UNIVERSIDADE FEDERAL DE OURO PRETO

DEPARTAMENTO DE COMPUTAÇÃO

David Silva Fernandes

REVISITANDO OS MODELOS DE PROGRAMAÇÃO DSM E ORIENTADO A TAREFAS

Ouro Preto, MG
2020

David Silva Fernandes

UNIVERSIDADE FEDERAL DE OURO PRETO

DEPARTAMENTO DE COMPUTAÇÃO

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Dr Joubert de Castro Lima

Coorientador: Ms André Luís Barroso Almeida

Ouro Preto, MG
2020

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

F363r Fernandes, David Silva .
Revisitando os modelos de programação DSM e orientado a tarefas. [manuscrito] / David Silva Fernandes. - 2020.
40 f.: il.: color., gráf., tab..

Orientador: Prof. Dr. Joubert de Castro Lima.
Coorientador: Prof. Me. André Luís Barroso Almeida.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Graduação em Ciência da Computação .

1. Programação (Computadores). 2. Programação orientada. 3. Java (Linguagem de programação de computador). 4. Middleware. I. Almeida, André Luís Barroso. II. Lima, Joubert de Castro. III. Universidade Federal de Ouro Preto. IV. Título.

CDU 004.42

Bibliotecário(a) Responsável: Celina Brasil Luiz - CRB6-1589

David Silva Fernandes

REVISITANDO OS MODELOS DE PROGRAMAÇÃO DSM E ORIENTADO A TAREFAS

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau em Bacharel em Ciência da Computação.

Aprovada em Ouro Preto, Setembro de 2020.

Dr Joubert de Castro Lima
Universidade Federal de Ouro Preto
Orientador

Ms André Luís Barroso Almeida
Instituto Federal de Minas Gerais, Ouro Preto
Coorientador

Túlio Spuri Teixeira de Oliveira
Gerencianet - Ouro Preto
Examinador

Dr André Ruiz Lins de Aquino
Universidade Federal de Alagoas - UFAL
Examinador

Resumo

Os modelos de programação DSM e Orientado a Tarefas muitas vezes utilizam a rede de comunicação desnecessariamente, pois de forma explícita realizam requisições entre clientes e provedores de serviço, mesmo quando as abstrações mantidas por tais modelos não tenham sofrido qualquer alteração. Há como evitar ou pelo menos reduzir o uso desnecessário da rede de comunicação nos modelos de programação DSM e Orientado a Tarefas? Para responder tal pergunta este trabalho descreve uma nova implementação de *middleware* para tais modelos de programação, onde ao invés de objetos, tais como variáveis globais, estruturas de dados e resultados de tarefas, serem obtidos via requisições explícitas de um cliente a um provedor de serviços, seja este um servidor ou um *peer* do *cluster*, estes objetos são enviados via notificações ao cliente solicitante apenas quando mudam seus estados internos, consequentemente de forma indireta ou implícita. Para testar a nova implementação proposta, foi utilizado o *middleware* JCL (Java Cá & Lá) integrado à plataforma Kafka que oferece suporte em escala ao padrão *publish/subscribe*. Os testes do JCL com e sem a integração ao Kafka mostraram que o esquema de notificação com uso de memórias locais é útil apenas para serviços não bloqueantes e com muitas leituras dos objetos compartilhados. Entretanto, quando o acesso bloqueante a objetos obriga o uso da rede de comunicação, a complexidade introduzida pela solução Kafka se torna ineficiente, chegando ao ponto de ser inviável à medida que o número de tópicos associados aos objetos compartilhados aumenta. A tolerância a falha do Kafka introduz uma sobrecarga de cerca de 15%, portanto aceitável quando comparada com a versão sem tal requisito.

Palavras-chave: Modelo de programação DSM. Modelo de programação orientado a tarefas. Padrão publish/subscribe. Java Cá & Lá. Kafka. Middleware.

Abstract

The DSM and Task Oriented programming models often use the communication network unnecessarily, as they explicitly make requests between clients and service providers, even when the abstractions maintained by such models have not changed. Is there a way to avoid or at least reduce the unnecessary use of the communication network in the DSM and Task Oriented programming models? To answer this question this work describes a new middleware implementation for such programming models, where instead of objects, like global variables, data structures and task results, be obtained via explicit requests from a client to a service provider, be it a server or a cluster peer, these objects are submitted via notifications to the client when they change their internal states, thus on an implicit or indirect way. To test the new implementation, it was used the JCL (Java Cá & Lá) integrated with Kafka platform that offers at scale publish/subscribe support. The JCL tests with and without Kafka integration demonstrated that the notification schema in conjunction with local memories is only useful for non-blocking services and with shared objects high number of reads. However, when shared objects blocking accesses force the utilization of the communication network, the Kafka complexities turn the solution inefficient, reaching the point of being unviable as the number of topics associated with shared objects increases. Kafka's fault tolerance introduces an overload of about 15%, therefore acceptable when compared to the version without such a requirement.

Keywords: DSM programming model. Task oriented programming model. Publish/subscribe pattern. Java Cá & Lá. Kafka. Middleware.

Lista de Ilustrações

Lista de Tabelas

Sumário

1 Introdução

O modelo de programação baseado em memória distribuída e compartilhada (DSM) (AMZA et al., 1996; PROTIC; TOMASEVIC; MILUTINOVIC, 1996; ANTONIU; BOUGÉ, 2001) cria um sistema de endereçamento global e único sobre um *cluster* de nós ou máquinas onde objetos, tais como variáveis globais de aplicações diversas, entradas de mapas ou itens de outras estruturas de dados, dados de sensoriamento ou mesmo resultados de tarefas submetidas ao *cluster*, são acessados de forma transparente por qualquer usuário do sistema. Requisitos de comunicação via troca de mensagens, serializações e de-serializações de objetos, concorrência para acessos bloqueantes aos objetos compartilhados e muitas vezes o serviço de tolerância a falhas dos objetos armazenados, incluindo o particionamento dos objetos em páginas, *frames* ou outras unidades de memória, são suportados pelo modelo DSM sem os explicitar ao programador, com o objetivo de simplificar e, consequentemente, agilizar o desenvolvimento de aplicações escaláveis e descentralizadas.

No modelo DSM há uma camada de software que é executada sobre o *cluster*, portanto sobre diversos nós autônomos e muitas vezes com diferentes sistemas operacionais, podendo tal camada ser um *middleware* ou um *framework* ou uma biblioteca, entretanto, seja qual for a implementação de tal camada, esta possui uma API (Interface de Programação de Aplicação) bem definida e normalmente simplificada. Um exemplo de API simplificada para o programador pode conter os serviços de inserção, consulta, atualização ou remoção (CRUD) de objetos no sistema de memória DSM. Como as aplicações que usam a API podem ser concorrentes, há necessidade do modelo DSM implementar algum mecanismo de controle de concorrência. Um dos mecanismos mais simples utiliza duas primitivas: *WRITE or READ* para reservar acesso bloqueante a um determinado objeto armazenado no sistema de memória DSM e a primitiva *RELEASE* para liberar o acesso a outra aplicação (PROTIC; TOMASEVIC; MILUTINOVIC, 1995). Um dos maiores desafios do modelo DSM é manter a coerência entre as diversas memórias privadas que cada nó do *cluster* possui, uma vez que cópias de objetos podem ser necessários para reduzir a latência e melhorar a escalabilidade/desempenho das aplicações, portanto inúmeros protocolos de coerência entre tais memórias foram desenvolvidos, dentre eles o sequencial, o baseado na primitiva *release*, o chamado *lazy-release* e o baseado na primitiva *synchronized* (PROTIC; TOMASEVIC; MILUTINOVIC, 1995). Uma das ideias mais comuns assume que o modelo DSM possui algum mecanismo de *cache* e este possui alguma solução que mantém a coerência entre as diversas *caches* locais.

O mecanismo de *cache*, assim como os demais desenvolvidos para resolver o problema de coerência de cópias de objetos armazenados no modelo DSM, adotam comunicação explícita entre solicitantes e provedores, ou seja, uma *cache* local solicita escrita às demais *caches* locais de outros nós do *cluster* para um determinado objeto O e contratos são estabelecidos, envolvendo,

novamente, comunicações explícitas para se saber qual nó irá atualizar O e quais esperarão numa fila de acesso. A comunicação entre nós de um *cluster* de forma explícita pode usar a rede de comunicação desnecessariamente. No exemplo de múltiplas *caches*, ter que encaminhar mensagens para as diversas memórias locais pode ser custoso e muitas vezes desnecessário.

O modelo de programação orientado a tarefas (SHAHRIVARI; SHARIFI, 2011; THOMAN et al., 2018) cria e gerencia uma abstração chamada tarefa, onde esta encapsula uma ou várias execuções assíncronas de um membro de uma classe ou de um objeto, incluindo, portanto, seus métodos. Tal modelo simplifica o desenvolvimento em ambientes paralelos ou distribuídos, entretanto elimina algumas vantagens de alguns paradigmas de programação, tal como o orientado a objetos, pois não aceita herança e polimorfismo. Em linhas gerais, programadores conseguem facilmente encapsular algoritmos sequenciais já existentes e os executar em um *cluster* como uma ou várias tarefas. Os resultados de tais tarefas são normalmente retornados para os nós que as submeteram, mas pode ocorrer de serem armazenados para utilização por outras tarefas, utilizando muitas vezes o modelo de memória DSM. Um exemplo de *middleware* que integra DSM e o modelo orientado a tarefas numa única solução é o Java Cá & Lá (JCL) (ALMEIDA et al., 2019; CIMINO et al., 2019), oferecendo transparentemente o gerenciamento de todo o ciclo de vida de uma tarefa, incluindo seu escalonamento no *cluster*.

1.1 O Problema

Os modelos de programação DSM e Orientado a Tarefas muitas vezes utilizam a rede de comunicação desnecessariamente, pois de forma explícita realizam requisições entre clientes e provedores de serviço, mesmo quando as abstrações mantidas por tais modelos não tenham sofrido qualquer alteração.

1.2 Hipótese

Há como evitar ou pelo menos reduzir o uso desnecessário da rede de comunicação nos modelos de programação DSM e Orientado a Tarefas? De forma mais precisa, há como usar a rede de comunicação apenas quando há atualizações nas abstrações mantidas por tais modelos de programação?

1.3 Alternativa de solução escolhida

Como alternativa de solução para validação da hipótese e, consequentemente, atenuação do problema foi escolhida a utilização do padrão *publish/subscribe* que insere um mecanismo de notificação de atualizações, portanto tal padrão pode reduzir significativamente as inúmeras comunicações desnecessárias quando não há atualizações em curso.

A pergunta tecnológica que passa a ser respondida é a seguinte: Como o padrão *publish/subscribe* pode beneficiar a escrita de *middlewares*, *frameworks* ou bibliotecas DSM/orientado a tarefas?

Para responder a pergunta acima há necessidade de reimplementar o modelo de programação DSM e o modelo de programação orientado a tarefas, objetivando avaliar o desempenho do esquema de notificação ou envio de mensagens de forma implícita quando este é integrado aos modelos de programação citados anteriormente. As premissas e transparências dos modelos DSM e orientado a tarefas devem ser preservadas nas reimplementações conduzidas.

1.4 Justificativas

Os modelos de programação DSM e o orientado a tarefas foram amplamente utilizados nas últimas décadas e ainda mantêm sua popularidade no desenvolvimento de aplicações descentralizadas (ZHU; WANG; LAU, 2002; WALKER et al., 2003; XIONG; WANG; XU, 2010; MARCHIONI; SURTANI, 2012; VEENTJER, 2013), pois estes não explicitam a troca de mensagens, o que torna o desenvolvimento menos árduo. O padrão *publish/subscribe* para notificar mudanças de estado ou contexto de objetos, logs, sensores, tabelas de banco de dados e diversas outras abstrações ou recursos computacionais, vem sendo adotado com enorme sucesso devido ao desenvolvimento nas últimas décadas de ferramentas que conseguiram garantir escalabilidade e resiliência em tal serviço (KREPS et al., 2011; SNYDER; BOSNANAC; DAVIES, 2011; VIDELA; WILLIAMS, 2012; CARLSON, 2013; KRISHNAN; GONZALEZ, 2015; MARCU et al., 2018; INTORRUK; NUMNONDA, 2019). Neste sentido, um estudo que busque integrar tais ferramentas de notificação de mudanças a soluções DSM/orientadas a tarefas sem alterar suas formas de programação é extremamente útil para sabermos até que ponto tal integração é factível e qual a eficiência do padrão *publish/subscribe* em atender acessos bloqueantes e não bloqueantes a objetos compartilhados. Consequentemente, torna-se possível avaliar o desempenho da ferramenta escolhida para integração.

