PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Rafael Oliveira Vasconcelos

Uma Abordagem Eficiente para Reconfiguração Coordenada em Sistemas Distribuídos de Processamento de Data Streams

PUC-Rio - Certificação Digital Nº 1312421/CA

Tese de Doutorado

Tese apresentada ao Programa de Pós-graduação em Informática da PUC-Rio como requisito parcial para obtenção do grau de Doutor em Informática.

Orientador: Prof. Markus Endler

Rio de Janeiro
Abril de 2017

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Rafael Oliveira Vasconcelos

Uma Abordagem Eficiente para Reconfiguração Coordenada em Sistemas Distribuídos de Processamento de Data Streams

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós-graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

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

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

Prof. Sérgio Colcher
Departamento de Informática – PUC-Rio

Profa. Flávia Coimbra Delicato
Departamento de Ciência da Computação – UFRJ

Prof. José Viterbo Filho
Departamento de Ciência da Computação e Pós-graduação – UFF

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 07 de abril de 2017

PUC-Rio - Certificação Digital Nº 1312421/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Rafael Oliveira Vasconcelos

Bacharel em Ciência da Computação pela Universidade Tiradentes (UNIT) e Mestre em Informática pelo Departamento de Informática (DI) da PUC-Rio.

Ficha Catalográfica

Vasconcelos, Rafael Oliveira

Uma abordagem eficiente para reconfiguração coordenada em sistemas distribuídos de processamento de data streams / Rafael Oliveira Vasconcelos ; orientador: Markus Endler. – 2017.

74 f. : il. color. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2017.

Inclui bibliografia

1. Informática – Teses. 2. Reconfiguração dinâmica. 3. Adaptabilidade. 4. Adaptação de software. 5. Comunicação móvel. 6. Processamento de fluxo de dados. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1312421/CA

PUC-Rio - Certificação Digital Nº 1312421/CA

Esta tese é dedicada aos meus pais, em especial ao meu amigo, confidente, pai e agora também anjo da guarda (*in memoriam*).

Sei que você lutou bravamente até a minha volta da defesa de proposta, mas infelizmente não pude lhe dar em vida a alegria de realizar o seu último desejo, ter um filho doutor.

Apesar de estar atrasado, sei que ainda é tempo, dou-lhe esta alegria agora. Sei que aí de cima, olhando por todos nós, você está contente e orgulhoso, pois, o seu filho, a partir de hoje, é o seu doutor.

Sentado nesta mesma cadeira que hoje escrevo este texto, eu prometi, e agora cumpri.

Hoje podemos falar juntos, *veni, vidi, vici*!

Fique em paz!

Agradecimentos

Eu tenho que agradecer aos meus pais por todo o apoio incondicional e eterno, vocês são os pilares que me sustentam. Foram 6 longos anos de pós-graduação.

O sucesso desta tese depende largamente do encorajamento e orientação de muitas pessoas. Aproveito esta oportunidade para expressar minha gratidão às pessoas que foram importantes para o sucesso deste projeto. Gostaria de agradecer ao meu orientador Markus Endler e aos meus amigos do LAC. Eu não posso dizer obrigado o suficiente por todo o suporte e ajuda. Sem eles eu não poderia materializar este projeto. As orientações e suporte que eu recebi foram vitais para o sucesso deste trabalho. Eu sou grato pelo constante apoio e ajuda que vocês me deram.

Eu também gostaria de agradecer ao Bruno por gentilmente me acolher em sua casa.

Agradeço ao CNPq e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Por fim, eu sou grato por todos aqueles que, de alguma forma, me ajudaram direta ou indiretamente nesta jornada.

PUC-Rio - Certificação Digital Nº 1312421/CA

Resumo

Vasconcelos, Rafael Oliveira; Endler, Markus. **Uma Abordagem Eficiente para Reconfiguração Coordenada em Sistemas Distribuídos de Processamento de Data Streams.** Rio de Janeiro, 2017. 74p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Ao mesmo tempo em que sistemas de processamento de fluxo de dados devem prover serviços de análise e manipulação de dados ininterruptamente (disponibilidade 24x7), eles comumente também precisam lidar com mudanças em seus ambientes de execução (e.g., alterar a topologia da rede) e nos requisitos que eles devem cumprir (e.g., adição de novas funções de processamento dos fluxos de dados). Por um lado, reconfiguração dinâmica de software (i.e., a capacidade de substituir parte do software em tempo de execução) é uma característica desejável. Por outro lado, sistemas de fluxo de dados podem sofrer com a interrupção e sobrecarga causada pela reconfiguração. Por conta da necessidade de reconfigurar (i.e., evoluir) o sistema ao mesmo tempo em que o sistema não pode ser interrompido (i.e., bloqueado), reconfiguração consistente e não bloqueante é ainda considerada um problema em aberto na literatura. Esta tese apresenta e valida uma abordagem não quiescente para reconfiguração dinâmica de software que preserva a consistência de sistemas de fluxo de dados distribuídos. A abordagem proposta permite que o sistema seja reconfigurado gradual e suavemente, sem precisar interromper o processamento do fluxo de dados ou atingir a quiescência. A avaliação indica que a abordagem proposta realiza reconfiguração distribuída consistentemente e tem um impacto desprezível sobre a diminuição na disponibilidade e no desempenho do sistema. Além disto, a implementação da abordagem proposta teve um desempenho melhor em todos os testes comparativos.

Palavras-chave

Reconfiguração dinâmica; Adaptabilidade; Adaptação de software; Comunicação móvel; Processamento de fluxo de dados.

PUC-Rio - Certificação Digital Nº 1312421/CA

Abstract

Vasconcelos, Rafael Oliveira; Endler, Markus (Advisor). **An Efficient Approach to Coordinated Reconfiguration in Distributed Data Stream Systems.** Rio de Janeiro, 2017. 74p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

While many data stream systems have to provide continuous (24x7) services with no acceptable downtime, they also have to cope with changes in their execution environments and in the requirements that they must comply (e.g., moving from on-premises architecture to a cloud system, changing the network technology, adding new functionality or modifying existing parts). On one hand, dynamic software reconfiguration (i.e., the capability of evolving on the fly) is a desirable feature. On the other hand, stream systems may suffer from the disruption and overhead caused by the reconfiguration. Due to the necessity of reconfiguring (i.e., evolving) the system whilst the system must not be disrupted (i.e., blocked), consistent and non-disruptive reconfiguration is still considered an open problem. This thesis presents and validates a non-quiescent approach for dynamic software reconfiguration that preserves the consistency of distributed data stream processing systems. Unlike many works that require the system to reach a safe state (e.g., quiescence) before performing a reconfiguration, the proposed approach enables the system to smoothly evolve (i.e., be reconfigured) in a non-disruptive way without reaching quiescence. The evaluation indicates that the proposed approach supports consistent distributed reconfiguration and has negligible impact on availability and performance. Furthermore, the implementation of the proposed approach showed better performance results in all experiments than the quiescent approach and Upstart.

Keywords

Dynamic reconfiguration; Adaptability; Software adaptation; Mobile communication; Data Stream Processing.

PUC-Rio - Certificação Digital Nº 1312421/CA

Sumário

1 Introdução 12
1.1 Descrição do problema 13
1.2 Cenário motivador 16
1.3 Objetivo e contribuições 17
1.4 Organização da tese 18

2 Fundamentação teórica 19
2.1 Principais aspectos sobre reconfiguração dinâmica de software 19
2.2 Suporte a reconfiguração dinâmica em processamento de fluxo de dados 23
2.3 Dependências direta e indireta 25
2.4 Trabalhos relacionados 26

3 Modelo do sistema 31
3.1 Arquitetura da solução proposta 32

4 Uma abordagem para reconfiguração dinâmica e distribuída de software 35
4.1 Gerenciamento de múltiplas versões 37
4.2 Gerenciamento de estado 40
4.3 Reconfiguração distribuída com múltiplas instâncias 42
4.4 Implementação 46

5 Avaliação de desempenho 48
5.1 Cenário de teste 49
5.2 Métricas 51
5.3 Avaliação de D-Joseph 52
5.3.1 Comparação entre o gerenciamento estático e o gerenciamento dinâmico de dependência 58
5.4 Comparação entre D-Joseph e outras abordagens 60

6 Conclusão 64
6.1 Resultados da tese 65

7 Referências bibliográficas 67

PUC-Rio - Certificação Digital Nº 1312421/CA

Lista de figuras

Figura 1. Cenário de um sistema de entrega de mensagem 14

Figura 2. Exemplo de arquitetura de um sistema para monitoramento de saúde. Adaptado de [40] 17

Figura 3. Trecho de código mostrando um ponto de atualização adicionado pelo desenvolvedor. Adaptado de [63] 21

Figura 4. Exemplo de mudança ocorrida no estado do componente User entre duas versões (v0 e v1). Adaptado de [62] 22

Figura 5. Exemplo de função de transformação responsável pela conversão do atributo forwardAddresses. Adaptado de [62] 22

Figura 6. Exemplo de transferência de estado automatizada. Adaptado de [4] 23

Figura 7. Exemplo de um sistema de processamento de fluxo de dados. Adaptado de [77] 24

Figura 8. Exemplo de dependência indireta entre os componentes Carga e Resposta 25

Figura 9. Arquitetura proposta 33

Figura 10. Exemplo de sistema parcialmente reconfigurado com dois fluxos possíveis no componente Descompressão 36

Figura 11. Fluxo de dados parcial do cenário motivador 37

Figura 12. Caminho de execução da tupla T no sistema parcialmente reconfigurado 38

Figura 13. Caminho de execução usando gerenciamento dinâmico onde só há dependências diretas 38

Figura 14. Gerenciamento dinâmico com a ocorrência de dependência indireta 39

Figura 15. Caminho de execução de uma tupla em trânsito 39

Figura 16. Caminho de execução de uma tupla gerada após a modificação das dependências e enquanto o sistema está parcialmente reconfigurado 40

Figura 17. Exemplo de estado compartilhado antes (a), durante (b) e depois (c) da reconfiguração de um componente 41

