,62. Conclui-se, portanto, que o desempenho do novo ContextNet para comunicação Outbound é satisfatório.

RTT/Exp ID	mtd_1	3
<= 10	18,43%	19,56%
<= 20	73,26%	56,32%
<= 30	3,83%	6,07%
<= 40	1,93%	2,41%
<= 50	0,88%	1,80%
<= 100	1,30%	4,23%
> 100	0,37%	9,60%

Figura 5.13: Comparação Configurações mtd_1 e 3

Para comparar as performances das comunicações Inbound e Outbound foi selecionada a configuração 3 do Cenário 1; e a mtd_1 do Cenário 2. Elas possuem o mesmo número de nós móveis conectados; ambas estão sendo executadas um nó apenas; e a frequência de envio de mensagens é a mesma nas duas.

Segundo a Figura 5.1 o RTT médio da configuração 3 é 49,45 milissegundos, significativamente superior ao da configuração mtd_1 que é de 14,25 milissegundos segundo a Figura 5.17. Observando os histogramas das figuras 5.14 e 5.15 que trazem a distribuição do RTT médio de cada nó notamos que em mtd_1 há uma quantidade muito maior de nós cujo RTT médio é inferior a 12,5 milissegundos, o que não se verifica na configuração 3. Pela Figura 5.13 sabemos que mais de 91% das mensagens que trafegam em mtd_1 tem RTT inferior a 20 milissegundos, enquanto na configuração 3 esse percentual cai

para 75%. Sendo assim, há indícios de que a comunicação Outbound tem uma performance melhor que a Inbound, contudo, mais configurações deveriam ser testadas para corroborar essa hipótese.

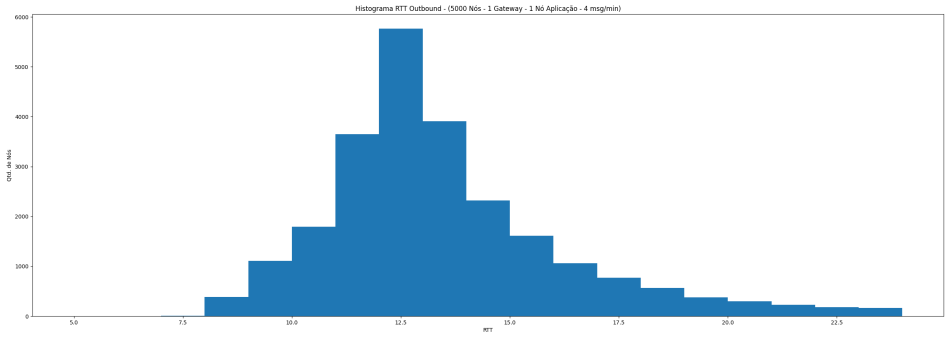

Figura 5.14: Histograma Configuração mtd_1

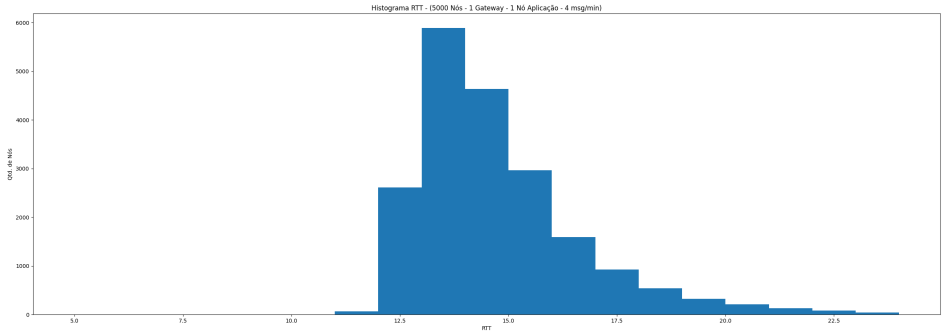

Figura 5.15: Histograma Configuração 3

5.5.1 Resultados Cenário 2, Cenário 3, Cenário 4

As configurações para os Cenários 2 (mtd_1), 3 (mtd_3, mtd_5), e 4 (mtd_2, mtd_4) estão expostas e identificadas na Figura 5.16. Em todas as configurações 5000 nós (NM) estão conectados e há um Gateway ativo (GW). Em 5.16 também está especificado se o MTD está ativo (MTD); o tempo que os nós móveis permanecem desconectados a cada desconexão (Tempo Desc.); e a parte (em percentual) dos nós móveis que desconecta.