O *middleware* JCL foi o escolhido em termos de solução DSM e orientado a tarefas. Sua escolha se justifica, uma vez que é uma das únicas soluções de *middleware* que integra as tecnologias Computação de Alto Desempenho (HPC - *High Performance Computing*) (ALMEIDA et al., 2019) e Internet das coisas (IoT - *Internet of Things*) (CIMINO et al., 2019). O Kafka (KREPS et al., 2011) se justifica devido a sua escalabilidade e utilização mundial.

1.5 Organização do Trabalho

Os demais capítulos do trabalho estão organizados da seguinte forma: no Capítulo 2 detalhamos fundamentos teóricos e explicações prévias que servem para orientação, análise e interpretação do trabalho. No Capítulo 3 há o estado-da-arte que mais se assemelha ao trabalho

sendo apresentado. No Capítulo 4 detalhamos como integramos o *middleware* JCL ao Kafka. Os experimentos são apresentados no Capítulo 5. Por fim, no Capítulo 6 fazemos as considerações finais e apontamos os trabalhos futuros.

2 Fundamentação Teórica

Este capítulo detalha fundamentos teóricos e explicações prévias que servem para orientação, análise e interpretação deste trabalho.

2.1 Sistema distribuídos

Várias são as definições de sistemas distribuídos na literatura. Neste trabalho usamos:

"Um sistema distribuído é um conjunto de computadores ou nós independentes que se apresenta a seus usuários como um sistema único e coerente." (TANENBAUM; STEEN, 2006)

Esta definição refere-se a duas características de sistemas distribuídos. A primeira característica enxerga um sistema distribuído como uma coleção de elementos computacionais disponíveis para colaborarem entre si. Um elemento computacional que será doravante referenciado por **processo** (STEEN; TANENBAUM, 2016). Um **processo** é sempre executado em um nó e este processo é independente porque executa sob um sistema operacional capaz de gerenciar **processos** e capaz de interconectar nós eficientemente, pois opera em rede. A segunda característica diz que os usuários, sejam eles pessoas físicas ou aplicações, acreditam que estão lidando com um sistema computacional centralizado, único. Isso indica que, de um modo ou de outro, os **processos** autônomos precisam colaborar para oferecer essa transparência aos usuários. Abaixo estão algumas características sobre sistemas distribuídos.

2.1.1 Transparências

Uma meta de um sistema distribuído é ocultar de seu usuário o fato de ele ser constituído pela combinação de vários nós fisicamente distribuídos em um *cluster*, criando a ilusão de ser um único sistema centralizado (STEEN; TANENBAUM, 2016). Um sistema distribuído que se apresenta para o usuário como um único sistema de computador é denominado **transparente**. A transparência pode ser avaliada a partir de diversas perspectivas.

- Localização - Usuários não conhecem onde os recursos estão localizados fisicamente no sistema, seja um recurso um objeto, um processo, uma impressora ou um arquivo, por exemplo;
- Migração - Recursos podem mover fisicamente sem que a forma como são acessados seja impactada;

- Relocação – Recursos são relocados enquanto estão sendo utilizados, sem que a forma como são acessados seja impactada;
- Replicação – Oculta a existência de cópias de um recurso;
- Concorrência – Usuários não percebem que outros usuários estão utilizando o mesmo recurso;
- Falha – Usuário não percebe que um recurso deixou de funcionar e que o sistema se recuperou da falha.

2.1.2 Desempenho

O desempenho de um sistema computacional é uma medida da capacidade ou rendimento desse sistema para realizar uma dada tarefa. Entre as atividades que formam essa tarefa há aquela que demanda mais recursos computacionais e, portanto, é a mais demorada. Esta atividade é chamada gargalo computacional do sistema e é a principal limitadora de seu desempenho. Gargalos computacionais são tipificados pelo consumo excessivo de algum elemento de hardware.

De forma geral, a limitação pode ser analisada em três aspectos (TANENBAUM; STEEN, 2006). Se a aplicação realiza muitas operações lógicas e aritméticas, mas há pouca comunicação com o sistema de memória e com os subsistemas de I/O, diz-se que a solução é *CPU-bound*. Caso a solução requeira muita comunicação com a memória para operações de escrita e leitura dizemos ser uma aplicação *Memory-bound*. Por fim, há as aplicações *I/O-bound* quando, por exemplo, muito acesso a disco é feito durante o ciclo de vida da aplicação.

Independentemente do tipo de aplicação, os desafios para garantir desempenho em sistemas distribuídos são enormes. Inicialmente, se busca evitar comunicação com a rede de dados, mas conforme podemos perceber a latência não é o único fator limitador de desempenho, pois o uso excessivo de disco em detrimento da rede de dados pode tornar o sistema ainda mais ineficiente. O mesmo não é verdade para o uso excessivo de CPU ou do sistema local de memória, uma vez que estes normalmente são bem mais rápidos do que o envio de mensagens pela rede de dados. Infelizmente, o volume de dados e a demora na execução das tarefas de forma sequencial requer distribuição de processamento e armazenamento, respectivamente, portanto não há como usar CPUs e memórias locais indefinidamente.

2.1.3 Escalabilidade

A **escalabilidade** de um sistema é medida pela capacidade de ampliar o desempenho do sistema pela adição de novos recursos computacionais, sem que haja perdas aparentes (HILL, 1990). Infelizmente, um sistema escalável muitas vezes apresenta perda de desempenho à medida que é ampliado. Isto ocorre devido ao custo inserido pela necessidade de controle e coordenação

dos novos recursos computacionais, sejam estes nós de um *cluster* ou unidades de disco ou unidades de impressão e praticamente qualquer outro hardware.

A escalabilidade é essencial para construir sistemas paralelos e distribuídos (CHEN; SUN, 2006), porque nem todas operações e aplicações requerem as mesmas melhorias. A escalabilidade pode ser definida em três eixos (ABBOTT; FISHER, 2009), formando um cubo. Toda solução escalável busca simplificar um dos seguintes eixos de tal cubo:

1. X - escalabilidade tradicional, a qual distribui a carga de trabalho total entre um dado número de processos;
2. Y - escalabilidade referente a extração e distribuição de serviços, isto é, uma decomposição funcional, tendo como um exemplo cada vez mais comum as soluções que utilizam a arquitetura de micro serviços distribuídos sob o *cluster*;
3. Z - escalabilidade referente ao particionamento de dados, envolvendo distribuição de dados entre processos, mas também o seu particionamento em unidades menores de memória (Ex. blocos, páginas ou *frames* de dados) para melhorar desempenho.

2.2 Sincronização

Em arquiteturas distribuídas, cada processo possui seu próprio relógio, com isto suas possibilidades de temporização e sincronização (STEEN; TANENBAUM, 2016). Caracterizados por capacidades de processamento distintas, é impossível dizer que os processos funcionam em mesma frequência, portanto o comum é processos perderem a sincronia gradualmente. Consequentemente, a assincronia pode causar falhas no sistema distribuído, pois aplicações podem ter que esperar tempos associados à arquivos, objetos e mensagens (TANENBAUM; STEEN, 2006). Logo, a sincronização entre processos é importante para garantir acesso correto a recursos, incluindo muitas vezes a causalidade entre as operações de acesso ao recurso.

A sincronização com um temporizador global e muitas vezes físico (Ex. servidor com relógio atômico de altíssima precisão) pode ser feita utilizando *Network Time Protocol* (Mills, 1991). A sincronização mútua entre processos pode utilizar temporizadores lógicos, de modo a garantir a sincronização apenas entre os processos, sem necessitar de um temporizador global (LAMPORT, 1978), uma vez que a necessidade primordial é acessar os recursos de forma correta, ou seja, sem corrompê-los. Há também como manter a ordem parcial entre processos que manipulam um recurso comum sob um *cluster* (FIDGE, 1987), portanto, por exemplo, a ordem de transações de saque e depósito numa determinada conta pode manter a causalidade mesmo entre processos concorrentes e em nós distintos do *cluster*.

2.3 Consistência e Replicação

A replicação é uma alternativa para conferir confiabilidade e, talvez, melhorar o desempenho do sistema (TANENBAUM; STEEN, 2006). A replicação melhora a resiliência na presença de falhas, pois cria cópias em memórias locais. Pode melhorar o desempenho, pois reduz a latência nas operações de escrita e leitura dos dados com memórias locais mais próximas de onde os processos estão em execução. É claro que ao replicar se cria automaticamente o problema de manter a coerência entre as réplicas ou cópias. A coerência se torna ainda mais desafiadora em sistemas descentralizados (GHOSH, 2014).

Há como manter a coerência entre réplicas focando nos dados em si, ou seja, focando na conta A ou B, independente do usuário do banco. Entretanto, há como focar no cliente, ou seja, no usuário do banco, independente das contas que ele possui e das operações que ele faz com outras contas (BERMBACH; KUHLENKAMP, 2013). Como exemplos da centralidade no cliente ou nos dados, há os protocolos: consistência contínua, desvio numérico, desvio de ordenação, consistência baseada em primário, escrita remota, escrita local, consistência baseada em quórum e algumas outras alternativas (TANENBAUM; STEEN, 2006). Também há outras estratégias, ou seja, sem o enfoque específico nos dados ou nos clientes, tais como consistência multidimensional e consistência adaptativa (BERMBACH; KUHLENKAMP, 2013).

2.4 Modelos de Programação

Nesta seção, são destalhados os modelos de programação usados tanto pelo programador final quanto para construir as reimplementações sendo apresentadas neste trabalho. Precisamente, são detalhados os modelos DSM, orientado a tarefas e, por fim, o baseado em eventos. No caso do modelo baseado em eventos, o enfoque foi dado ao padrão *publish/subscribe*, pois este tem sido implementado eficientemente em larga escala nos últimos anos.

2.4.1 Orientado a Tarefas

O modelo de programação orientado a tarefas permite programar aplicações tanto para multi-processadores, quanto para sistemas de computação distribuída (SHAHRIVARI; SHARIFI, 2011). Uma tarefa é definida como a soma de trechos de código e os dados envolvidos pelas operações presentes em tal trecho de código. Desta forma, uma tarefa pode ocupar um ou mais processos durante sua execução. Tratar tarefas como um nível de programação mais elevado facilita a programação, pois não há preocupação em enviar mensagens entre processos ou inicializar *threads* de execução, mas apenas instanciar tarefas que são transparentemente resolvidas pela camada de software, seja esta um *middleware*, *framework* ou biblioteca.

Várias são as definições de tarefa na literatura. Neste trabalho nos referimos a tarefa como sendo:

"Um conjunto de atividades somadas aos dados envolvido em seu processamento, que podem ser atribuídas e executadas em máquinas remotas, sendo que posterior à sua execução, o resultado permanece disponível para a máquina dona da tarefa" (SHAHRIVARI; SHARIFI, 2011)

Tarefas são normalmente assíncronas, portanto quem as submete pode realizar outras tarefas enquanto não precisa dos resultados das tarefas previamente submetidas. Há sempre mecanismos de sincronização quando as tarefas são assíncronas, onde se cria barreiras totais ou parciais que são úteis para a construção de mecanismos de causalidade entre tarefas.

É importante ressaltar que tarefas também encapsulam objetos, portanto o paradigma orientado a objetos também é utilizado em conjunto com a implementação de tarefas. Entretanto, as elegâncias e particularidades da orientação objetos são perdidas em pró da simplicidade de programação no modelo orientado a tarefas, portanto polimorfismo, herança e sobrecarga de métodos inexistem quando tarefas entram em cena.