Figura 18. Ciclo para reconfiguração de um estado v1 para v2 41

Figura 19. Reconfiguração parcial inconsistente no fluxo tracejado em vermelho 43

Figura 20. Reconfiguração parcial consistente 43

Figura 21. Sistema reconfigurado 44

Figura 22. Sistema parcialmente reconfigurado com três versões em paralelo 45

PUC-Rio - Certificação Digital Nº 1312421/CA

Figura 23. Sistema após reconexão do nó e conclusão das duas reconfigurações 46
Figura 24. Implantação física utilizada nos testes. O Gerente de Reconfiguração e o Gateway foram omitidos por questões de legibilidade 49
Figura 25. Abordagem para identificar choque séptico 50
Figura 26. Nova abordagem recomendada para identificar choque séptico 50
Figura 27. Vazão no cenário com 3.000 CNs, 15.000 tuplas/s e gerenciamento estático 54
Figura 28. Vazão no cenário com 3.000 CNs, 15.000 tuplas/s e gerenciamento dinâmico 55
Figura 29. Latência no cenário com 300 CNs, 15.000 tuplas/s e gerenciamento estático de dependência 55
Figura 30. Latência no cenário com 300 CNs, 15.000 tuplas/s e gerenciamento dinâmico de dependência 56
Figura 31. Sobrecarga em relação ao tempo de processamento imposta por D-Joseph 57
Figura 32. Sobrecarga em relação à vazão imposta por D-Joseph 57
Figura 33. Sobrecarga em relação à latência imposta por D-Joseph 58
Figura 34. Dependências entre os componentes 59
Figura 35. Vazão do sistema usando quiescência, D-Joseph e Upstart 61
Figura 36. Latência no cenário com 3.000 CNs usando quiescência 62
Figura 37. Latência no cenário com 3.000 CNs usando Upstart 62
Figura 38. Latência no cenário com 3.000 CNs usando D-Joseph 63

PUC-Rio - Certificação Digital Nº 1312421/CA

Lista de tabelas

Tabela 1. Parâmetros dos cenários de avaliação 52

Tabela 2. Tempo de atualização para cada cenário avaliado 53

Tabela 3. Comparação entre o gerenciamento estático e o gerenciamento dinâmico de dependência 59

Tabela 4. Tempo de atualização dos trabalhos comparados 60

PUC-Rio - Certificação Digital Nº 1312421/CA

1
Introdução

Muitos sistemas de processamento de fluxo [1]–[3] devem prover serviços ininterruptamente, ou seja, implementar uma disponibilidade 24x7, onde interrupções não são aceitáveis [4] [5]. Apesar disto, tais sistemas comumente precisam lidar com mudanças em seus ambientes de execução (e.g., alterar a topologia da rede) e nos requisitos de aplicação que eles devem cumprir (e.g., adição de novas funções de processamento dos fluxos de dados) [6]. Os autores em [6] enfatizam que mudanças de software são difíceis de serem previstas em tempo de projeto. Além disso, a execução contínua do serviço dificulta a correção de erros (i.e., *bugs*) e a adição de novas funcionalidades em tempo de execução, pois requer a substituição de partes do programa/sistema em execução de uma versão para outra, sem interromper o serviço sendo prestado [7] [8]. Ertel e Felber [7] explicam ainda que algumas abordagens de reconfiguração dinâmica (também denominadas de adaptação dinâmica, *live update* ou evolução dinâmica por alguns autores) requerem a inicialização de um novo processo em outro computador, sendo que o custo de hardware redundante pode ser consideravelmente alto, e requerem a transferência e sincronização de estados entre os componentes sendo permutados (i.e., atualizados), [7] [9] [10].

Apesar da extensa pesquisa em reconfiguração dinâmica de software [11]–[14], reconfiguração consistente (i.e., uma reconfiguração onde "... *o sistema pode continuar processando normalmente ao invés de avançar para um estado errôneo*..." [15]) ainda é um desafio em aberto [5] [16] [17]. Uma abordagem comum é levar o componente que será modificado para um estado seguro, como por exemplo o estado quiescente (i.e., de inatividade parcial) [15], antes de reconfigurar o sistema [15] [18] [19]. Nesse caso, uma tarefa da reconfiguração é conduzir o sistema para um estado consistente e preservar a execução correta das funções em andamento colocando as partes afetadas no estado seguro, como o quiescente [6]. Além disto, a reconfiguração dinâmica deve também minimizar a interrupção do sistema (i.e., *disruption*) e o tempo necessário para atualizar o

PUC-Rio - Certificação Digital Nº 1312421/CA

sistema (i.e., *timeliness*) [15] [20]. Ertel e Felber [7] argumentam ainda que coordenar (i.e., orquestrar) a reinicialização de todos os componentes permutados ou adicionados é ainda uma tarefa desafiadora caso o sistema não possa ser interrompido.

Alinhado com os requisitos supracitados, aplicações no domínio de processamento de fluxo de dados requerem um processamento contínuo e *online* de grandes volumes de dados, originados de inúmeros dispositivos distribuídos (e possivelmente móveis), para fazer a análise *online* a partir de consultas complexas sobre o fluxo dados, ou eventos [1] [21] [22]. Sistemas de transporte inteligente, monitoramento de elementos de rede, monitoramento de ações em bolsa de valores, gerenciamento inteligente de energia e logística são alguns exemplos de áreas que requerem processamento de fluxo de dados [1], [21]–[24]. Assim, embora reconfiguração dinâmica seja uma característica desejável, tais sistemas não devem sofrer degradação de desempenho por conta de potenciais interrupções e sobrecarga (i.e., *overhead*) causadas pela reconfiguração dinâmica.

Esta tese apresenta uma abordagem para reconfiguração dinâmica de software para sistemas de processamento de fluxo de dados, que não usa a noção de quiescência. A abordagem proposta permite a execução concorrente de múltiplas versões de um componente de software (i.e., função de processamento do fluxo de dados). Resumidamente, a proposta é baseada na ideia de que uma tupla (i.e., um dado/evento que é enviado de um componente para outro no fluxo) deve ser processada por uma versão específica de cada componente. Entretanto, não há problema em atualizar um componente enquanto uma tupla circula no sistema, visto que o sistema mantém a versão antiga e a versão nova do componente, e de todos os seus componentes dependentes, até que todas as tuplas da versão antiga tenham sido processadas.

PUC-Rio - Certificação Digital Nº 1312421/CA

1.1 Descrição do problema

Sistemas de processamento de fluxo (*Stream Processing Systems)* são compostos por componentes distribuídos, em que cada componente processa uma porção do fluxo de dados (inseridos no sistema). Em tais sistemas, há uma dependência de execução entre os componentes (i.e., dependência entre artefatos de software) [25]. Além disto, cada tipo de componente pode ter múltiplas

instâncias distribuídas. A dependência de execução entre componentes difere da dependência de implantação já que é determinada também pelos dados em trânsito no sistema e pelos fluxos de controle em tempo de execução, em vez de somente dependências estáticas [6] entre os componentes de software. Estes sistemas precisam alcançar um estado seguro e manter a consistência entre os estados dos componentes durante, e depois, da reconfiguração dinâmica. Ao mesmo tempo, por conta da natureza contínua do fluxo de dados, não é possível bloquear os componentes envolvidos, ou esperar que alcancem um estado quiescente. Além disto, este tipo de sistema deve lidar com adaptações coordenadas de muitas – e possivelmente distribuídas – instâncias de um mesmo tipo de componente. O exemplo da Figura 1 ilustra um cenário simples de uma aplicação de chat (bate-papo), proposto por Ghafari *et al.* [18], onde os remetentes interagem com os destinatários através da troca de mensagens, que sofrem compressão nos remetentes e descompressão nos destinatários. Neste exemplo, tanto o nó remetente como o nó destinatário têm suas próprias instâncias locais do componente de *Compressão* e *Descompressão*.

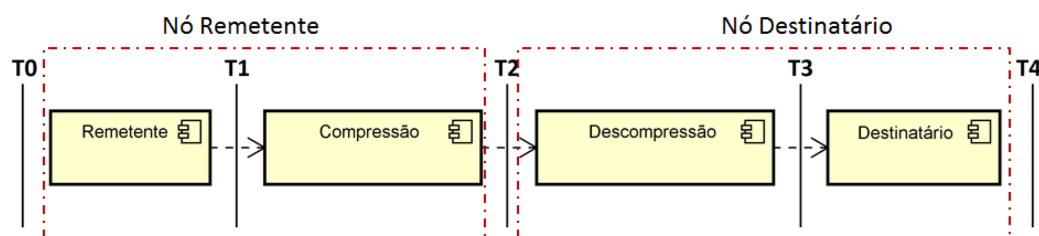

Figura 1. Cenário de um sistema de entrega de mensagem

Sempre que o tipo de componente *(Des)Compressão* é atualizado por uma versão mais nova que é incompatível com a versão anterior, todas as instâncias deste tipo também devem ser atualizadas para a nova versão, e isso deve ser feito de modo coordenado para garantir a consistência global do sistema. Em outras palavras, um *middleware* (i.e., plataforma de reconfiguração) deve garantir que nenhuma das mensagens comprimidas pela versão antiga do componente *Compressão* seja descomprimida pela versão nova do componente *Descompressão* (ou vice-versa), para assim evitar que o sistema atinja um estado inconsistente, como discutido por Ghafari *et al.* [18]. Deste modo, o principal desafio desta adaptação distribuída é que, havendo *N* instâncias distribuídas do componente *(Des)Compressão*, o *middleware* precisa gerenciar as dependências dinâmicas entre todas as instâncias dos componentes, para assim garantir a

PUC-Rio - Certificação Digital Nº 1312421/CA

consistência local/global ao atualizar os componentes [6]. Consistência local considera apenas a consistência entre os estados dos componentes em um único nó, enquanto que consistência global compreende o estado local de todas as instâncias dos componentes distribuídos e suas mensagens em trânsito [6]. Nas situações/aplicações em que não é possível bloquear todos os *N* remetentes, caso o *middleware* não gerencie todas as mensagens, i.e., os dados no fluxo de dados, em trânsito enviadas pela versão antiga antes de substituir todas as *N* instâncias do componente *(Des)Compressão*, alguns nós poderiam usar a versão nova do componente *Descompressão* para descomprimir uma mensagem comprimida pela versão anterior do componente *Compressão* (ou vice-versa). Portanto, é preciso coordenar a substituição das instâncias do componente *(Des)Compressão* em todos os *N* nós. Além disto, um nó que fica desconectado/indisponível durante o processo de adaptação do sistema pode atrasar a evolução do resto do sistema para a nova versão, uma vez que o *middleware* não pode concluir a reconfiguração do sistema.