ID	NM	GW	MTD	Tempo Desc.	%MN Desc.
mtd_1	5000	1	Inativo	0	0
mtd_2	5000	1	Ativo	30000	30
mtd_3	5000	1	Inativo	30000	30
mtd_4	5000	1	Ativo	30000	70
mtd_5	5000	1	Inativo	30000	70

Figura 5.16: Configurações dos Cenários 2, 3, 4

ID	TE	TE Desc	100%TE Desc	RTT Fixo	Desvio RTT Fixo	RTT Desc	Desvio RTT Desc
mtd_1	99,93%			14,259	13,625		
mtd_2	99,99%	99,96%	99,32%	33,370	82,147	3862,168	10095,458
mtd_3	95,81%	85,23%	0,00%	21,376	39,602	24,258	40,620
mtd_4	99,98%	99,97%	99,36%	71,426	167,212	3450,551	9208,724
mtd_5	90,07%	85,27%	0,00%	32,810	55,228	37,020	61,339

Figura 5.17: Resultados dos Cenários 2, 3, 4

Na Figura 5.17 são apresentados os resultados dos testes com os Cenários 2, 3, e 4. Ela apresenta para cada configuração a taxa de entrega total das mensagens (TE); a taxa de entrega das mensagens endereçadas aos nós que desconectam (TE Desc); a parte (percentual) dos nós que desconectam cuja taxa de entrega é de 100%; a média (RTT Fixo) e o desvio padrão (Desvio RTT Fixo) do round-trip-time dos nós fixos; e a média (RTT Desc) e o desvio padrão (Desvio RTT Desc) do round-trip-time dos nós que desconectam.

RTT/Exp ID	mtd_2	mtd_3	mtd_4	mtd_5
<= 20	64,88%	73,37%	58,86%	61,49%
<= 50	11,67%	17,61%	10,77%	20,36%
<= 100	4,13%	6,20%	4,65%	10,42%
<= 200	2,05%	2,00%	3,41%	5,23%
<= 500	1,82%	0,71%	4,35%	2,26%
<= 1000	0,61%	0,09%	2,69%	0,23%
<= 2000	0,04%	0,02%	0,69%	0,02%
> 2000	14,80%	0,00%	14,56%	0,00%

Figura 5.18: Comparação Configurações Outbound

A Figura 5.18 traz a distribuição de todos os RTT das configurações mtd_2, mtd_3, mtd_4 e mtd_5 em faixas de milissegundos. Pela Figura 5.18 sabe-se que por exemplo que 64,88% dos RTT da configuração mtd_2 foram menores que 20 milissegundos; e 14,80% foram maiores que 2000 milissegundos.

Comparando mtd_1 com mtd_3 e mtd_5 avaliamos o impacto das falhas na rede no RTT e na TE. Segundo a Figura 5.17 a TE caiu de 99,93% em mtd_1 para 95,81% em mtd_3; e 90,07% em mtd_5. Observou-se também um aumento significativo no RTT até dos nós fixos quando há falhas na rede: em mtd_1 a média foi 14,25 milissegundos já em mtd_3 21,37 e em mtd_5 32,81. Ou seja, falhas na rede impactam negativamente até no desempenho dos nós que não falham, e diminuem a confiabilidade.

Comparando mtd_1 com mtd_2 e mtd_4 avaliamos o impacto do MTD e das falhas de rede no RTT e na TE. Segundo a Figura 5.17 a TE se manteve praticamente idêntica em mtd_1, mtd_2 e mtd_4 e de 99,9%, ou seja, o MTD tornou a comunicação de um ambiente com falhas tão confiável quanto a de um ambiente sem. Em mtd_2 e mtd_4 há um aumento significativo no RTT quando comparado com mtd_1, contudo, mesmo com esse aumento o desempenho da comunicação ainda é satisfatório. Como pode-se observar na

Figura 5.18 mais de 58% das mensagens trafegadas em mtd_2 e mtd_4 tem RTT inferior a 20 milissegundos.