2.4.2 Memória Distribuída e Compartilhada (*Distributed Shared Memory* - DSM)

DSM é uma arquitetura de memória que endereça blocos de memórias, postos separados geograficamente, em um único espaço local virtual de memória (PROTIC; TOMASEVIC; MILUTINOVIC, 1996). O termo *distributed* deduz que para o usuário exista apenas um único espaço de memória, quando internamente a composição ocorre por diversos endereços físicos de memórias locais e interligadas por redes de comunicação, sendo toda esta infraestrutura gerenciada pelo modelo de programação baseado em DSM. Desta forma, tal modelo esconde os mecanismos de comunicação entre os processos, exibindo ao programador uma API suficiente para consolidar operações feitas em memória típicas, bem como prover mecanismos transparentes para escalabilidade e custo-benefício (PROTIC; TOMASEVIC; MILUTINOVIC, 1995).

A arquitetura DSM pode ser implementada no nível de software, hardware ou em ambos (PROTIC; TOMASEVIC; MILUTINOVIC, 1996). No nível de software, existem abordagens baseadas em páginas onde os blocos de memórias virtuais comportam tamanho fixo, ou as baseadas em objetos onde os blocos de memória compartilham objetos de tamanhos variados. Há também as abordagens baseadas em tuplas onde a memória virtual armazena objetos identificando-os por pares *key-value* (PROTIC; TOMASEVIC; MILUTINOVIC, 1995).

Bem como nos modelos de memória convencionais, a arquitetura DSM também deve lidar com a consistência da memória (PROTIC; TOMASEVIC; MILUTINOVIC, 1996), apresentados na seção 2.3. A consistência pode ser preservada utilizando a abordagem sequencial onde as operações são tratadas na mesma ordem em que foram submetidas. Outra alternativa é utilizar a abordagem versionada onde premissas de uma tarefa assíncrona devem ser conhecidas antes que a tarefa seja finalizada. Em linhas gerais, há inúmeras tentativas de garantir a coerência dos dados para cada etapa de execução das aplicações.

2.4.3 Baseado em Eventos

O modelo de programação baseado em eventos, assim como qualquer outro modelo de programação, pode ser implementado de diferentes formas ou padrões. Neste trabalho, o enfoque é o padrão *publish/subscribe*, mas há outros, como os *Listeners*, amplamente usados em interfaces gráficas (GUIs). De uma forma geral, no modelo baseado em evento há quem gera o evento e quem se interessa pelo evento gerado. Tal modelo pode ser implementado fracamente acoplado entre geradores de evento e interessados em evento, que é o enfoque deste trabalho, mas isto pode não ser o foco, como é o caso dos *Listeners* e *Observers*, por exemplo. O trabalho (MEIER; CAHILL, 2005) mostra uma classificação entre os sistemas com suporte a programação baseada em eventos, detalhando as soluções distribuídas com tal propósito.

Publish/Subscribe é um padrão de comunicação em sistemas distribuídos por troca de mensagens. Nele, *publishers* publicam notificações sobre um tópico em um canal quando este tópico muda de estado ou segundo alguma regra do domínio da aplicação, sendo um tópico qualquer objeto ou recurso, portanto qualquer sensor, atuador ou variável ou um vértice de um grafo ou uma impressora e praticamente tudo que se almeje monitorar. Já os *subscribers* se inscrevem para tópicos para consumir notificações desejadas, incluindo suas mudanças de estado.

Uma das maiores vantagens do padrão *publish/subscribe* é o fraco acoplamento entre *publishers* e *subscribers*, onde o tópico se mantém resiliente mesmo em cenários onde ambos não estejam online (EUGSTER et al., 2003). Aplicações de emails, redes sociais diversas e integração de sistemas podem ser construídas eficientemente usando tal padrão. Apache Kafka (KREPS et al., 2011) e Google Pub/Sub (GOODMAN et al., 2011) são exemplos de soluções distribuídas que implementam tal padrão, são resilientes, ou seja, os tópicos são tolerantes a falhas, e que operam em escala, neste caso que funcionam para aplicações com dezenas ou até mesmo centenas de milhões de usuários.

2.5 Java Cá & Lá - JCL

O Java Cá & Lá (JCL) é um *middleware* orientado a tarefas e com suporte a DSM (ALMEIDA et al., 2019). Ele também permite a integração com *brokers* MQTT (LIGHT, 2017), portanto ele consegue se inscrever a tópicos, submeter notificações diversas e as receber. De uma forma geral, o JCL é o primeiro *middleware* a integrar as tecnologias IoT e HPC (CIMINO et al., 2019).

A Figura 2.1 ilustra a arquitetura do JCL em um *cluster*, sendo este composto pelos componentes: JCL_Server, JCL_Host, JCL_SuperPeer e JCL_User. Tais componentes são capazes de armazenar ou instanciar objetos Java ou Android remotamente, assim como executar tarefas que encapsulam objetos Java ou Android em ambientes distribuídos compostos por diferentes *clusters*, onde tais ambientes sequer enxergam uns aos outros. Quem provê esta interconexão de *clusters* é o componente JCL_SuperPeer. Com o desenvolvimento de componentes JCL_Host para

Java, Android e Arduino torna-se possível sensoriar e atuar em ambientes remotos a partir de uma API simples e totalmente integrada aos serviços de armazenamento e processamento já existentes desde sua versão HPC. O JCL_Server é responsável por gerenciar os demais componentes da arquitetura, contudo sem virar o gargalo, pois este delega permissões aos componentes JCL_Hosts e JCL_SuperPeers.

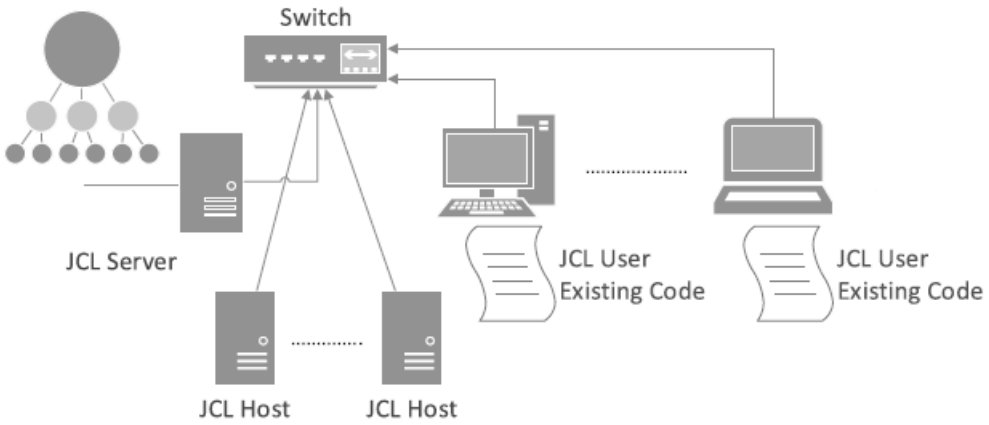

Figura 2.1 – Implantação do JCL (ALMEIDA et al., 2019)

O JCL_User é a fachada ou API do JCL, portanto cabe ao programador incluir tal componente em sua aplicação. É nele que os métodos são invocados para execuções remotas ou para instanciação de variáveis ou para criação de mapas distribuídos. Os serviços de sensoriamento, sensibilidade de contexto e atuação em ambientes remotos também podem ser feitos via API e com o JCL_User. Cabe ao *kernel* JCL decidir onde os objetos são armazenados e onde as tarefas são executadas. O programador também pode optar por executar uma tarefa num nó específico do *cluster*. Os sensores podem ser configurados remotamente e programaticamente, assim como a adição de contextos diversos. O fato do JCL ter sido desenhado inicialmente para HPC e depois ter incorporado serviços IoT faz com que ele possua os modelos de programação DSM e orientado a tarefas para o desenvolvimento de aplicações IoT, ou seja, no JCL não há apenas o modelo de eventos usando o padrão *publish/subscribe* para desenvolver IoT.

Atualmente, o JCL implementa a obtenção explícita dos objetos e dos resultados das tarefas, assim como dos pares *key-value* de seu mapa distribuído. Cabe ao JCL_User encontrar onde está armazenado o objeto, seja ele uma variável, a entrada do mapa, o resultado de uma determinada tarefa ou uma informação de sensoriamento, ou seja, encontrar um JCL_Host no *cluster* e obter uma cópia do objeto para o nó onde o JCL_User está em execução. Isto ocorre mesmo para os acessos não bloqueantes aos objetos e mesmo quando já há uma cópia de tais objetos no nó JCL_User. A ideia deste trabalho é adicionar memórias locais ao JCL e inverter a lógica de obtenção de objetos de inúmeros serviços de sua atual API sem modificar as assinaturas existentes, ou seja, sem causar qualquer impacto para as aplicações já codificadas e em execução.

2.6 Apache Kafka

O Kafka é um sistema distribuído para troca de mensagens que consegue coletar e enviar grande volume de dados de log com baixa latência (KREPS et al., 2011). Há três componentes no Kafka: *cluster*, *producer* e *consumer*, ilustrados na Figura 2.2. O *cluster* é o componente central que recebe tarefas das aplicações produtoras, tais como mensagens de entrada, e as encaminha para aplicações consumidoras interessadas em mensagens devidamente organizadas em tópicos. O Kafka permite transformar dados processados pelo *cluster*, aplicando regras de negócios do domínio da aplicação. Há também como armazenar os dados publicados no *cluster* com tolerância a falhas.

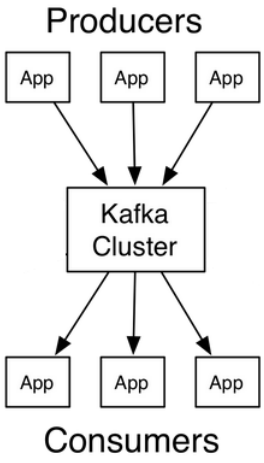

Figura 2.2 – Implantação do Kafka (GARG, 2013)

A entrega de mensagens é feita utilizando filas ou o padrão *publish/subscribe*. O padrão *publish/subscribe* permite que um registro seja enviado a vários consumidores via *broadcast*, contudo não é possível escalonar o processamento, visto que o envio ocorre para todos os inscritos (EUGSTER et al., 2003). Ao manter as duas estratégias, precisamente filas e *publish/subscribe*, o Kafka escala o processamento e suporta múltiplos inscritos.

3 Revisão Bibliográfica

Neste capitulo, é descrito a literatura que mais se assemelha ao trabalho sendo apresentado. Uma busca nos repositórios IEEE, ACM e ISI Web of Science foi realizada usando as expressões:

- "(DSM *OR Distributed Shared Memory*) *AND Task Oriented AND publish-subscribe*"
- "(DSM *OR Distributed Shared Memory*) *AND publish-subscribe*"
- "*Task Oriented AND publish-subscribe*"

O mesmo foi feito no Google Acadêmico. Até o momento não foi encontrado trabalho correlato sobre reimplementações do modelo orientado a tarefas usando o padrão *publish/subscribe* para notificar submissão de tarefas ou obtenção de resultados de tarefas em *clusters*.

O trabalho que mais se assemelha ao nosso foi publicado em (MAZZUCCO et al., 2009). Os autores implementam um *middleware* DSM utilizando como base um *middleware* orientado a mensagem (MOM), portanto onde *publishers* e *subscribers* não precisam se conhecer e não precisam estar online ao mesmo tempo. Os serviços DSM são oferecidos a objetos Java usando escritas locais e remotas em dois sistemas de memória. Há interfaces a serem estendidas para construir os objetos a serem armazenados no sistema DSM. O modelo de consistência entre as memórias é o PRAM (Pipelined RAM (LIPTON; SANDBERG, 1988)), onde há uma escrita local e rápida após salvar uma cópia do objeto. A cópia serve para o caso de *rollback*, pois após atualizar localmente um objeto, há a publicação do mesmo no *cluster*, portanto pode haver outras publicações em andamento e por isto as cópias são fundamentais. Há efeitos colaterais nas aplicações que fizerem uso dos objetos que tiveram operações de *rollback*, portanto a consistência PRAM pode se tornar custosa computacionalmente.