Enquanto o conceito de *adaptação transacional* [15] [26] é suficiente e apropriado para garantir consistência quando é considerada apenas uma única instância de cada tipo de componente, tal conceito não é suficiente para sistemas envolvendo componentes com instâncias múltiplas e distribuídas. Por exemplo, no caso da Figura 1, uma transação deve iniciar no instante T0 e terminar somente no instante T4. Apesar do sistema estar em um estado seguro para ser atualizado antes do instante T1, se no instante T1 o nó destinatário receber uma mensagem de reconfiguração para substituir os componentes de *(Des)Compressão* mas esta mensagem só for entregue para o nó remetente no instante T2, a reconfiguração levará o sistema para um estado inconsistente já que ambos os nós interagirão usando versões diferentes de *Compressão* e *Descompressão*. Analisando somente o destinatário, sua instância local do componente *Descompressão* estaria em um estado seguro até T2, instante no qual o componente *Descompressão* é executado. Em princípio, seria possível aplicar o conceito de *adaptação transacional* a múltiplas instâncias de um tipo de componente, entretanto transações impõem uma sobrecarga considerável ao sistema visto que um remetente teria que iniciar uma transação para cada nó destinatário, e para cada mensagem enviada. O uso de transações implicaria em dois problemas: (i) para cada troca de mensagem, seria necessário iniciar uma transação com os destinatários, requerendo assim trocas de

PUC-Rio - Certificação Digital Nº 1312421/CA

mensagens com todos os destinatários, o que não seria escalável e tornaria a adaptação inviável em um cenário com muitos (e.g., milhares) destinatários, e (ii) o *middleware* deve estar ciente dos destinatários de cada mensagem – muitas vezes apenas a camada da aplicação está ciente dos destinatários – para poder iniciar uma transação com todos os destinatários independentemente do fato de uma reconfiguração ser necessária neste momento, o que.

1.2
Cenário motivador

Nos últimos anos, os sistemas de monitoramento de pacientes através de dispositivos vestíveis (i.e., com sensores que medem, entre outras informações, os sinais vitais) têm atraído atenção e investimentos do setor de saúde e da comunidade acadêmica [27]–[29]. Os dispositivos (i.e., sensores) podem ser aplicados em roupas inteligentes, pulseiras de monitoramento ou diretamente no corpo [30]–[32]. Tais sensores podem, por exemplo, monitorar o nível de glicose, pressão arterial, temperatura do corpo e pele, frequência cardíaca e respiratória, e saturação de oxigênio no sangue, além de poder realizar um eletrocardiograma [28] [29] [33]. O cenário motivador consiste de um hospital que monitora e trata milhares de pacientes, como por exemplo o *Clinical Centre of Serbia*, com 3.500 leitos [34].

Tipicamente, os sensores vestíveis (i.e., *wearables*) possuem alguma interface sem fio de curto alcance e baixo consumo de energia (e.g., *Bluetooth* ou *ZigBee*), e não possuem acesso direto à internet. Para tal, os sensores são comumente interconectados através de um nó móvel (e.g., *smartphone*) que age como uma espécie de *hub* móvel que permite que os dados dos sensores sejam pré-processados e transmitidos para servidores remotos através da internet (ou outra rede, como uma intranet, por exemplo), como mostrado na Figura 2 [33], [35], [36].

A necessidade de reconfiguração dinâmica em tais sistemas emerge de três demandas principais: (i) a troca de sensores por modelos mais modernos, (ii) o melhoramento contínuo dos protocolos, i.e., técnicas ou abordagens, utilizados no monitoramento dos pacientes, e (iii) atualização do aplicativo que executa no nó móvel. Em todos os casos, o sensoriamento e o processamento dos dados não podem ser interrompidos por conta da criticidade do monitoramento, visto que os

PUC-Rio - Certificação Digital Nº 1312421/CA

dados precisam ser processados continuamente para que alertas possam ser gerados em tempo real para os profissionais de saúde [37] [38]. O sucesso de tais aplicações depende do monitoramento contínuo e ininterrupto das pessoas/pacientes [39].

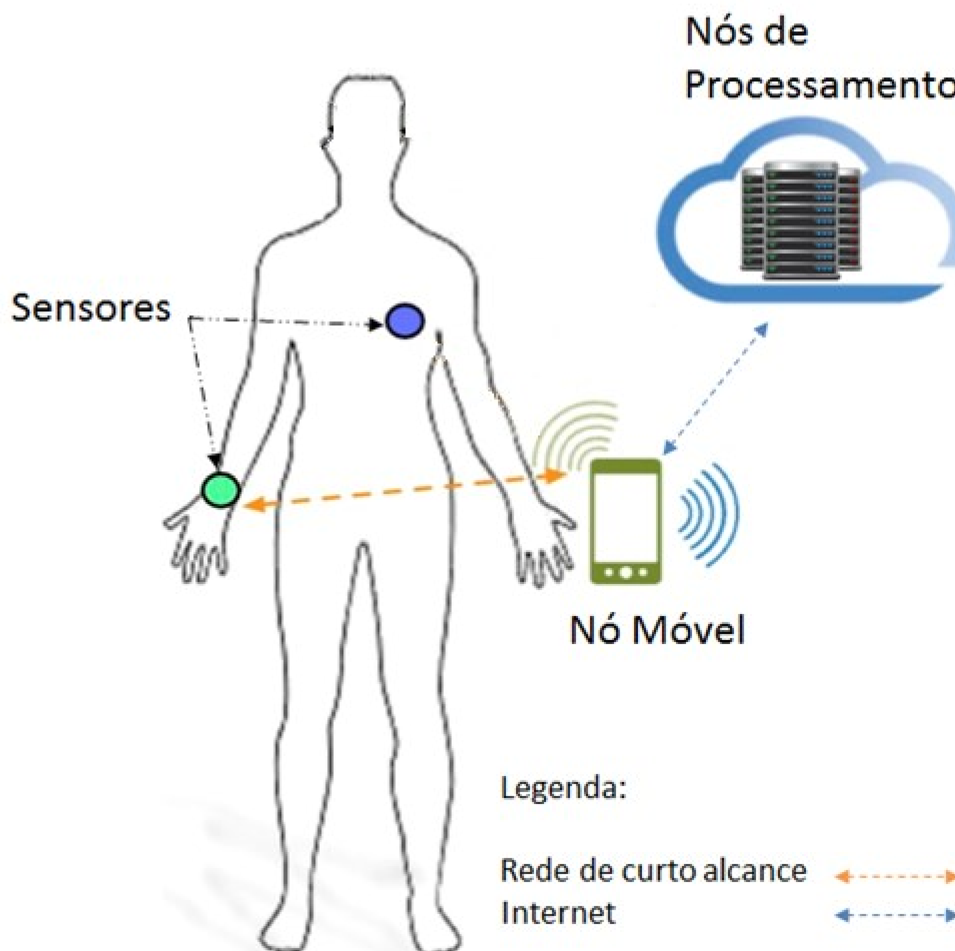

Figura 2. Exemplo de arquitetura de um sistema para monitoramento de saúde. Adaptado de [40]

1.3
Objetivo e contribuições

A fim de resolver os problemas supracitados, esta tese propõe e valida uma abordagem não quiescente para reconfiguração dinâmica e coordenada de software que preserva a consistência global em sistemas de fluxo de dados distribuídos. Esta abordagem (i) garante que todos itens de dados (de um fluxo de dados) são processados exatamente uma vez, (ii) não necessita atingir a quiescência para realizar uma reconfiguração de software, (iii) lida com nós móveis que podem aparecer e desaparecer a qualquer momento, e (iv) suporta

PUC-Rio - Certificação Digital Nº 1312421/CA

componentes com e sem estado. Mais especificamente, as principais contribuições deste trabalho são:

1. Um modelo para reconfiguração dinâmica distribuída que permite a um sistema evoluir sem a necessidade que seus nós alcancem o estado quiescente, evitando assim um atraso no processamento de alguns dados do fluxo de dados;
2. Desenvolvimento de um protótipo do modelo, o qual é executado em uma arquitetura *mobile-cloud*, implementado como um middleware em Java que permite reconfiguração dinâmica consistente e de modo não quiescente em sistemas de fluxo de dados.

1.4
Organização da tese

O restante da tese é organizado como segue:

- O Capítulo 2 apresenta uma visão geral acerca dos principais conceitos abordados nesta tese e revisa os principais trabalhos relacionados a esta tese;
- O Capítulo 3 apresenta o modelo e a arquitetura da solução proposta para reconfiguração dinâmica de software (de processamento de fluxos de dados).
- O Capítulo 4 descreve em detalhes a abordagem proposta para permitir reconfiguração dinâmica não quiescente em sistemas de processamento de fluxos;
- O Capítulo 5 avalia o protótipo que implementa a abordagem proposta e o compara com dois outros trabalhos;
- O Capítulo 6, por fim, revisa e discute as ideias centrais apresentadas na tese e propõe linhas de trabalhos futuros, além de listar os resultados obtidos.

PUC-Rio - Certificação Digital Nº 1312421/CA

2
Fundamentação teórica

Este capítulo apresenta os principais conceitos relacionados a reconfiguração dinâmica de software, e suporte a reconfiguração dinâmica em processamento de fluxo de dados, bem como apresenta o modelo do sistema.