Comparando os pares de configurações mtd_2 e mtd_3; e mtd_4 e mtd_5 pode-se avaliar o quão efetivo é o MTD; e qual o impacto dele na RTT e na TE em cenários de falhas. Observou-se um aumento do RTT Desc em mtd_2 e mtd_4 que passou de dezenas de milissegundos em mtd_3 e mtd_5 para milhares de milissegundos; comportamento esperado uma vez que em mtd_2 e mtd_4 os nós recebem mensagens endereçadas a eles em momentos de desconexão.

Segundo a Figura 5.17 entre os nós que desconectam, a taxa de entrega (TE Desc) aumentou de aproximadamente 85% em mtd_3 e mtd_5 para 99,96% em mtd_2 e mtd_4. Além disso, em mtd_2 e mtd_4 mais de 99% dos nós que desconectavam receberam 100% das mensagens a eles endereçadas enquanto em mtd_3 e mtd_5 esse percentual é 0%. Ou seja, com o MTD houve um aumento considerável na confiabilidade da comunicação em um cenário onde há falhas na rede.

Analisando 5.18 observamos que a distribuição dos RTT nas faixas até 2000 milissegundos é muito semelhante entre mtd_2, mtd_3, mtd_4, mtd_5. Na faixa acima de 2000 milissegundos há uma grande diferença entre as configurações que possuem MTD ativo (mtd_2 e mtd_4); e as que não possuem (mtd_3 e mtd_5). Em mtd_2 e mtd_4 mais de 14% das mensagens possuem RTT acima de 2000 milissegundos; enquanto em mtd_2 e mtd_4 não há RTT com essa duração. Essas mensagens cujo RTT tem duração acima de 2000 milissegundos muito provavelmente são mensagens reencaminhadas pelo MTD, e portanto, mensagens cujo atraso já era esperado.

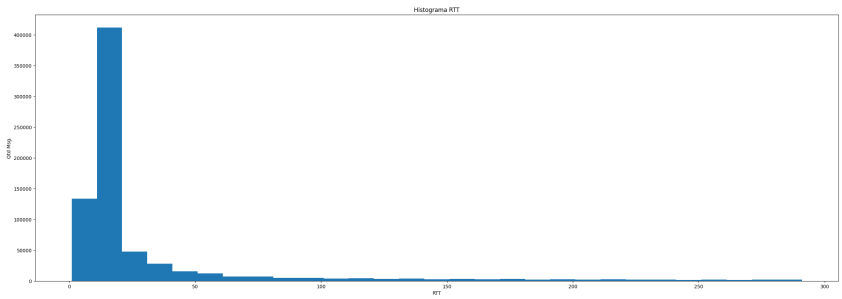

Figura 5.19: Histograma RTT Médio Nós Fixos - Configuração mtd_4

Observando a semelhança entre as distribuições do RTT médio dos nós Fixos expostas nas figuras 5.19 e 5.20, podemos afirmar que o comportamento dos nós fixos é muito parecido com e sem o MTD ativo apesar do aumento significativo no RTT médio total com MTD como exposto em 5.17.

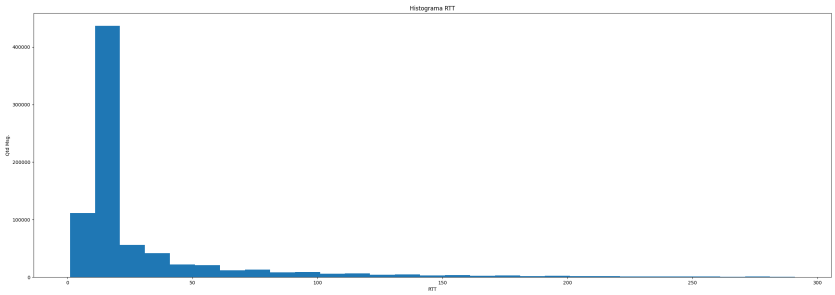

Figura 5.20: Histograma RTT Médio Nós Fixos - Configuração mtd_5

Comparando as configurações mtd_3 e mtd_5, notamos que mesmo com o aumento da quantidade de nós que desconectam a TE Desc se mantém praticamente constante como exposto em 5.17. O mesmo pode ser observado comparando as configurações mtd_2 e mtd_4, o que nos leva a concluir que a quantidade de nós que desconectam tem pouca influência sobre a TE Desc.