O nosso trabalho se difere em alguns aspectos do trabalho (MAZZUCCO et al., 2009). Primeiro, não há necessidade do programador implementar interfaces extras para construir os objetos a serem armazenados na arquitetura DSM, o que gera dependência, refatoração de código e manutenibilidade da aplicação desenvolvida. O *middleware* usado neste trabalho, chamado JCL, usa computação reflexiva do Java, portanto não só armazena, mas também instancia quaisquer objetos remotamente. Segundo, usamos o padrão *publish/subscribe* para reimplementar não apenas o modelo de programação DSM, mas também o modelo orientado a tarefas, precisamente a submissão e obtenção dos resultados de tarefas de tal modelo. Por fim, ainda não há nenhum relato na literatura sobre o uso de soluções líderes de mercado, portanto mundialmente utilizadas, para reimplementar soluções DSM ou orientadas a tarefas. Consequentemente, não se sabe até que ponto a integração é factível e qual a eficiência de soluções, tais como Kafka ou Google

Pub/Sub, na implementação de um espaço de memória compartilhado, distribuído e transparente para os programadores.

O trabalho (EUGSTER; GUERRAOUI; SVENTEK, 2000) implementa algumas estruturas do *Java Collections Framework* (JCF), precisamente listas e conjuntos, de forma assíncrona e distribuída. Os autores fazem uso da ideia introduzida no pacote Java *Concurrent*, ou seja, fazem uso do objeto *Future*, onde resultados de *threads* locais são obtidos assincronamente e via espera condicional, evitando, portanto, a ineficiente espera ocupada. No trabalho, o programador recebe objetos *Future* como resultados de manipulações de tais estruturas de dados em um *cluster*. Internamente, há adoção do padrão *publish/subscribe* para atualizações em nós do *cluster* de forma implícita ou indireta, ao contrário do que é adotado no JCF tradicional ou no STL do C++, por exemplo.

Nosso trabalho se difere de (EUGSTER; GUERRAOUI; SVENTEK, 2000), pois não propomos apenas a implementação de estruturas de dados distribuídas usando o padrão *publish/subscribe*, mas também reimplementamos a submissão e obtenção dos resultados de tarefas, assim como o armazenamento e instanciação remota de variáveis globais de aplicações orientadas a objetos em Java. Também não introduzimos objetos como *Future*, pois mantemos a API das estruturas de dados (Ex. Java Map) e a API DSM do *middleware* JCL, que pode ou não usar o objeto *Future* do Java. Neste trabalho, o padrão *publish/subscribe* é transparente ao programador, portanto este enxerga apenas os modelos de programação DSM e orientado a tarefas.

O trabalho de (CUDENNEC, 2019) inverte a ideia do trabalho sendo apresentado, ou seja, em (CUDENNEC, 2019) os autores reimplementam o padrão *publish/subscribe* usando as primitivas do modelo DSM. Os *publishers* e *subscribers* escrevem e leem à partir de um endereço compartilhado e distribuído sobre um *cluster* de nós. As diferenças com o nosso trabalho são evidentes.

No trabalho (EUGSTER et al., 2003) há uma detalhada explicação sobre as muitas facetas do padrão *publish/subscribe*. Os autores citam algumas soluções assíncronas de espaços compartilhados e distribuídos (Ex. JavaSpaces, TSpaces e WCL), mas todas se diferenciam substancialmente deste trabalho, pois em nosso trabalho mantemos o acesso síncrono ou assíncrono de objetos do modelo DSM. Internamente, há uma implementação do padrão *publish/subscribe* com esquemas de notificação e replicação de objetos que buscam reduzir a latência para as operações de leitura e escrita nos diversos nós do *cluster*. Estendemos o padrão *publish/subscribe* para garantirmos sincronizações em operações de escrita nos tópicos criados pelos programadores, portanto mantemos o modelo DSM e buscamos usar as vantagens do padrão de notificações ou indireções de mensagens.

4 Desenvolvimento

Este capítulo apresenta as mudanças feitas para integrar os serviços do JCL ao Kafka. A API JCL é composta por vários serviços para HPC e IoT. Neste trabalho, os métodos da API JCL que manipulam objetos foram reimplementados para validar a hipótese que é possível tal integração. Os métodos reimplementados são: $JCL.instantiateGlobalVar$, $JCL.getValue$, $JCL.getValueLocking$, $JCL.setValueUnlocking$, $JCL.execute$, ***JCL.executeAll***, ***JCL. executeAllCores***, $JCL.executeOnHost$, $JCL.getResultBlocking$ e ***JCL.getAllResultsBlocking***. No mapa distribuído JCL foram reimplementados os métodos: $JCLhashmap.get$, ***JCLhashmap.getLocking***, $JCLhashmap.put$ e $JCLhashmap.putAll$.

4.1 Reimplementação do método *JCL.instantiateGlobalVar*

O JCL permite a instanciação ou armazenamento de um objeto, que neste caso é uma variável global, no *cluster*. Para tal, o programador deve invocar o método da API denominado $JCL.instantiateGlobalVar$. Há como invocar tal método com os argumentos $key, value$, onde estes representam o identificador e valor da variável, respectivamente. Há também a variação da assinatura do método com os argumentos $key, className, jars, args$, onde neste caso o JCL instancia um objeto remotamente usando key como identificador, $className$ como o nome da classe a ser usada para instanciar o objeto, $jars$ são as possíveis dependências necessárias para tal instanciação e $args$ os argumentos necessários durante a instanciação do objeto, precisamente por seu construtor. O método $JCL.instantiateGlobalVar$ possui duas versões: i) síncrona, que retorna *true* se deu certo a instanciação ou *false* caso contrário; ii) assíncrona, que retorna um objeto do tipo *Future* do Java para que o programador possa obter o resultado da instanciação num momento futuro em seu código.

Na versão JCL sem integração ao Kafka, o componente JCL_User calculava a função *hash* do apelido (key) do objeto no *cluster* e em seguida calculava o módulo de tal valor dividido pelo tamanho do *cluster*, indicando rapidamente um componente JCL_Host para hospedar tal variável. Perceba que tal estratégia evita o acesso ao componente JCL_Server, reduzindo gargalos e centralizações. A limitação, conforme relata os autores em (ALMEIDA et al., 2019), é não haver garantia de uniformidade no armazenamento dos objetos no sistema DSM. O tamanho do objeto e a disponibilidade de memória dos JCL_Hosts também não são considerados, por exemplo, na estratégia de alocação.

O método $JCL.instantiateGlobalVar$, porção JCL_Host, recebe os argumentos do JCL_User e procede uma chamada a API JCL internamente, mais precisamente se chama o método $JCL.execute$, que cria uma tarefa assíncrona no mesmo JCL_Host. Isto ocorre para todas as variações do método $JCL.instantiateGlobalVar$. Para que se implemente a versão síncrona

do armazenamento ou instanciação de objetos no JCL sem integração ao Kafka, basta criar uma barreira de sincronização após a chamada ao método $execute$. Após execução, basta chamar o método $JCL.getResultBlocking$ com o identificador da tarefa para aguardar seu término ou obter seu resultado imediatamente, caso a tarefa já tenha terminado sua execução. Para a versão assíncrona do $JCL.instantiateGlobalVar$, basta retornar um objeto do tipo *Future*, ou seja, basta retornar a chamada ao método $JCL.execute$. A tarefa do *kernel* que armazena variáveis no JCL recebe o objeto que é a variável a ser armazenada no JCL_Host e seu nome, portanto basta inserir tal par *key-value* em um mapa local, existente em cada JCL_Host do *cluster*. Caso haja demanda por instanciar um objeto remotamente e não apenas armazená-lo, o JCL usa computação reflexiva do Java para tal procedimento antes de armazenar os valores no mapa local. Os argumentos $className$, $jars$ e $args$ são usados em tal procedimento.

Já no novo componente JCL_User, integrado ao Kafka, a API do método foi mantida, conforme ilustra o algoritmo 1, mas para que a variável global JCL seja armazenada ou mesmo instanciada remotamente de forma síncrona, a comunicação JCL_User e JCL_Host foi removida. O componente JCL_Host na nova versão JCL passa a não mais cuidar do serviço de armazenamento. O JCL_User na instanciação ou armazenamento de variáveis globais de forma síncrona passa a publicar diretamente o objeto e seu identificador para um tópico que permanece armazenado em um servidor Kafka denominado *worker*, portanto os *workers* Kafka assumem o papel dos JCL_Hosts usados anteriormente para armazenamento. Para que os demais JCL_Users vejam tal modificação, o Kafka notifica tais componentes inscritos ao tópico responsável pela variável global JCL. O método $JCL.instantiateGlobalVar$ assíncrono ainda usa o componente JCL_Host para proceder a instanciação ou mesmo o armazenamento remoto e, conforme previamente explicado, há internamente no JCL_Host uma chamada do método $JCL.execute$, portanto tal integração é detalhada uma única vez na Seção 4.5.

Algorithm 1 Instanciação ou armazenamento de variável no *cluster* JCL de forma síncrona

Data: key, value

1 JCL_User$_i$ instancia um objeto, caso necessário, usando os parâmetros $className$, $jars$ e $args$
2 JCL_User$_i$ produz a tupla T (key, value)
3 JCL_User$_i$ publica T para o tópico key
4 Servidor Kafka recebe a publicação de T
5 Servidor Kafka persiste T
6 Servidor Kafka notifica JCL_User inscrito no tópico key
7 JCL_User$_k$ recebe T
8 JCL_User$_k$ atualiza memória local com T

4.2 Reimplementação do método *JCL.getValue*

No JCL há como recuperar uma variável armazenada ou instanciada previamente de algumas maneiras. Uma delas é a obtenção sem bloqueio, precisamente usando o método $JCL.getValue$. Tal método é sempre síncrono e retorna o objeto com nome igual a key ou

resultado nulo. Este método não serve para aplicações que almejam alterar estados de objetos em ambientes concorrentes, pois não há garantias que estes estados não irão se corromper com a presença de concorrência.

Na versão anterior do JCL, o método $JCL.getValue(key)$ faz o mesmo processo de identificação de JCL_Host explicado para instanciação/armazenamento de variáveis, ou seja, se calcula o módulo do *hash* do key da variável dividido pelo tamanho do *cluster*. Uma vez identificado o JCL_Host, o método $JCL.getValue()$, porção JCL_User, procede solicitação pela rede de comunicação do respectivo objeto. Na porção JCL_Host do método há a aquisição do objeto no mapa local usado para armazenar variáveis globais JCL.

Já na versão integrada ao Kafka, há o conceito de memória local com as variáveis globais do JCL. O método $JCL.getValue(key)$ busca o objeto localmente na máquina onde o JCL_User está em execução, pois este é publicado após uma chamada $JCL.instantiateGlobalVar$. Caso o objeto ainda não tenha sido publicado aos interessados por tal variável, o método $JCL.getValue(key)$ faz com que a aplicação requisitante bloqueie sua execução até que o conteúdo esteja disponível em memória local.

Algorithm 2 Leitura local de variável global

Data: key

9 JCL_User$_i$ requisita a tupla T de chave key

10 JCL_User$_i$ obtém T em sua própria memória local

11 caso T não exista localmente, bloqueia até o término da publicação nas memórias locais do *cluster*

Conforme podemos perceber, o componente JCL_Host não participa do fluxo do método $JCL.getValue$ síncrono. A versão assíncrona é explicada mais à frente com o detalhamento do fluxo de execução do método $JCL.execute$ e suas variações.

4.3 Reimplementação do método *JCL.getValueLocking*

Para as aplicações que almejam alterar estados das variáveis globais JCL em ambientes concorrentes, o JCL implementa os métodos $JCL.getValueLocking$ e $JCL.setValueUnlocking$, os quais garantem acessos seguros, ou seja, de forma que acessos concorrentes não corrompam os valores das variáveis.

Na versão com Kafka, a aquisição de uma variável global é feita utilizando a própria estrutura de processamento e armazenamento de mensagens do Kafka. As mensagens enviadas para um tópico são armazenadas em um *log* ordenado e imutável (KLEPPMANN; KREPS, 2015). Cada mensagem inserida no *log* assume um identificador inteiro sequencial único caracterizando sua posição de chegada. O Kafka garante que não há identificadores repetidos em um mesmo *log*, pois o acesso ao *log* é serializado, isto é, apenas uma única *thread* escreve por vez. Desta forma, a *thread* que acessar o *log* primeiro possui menor identificador do que as posteriores e assim

sucessivamente. Este mecanismo abre diversas possibilidades de implementações de diversos protocolos usando o *log* Kafka, incluindo o baseado nas primitivas *acquire/release* que optamos por usar.