2.1
Principais aspectos sobre reconfiguração dinâmica de software

Reconfiguração dinâmica de software, ou para alguns pesquisadores evolução dinâmica de software, é um processo evolutivo que pode ser definido como mudanças progressivas e discretas, ao longo do tempo, nas características, atributos ou propriedades de um artefato de software (e.g., método, classe, componente ou módulo de um sistema), como proposto por Lehman e Framil [41] [42], sem que a execução do software (ou serviço) seja interrompida [15] [19] [43] [44]. A reconfiguração dinâmica de software tem como objetivo corrigir, melhorar, estender ou reduzir um sistema que já foi implantado e está em execução [13] [17] [45]. A definição aqui dada para reconfiguração dinâmica de software é semelhante à encontrada na literatura utilizando outros termos e nomenclatura, já que não há um consenso na comunidade acerca da nomenclatura empregada [17] [23].

De modo geral, os pesquisadores têm classificado reconfiguração dinâmica em duas abordagens, reconfiguração paramétrica e reconfiguração composicional (ou estrutural) [13], [46]–[48]. A reconfiguração paramétrica permite a modificação do comportamento do sistema a partir da alteração dos valores de propriedades ou atributos do sistema, sem que seja alterado o código do software (i.e., sem modificações estruturais ou algorítmicas). Um exemplo clássico de reconfiguração paramétrica ocorre no protocolo TCP (*Transmission Control Protocol*), onde a modificação do valor de alguns parâmetros permite que o protocolo mude o seu comportamento em face de um congestionamento no meio físico [46] [47] [49]. Reconfiguração paramétrica pode ainda ser vista como um

PUC-Rio - Certificação Digital Nº 1312421/CA

modo de fazer *tuning* (e.g., fazer o ajuste fino de um SGBD – Sistema Gerenciador de Banco de Dados) [13] [46]. Na área de processamento de fluxo de dados, vários *middlewares* têm capacidade de realizar reconfiguração paramétrica com os mais diversos propósitos, como aumento de desempenho ou acurácia, e suporte a tolerância a falha, por exemplo [3], [50]–[53].

Apesar dos benefícios da reconfiguração paramétrica, ela não permite que sejam introduzidas mudanças não previstas durante o projeto inicial do sistema [17] [47]. Por outro lado, a adaptação composicional vai além da simples modificação de propriedades ou seleção de componentes (i.e., seleção de diferentes estratégias implementadas durante o projeto inicial). Ela permite que seja feita a adição/remoção/atualização de componentes ou módulos de software em tempo de execução para lidar com novos requisitos que podem surgir depois da implantação do sistema [46] [54]. Deste modo, a reconfiguração composicional permite a modificação em nível estrutural do sistema, possibilitando assim a inclusão de novas estratégias e algoritmos que não foram previstas no projeto inicial do sistema [47] [48]. Alguns exemplos de reconfiguração composicional são a correção de *bugs* (i.e., erros), adição – ou remoção – de funcionalidades, e a substituição de um comportamento por outro – e.g., substituição das estratégias de processamento do fluxo de dados por novas que foram desenvolvidas após a implantação do sistema.

Pesquisadores como Costa-Soria [17] e Buckley [55] explicam que a reconfiguração dinâmica pode ser realizada de modo reativo ou proativo. No primeiro modo, a reconfiguração é iniciada por um agente externo, comumente o desenvolvedor ou administrador do sistema. Este tipo de reconfiguração é ideal para introduzir mudanças não previstas durante o projeto, [17]. Na reconfiguração reativa, o sistema não tem inteligência para decidir quando realizar uma reconfiguração, sendo assim, o sistema recebe e executa uma ordem de reconfiguração de algum agente externo. Já a reconfiguração proativa é iniciada autonomamente pelo sistema em face de alguma condição ou ocorrência de um evento. Para tal, o sistema, tipicamente classificado como autonômico, deve possuir sensores e uma base de conhecimento para que as ações sejam tomadas autonomicamente pelo sistema, [55]–[60].

As principais questões que devem ser tratadas para que um software possa ser reconfigurado dinamicamente são: (i) atingir um estado seguro no qual o

PUC-Rio - Certificação Digital Nº 1312421/CA

sistema possa ser atualizado consistentemente, e (ii) transferir o estado entre os componentes que estão sendo substituídos [17]. Como a reconfiguração dinâmica pode gerar uma inconsistência e fazer com que o sistema antiga um estado errôneo no momento da substituição dos componentes, o sistema – ou preferencialmente apenas suas partes afetadas – deve estar em um estado seguro no qual as mudanças realizadas preservem a corretude das atividades – i.e., transações ou processamento – que estão em curso [6] [17] [61] [62]. Por outro lado, a abordagem (i.e., os critérios) para garantir que o sistema entre em um estado seguro deve minimizar a interrupção do sistema, levando assim um compromisso (*tradeoff*) entre manutenção da consistência (i.e., corretude) e disponibilidade do sistema [15] [17] [18] [44].

Enquanto algumas abordagens detectam automaticamente o momento em que o sistema atinge um estado seguro para ser reconfigurado, outras requerem que o desenvolvedor do sistema adicione explicitamente pontos de atualização (*update points*) específicos no código informando quando uma reconfiguração pode ser realizada (Figura 3) [63]–[66]. O problema com a adição de pontos de atualização é que o desenvolvedor precisa escolher os pontos de atualização, em várias instâncias, e refletir sobre a corretude do sistema nestes pontos de atualização, o que torna a tarefa difícil e propensa a erros, especialmente à medida que novos pontos de atualização são adicionados [63] [67].

```
1 void *thread_entry(void *arg) {
2   /* thread init code */
3   while (1) {
4     qbench_update(); /* update point */
5     /* loop body: typically handles a single program event */
6   }
7 }
```

ponto de atualização

Figura 3. Trecho de código mostrando um ponto de atualização adicionado pelo desenvolvedor. Adaptado de [63]

Outra questão comum na reconfiguração dinâmica é a transferência de estado [68]. A transferência de estado deve extrair o estado relevante da versão antiga do componente que está sendo atualizado, para posteriormente inicializar o estado da nova versão do componente, potencialmente realizando transformações estruturais ou semânticas no estado do componente em tempo de execução [17] [69] [70], como exemplificado na Figura 4, onde o tipo do atributo

PUC-Rio - Certificação Digital Nº 1312421/CA

forwardAddresses foi alterado de *String[]* para *EmailAddress[]*. Neste caso, para que a transferência de estado seja realizada corretamente, deverá haver uma função de transformação responsável por realizar a conversão do tipo *String[]* para *EmailAddress[]*.

```
1 class User{
2   String userName;
3   String password;
4   String[] forwardAddresses;
5 }
```

(a) v0

```
1 class User{
2   String userName;
3   String password;
4   EmailAddress[] forwardAddresses;
5 }
```

(b) v1

Figura 4. Exemplo de mudança ocorrida no estado do componente *User* entre duas versões (v0 e v1). Adaptado de [62]

```
1 public void updateObject(String[] p1){
2   final int length = p1.length;
3   forwardAddresses = new EmailAddress[length];
4   for(int i=0;i<length;i++){
5     forwardAddresses[i] = new EmailAddress(p1[i]);
6   }
7 }
```

Figura 5. Exemplo de função de transformação responsável pela conversão do atributo *forwardAddresses*. Adaptado de [62]

De modo geral, a transferência de estado pode ser delegada ao desenvolvedor do sistema (i.e., transferência de estado manual) ou automatizada (i.e., o *middleware* que provê as funcionalidades de reconfiguração dinâmica é capaz de realizar as transformações estruturais ou semânticas) [5] [19]. Na transferência de estado delegada, o desenvolvedor fica responsável por fornecer uma função de transformação responsável pela transformação do estado, como exemplificado na Figura 5, enquanto o *middleware* automatiza a migração da versão antiga para a versão nova. Assim, o *middleware* repassa o estado da versão antiga para a função de transformação. Esta, por sua vez, deve mapear o estado do componente antigo para o estado da nova versão [17]. Por outro lado, a transferência automatizada é capaz de fazer o mapeamento entre os estados autonomamente, sem qualquer intervenção do desenvolvedor. Entretanto, a depender das mudanças realizadas, a completa automação pode não ser viável, uma vez que nem sempre a semântica do estado pode ser automaticamente derivada a partir do código fonte [17] [71] [72]. No exemplo da Figura 6, o *middleware* não foi capaz de identificar que o atributo *addr* foi renomeado para *useraddress* e teve seu tipo alterado de *short* para *int*.

PUC-Rio - Certificação Digital Nº 1312421/CA

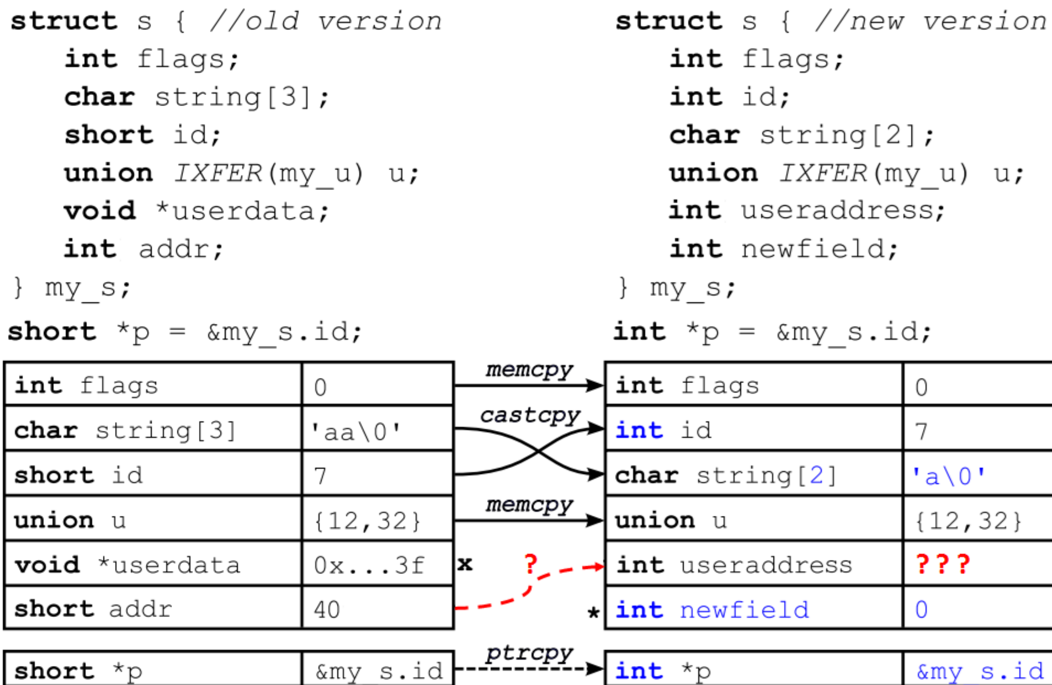

Figura 6. Exemplo de transferência de estado automatizada. Adaptado de [4]