5.6 Testes de comunicação em grupo (Outbond)

Os testes de comunicação em grupo são testes de performance com nós móveis estáticos cujas informações de contexto variam.

Conectado ao Core também estão até três nós Group Definer (GD1, GD2, GD3), com lógicas de pertencimento diferentes e não exclusivas. Um nó pode estar em qualquer combinação dos grupos por GD1, GD2, GD3 definidos a qualquer momento.

Em todos os cenários a aplicação envia mensagens a cada um dos grupos de nós móveis com determinada periodicidade. Os nós móveis respondem as mensagens enviada pela aplicação; e enviam com determinada periodicidade suas informações de contexto ao Gateway.

Estes testes têm como métricas o round-trip time (RTT) das mensagens enviadas pela aplicação; o intervalo entre a entrega de determinada mensagem de grupo a diferentes nós móveis que pertençam a mesma máquina (TA); o percentual de entrega correta das mensagens de grupo (TC). O RTT é medido pelo processo que controla a aplicação enquanto o TA e o TC são medidos pelo processo que controla os nós móveis de uma determinada máquina. Entende-se que uma entrega de uma mensagem de grupo é correta se no momento em que ela ocorre o grupo ao qual ela foi endereçada é o mesmo grupo ao qual o nó móvel pertence.

Os testes nesse cenário têm como objetivo verificar o quão veloz (RTT), "simultânea"(TA) e precisa (TC) é a comunicação em grupo para uma determinada quantidade de nós a uma determinada taxa de envio de mensagens

(TM). Nos testes deste cenário (Cenário 5) foram conectados 5000 nós; e a taxa de envio de mensagens (TM) foi de 4 por minuto. Foi utilizado um broker Kafka e um Gateway.

No cenário 5 para cada nó a cada 60 segundos há mudança nas informações de contexto e consequentemente mudança nos grupos aos quais o nó pertence.

Foram feitos testes com um Group Definer (GD1) e com três Group Definires (GD1, GD2, GD3). Cada um dos Group Definires define dois grupos e cada nó só pode pertencer a um desses dois grupos num dado momento. Ou seja, nos testes em que há três Group Definires os nós pertencem a todo momento a três grupos; nos testes em que há um Group Definer os nós pertencem a todo momento a um único grupo.

Comparando os testes com um e três Group Definires pretende-se avaliar se a existência de mais de um Group Definer ativo influência na performance (RTT), na simultaneidade (TA) e na precisão (TC).

O período de observação se iniciou quando todos os nós estavam conectados ao Gateway e terminou quando a aplicação havia enviado 100 mensagens para cada grupo.

5.6.1 Resultados Cenário 5

Para o Cenário 5 foram selecionadas 2 configurações diferentes, uma com um Group Definer (1GD); e outra com três Group Definer (3GD) com lógicas distintas.

A Figura 5.21 trás um resumo dos resultados dos testes do Cenário 5. Para cada uma das configurações a Figura 5.21 apresenta a média (Média RTT) e o desvio padrão (Desvio RTT) do round-trip-time de todas as mensagens trafegadas; o percentual de entrega correta das mensagens de grupo (TC); e a média (Média TA) e o desvio padrão (Desvio TA) do intervalo entre a entrega de determinada mensagem de grupo a diferentes nós móveis.

Métricas	3GD	1GD
Média RTT	2532,18	1019,38
Desvio RTT	1249,80	575,54
TC	89,76%	97,05%
%100 TC MN	40,42%	81,31%
Média TA	156,19	76,77
Desvio TA	738,72	267,74

Figura 5.21: Resultados Cenário 5

Pelos resultados da 5.21 pode-se afirmar que o RTT médio do Cenário 5 é muito superior aos dos demais cenários. Esse RTT médio superior a 1000 milissegundos pode ser fruto de uma limitação da aplicação devido ao grande número de mensagens praticamente simultâneas que ela recebe.