A aquisição e bloqueio de uma variável global foi implementada no componente JCL_User, portanto também não há utilização do componente JCL_Host nos acessos concorrentes. O controle de acesso a variável foi desenhado para que, por meio de publicações e notificações feitas concorrentemente entre JCL_Users e o Kafka, a semântica de *acquire/release* seja mantida de forma atômica, inibindo com isto que mais de uma *thread* obtenha a mesma variável JCL no *log* Kafka ao mesmo tempo.

Internamente, as *threads* que almejam obter uma variável global X publicam mensagens informando interesse de aquisição - primitiva *acquire* de X. Essas mensagens são inseridas no *log* de mensagens gerenciado pelo tópico X, assumindo identificadores sequenciais. A *thread* que enviou a mensagem inserida mais cedo acessa a variável global X e a bloqueia até que uma mensagem de liberação para X seja publicada pela *thread* que a conseguiu bloquear. As demais *threads* que enviaram mensagens após o bloqueio de X conseguem saber que há um bloqueio ao receber as notificações com identificadores maiores do que o usado no bloqueio. Caso apenas um JCL_User esteja interessado no bloqueio, apenas um identificador é enviado e este é o menor. O algoritmo 3 descreve como uma aquisição é feita.

Todo bloqueio de variável no JCL sempre inicia com a geração de um *token* único. Esse *token* caracteriza unicamente cada tentativa de aquisição a uma variável global. O *token* é então publicado para o Kafka, especificamente para o *worker* responsável pelo tópico que armazena no *log* de publicações da variável requisitada. A publicação é adicionada ao *log* e recebe um identificador sequencial único, correspondente a posição que o *token* assumiu. Os JCL_Users inscritos no tópico, isto é, interessados na variável global, recebem a notificação com o valor do *token*, conhecendo a tentativa de aquisição. O JCL_User que publicou o *token* o conhecia previamente, portanto consegue verificar se o mesmo é o de menor índice dentro do *log*. Caso seja, o JCL_User em questão prossegue para a aquisição de acesso da variável e os demais JCL_Users aguardam até que um outro *token* de *release* seja publicado. O próximo JCL_User na fila de espera consegue obter acesso apenas seguindo os valores dos *tokens*, ou seja, basta seguir o mesmo procedimento para que o mecanismo de *acquire/release* proceda corretamente.

Algorithm 3 Aquisição de variável no *cluster* JCL

Data: nickname

JCL_User$_i$ requisita a tupla T de chave key
JCL_User$_i$ gera *token*
JCL_User$_i$ publica *token* para o tópico key
Servidor Kafka recebe a publicação de *token*
Servidor Kafka persiste *token*
Servidor Kafka notifica JCL_User inscrito no tópico key
JCL_User$_i$ recebe *token*
JCL_User$_i$ armazena localmente *token* e posição de chegada do *token* no *log* do tópico key
while *token não for o de menor posição de chegada* **do**
    JCL_User$_i$ aguarda
JCL_User$_i$ obtém T em sua memória local

4.4 Reimplementação do método *JCL.setValueUnlocking*

Na versão com Kafka, a liberação de uma variável global - primitiva *release* - é feita usando publicações para tópicos no Kafka. Dado que um JCL_User adquiriu uma variável, este publica uma mensagem informando a liberação da mesma usando seu identificador key. Os demais JCL_Users inscritos no tópico correspondente ao key são notificados sobre a publicação e atualizam suas respectivas memórias locais. Após tal publicação, a variável torna-se desbloqueada e apta a ser adquirida pelo próximo JCL_User interessado e registrado no *log* Kafka da variável em questão. Para isto, basta seguir os valores dos *tokens* publicados anteriormente.

Algorithm 4 Atualização e desbloqueio de variável no *cluster* JCL

Data: key, value

JCL_User$_i$ produz a tupla T (key, value)
JCL_User$_i$ publica T com informação de desbloqueio da variável com identificador key para o tópico key
Servidor Kafka recebe a publicação de T
Servidor Kafka persiste T
Servidor Kafka notifica JCL_User inscrito no tópico key
JCL_User$_k$ recebe T com informação de desbloqueio da variável com identificador key para o tópico key
JCL_User$_k$ atualiza memória local com T

De forma análoga ao método $JCL.getValueLocking$, o componente JCL_Host não participa do fluxo de execução do método $JCL.setValueUnlocking$ usando o Kafka, pois cabe ao Kafka toda a tarefa de armazenamento na nova versão JCL.

4.5 Reimplementação dos métodos *JCL.execute*, *JCL.executeAll*, *JCL.executeAllcores* e *JCL.executeOnHost*

Além de instanciar e armazenar variáveis, o JCL permite executar tarefas de forma assíncrona e para isto há várias alternativas. De uma forma geral, o componente JCL_User sempre se comunica com um ou vários JCL_Hosts para proceder a execução de tarefas. O componente JCL_User não sofreu alteração na versão JCL integrada ao Kafka para execução de tarefas e para instanciação/armazenamento de variáveis de forma assíncrona. Para todos estes casos o componente JCL_User continua a se comunicar com um componente JCL_Host no *cluster* JCL.

Algorithm 5 Execução de tarefa no *cluster* JCL

Data: Class name, Class method name, Args
30 JCL_User$_i$ requisita execução de tarefa
31 JCL_User$_i$ recebe um *Future* como *ticket* da execução da tarefa
32 JCL_Host$_i$ recebe requisição de execução de tarefa
33 JCL_Host$_i$ executa tarefa
34 JCL_Host$_i$ publica resultado de execução da tarefa

Na versão integrada ao Kafka, o componente JCL_User continua a receber um objeto do tipo *Future* oriundo de um JCL_Host, o que garante a computação assíncrona no JCL. Internamente, no JCL_Host a tarefa é executada e tão logo esta termine, o seu resultado é publicado para um tópico armazenado no Kafka (Algoritmo 5). Para que os JCL_Users conheçam o resultado da execução, estes manifestam seus interesses quando chamam o método get do objeto *Future* retornado em cada chamada assíncrona. O mesmo ocorre quando JCL_Users chamam $JCL.getValueLocking$, $JCL.getValue$ ou $JCL.getResultBlocking$. Em suma, quando um JCL_User usa o identificador de uma variável ou de um resultado de uma tarefa ele está manifestando interesse naquele tópico, portanto se inscrevendo neste tópico, caso ainda não esteja inscrito.

Note que foi decisão arquitetural utilizar um tópico por resultado de tarefa ou variável global criada ou mantida no JCL, pois assumimos que são tópicos distintos e com interesses não necessariamente correlacionados. O número de tópicos aumenta quando muitos objetos são mantidos no ambiente DSM e isto é um limitador, conforme aponta os experimentos realizados. No Capítulo 5 há uma discussão sobre os pontos positivos e negativos de tal desenho arquitetural, assim como os resultados obtidos em diversos cenários experimentais com e sem a integração com o Kafka.

As variações do método $JCL.execute$, precisamente as chamadas da API $JCL.executeAll$ e $JCL.executeAllCores$, onde há possibilidade de executar inúmeras tarefas no *cluster*, são extensões da explicação acima, onde ao invés de uma publicação para um tópico se realiza inúmeras publicações para diversos tópicos, sendo uma para cada tarefa submetida. O custo

computacional de tais variações do método $JCL.execute$ é enorme, conforme demonstra os experimentos realizados e detalhados no Capítulo 5. O método $JCL.executeOnHost$ possui a mesma lógica do método $JCL.execute$ sob ponto de vista de integração ao Kafka, pois este permite que o programador escolha em qual JCL_Host executar uma tarefa.

4.6 Reimplementação dos métodos *JCL getResultBlocking* e *JCL getAllResultsBlocking*

Após submeter tarefas, o programador pode recuperar seus resultados. Para isto, há os métodos $JCL.getResultBlocking$ e $JCL.getAllResultsBlocking$. Na versão com o Kafka, há o conceito de memória local, conforme explicado anteriormente. Tal memória é usada para o armazenamento das variáveis globais e também para o armazenamento dos resultados de execuções de tarefas no JCL. O método $JCL.getResultBlocking$ busca o resultado de uma tarefa localmente, pois este é publicado após uma chamada de algum método de execução no JCL_Host. Caso o resultado ainda não tenha sido publicado aos interessados, o método $JCL.getResultBlocking$ bloqueia a aplicação até que o resultado esteja disponível na memória local. Na versão Kafka implementamos uma espera condicional no JCL_User, pois o Kafka é assíncrono ao manipular tópicos.

A assinatura $JCL.getAllResultsBlocking$ realiza o mesmo procedimento explicado anteriormente, contudo para obtenção de inúmeros resultados de tarefas, pois trata-se de uma barreira de sincronização total. Uma lista de objetos do tipo *Future* deve ser usada para a chamada $JCL.getAllResultsBlocking$.

Algorithm 6 Leitura local de resultado de execução de tarefa

Data: Future

35 JCL_User$_i$ requisita resultado usando o objeto *Future*, precisamente seu método get

36 JCL_User$_i$ obtém resultado usando memória local

37 caso resultado da tarefa não exista localmente, aguarde até o término da publicação nas memórias locais do *cluster*

Na porção JCL_Host dos métodos $JCL.getResultBlocking$, $JCL.getValueLocking$ e $JCL.getValue$, todos integrados ao Kafka, não ocorre alteração para obtenção de resultados, pois estes efetuam apenas a publicação. Conforme explicado anteriormente, é no componente JCL_User que as modificações para obtenção de dados no *cluster* é feita, pois tal componente deve esperar pela notificação enviada pelo servidor - *worker* - Kafka.

Há variações do método $JCL.getResultBlocking$, mas sem bloqueio da aplicação. São eles: $JCL.getResult$ e $JCL.getAllResults$. Basicamente, há a checagem da memória local no JCL_User sem qualquer bloqueio da aplicação para o caso do resultado da tarefa ainda não existir. Devido a simplicidade de implementação, tais métodos não são detalhados neste trabalho.

4.7 Reimplementação do mapa distribuído *JCL HashMap*

JCL implementa uma versão distribuída da interface Java Map, permitindo ao programador utilizar estruturas de dados familiares a comunidade Java (ALMEIDA et al., 2019). Sua utilização requer refatorações mínimas, pois há necessidade apenas de substituir os subtipos de objetos Map (Ex. Tree Map, Hash Map, etc.) pelo subtipo JCL HashMap, o que faz com que o armazenamento que antes era feito local torne-se distribuído. Os métodos get, $getLocking$, put e $putAll$ sofreram algum tipo de alteração na integração com o Kafka.

Internamente, para implementar os métodos get, $getLocking$ e put se utiliza os métodos já explicados $getValue$, $getValueLocking$, $setValueUnlocking$, respectivamente. Isto se deve ao fato do mapa JCL ser um conjunto de variáveis globais com um mesmo prefixo, que no caso é o nome do mapa. O método $putAll$ é um conjunto de chamadas do método put e, consequentemente, $setValueUnlocking$ previamente explicado. Em linhas gerais, o mapa JCL é apenas uma metáfora que não existe internamente no JCL, havendo apenas um conjunto de variáveis globais com seus respectivos identificadores.