PUC-Rio - Certificação Digital Nº 1312421/CA

2.2 Suporte a reconfiguração dinâmica em processamento de fluxo de dados

Processamento de fluxo de dados é um paradigma computacional com foco na análise, agregação e transformação de grandes volumes de fluxos de dados que são continuamente atualizados [1] [3] [73]. O fluxo de dados é uma sequência contínua e *online* de itens, que em princípio não têm fim (i.e., *unbounded*), onde não é possível controlar a ordem dos dados produzidos e processados [74] [75]. Deste modo, os dados são processados *on the fly* à medida que se deslocam dos nós de origem (i.e., nós produtores dos dados) para os nós consumidores, passando por vários nós distribuídos de processamento, que selecionam, filtram, classificam ou manipulam os dados [76]. Este modelo de processamento é comumente representado por um grafo onde os vértices podem ser nós de origem que produzem os dados (i.e., *source nodes*), nós de processamento que utilizam operadores para realizarem a análise do fluxo de dados, ou nós consumidores (i.e., *sink nodes*) que consomem o fluxo de dados processado. Já as arestas definem possíveis caminhos de dados (i.e., canais de fluxo ou *stream channels*) entre os vértices, como exemplificado na Figura 7.

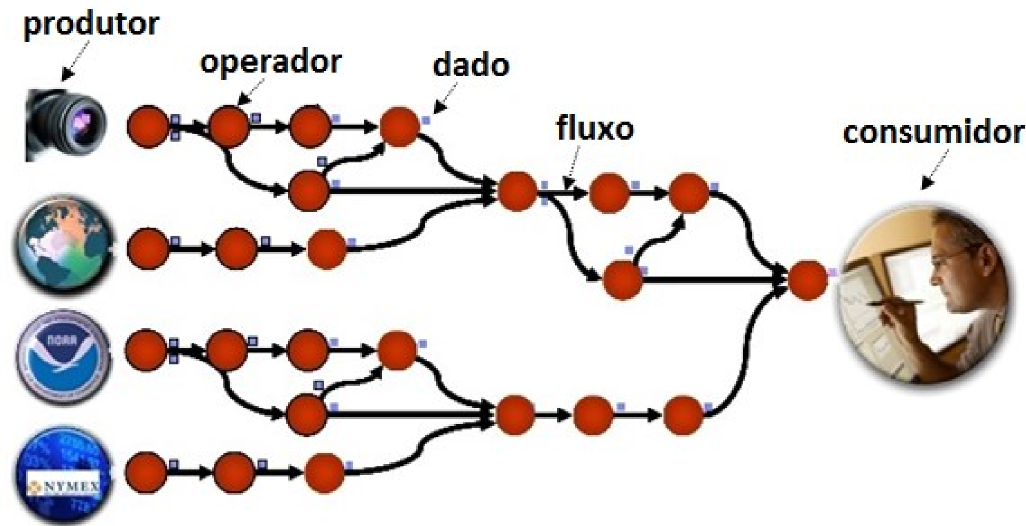

Figura 7. Exemplo de um sistema de processamento de fluxo de dados. Adaptado de [77]

Para lidar com a alta demanda por processamento, sistemas de processamento de fluxo comumente empregam o paradigma SIMD (*Single Instruction, Multiple Data*) de paralelismo e usam múltiplas instâncias de um operador (i.e., unidades de processamento), onde cada instância de operador é responsável por processar um subconjunto do fluxo de dados independentemente do restante do fluxo de dados. Por este motivo, muitos sistemas de processamento de fluxo são inerentemente distribuídos e podem consistir de dezenas ou centenas de operadores distribuídos em um grande número de nós de processamento, onde cada nó de processamento executa um ou mais operadores [76] [77].

O trabalho [22] propõe oito regras gerais que sustentam o processamento de fluxo de dados, entretanto as mais importantes relacionadas a esta tese são: manter os dados em movimento, gerar resultados previsíveis, garantir a disponibilidade dos dados, e processar e responder instantaneamente. Para satisfazer estas regras, mesmo na presença de reconfiguração dinâmica, uma plataforma de reconfiguração deve ser confiável (i.e., não produzir resultados errôneos e reduzir a disponibilidade do sistema) e não interromper o fluxo.

Os trabalhos [3] [7] [51] [52] [78] [79] são exemplos de abordagens para processamento de fluxo de dados com algum suporte a reconfiguração dinâmica. Entretanto, dentre estes trabalho, apenas o trabalho de Ertel e Felber [7] possui suporte a reconfiguração composicional, tendo os demais suporte apenas para reconfiguração paramétrica. Apesar do trabalho [79] permitir que o administrador do sistema possa definir suas próprias regras de reconfiguração (i.e., lógica de

PUC-Rio - Certificação Digital Nº 1312421/CA

orquestração), estas regras podem realizar apenas modificações através da chamada de comandos de controle previamente suportados no *System S*. Deste modo, não é possível, por exemplo, corrigir o código de um operador caso seja descoberto um *bug* após a implantação do sistema.

2.3 Dependências direta e indireta

Em muitos sistemas, como por exemplo os de processamento de fluxo de dados, os componentes têm dependências diretas e indiretas. A dependência direta ocorre quando dois componentes *A* e *B* são adjacentes (i.e., há uma aresta no grafo interconectando-os) e além disto, eles possuem alguma dependência de software. No exemplo da Figura 1, os componentes *Compressão* e *Descompressão* têm uma dependência direta, como já explicado na seção 1.1. A dependência indireta, por sua vez, é uma dependência entre componentes *X* e *Z*, os quais não estão diretamente ligados (i.e., não há uma aresta no grafo interconectando-os) [7]. Assim, existe pelo menos um componente intermediário (i.e., vértice intermediário) *Y* que permite que uma tupla saindo de *X* chegue em *Z*. No exemplo dado no trabalho [7] e ilustrado na Figura 8, o componente *Resposta* depende do componente *Carga*, entretanto, há um componente entre eles. Os autores [7] explicam que para substituir o componente *Carga* por sua versão não bloqueante (*NIO – Non-blocking IO*), é preciso substituir também o componente *Resposta* por sua versão NIO, uma vez que há uma dependência indireta entre os tipos de componentes *Carga* e *Resposta*. Além disto, como podem existir tuplas em trânsito entre os componentes *Carga* e *Resposta*, o sistema deve garantir que todas estas tuplas sejam processadas pela versão correta do componente *Resposta*.

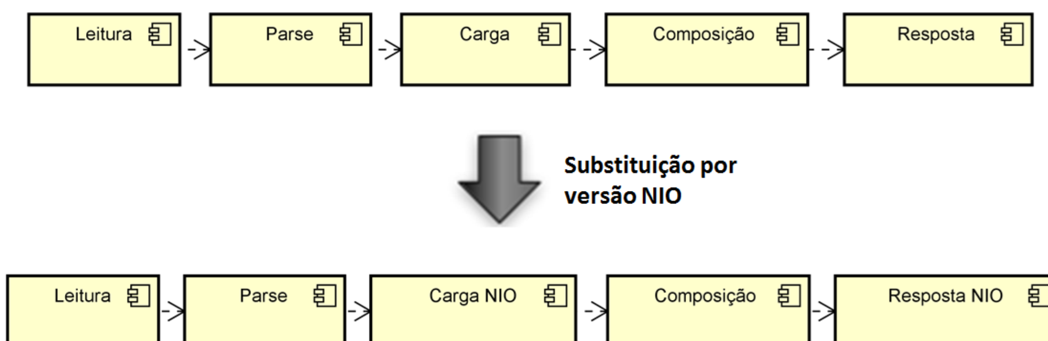

Figura 8. Exemplo de dependência indireta entre os componentes *Carga* e *Resposta*

PUC-Rio - Certificação Digital Nº 1312421/CA

A importância de mapear as dependências entre os componentes está no fato que a reconfiguração de um componente pode causar conflitos ou inconsistências no sistema. Assim como na engenharia de software é analisada a dependência entre classes (ou outros artefatos de software, como módulos, por exemplo) para que se saiba a priori quais partes do sistema podem ser afetadas por conta de uma modificação, sistemas dinamicamente reconfiguráveis para o processamento de fluxo de dados também precisam conhecer as dependências – diretas e indiretas - entre os componentes. Outro ponto relevante na avaliação das dependências é que as alterações de software feitas em um sistema reconfigurável são realizadas enquanto o sistema está em execução. Para tal a abordagem proposta avalia em tempo de execução as dependências entre os componentes para garantir que não haja inconsistências por conta de uma reconfiguração.

2.4 Trabalhos relacionados

PUC-Rio - Certificação Digital Nº 1312421/CA

Reconfiguração de *software* em tempo de execução é um tópico de pesquisa que combina questões e abordagens de áreas como engenharia de software [80] [81], linguagens de programação [64] [82] e sistemas operacionais [4] [66]. Entretanto, um problema comum é a identificação de estados no qual o sistema está seguro e pronto para evoluir [6]. Os autores Ertel e Felber [7] propõem um *framework* para sistemas que são modelados usando programação baseada em fluxo (*flow-based programming* – FBP) [83]. A ideia por trás do FBP é modelar o sistema como um grafo acíclico direcionado onde os operadores (vértices) são funções que processam o fluxo de dados e arestas que definem as portas de entrada e saída de cada operador. Como as mensagens são entregues em ordem, o trabalho [7] encaminha mensagens especiais informando quando um componente (i.e., operador) está em um estado seguro para ser reconfigurado. Quando um componente recebe esta mensagem, ele é substituído pela nova versão.