Diferentemente do que ocorre nos demais Cenários, no Cenário 5 toda a comunicação Aplicação - Nó Móvel é através de mensagens de grupo que teoricamente são recebidas e respondidas no mesmo instante pelos Nós Móveis. Na teoria neste cenário todos os nós de um determinado grupo respondem a aplicação no mesmo momento fazendo com que ela tenha que processar um grande número de mensagens que chegam em um intervalo curtíssimo, o que naturalmente causa atraso nos registros de chegada.

Ainda na Figura 5.21 nota-se que o TC é 89,7% em 1GD; e 97% em 3GD e que 40,42% dos nós em 1GD e 81,31% em 3GD receberam todas as mensagens corretas, ou seja, a comunicação de grupo é precisa no novo ContextNet. A média do TA ficou em 156,19 milissegundos em 3GD e 76,77 milissegundos em GD; intervalos razoáveis de recebimento.

5.7 Discussões

Neste subcapítulo serão discutidos fatores que influenciam na execução dos testes, mas que não estão sendo avaliados por eles ou que fogem ao escopo deste trabalho.

Como relatado nos subcapítulos anteriores, o MR-UDP é um protocolo que consome muita memória da máquina virtual Java no momento da primeira conexão. Para evitar que a máquina virtual Java falhe foi estabelecido um intervalo entre as conexões. Esse intervalo garante que o *Garbage Collector* possa atuar e aumentar a memória disponível para a próxima conexões, contudo, torna a execução dos experimentos muito lenta. Foram testados todos os tipos de *Garbage Collector* que a máquina virtual Java possui nativamente a sua disposição; e após os experimentos concluiu-se que o *Garbage First Garbage Collector* [4] era o mais apropriado para uso com o Gateway, e portanto, com o MR-UDP, pois permitia que o intervalo entre as conexões fosse menor que os demais. Além do tipo de *Garbage Collector* há diversas configurações da máquina virtual Java que podem influenciar no desempenho das aplicações mas que não foram avaliadas por este trabalho.

Durante os experimentos observou-se também que o protocolo MR-UDP rompia as conexões casos elas ficassem sem trafegar mensagens por alguns minutos. Esse comportamento do protocolo exigiu que em todos os cenários durante o processo de conexão dos nós móveis ao Gateway (antes do início da

observação), mensagens Core - Nó Móvel ou Nó Móvel - Core fossem enviadas para ou por todos os nós móveis em intervalos menores do que um minuto.

Tanto nas aplicações quanto no Gateway foram utilizadas *Thread Pools* [5] no gerenciamento das *Threads* paralelas. Antes da execução dos cenários de teste foram feitos experimentos variando o tamanho das *Thread Pools* dos Gateways e das aplicações e medindo o RTT. Estes experimentos foram feitos para obtenção de um tamanho eficaz, ou seja, aquele tamanho que tornaria o RTT satisfatório.

Como esperado, observou-se que o RTT não é inversamente proporcional ao tamanho da *Thread Pool*. Observou-se também que o tamanho eficaz depende da quantidade de nós móveis no experimento. Com esses experimentos encontrou-se um tamanho eficaz para o Gateway - o mesmo para todos os testes - e para cada nó aplicação. Há um tamanho ideal, aquele que tornaria o RTT mínimo, para cada *Thread Pool* dos testes acima descritos, contudo, foge ao escopo desse trabalho encontra-los.

6
Conclusão e Trabalhos Futuros

À medida que IoT está se tornando cada vez mais popular tanto na indústria quanto na rotina das pessoas, aumentam os desafios das aplicações que o suportam. Neste contexto, este trabalho reconstruiu o *middleware* ContextNet cuja função é dar suporte a essas aplicações endereçando alguns dos desafios que elas enfrentam.