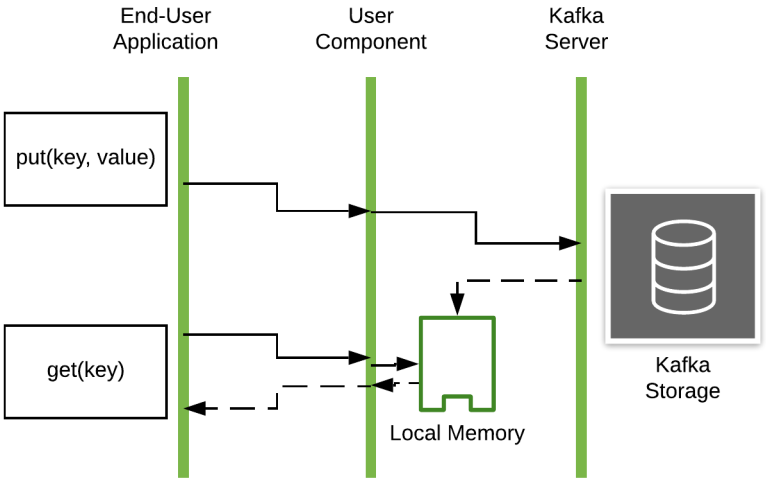

Figura 4.1 – JCL HashMap

Cada variável da JCL Hash Map possui um identificador único fornecido pelo programador na sua criação. Com este identificador único, qualquer aplicação JCL pode acessar o mapa previamente criado no *cluster*. Múltiplos mapas JCL podem ter chaves(*keys*) idênticas no *cluster* JCL, contudo, mapas diferentes devem possuir identificadores distintos para evitar sobreposições no *cluster*. Para permitir a implementação eficiente de alguns métodos da interface Java Map, tais como $clear()$, $containsKey(Objectkey)$ e $containsValue(Objectvalue)$, a lista de todas as chaves(*keys*) é armazenada em um único tópico ao qual o JCL_User se inscreve ao instanciar o mapa. Todo JCL_User que manifesta interesse por alguma entrada de um determinado mapa

também se inscreve no tópico de chaves (*keys*) para ser notificado quando o mapa sofre alguma alteração.

5 Experimentos e Avaliações

Este capítulo objetiva avaliar a nova implementação do *middleware* JCL integrada ao Kafka, comparando-a com sua versão mais recente em operação sem integração [1].

5.1 Configuração do ambiente

Os experimentos foram executados num *cluster* composto por 8 máquinas *octo core* e com 16 GB de RAM cada. As maquinas possuem sistema operacional Linux Ubuntu 16.04 LTS 64 bits. As máquinas possuem processadores Intel(R) Core(TM) i7-3770 CPU operando a 3.40GHz e memória RAM 16Gib DDR3 Síncrono 1600 MHz. As máquinas são interligadas por *switch* Gigabit 1GigE padrão Ethernet (IEEE 802.3-2005).

Os experimentos foram executados 5 vezes para que a média fosse obtida, tentando diluir as interferências diversas de rede de comunicação, sistema operacional e *softwares* básicos. Os experimentos coletaram a métrica *runtime* que representa o período que o programa de computador permanece em execução. No nosso cenário, significa o tempo que a aplicação teste, que inclui o componente JCL_User, leva para invocar a API JCL, portanto inclui o tempo de rede e o tempo de processamento remoto onde um ou vários JCL_Hosts ou Kafka *workers* são contactados. O Kafka foi executado sem tolerância a falhas para melhor desempenho, pois o JCL nativo não possui tal requisito. Quando há tolerância a falhas no Kafka para avaliarmos tal sobrecarga, há explicitamente tal informação no experimento realizado. Há apenas uma aplicação cliente testando o *cluster* com as 8 máquinas, portanto tal aplicação é *multithread* para garantir a execução da API JCL concorrentemente.

5.2 Experimentos do serviço de armazenamento

No primeiro cenário, foram testados os serviços de armazenamento e manipulação de variáveis globais e pares *key-value* do mapa distribuído do JCL. O *runtime* foi obtido em diversas rodadas de execuções dos métodos $getValue$, $getValueLocking$, $setValueUnlocking$, $JCLHashMap.get$, $JCLHashMap.getLock$ e $JCLHashMap.putUnlock$ utilizando as versões com e sem integração ao Kafka.

5.2.1 Acesso a variáveis sem bloqueio

Para avaliar a leitura de variáveis armazenadas pelo JCL com e sem integração ao Kafka, foram feitas oito mil, dezesseis mil e oitenta mil leituras de cem variáveis instanciadas em dois,

[1] site Java Cá & Lá - www.javacaela.org

quatro e oito JCL_Hosts para a versão não integrada ao Kafka e em dois, quatro e oito servidores Kafka (Kafka *workers*) para a versão JCL integrada.

A versão JCL integrada ao Kafka se mostrou mais eficiente quando quando há muita leitura das cem variáveis instanciadas (Tabela 5.1). Uma justificativa para tal comportamento é que a sobrecarga causada pelo número de inscrições em tópicos e notificações feitas pelo Kafka se torna vantajosa com o uso da memória local apenas quando dezenas de milhares de leituras são feitas. No caso do JCL sem integração ao Kafka, o acesso a rede de comunicação é obrigatório, portanto sempre há 8k, 16k e 80k comunicações remotas com o componente JCL_Host. O que é importante ressaltar é que para compensar o uso do Kafka com a memória local a variável deve sofrer muita leitura, pois caso contrário sua sobrecarga é enorme.

Foram realizados experimentos com os tipos *INTEGER* e *COLLECTION* com cem itens em cada *COLLECTION* e os resultados foram similares, portanto tais resultados são apresentados apenas no Anexo deste trabalho.

Tabela 5.1 – GET não bloqueante (segundos) STRING

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
8k GETS	2,655	18,605	3,084	12,077	2,706	11,082
16K GETS	4,523	19,106	4,819	12,817	4,461	11,635
80K GETS	19,335	21,159	19,810	13,017	18,833	12,710

Um aspecto importante a ser avaliado em leituras sem bloqueio é o comportamento da versão JCL integrada ao Kafka ao variar a quantidade de variáveis globais. Neste experimento, o número de Kafka *workers* foi fixado em oito e a quantidade de variáveis nas quais as leituras são feitas variaram entre dez, vinte e cem. Foram coletados o *runtime* para oito mil, dezesseis mil e oitenta mil leituras sem bloqueio.

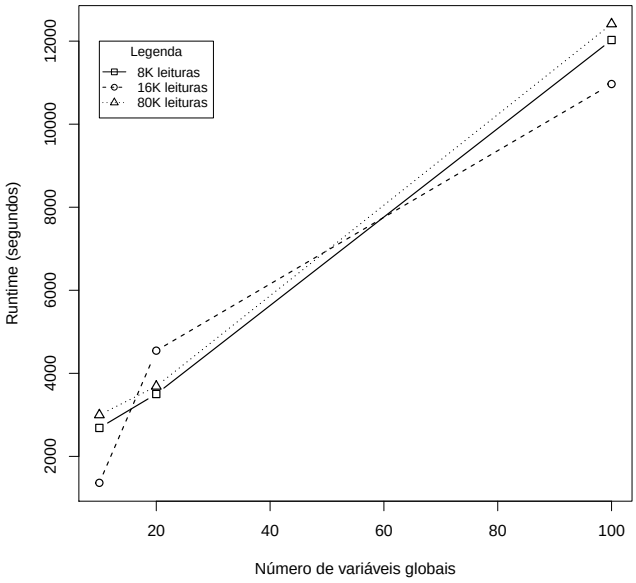

Figura 5.1 – GET sem bloqueio para diferentes quantidades de variáveis na versão JCL integrada ao Kafka

A Figura 5.1 mostra que o *runtime* aumenta à medida que a quantidade de variáveis lidas aumenta, independentemente do número de leituras feitas. A justificativa é o número de inscrições feitas em tópicos Kafka que armazenam as variáveis globais. É custoso inscrever um JCL_User em um tópico para que ocorra o carregamento da variável em memória local e consequentemente a notificação que destrava a espera para a primeira leitura. Após realizada a primeira leitura e o desbloqueio da aplicação, as demais leituras são extremamente rápidas, pois utilizam memória local, portanto torna-se praticamente irrelevante a quantidade de leituras realizadas sob ponto de vista do tempo de acesso a tais variáveis. No JCL sem a integração ao Kafka e, consequentemente, sem memórias locais o *runtime* aumentaria à medida que o número de leituras aumentasse.

5.2.2 Acesso ao mapa sem bloqueio

Para avaliar a leitura de pares *key-value* de um mapa JCL com e sem integração ao Kafka, foram feitas oito mil, dezesseis mil e oitenta mil leituras de cem entradas em tal mapa armazenadas em dois, quatro e oito JCL_Hosts para a versão não integrada ao Kafka e em dois, quatro e oito servidores Kafka (Kafka *workers*) para a versão JCL integrada ao Kafka.

Tabela 5.2 – GET não bloqueante em um mapa (segundos) COLLECTION

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
8k GETS	2,353	13,077	2,532	11,460	2,425	10,551
16K GETS	4,131	12,899	4,497	11,743	4,392	11,187
80K GETS	18,985	11,615	19,365	11,471	19,217	11,841

Conforme esperado, o comportamento foi similar ao acesso não bloqueante a variáveis globais JCL. Isto se deve ao fato do mapa ser implementado como diversas variáveis globais, ou seja, cada par *key-value* é uma variável no *cluster*, portanto a mesma lógica é aplicada. Novamente, o JCL integrado ao Kafka se comporta superior a versão não integrada apenas quando o número de leituras é 80k (Tabela 5.2). Devido ao tipo *COLLECTION* conter cem itens internamente, este é maior do que a *STRING* usada no experimento ilustrado na Tabela 5.1, portanto o ganho da versão integrada ao Kafka ocorre já com dois JCL_Hosts no *cluster*. A justificativa é a mesma para tal comportamento. Em linhas gerais, apenas entradas de um mapa JCL com muitas leituras compensarão a infraestrutura do Kafka, incluindo a adoção de memória local.

Os resultados experimentais do mapa com outros tipos, precisamente *STRING* e *INTEGER* estão no Anexo deste trabalho, pois o comportamento foi similar ao apresentado nesta seção.

5.2.3 Acesso a variáveis com bloqueio

Para avaliar o acesso bloqueante de variáveis armazenadas pelo JCL com e sem integração ao Kafka, foram feitas oitocentas, mil e seiscentas e oito mil leituras de cem variáveis instanciadas em dois, quatro e oito JCL_Hosts para a versão não integrada ao Kafka e em dois, quatro e oito *workers* Kafka para a versão JCL integrada.

Tabela 5.3 – GET + SET bloqueante (segundos) INT

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k GET/SET	1,582	24,688	1,491	20,954	1,536	19,524
1.6K GET/SET	2,567	31,786	2,654	24,262	2,647	23,367
8K GET/SET	10,608	53,288	11,816	41,219	10,936	43,702

A Tabela 5.3 mostra que, para acessos bloqueantes a valores de variáveis globais, sejam estas *INT*, *STRING* ou *COLLECTION*, a versão JCL sem integração ao Kafka é muito eficiente em relação a versão com integração ao Kafka. A discrepância entre valores pode chegar a 20 vezes e isto se deve a sobrecarga do Kafka na primitiva *acquire/release* implementada com a utilização de seu *log*. Após aquisição da variável há notificação e após liberação da mesma há

nova notificação, portanto com o Kafka as notificações ocorrem duas vezes e mesmo assim o interessado na variável precisa acessar a rede de comunicação para o bloqueio, não sendo útil a memória local. Nos experimentos, ambas as versões JCL tiveram que acessar a rede de comunicação 0.8k, 1.6k e 8k vezes, portanto o JCL nativo, que não possui memória local, mas mantêm apenas uma fila para um mapa em RAM em um JCL_Host facilmente detectado no *cluster*, se mostrou muito mais rápido.

Devido aos resultados com os tipos *STRING* e *COLLECTION* terem sido similares, optamos por apresentá-los apenas no Anexo deste trabalho.

5.2.4 Acesso ao mapa com bloqueio

Para avaliar o acesso bloqueante ao mapa armazenado pelo JCL com e sem integração ao Kafka, foram feitas oitocentas, mil e seiscentas e oito mil leituras de cem pares *key-value* de um mapa JCL armazenados em dois, quatro e oito JCL_Hosts para a versão não integrada ao Kafka e em dois, quatro e oito *workers* Kafka para a versão JCL integrada ao Kafka.