Apesar das vantagens da proposta [7], ela não lida com nós que podem aparecer ou desaparecer a qualquer momento e também não assume que alguns vértices podem ter versões diferentes. Desta forma, a limitação principal do trabalho [7] é que ou todos os componentes realizam a reconfiguração ou nenhum pode prosseguir com a reconfiguração, de modo parecido como acontece com uma transação que tem as propriedades ACID (*Atomicity, Consistency, Integrity*

and Durability) [84]. Como demonstrado por Nouali-Taboudjemat, Chehbour e Drias [84], tal abordagem não tem uma boa taxa de convergência (i.e., a reconfiguração falha na maioria dos casos) em cenários que têm falhas ou desconexões frequentes, tais como sistemas que lidam com dispositivos móveis. Como a transação deve ser atómica, i.e., ou todos os nós realizam a reconfiguração ou nenhum realiza, na maioria dos casos a reconfiguração não pode ser realizada. Por fim, enquanto um componente é modificado para assumir uma versão mais nova, ele é permanece impossibilitado de processar o fluxo de dados, causando assim uma contenção temporária em parte do sistema.

Também focado em sistemas de processamento de fluxos dinamicamente reconfiguráveis, a pesquisa de Schweppe, Zimmermann e Grill [73] propõe um método flexível para o processamento a bordo de carros de dados de sensores que é modelado como um sistema de processamento de fluxo de dados. No que diz respeito a questões de reconfiguração dinâmica, esta abordagem é capaz de trocar parâmetros de componentes, a topologia do sistema (i.e., a estrutura do grafo), ou o modo como os componentes armazenam seus dados. Quando um parâmetro de um componente é atualizado, os autores [73] usam um mecanismo de transferência de estado para realizar uma atualização (i.e., transformação) do estado antigo para o novo estado do componente. No entanto, a proposta não permite a modificação de um componente por uma versão mais nova (i.e., não permite a adaptação composicional). Por fim, os sistemas considerados pelos autores são sempre implantados em um único nó (i.e., um carro), ao contrário da abordagem proposta por esta tese, onde o sistema é distribuído e tem múltiplas instâncias de um tipo de componente.

O trabalho pioneiro de Kramer e Magee [15] propôs e provou que o critério de *quiescência* garante a consistência do sistema durante o processo de atualização. O modelo dos autores representa o sistema distribuído como um grafo direcionado cujos nós interagem por meio de transações (i.e., uma sequência de mensagens que deve ser executada atomicamente). O ponto fraco deste trabalho é que ele causa uma interrupção considerável uma vez que bloqueia toda a computação potencialmente dependente durante a evolução do sistema. Por conta da interrupção causada pelo quiescência, surgiu *Tranquillity* [26], uma alternativa mais fraca (i.e., relaxada) que o critério de quiescência. A ideia por trás do *tranquility* é que a reconfiguração pode prosseguir mesmo que exista uma

PUC-Rio - Certificação Digital Nº 1312421/CA

transação em execução, desde que o componente a ser reconfigurado não esteja envolvido na transação. Alguns pesquisadores argumentam que *tranquility* poderia permitir atualizações inseguras se uma sub-transação fosse iniciada por uma outra sub-transação que não está diretamente conectada (i.e., não tem dependência direta) com o componente que iniciou a transação raiz [6]. Outra desvantagem de ambos, quiescência e *tranquillity*, é que eles bloqueiam parte do sistema para substituir os componentes por suas novas versões, o que por sua vez pode causar uma interrupção significativa do sistema [6] [18].

Rubah [65] utiliza a técnica de pontos de atualização para realizar a reconfiguração no sistema. A vantagem é que a plataforma de reconfiguração não precisa encontrar um estado seguro para realizar a reconfiguração e nem precisa se preocupar com versões diferentes coexistindo no nó. Por outro lado, fica a cargo do desenvolvedor do sistema adicionar os pontos de atualização que indicam os momentos que o sistema está em um estado quiescente. Como discutido na seção 2.1, a medida que o sistema fica mais complexo, mais difícil é para o programador encontrar pontos de atualização válidos, o que pode aumentar possibilidade da adição de um ponto de atualização em um local não apropriado do sistema. Como vantagens, Rubah permite que a reconfiguração seja realizada no momento que foi solicitada a reconfiguração ou adiada até o momento que o componente seja utilizado (i.e., abordagem denominada de *lazy*).

Enquanto a *quiescência* [15] e *tranquillity* [26], por exemplo, bloqueiam o sistema para permitir sua evolução, outros trabalhos são capazes de executar concorrentemente ambas as versões, a antiga e a nova [6] [18] [44] [85]. As propostas de Ma *et al.* [6] e Ghafari *et al.* [18] garantem que, enquanto uma reconfiguração é executada, qualquer transação existente com todas suas sub-transações é executada inteiramente pela configuração antiga ou nova do sistema (i.e., versão antiga ou nova) [6].

O trabalho de Ma *et al.* [6] gerencia as dependências entre os componentes por meio de grafos direcionados onde vértices representam componentes versionados e arestas representam as dependências. As maiores desvantagens são a falta de mecanismos para garantir a consistência do estado entre as versões antiga e nova, e o *overhead* necessário para manter o grafo que representa a configuração do sistema [18] [61]. Com intuito de resolver o problema do *overhead*, Ghafari *et al.* [18] escolheram usar o *tempo de evolução* (i.e., o

PUC-Rio - Certificação Digital Nº 1312421/CA

timestamp no qual a reconfiguração foi realizada) como mecanismo para decidir se uma transação deve ser executada pela versão antiga ou nova. Assim, uma transação iniciada antes do tempo de evolução é executada pela versão antiga, caso contrário é executada pela versão nova. Apesar do tempo de evolução causar pouco impacto no desempenho do sistema, sincronização de relógio é um problema bem conhecido em sistemas distribuídos. Kshemkalyani e Singhal [86] explicam que os relógio podem facilmente variar e acumular erros significativos, e que relógio físico impõe sérios problemas para aplicações que dependem de uma noção de tempo sincronizado. De modo semelhante aos trabalhos de Ma *et al.* [6] e Ghafari *et al.* [18], POLUS [81] permite que ambas versões de um componente e seus estados possam coexistir usando como base protocolos de coerência de memória cache. Por conta da coexistência de estados de diferentes versões, POLUS propõe funções de sincronização para garantir que modificações na versão de um estado sejam replicadas na outra versão de modo a garantir a consistência do sistema.

Uma das inovações do Upstart [44] [85] é permitir que o sistema continue operando enquanto está sendo reconfigurado (i.e., está parcialmente reconfigurado), assim como é feito na abordagem proposta por esta tese. Upstart permite que a reconfiguração seja feita gradualmente de tal modo que nós com diferentes versões possam coexistir sem causar erro ou inconsistência no sistema. Para tanto, Upstart utiliza interceptadores (i.e., uma espécie de *proxy*) para cada uma das versões que possam coexistir no sistema. Assim, quando é realizada uma chamada a um determinado componente, a chamada é tratada por interceptadores que fazem as alterações necessárias para que de fato a requisição possa ser executada pelo componente que está atualmente implantado no nó. Apesar de não requerer um estado seguro para que o sistema seja reconfigurado, Upstart ainda necessita bloquear o componente a ser reconfigurado para que as modificações em um nó (i.e., adição/remoção dos interceptadores e substituição do componente de uma versão i para $i + 1$) sejam feitas consistentemente.

O trabalho de Giuffrida e Tanenbaum [67] propõe uma atualização cooperativa no qual o componente a ser substituído é notificado sobre a reconfiguração e coopera com a plataforma de reconfiguração (i.e., *middleware* de reconfiguração) a fim de progredir para um estado seguro antes da reconfiguração ser de fato executada. A desvantagem desta abordagem é que o componente tem

PUC-Rio - Certificação Digital Nº 1312421/CA

ciência da reconfiguração e coopera ativamente com a plataforma de reconfiguração, o que pode causar erro na reconfiguração caso o componente a ser reconfigurado não coopere de maneira adequada. Além disto, o trabalho [67] não discute problemas relacionados à reconfiguração em nós distribuídos.

Siniavine e Goel [87] e os autores de MCR (*Mutable Checkpoint-Restart*) [5] [19] utilizam técnicas de *checkpoint* [86] para realizar reconfiguração de software. No momento de uma reconfiguração, a ideia central em ambos os trabalhos é esperar que o sistema atinja a quiescência, para então realizar o *checkpoint* dos componentes afetados, inicializar os novos componentes e por fim restaurar o *checkpoint*. A abordagem de Siniavine e Goel [87] foi desenvolvida especificamente para realizar *patches*, enquanto MCR é mais genérico e pode ser utilizado para outros tipos de reconfiguração. Além disto, MCR utiliza uma técnica de transferência de estado que é capaz de automatizar parte do processo da transformação dos estados, enquanto Siniavine e Goel [87] lidam com reconfigurações que não necessitam alterar o estado dos componentes. O problema com ambos os trabalhos é que o sistema ainda precisa alcançar a quiescência para realizar uma reconfiguração, além de não haver suporte para sistemas distribuídos.

Até o momento da escrita desta tese, não são conhecidas pesquisas que lidam com o problema de reconfiguração de software em tempo de execução de um tipo de componente que tem um conjunto dinâmico de instâncias distribuídas espalhadas no sistema de processamento de fluxo de dados distribuído, e que além disto, garantam a consistência global do sistema, causem pouca interrupção no sistema e suportem nós móveis. Isto é, cada tipo de componente pode ser implantado em muitos nós distribuídos (e possivelmente móveis), em vez de em um único nó. Deste modo, a reconfiguração para uma nova versão de um componente requer a modificação coordenada de muitas instâncias de componentes implantadas em numerosos nós (móveis) que podem ficar temporariamente desconectados/indisponíveis.