Os testes realizados neste trabalho indicam que a comunicação no novo ContextNet é *real-time*. Além disso, foi mostrado que ele suporta grandes quantidades de nós móveis e permite a paralelização das aplicações IoT. Nesse estudo, conclui-se que a paralelização de algumas aplicações IoT pode ser fator chave para sua viabilidade e que há indícios que a paralelização das aplicações melhora o desempenho.

Os experimentos também mostraram que algumas das funcionalidades adicionadas, como o armazenamento temporário de mensagens endereçadas a nós móveis desconectados, desempenharam da forma que era esperado quando foram idealizadas.

O objetivo de facilitar o uso do ContextNet por parte dos desenvolvedores também foi cumprido por este trabalho, uma vez que a imagem de todos os componentes apresentados está publicada e disponível para uso.

Os resultados dos testes nos mostraram que não há diferença significativa em termos de velocidade entre a comunicação *intra-Core* do antigo e do novo ContextNet. Tanto o DDS quanto o Kafka desempenharam bem, contudo, o Kafka possui diversos recursos que o tornam preparado para atender a aplicações, cujas necessidades estejam além das básicas, além de simplificar a implementação por parte do desenvolvedor.

Os princípios do ContextNet aplicados com o Kafka permitiram a construção de um *middleware* que atende as necessidades da comunicação móvel e das aplicações IoT; e não somente de um barramento veloz para comunicação.

Há diversos cenários de teste não explorados por este trabalho e que poderiam ser realizados para que a avaliação das novas funcionalidades e do desempenho do novo ContextNet fosse ainda mais completa. Para os trabalhos futuros pretende-se analisar cenários com e sem o PoA-Manager; repetir os cenários já avaliados com uma quantidade maior de variáveis para aferir mais

precisamente qual o impacto de cada uma; testar aplicações cuja necessidade de processamento seja significativa e cenários com comunicação *Inbound* e *Outbound* simultaneamente.

Também faz parte dos trabalhos futuros documentar detalhadamente como fazer uso dos componentes do ContextNet Kafka Core; como construir e paralelizar aplicações; e como migrar as aplicações desenvolvidas do antigo para o novo ContextNet.

Referências bibliográficas

[1] ENDLER, MARKUS AND E SILVA, FRANCISCO SILVA. **Past, present and future of the contextnet iomt middleware**. Open Journal of Internet Of Things (OJIOT), 4(1):7–23, 2018.

[2] KAI WAEHNER. **Event streaming and apache kafka in telco industry**, 2020. Last accessed August 2020.

[3] **Dds what is dds?**, 2021.

[4] **Garbage first garbage collector tuning**.

[5] **Thread pool executor**.

[6] DÍAZ, MANUEL AND MARTÍN, CRISTIAN AND RUBIO, BARTOLOMÉ. **-coap: An internet of things and cloud computing integration based on the lambda architecture and coap**. In: 11TH COLLABORATIVE COMPUTING: NETWORKING, APPLICATIONS, AND WORKSHARING, p. 195–206, 2015.

[7] RANJAN, YATHARTH AND KERZ, MAXIMILIAN AND RASHID, ZULQARNAIN AND BÖTTCHER, SEBASTIAN AND DOBSON, RICHARD AND FOLARIN, AMOS. **Poster: Radar-base: A novel open source m-health platform**. In: UBICOMP '18: THE 2018 ACM INTERNATIONAL JOINT CONFERENCE ON PERVASIVE AND UBIQUITOUS COMPUTING, p. 223–226, 2018.

[8] SNEPPE, M. AND NAMIOT, D.. **On mobile cloud for smart city applications**. In: ARXIV PREPRINT ARXIV:1605.02886, 2016.

[9] DAVID L., VASCONCELOS R., ALVES L., ANDRÉ R. AND ENDLER M.. **A dds-based middleware for scalable tracking, communication and collaboration of mobile nodes**. Journal of Internet Services and Applications, 4(16), 2013.

[10] **Kafka 2.6 documentation**, 2020. Last accessed September 2020.

[11] SILVA, L. D. N. AND ENDLER, M. AND RORIZ JR., M.. **MR-UDP: Yet Another Reliable UserDatagram Protocol, now for Mobile Nodes**. Technical report, Departamento de Informática, PUC-Rio, 2013.