Tabela 5.4 – GET + PUT bloqueante em mapa (segundos) COLLECTION

	JCL Hosts ou servidores Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k GET/PUT	2,123	26,697	1,936	20,390	1,897	20,201
1.6K GET/PUT	3,728	28,882	3,549	30,421	3,647	29,346
8K GET/PUT	14,649	54,849	15,243	48,916	15,290	44,946

Conforme esperado, o JCL Kafka possui uma sobrecarga muito elevada em relação ao JCL nativo. Os resultados para o tipo *COLLECTION* com cem itens cada demonstraram que a implementação das primitivas *acquire/release* feitas sob o *log* Kafka não obtiveram êxito. O uso de outras políticas de bloqueio podem não surtir efeito quando realmente há 0.8k, 1.6k e 8k alterações de cem entradas de um mapa distribuído ou cem variáveis armazenadas em *workers* Kafka. Um bloqueio otimista que permita escrever na memória local e realizar *rollback*, caso o bloqueio não tenha ocorrido na ordem esperada, pode reduzir um pouco o *runtime*, mas também pode piorar o atual comportamento se houver 0.8k, 1.6k e 8k operações de *rollback* (pior caso). Além disto, há consumo de memória extra para salvar os estados passados dos objetos para o caso de precisar retroagir.

5.2.5 Acesso ao Kafka tolerante a falhas

Um aspecto importante a ser avaliado é o comportamento da versão JCL integrada ao Kafka ao variar o grau de tolerância a falha. Neste experimento, o número de *workers* Kafka está fixado em oito, a quantidade de leituras variam entre oitocentas, mil e seiscentas e oito mil. Os

fatores de replicação do armazenamento feito pelo Kafka foram um e dois, ou seja, o mesmo tópico esteve armazenado em um *worker* Kafka no primeiro experimento e em dois *workers* no segundo experimento. Os resultados são apresentados na Tabela 5.5.

O aumento no *runtime* não ultrapassou 15% ao inserir replicação nível dois, portanto a solução Kafka demonstra sua eficiência neste quesito. É importante ressaltar que isto não melhora o desempenho insatisfatório na implementação das primitivas *acquire/release* apresentadas nas seções anteriores.

Tabela 5.5 – GET + SET bloqueante (segundos) COLLECTION. Kafka sem replicação e com replicação nível dois

	Fator de replicação	
	1	2
0.8k GET/SET	21,482	23,242
1.6K GET/SET	26,815	29,257
8K GET/SET	48,193	55,172

5.3 Experimentos do serviço de execução de tarefas

No segundo cenário de experimentos foram testados os serviços de execuções de tarefas. O *runtime* foi obtido em diversas rodadas de execuções dos métodos $JCL.execute$, $JCL.executeAll$ e $JCL.executeAllCores$, também utilizando as versões do JCL com e sem integração ao Kafka.

Cada execução de tarefa considera também a obtenção do seu resultado, mesmo que o JCL adote computação assíncrona para o processamento de tarefas no *cluster*. Sempre após uma chamada a API para execução de uma tarefa é feita uma segunda chamada ao método $getResultBlocking$, sendo este responsável por criar uma barreira de sincronização para a espera do resultado da tarefa. A tarefa usada nos experimentos é uma PA (Progressão Aritmética), portanto três argumentos do tipo inteiro são repassados a cada chamada da API JCL em ambas versões.

5.3.1 *JCL.execute*

Para avaliar a execução de tarefas usando o método $JCL.execute$, foram submetidas oitocentos, um mil e seiscentos, e oito mil execuções em dois, quatro e oito JCL_Hosts para a versão não integrada ao Kafka e a mesma quantidade de servidores Kafka (Kafka *workers*) para a versão JCL integrada.

Tabela 5.6 – EXECUTE (segundos)

	JCL Hosts ou servidores Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k EXECUTE	280	30,281	573	68,012	764	134,512
1.6K EXECUTE	355	28,200	592	55,589	885	112,478
8K EXECUTE	272	26,502	609	51,202	759	102,437

Nestes experimentos de execução de tarefas a diferença de *runtime* entre as versões JCL foi ainda mais discrepante. Quanto maior o *cluster* maior o *runtime* para executar as mesmas 0.8k, 1.6k e 8k tarefas, assim como a obtenção de seus resultados. Isto se deve em boa parte devido ao aumento da comunicação na rede de dados. No caso específico do JCL integrado ao Kafka, há o fluxo de publicação para a execução da tarefa e há o fluxo da primitiva *acquire/release* para a realização da chamada $getResultBlocking$, pois esta, conforme já explicado, cria uma barreira de sincronização para espera do resultado da tarefa.

São criados 0.8k, 1.6k e 8k tópicos no Kafka e isto o degrada excessivamente. A escolha por um tópico por cada tarefa submetido ao *cluster* se deve ao fato do JCL ser colaborativo, portanto multi-usuário. Uma mesma aplicação ou aplicações distintas podem almejar o resultado de uma tarefa e não seria interessante agrupar as tarefas por usuário, por *thread* ou qualquer outra abstração, pois nestes casos haveria notificações desnecessárias para inscritos em tópicos que não possuem qualquer interesse no mesmo. Em linhas gerais, a sobrecarga do Kafka chega a milhares de vezes quando milhares de tópicos são criados e devem ser publicados, assim como milhares de bloqueios devem ser feitos para obtenção de inúmeros resultados de tarefas. Note que tal comportamento ocorre mesmo com poucos inscritos nos tópicos, pois conforme explicado há apenas uma aplicação *multi-thread* realizando o papel de cliente nos experimentos, portanto há poucas notificações a serem feitas em termos de *subscribers*.

5.3.2 *JCL.executeAll*

Para avaliar a execução de tarefas sendo executas em todo o *cluster* com o método $JCL.executeAll$, foram feitas oitocentos, um mil e seiscentos, e oito mil execuções em dois, quatro e oito JCL_Hosts para a versão não integrada ao Kafka e as mesmas execuções em dois, quatro e oito servidores Kafka (Kafka *workers*) para a versão JCL integrada.

Tabela 5.7 – EXECUTE ALL (segundos)

	JCL Hosts ou servidores Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k EXECUTE ALL	486	57,198	766	102,437	1,285	273,631
1.6K EXECUTE ALL	736	99,641	1,315	205,421	1,847	389,717
8K EXECUTE ALL	978	170,774	1,401	346,095	2,708	798,261

O método $JCL.executeAll$ atribui uma mesma tarefa ou tarefas distintas a cada máquina do *cluster*, evitando concorrência em cada máquina. Como apenas uma tarefa é processada por máquina, num *cluster* com oito máquinas tivemos (0.8k $\times$ 8), (1.6k $\times$ 8) e (8k $\times$ 8) tarefas submetidas no experimento, ou seja, oito vezes mais tarefas do que o experimento da Seção 5.3.1, portanto se espera um comportamento oito vezes pior.

A Tabela 5.7 mostra que o fator de multiplicação de oito vezes ocorreu apenas quando 8k tarefas do tipo $JCL.executeAll$ foram submetidas, portanto quando o Kafka teve que criar 64k tópicos e realizar 64k fluxos para atender a primitiva *acquire/release* implementada com seu *log*. Nos demais cenários, a piora no *runtime* foi em torno de três vezes para oito vezes mais execuções de tarefas, portanto satisfatória nesta perspectiva.

5.3.3 *JCL.executeAllCores*

A última chamada da API JCL para execução de tarefas é a mais custosa e se chama $JCL.executeAllCores$. Por conta disto, foram feitas apenas quarenta execuções em oito JCL_Hosts para a versão não integrada ao Kafka e as mesmas quarenta em oito servidores Kafka (Kafka *workers*) para a versão JCL integrada.

Tabela 5.8 – EXECUTE ALL CORES (segundos)

	JCL Hosts ou servidores Kafka	
	8	
	JCL	JCL Kafka
0.04k EXECUTE ALL CORES	612	99,417

Neste cenário ocorreram (40 $\times$ 8 $\times$ 8) tarefas submetidas ao *cluster*, pois cada máquina é *octocore* e a primitiva $JCL.executeAllCores$ atribui uma tarefa por *core* do *cluster*, portanto 64 tarefas a cada chamada da API JCL. Conforme esperado, a versão integrada possui enorme degradação devido ao Kafka e seus procedimentos internos, assim como a implementação da primitiva de bloqueio implementada neste trabalho. Os resultados foram dezenas de milhares de vezes piores da versão integrada ao Kafka frente a versão já existente do JCL, portanto para os serviços de execução de tarefas e obtenção de seus resultados o Kafka parece não ser mesmo uma boa opção, em especial quando há acessos bloqueantes aos dados.

5.4 Discussões

Os resultados experimentais mostraram que a integração com o Kafka traz enormes sobrecargas, não somente em termos de sua política de criação de tópicos, inscrição em tópicos e publicação de mudanças nos mesmos - padrão *publish/subscribe* - mas também na implementação da estratégia de bloqueio baseada nas primitivas *acquire/release* feitas neste trabalho. O uso do *log* armazenado em disco é outro limitante que traz consideráveis acréscimos ao *runtime*.

Por outro lado, com a adoção de uma solução baseada no padrão *publish/subscribe* as notificações indiretas passam a ocorrer, tornando mais flexível a propagação das atualizações de objetos e resultados de tarefas ou mesmo a submissão destas num *cluster* que pode ter seu tamanho redimensionado durante o ciclo de vida da aplicação. Outra vantagem é que a aplicação responsável pela criação, armazenamento ou submissão das tarefas não precisa estar ativa para que outras aplicações possam fazer uso dos objetos criados ou dos resultados das tarefas anteriormente encaminhadas. Por fim, a tolerância a falha de soluções baseadas em tópicos traz resiliência para as aplicações DSM e orientadas a tarefas, algo extremamente útil para cenários onde empresas precisam operar em escala e com taxas muito baixas de *down-time*.

O uso do JCL integrado ao Kafka serviu para demonstrar que o modelo de programação DSM pode ter vantagens quando o número de leituras de objetos é muito grande ou quando o número de inscritos em tópicos aumenta e se torna esparso pelo mundo. Entretanto, quando o acesso passa a ser de escrita e concorrente a solução tende a não escalar, pois os custos de publicação e bloqueio se tornam impeditivos.

Em relação ao modelo de programação orientado a tarefas, parece que soluções como o Kafka seriam úteis apenas para publicar a execução de tais tarefas a possíveis interessados nestas abstrações que estejam igualmente localizados de maneira esparsa em ambientes diversificados e heterogêneos, incluindo *grids* computacionais com diversos protocolos e inúmeras políticas de segurança. A porção do modelo de programação onde ocorre o bloqueio para obtenção dos resultados das tarefas sofre das mesmas limitações apresentadas para o modelo DSM, portanto cabe muita ressalva a utilização do padrão *publish/subscribe* neste contexto.

6 Conclusão

Neste trabalho apresentamos uma reimplementação dos modelos de programação DSM e orientado a tarefas. Tais modelos são normalmente implementados usando comunicação explícita ou direta entre nós de um *cluster* para se manipular objetos, submeter tarefas ou mesmo obter resultados de tarefas. Isto pode gerar uso desnecessário da rede de comunicação, pois nem sempre que o provedor de serviço é consultado há atualizações das abstrações mantidas por tais modelos, portanto cópias idênticas podem ser obtidas desnecessariamente. Foi investigado a possibilidade de uma solução baseada no padrão *publish/subscribe* ser usada na reimplementação de tais modelos de programação, sem alterar suas APIs e consequentemente sem refatoração dos códigos existentes. Com isto, a comunicação indireta por meio de notificações e inscrições em tópicos passou a ser o desenho arquitetural da solução. As primitivas *acquire/release* foram reimplementadas usando a solução de mensageria escolhida. A ideia foi notificar os interessados nos objetos ou tarefas ou resultados de tarefas apenas quando atualizações ocorressem, evitando o uso da rede de comunicação quando cópias atualizadas existissem em memória local.

Kafka e JCL foram as soluções escolhidas para validação da hipótese deste trabalho. O Kafka é um dos líderes mundiais e o JCL uma das únicas soluções que integra duas tecnologias fundamentais: HPC e IoT. Na literatura não encontramos estudos sobre o uso do Kafka para objetivo deste trabalho, portanto os resultados servem como um caso de estudo de tal plataforma. Alguns trabalhos acadêmicos foram encontrados com objetivos similares ao deste trabalho, mas nenhum realmente endereçou solução para o problema apresentado neste estudo, portanto os resultados expostos são inéditos também sob ponto de vista teórico e não apenas tecnológico.