PUC-Rio - Certificação Digital Nº 1312421/CA

3
Modelo do sistema

Inspirada no trabalho [73], representaremos o sistema de processamento de fluxo de dados, como um grafo acíclico direcionado que consiste de múltiplos operadores (i.e., vértices) implantados em nós distribuídos. Formalmente, o grafo G = (V, E) consiste de vértices e arestas. Um vértice representa um operador sobre o fluxo e uma aresta representa um canal de fluxo (i.e., *stream channel*). Uma aresta $e = (v1, v2)$ interconecta a saída de um vértice $v1$ com a entrada de um vértice $v2$. Um vértice sem portas de entrada (i.e., sem arestas de entrada) é referido como vértice produtor. Analogamente, vértices sem portas de saída são chamados vértices consumidores. Por fim, vértices com ambas portas de entrada e saída são chamados de vértices intermediários.

A tupla t = (*valor*, *caminho**) consiste de um valor (*valor*) e um caminho de execução (*caminho**) que contém as versões dos operadores que uma tupla t percorreu através de G. Por exemplo, uma tupla t que saiu do vértice produtor *VP1* para o vértice consumidor *VC1* através dos vértices intermediários *O1* e *O2* contém *caminho* = {*VP1*, *O1*, *O2*}. O atributo *valor* da tupla t é transformado (i.e., processado) ao longo do grafo. Um fluxo f = (t*) entre $v1$ e $v2$ consiste de uma sequência ordenada de tuplas t* onde $t1 < t2$ representa que $t1$ foi enviada antes de $t2$ por um nó $n1$. Um vértice é composto por funções $f^{seleção}$, $f^{saída}$ e $f^{atualização}$, e por um estado interno no caso de um vértice com estado. Quando um vértice $v1$ gera uma tupla (i.e., envia a tupla através da porta de saída), seus vértices sucessores (i.e., os vértices que recebem o fluxo de $v1$) recebem a tupla através da função $f^{seleção}$, a qual é responsável por selecionar, ou não, a tupla para ser processada pela função $f^{atualização}$.

Para padronizar os termos e noções utilizadas nesta tese, um vértice (ou operador) será genericamente referido como componente [88]. Um nó é qualquer dispositivo físico (e.g., computador ou *smartphone*) que executa um componente. Um nó de processamento (*Processing Node* – PN), por sua vez, é um nó que contém pelo menos um operador intermediário (i.e., um operador com portas de

PUC-Rio - Certificação Digital Nº 1312421/CA

entrada e saída). Além disto, como sistemas de fluxo de dados devem ser elásticos para se adaptarem às variações no volume dos fluxos de dados, esta tese considera que alguns nós de processamento dividem suas cargas de trabalho entre si [24] [89] [90].

Como principais suposições sobre o sistema e rede subjacente, esta tese considera que qualquer nó cliente (i.e., *Client Node* – CN) é capaz de entrar ou sair do sistema a qualquer momento, mensagens são entregues confiavelmente em ordem FIFO (*First In*, *First Out*) entre dois nós, os nós executam seus componentes corretamente, e todos os componentes por si só não introduzem falhas que podem levar o sistema a um estado inconsistente. Também é assumido que não existem falhas bizantinas e que se qualquer CN falhar (*fail-stop*), o sistema é capaz de detectar em tempo hábil a falha do nó [86]. Apesar de um CN poder sair do sistema por conta de uma falha (*fail-stop*) ou desconexão, o sistema como um todo permanece operacional. Por fim, os nós localizados na nuvem não falham. Como o componente precisa ter apenas um método para ser capaz de processar as tuplas, assume-se que a interface provida do componente não muda, entretanto, os demais métodos podem sofrer modificação. Em relação à quantidade de versões que podem existir no sistema, é considerado que a ocorrência de múltiplas versões são situações excepcionais, onde o sistema se encontra parcialmente reconfigurado. Deste modo, a quantidade de versões em paralelo esperada no sistema deve ser inferior a 10.

PUC-Rio - Certificação Digital Nº 1312421/CA

3.1 Arquitetura da solução proposta

Levando em consideração que muitos dos atuais sistemas distribuídos seguem o paradigma de comunicação aplicativo móvel-nuvem (*mobile-cloud*) [91] [92], a arquitetura proposta aqui (Figura 9) é composta por nós clientes (CNs), os quais podem ser nós móveis ou estacionários, e nós de processamento (PNs) implantados na nuvem. Os CNs são interconectados à nuvem através de um *gateway* (GW), o qual por sua vez encaminha o fluxo dos CNs para os PNs. Atualmente, espera-se que muitos dos CNs sejam dispositivos móveis, os quais introduzem novos problemas, tal como a alta taxa de conexão/desconexão, e.g., um nó móvel pode ficar (in)disponível a qualquer hora.

Considerado que esta tese modela o sistema como um sistema de fluxo de dados distribuído, alguns componentes de *software* são responsáveis por tarefas de comunicação, enquanto outros são responsáveis por tarefas de processamento, i.e., análise, agregação e transformação do fluxo de dados. O GW, por exemplo, é um nó responsável por redirecionar o fluxo de dados dos CNs externos à nuvem para PNs executando – em máquinas virtuais – dentro da nuvem, ou no sentido oposto, dos PNs para os CNs, e por interconectar os CNs ao RM (*Reconfiguration Manager*, ou Gerente de Reconfiguração). Um CN, por sua vez, tem alguns componentes de comunicação para permitir a interação com o GW, ao mesmo tempo que também pode ter componentes de processamento que realizam alguma tarefa de pré-processamento antes do fluxo ser enviado para a nuvem.

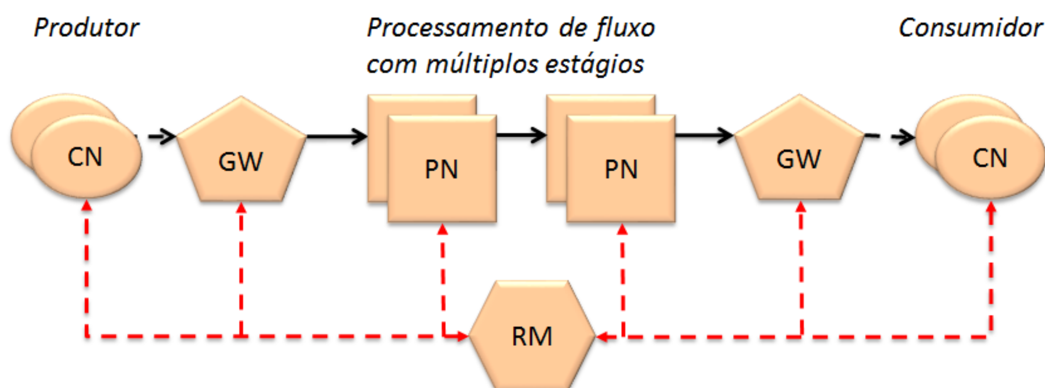

Figura 9. Arquitetura proposta

Além dos nós já mencionados, o Gerente de Reconfiguração gerencia a implantação dos componentes de software e coordena a execução da reconfiguração realizada nos nós, sejam eles CNs ou PNs. O Gerente de Reconfiguração, o qual é implantado na nuvem, é responsável por coordenar (i.e., iniciar e orquestrar a execução de todas as operações que englobam uma reconfiguração distribuída) o processo de reconfiguração em todo o sistema (e.g., a implantação de novos componentes de software) em vários nós. Por exemplo, se a reconfiguração for a implantação de uma nova versão de um componente, o Gerente de Reconfiguração envia a nova versão do componente para os nós, verifica se todos eles implantaram corretamente a nova versão, e então realiza a remoção da versão antiga. As linhas tracejadas em vermelho representam o canal de controle da reconfiguração entre o Gerente de Reconfiguração e os demais nós, enquanto as linhas em preto representam o fluxo de dados do sistema. Portanto,

PUC-Rio - Certificação Digital Nº 1312421/CA

todas as reconfigurações realizadas nos nós são iniciadas e orquestradas pelo Gerente de Reconfiguração.

PUC-Rio - Certificação Digital Nº 1312421/CA

4
Uma abordagem para reconfiguração dinâmica e distribuída de software

Este capítulo apresenta uma abordagem não quiescente para reconfiguração dinâmica em sistemas de processamento de fluxo de dados distribuídos. A abordagem proposta é baseada na ideia de que um dado produzido por um CN deve ser inteiramente processado por uma versão específica de cada componente (i.e., função $f^{atualização}$). Ou seja, o dado deve ser processado ou pela configuração antiga ou pela configuração nova do sistema. Entretanto, não há problema em atualizar (i.e., reconfigurar) um componente *C* enquanto uma tupla (i.e., mensagem) *T* atravessa o sistema (i.e., o grafo), uma vez que o sistema mantém a versão antiga e nova de *C*, e dos seus componentes dependentes, até que todas as tuplas oriundas da versão antiga sejam drenadas do sistema. Assim, é mantida a configuração antiga e nova do sistema para que os dados possam ser corretamente processados. Diferentemente de outros trabalhos, como Makris e Ryu [66], a abordagem proposta por esta tese não requer que o sistema atinja a quiescência (ou um estado) para reconfigurar uma função $f^{atualização}$.

No cenário mostrado na Figura 1, existe uma dependência entre os componentes *Compressão* e *Descompressão* uma vez que eles devem usar algoritmos compatíveis para que a troca de mensagens (i.e., tuplas) através da rede seja realizada corretamente. Deste modo, caso a reconfiguração dos componentes *(Des)Compressão* aconteça antes que qualquer nó remetente (i.e., nó produtor) envie uma mensagem, todas as mensagens serão processadas pela versão nova do componente *Descompressão*, uma vez que os remetentes usaram a versão nova do componente *Compressão*. Entretanto, caso a reconfiguração aconteça enquanto os remetentes enviam mensagens (i.e., o envio das mensagens é um fluxo contínuo), algumas mensagens devem ser processadas pela versão nova do componente *Descompressão* (se e somente se a mensagem foi enviada pela nova versão do componente *Compressão*), enquanto outras mensagens devem sem processadas pela versão antiga do componente *Descompressão*, como ilustrado na Figura 10.

PUC-Rio - Certificação Digital Nº 1312421/CA

No momento em que existem alguns remetentes com a versão antiga e outros com a versão nova, os nós destinatários devem ter implantados ambas as versões do componente *Descompressão* para serem capazes de receberem corretamente mensagens de qualquer nó remetente. Portanto, ambas versões do componente *Descompressão* coexistem nos nós destinatários, e podem ser executadas em paralelo, enquanto o sistema está sendo reconfigurado.