Os experimentos realizados testaram os serviços dos modelos de programação em avaliação, precisamente os serviços de armazenamento de objetos e processamento de tarefas. Uma versão do JCL sem integração ao Kafka foi testada contra a versão integrada e vários testes foram conduzidos num *cluster* de máquinas multiprocessadas. Os resultados demonstraram que o Kafka se torna útil quando a leitura de objetos aumenta, ou seja, quando o interesse pelo objeto compensa o custo do Kafka em criar tópico, inscrever interessados no tópico e publicar mudanças ocorridas neste tópico aos inscritos no mesmo. O uso de memórias locais nas leituras recorrentes representa uma enorme redução da latência, o que auxilia e muito nos resultados positivos da versão JCL Kafka. Entretanto, quando há bloqueio de objetos para o acesso concorrente de escrita nos mesmos, o JCL integrado ao Kafka perde em todos os cenários, ou seja, em todos os tamanhos de *cluster* e em todas as quantidades de chamadas a API. A justificativa é que a memória local passa a não ser mais útil e o acesso a rede de comunicação ocorre a toda chamada ao JCL. Mais do que isto, o Kafka tem que publicar duas vezes para que as primitivas *acquire/release* sejam implementadas e seu *log* é também acessado duas vezes para que *tokens* de controle sejam inseridos. Tudo isto faz com que o JCL integrado ao Kafka chegue a ficar vinte vezes mais lento

do que a versão sem integração.

Os resultados dos serviços de submissão e obtenção de resultados de tarefas não foram promissores para o Kafka e, consequentemente, para soluções baseadas no padrão *publish/subscribe*. A execução de tarefas de forma assíncrona no JCL chegou a registrar *runtime* com diferença de milhares de vezes entre as versões testadas. Infelizmente a integração se mostrou inviável, pois a criação de tópicos invalida qualquer tentativa de competitividade com soluções nativamente orientadas a tarefas. Se uma tarefa fizer com que o Kafka crie um tópico, o mesmo tem que criar milhares de tópicos durante a execução de uma aplicação orientada a tarefas e isto o degrada demasiadamente.

Como trabalhos futuros outras soluções de controle de mensagens baseadas no padrão *publish/subscribe*, tais como Apache ActiveMQ, RabbitMQ, RocketMQ, Google Cloud Pub/Sub e Apache Pulsar, podem ser usadas no JCL para uma avaliação comparativa. Outros protocolos para garantir a coerência entre as memórias locais podem ser implementados. A verificação de alternativas otimistas para controle de concorrência que usam operações de *rollback* para os casos de insucesso de bloqueio podem ser testadas para tentar agilizar as escritas de objetos compartilhados. Outras estratégias de criação de tópicos podem ser avaliadas, pois a atual não compartilha variáveis ou resultados de tarefas num mesmo tópico e isto gerou enormes gargalos.

Referências

ABBOTT, M. L.; FISHER, M. T. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. [S.l.]: Pearson Education, 2009.

ALMEIDA, A. L. B. et al. A general-purpose distributed computing java middleware. *Concurrency and Computation: Practice and Experience*, v. 31, n. 7, p. e4967, 2019. E4967 cpe.4967. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4967>.

AMZA, C. et al. Treadmarks: Shared memory computing on networks of workstations. *Computer*, IEEE, v. 29, n. 2, p. 18–28, 1996.

ANTONIU, G.; BOUGÉ, L. Dsm-pm2: A portable implementation platform for multithreaded dsm consistency protocols. In: SPRINGER. *International Workshop on High-Level Parallel Programming Models and Supportive Environments*. [S.l.], 2001. p. 55–70.

BERMBACH, D.; KUHLENKAMP, J. Consistency in distributed storage systems. In: SPRINGER. *International Conference on Networked Systems*. [S.l.], 2013. p. 175–189.

CARLSON, J. L. *Redis in action*. [S.l.]: Manning Publications Co., 2013.

CHEN, Y.; SUN, X. Stas: A scalability testing and analysis system. In: *Cluster Computing*. [S.l.: s.n.], 2006.

CIMINO, L. de S. et al. A middleware solution for integrating and exploring iot and hpc capabilities. *Software: Practice and Experience*, v. 49, n. 4, p. 584–616, 2019. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2630>.

CUDENNEC, L. Merging the publish-subscribe pattern with the shared memory paradigm. In: MENCAGLI, G. et al. (Ed.). *Euro-Par 2018: Parallel Processing Workshops*. Cham: Springer International Publishing, 2019. p. 469–480. ISBN 978-3-030-10549-5.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, n. 2, p. 114–131, jun. 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/857076.857078>.

EUGSTER, P. T.; GUERRAOUI, R.; SVENTEK, J. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 2000. p. 252–276.

FIDGE, C. J. *Timestamps in message-passing systems that preserve the partial ordering*. [S.l.]: Australian National University. Department of Computer Science, 1987.

GARG, N. *Apache Kafka*. [S.l.]: Packt Publishing, 2013. ISBN 1782167935, 9781782167938.

GHOSH, S. *Distributed systems: an algorithmic approach*. [S.l.]: Chapman and Hall/CRC, 2014.

GOODMAN, B. D. et al. *Pub/sub message invoking a subscribers client application program*. [S.l.]: Google Patents, 2011. US Patent 7,890,572.

HILL, M. D. What is scalability? *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 18, n. 4, p. 18–21, dez. 1990. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/121973.121975>.

INTORRUK, S.; NUMNONDA, T. A comparative study on performance and resource utilization of real-time distributed messaging systems for big data. In: IEEE. *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. [S.l.], 2019. p. 102–107.

KLEPPMANN, M.; KREPS, J. Kafka, samza and the unix philosophy of distributed data. IEEE, 2015.

KREPS, J. et al. Kafka: A distributed messaging system for log processing. In: *Proceedings of the NetDB*. [S.l.: s.n.], 2011. p. 1–7.

KRISHNAN, S.; GONZALEZ, J. L. U. Google cloud pub/sub. In: *Building Your Next Big Thing with Google Cloud Platform*. [S.l.]: Springer, 2015. p. 277–292.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359545.359563>.

LIGHT, R. A. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, v. 2, n. 13, p. 265, 2017.

LIPTON, R. J.; SANDBERG, J. S. *PRAM: A scalable shared memory*. [S.l.]: Princeton University, Department of Computer Science, 1988.

MARCHIONI, F.; SURTANI, M. *Infinispan data grid platform*. [S.l.]: Packt Publishing Ltd, 2012.

MARCU, O.-C. et al. Kera: Scalable data ingestion for stream processing. In: IEEE. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.], 2018. p. 1480–1485.

MAZZUCCO, M. et al. Engineering distributed shared memory middleware for java. In: . [S.l.: s.n.], 2009. v. 5870, p. 531–548.

MEIER, R.; CAHILL, V. Taxonomy of distributed event-based programming systems. *The Computer Journal*, OUP, v. 48, n. 5, p. 602–626, 2005.

Mills, D. L. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, v. 39, n. 10, p. 1482–1493, Oct 1991.

PROTIC, J.; TOMASEVIC, M.; MILUTINOVIC, V. A survey of distributed shared memory systems. In: IEEE. *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. [S.l.], 1995. v. 1, p. 74–84.

PROTIC, J.; TOMASEVIC, M.; MILUTINOVIC, V. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, IEEE, v. 4, n. 2, p. 63–71, 1996.

SHAHRIVARI, S.; SHARIFI, M. Task-oriented programming: A suitable programming model for multicore and distributed systems. In: IEEE. *2011 10th International Symposium on Parallel and Distributed Computing*. [S.l.], 2011. p. 139–144.

SNYDER, B.; BOSNANAC, D.; DAVIES, R. *ActiveMQ in action*. [S.l.]: Manning Greenwich Conn., 2011. v. 47.

STEEN, M.; TANENBAUM, A. S. A brief introduction to distributed systems. *Computing*, Springer-Verlag, Berlin, Heidelberg, v. 98, n. 10, p. 967–1009, out. 2016. ISSN 0010-485X. Disponível em: <https://doi.org/10.1007/s00607-016-0508-7>.

TANENBAUM, A. S.; STEEN, M. v. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0132392275.

THOMAN, P. et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, Springer, v. 74, n. 4, p. 1422–1434, 2018.

VEENTJER, P. *Mastering Hazelcast*. [S.l.]: Hazelcast, 2013.

VIDELA, A.; WILLIAMS, J. J. *RabbitMQ in action: distributed messaging for everyone*. [S.l.]: Manning, 2012.

WALKER, S. M. et al. *RAFDA: A Policy-Aware Middleware Supporting the Flexible Separation of Application Logic from Distribution*. [S.l.], 2003. Technical Report CS/06/2.

XIONG, J.; WANG, J.; XU, J. Research of distributed parallel information retrieval based on jppf. In: *International Conference of Information Science and Management Engineering*. [S.l.: s.n.], 2010. p. 109–111.

ZHU, W.; WANG, C.; LAU, F. C. M. JESSICA2: a distributed java virtual machine with transparent thread migration support. In: *International Conference on Cluster Computing*. [S.l.: s.n.], 2002.

Anexos

ANEXO A – Tabelas extras

Este capítulo apresenta tabelas com resultados extras, não apresentadas no Capítulo 5. As tabelas servem para reforçar os resultados apresentados, pois os comportamentos observados foram similares para os três tipos de dados testados: *INTEGER*, *STRING* e *COLLECTION*.

Tabela A.1 – GET não bloqueante (segundos) INT

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
8k GETS	2,726	18,161	2,863	12,536	2,659	12,027
16K GETS	4,632	17,269	4,784	13,143	4,556	10,967
80K GETS	12,828	20,243	19,726	13,639	18,833	12,328

Tabela A.2 – GET não bloqueante (segundos) COLLECTION

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
8k GETS	2,623	18,873	3,008	15,041	2,646	12,027
16K GETS	4,535	18,631	4,715	13,144	4,583	10,967
80K GETS	20,204	20,764	19,726	16,241	18,880	12,413

Tabela A.3 – GET não bloqueante em mapa (segundos) INT

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
8k GETS	2,327	12,603	2,458	11,936	2,425	10,191
16K GETS	4,129	12,022	4,353	11,740	4,255	10,244
80K GETS	18,779	10,929	19,612	12,019	18,550	10,845

Tabela A.4 – GET não bloqueante em mapa (segundos) STRING

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
8k GETS	2,361	12,296	2,491	11,676	2,364	10,648
16K GETS	4,166	12,804	4,535	11,907	4,167	10,285
80K GETS	18,714	12,339	18,971	11,225	18,450	11,351

Tabela A.5 – GET + SET bloqueante (segundos) STRING

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k GET/SET	1,688	23,741	1,583	20,593	1,589	19,384
1.6K GET/SET	2,664	30,334	2,585	24,860	2,672	26,608
8K GET/SET	10,533	55,601	10,802	49,565	10,550	55,601

Tabela A.6 – GET + SET bloqueante (segundos) COLLECTION

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k GET/SET	1,558	26,218	1,609	23,052	1,670	21,482
1.6K GET/SET	2,639	32,070	2,749	24,780	2,718	26,815
8K GET/SET	10,908	54,139	11,222	49,757	11,121	48,193

Tabela A.7 – GET + PUT bloqueante em mapa (segundos) INT

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k GET/PUT	1,994	21,934	1,775	19,227	1,833	21,296
1.6K GET/PUT	3,478	30,770	3,565	25,427	3,557	23,965
8K GET/PUT	14,525	49,572	15,318	49,440	15,300	46,021

Tabela A.8 – GET + PUT bloqueante em mapa (segundos) STRING

	JCL Hosts ou workers Kafka					
	2		4		8	
	JCL	JCL Kafka	JCL	JCL Kafka	JCL	JCL Kafka
0.8k GET/PUT	1,825	24,514	1,786	20,774	1,882	17,022
1.6K GET/PUT	3,455	31,873	3,435	27,713	3,535	27,517
8K GET/PUT	14,736	53,119	15,050	49,754	15,068	47,676