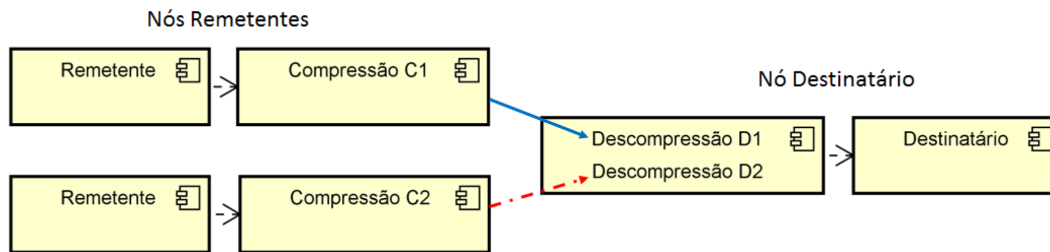

Figura 10. Exemplo de sistema parcialmente reconfigurado com dois fluxos possíveis no componente *Descompressão*

Cada componente tem uma ou muitas funções $f^{seleção}$, $f^{saída}$ e $f^{atualização}$, e os componentes podem ter interdependências. A vantagem de permitir que um componente tenha mais de uma $f^{atualização}$ executando paralelamente é que, em face de uma reconfiguração, a nova função é capaz de processar parte do fluxo de dados enquanto a versão antiga ainda está em uso, e por isto não pode ser desativada. Consequentemente, quando uma tupla *T* é recebida por uma função $f^{seleção}$, ela deve escolher a função $f^{atualização}$ correta para processar *T*. Para fazer isso, a função $f^{seleção}$ verifica o *caminho de execução* de *T* quando há mais de uma função $f^{atualização}$, caso contrário não existe necessidade de verificar o *caminho* já que só existe uma função $f^{atualização}$ para processar *T*. As funções $f^{seleção}$ e $f^{saída}$ representam as portas de entrada e saída de um componente, respectivamente, enquanto a função $f^{atualização}$ é o algoritmo responsável por realizar a transformação (i.e., processamento) no fluxo de dados de entrada. Assim, a abordagem é capaz de reconfigurar os algoritmos que processam os fluxos de dados (i.e., funções $f^{atualização}$) e a topologia do sistema através da reconfiguração das funções $f^{seleção}$ e $f^{saída}$. Além disto, cada componente mantém uma tabela de dependências que o componente tem com os demais componentes do sistema.

PUC-Rio - Certificação Digital Nº 1312421/CA

4.1 Gerenciamento de múltiplas versões

No exemplo da Figura 11 por questões de didática, considera-se que só haja dependências diretas entre os componentes. A Figura 11 mostra o fluxo de dados parcial de uma tupla *T* quando o sistema tem as funções $f^{atualização}$ *Componente A1*, *Componente B1*, *Componente C1*, *Componente D1* e *Componente E1*. A Figura 12 mostra que as versões *Componente B2*, *Componente C2*, *Componente D2* e *Componente E2* foram adicionadas no sistema, as quais são incompatíveis com as versões anteriores, e que a tupla *T* é transformada (i.e., processada) pelas funções $f^{atualização}$ *Componente D1* e *Componente E1* para que seja mantida a consistência do sistema. Assim, na Figura 12, quando a tupla *T* chega na função $f^{seleção}$ do componente que contém as funções $f^{atualização}$ *Componente D1* e *Componente D2*, a função $f^{seleção}$ verifica que a tupla *T* veio da função $f^{atualização}$ *Componente 1* e então *T* é transformada por *Componente D1*. O mesmo procedimento ocorre quando *T* chega no *Componente E*. Sendo assim, cada componente deve ter ciência das suas dependências e do caminho de execução da tupla para ser capaz de escolher a função $f^{atualização}$ correta.

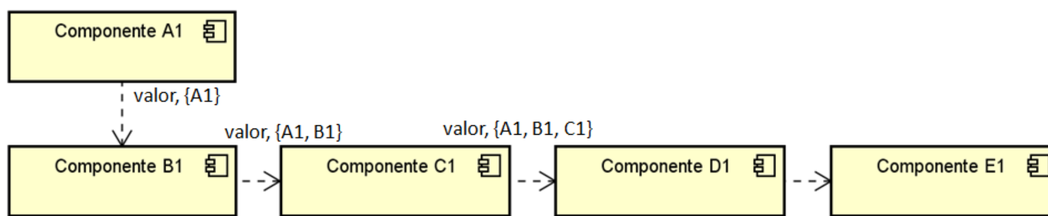

Figura 11. Fluxo de dados parcial do cenário motivador

As dependências podem ser gerenciadas usando duas abordagens, gerenciamento estático ou dinâmico. O gerenciamento estático, o qual é mais simples, não leva em consideração os próximos componentes dependentes (i.e., os componentes dependentes no fluxo abaixo, ou seja, no *downstream*) para gerar o caminho de execução de uma tupla. Assim, sempre que um componente processa uma tupla *T*, a versão da função $f^{atualização}$ do componente que processou *T* é adicionada no caminho de execução de *T*, como mostrado na Figura 11 e na Figura 12. Por fim, quando *T* chega em um próximo componente, como por exemplo no componente que contém as funções $f^{atualização}$ *Componente D1* e *D2*, sua função $f^{seleção}$ verifica o caminho de execução de *T* para decidir qual é a função $f^{atualização}$ correta para processar *T*. Para tal, cada componente tem uma lista com

PUC-Rio - Certificação Digital Nº 1312421/CA

todas as versões dos componentes dos quais ele depende (i.e., todos os componentes no fluxo acima – *upstream* – que um componente *C* depende). No exemplo da Figura 8, o componente *Resposta* tem na sua lista de dependência o componente *Carga*, já o componente *Resposta NIO* depende do componente *Carga NIO*.

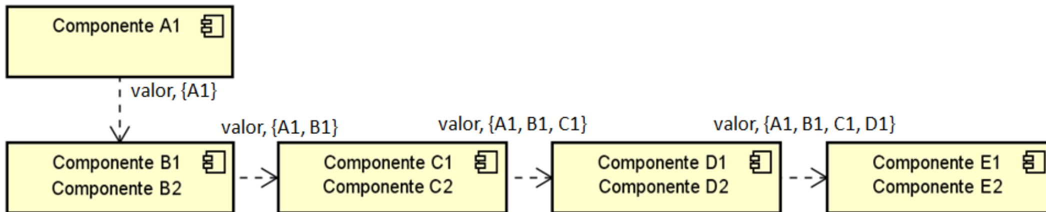

Figura 12. Caminho de execução da tupla *T* no sistema parcialmente reconfigurado

A abordagem dinâmica para gerenciamento das dependências, por outro lado, verifica se há algum componente dependente no *downstream* antes de adicionar a versão da função $f^{atualização}$ no caminho de execução. Além disto, em cada componente, o caminho de execução é analisado para que se possa descartar as versões que não têm mais componentes dependentes. Na Figura 13, por exemplo, a versão do componente anterior sempre é removida já que não existem dependências indiretas, ficando assim somente a versão do componente corrente no caminho de execução. Caso seja considerado que há uma dependência indireta entre os componentes *Componente A* e *Componente E*, como ilustrado na Figura 14, *A1* não pode ser removido do caminho de execução já que o *Componente E1* é dependente do *Componente A1*. Deste modo, em cada componente do fluxo é verificado que existe um componente no *downstream* (i.e., o *Componente E*) que é dependente de *A1* e, portanto, *A1* não pode ser removido do caminho de execução.

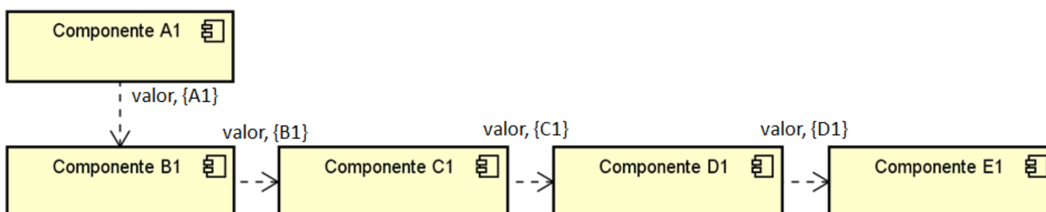

Figura 13. Caminho de execução usando gerenciamento dinâmico onde só há dependências diretas

A vantagem de empregar o gerenciamento estático é que ele é simples, tem baixo custo de execução e a modificação das dependências não afeta o sistema, uma vez que o caminho de execução mantém todos os componentes que a tupla atravessou. Assim, a reconfiguração é realizada de forma mais simples e rápida. Entretanto, caso o caminho de execução cresça (i.e., existem vários componentes

PUC-Rio - Certificação Digital Nº 1312421/CA

intermediários entre o nó produtor e o nó consumidor), pode haver degradação no desempenho por conta do custo de rede, por exemplo, ocasionado pelo caminho de execução excessivamente grande.

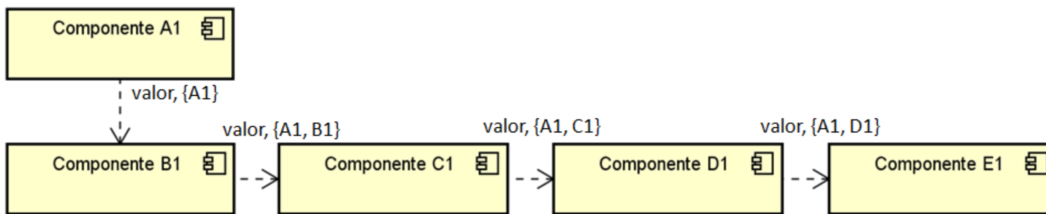

Figura 14. Gerenciamento dinâmico com a ocorrência de dependência indireta

Por outro lado, o gerenciamento dinâmico de dependência tem a vantagem de não desperdiçar rede e memória, já que o caminho de execução contém apenas informação útil, o que é uma vantagem para caminhos de execução grandes. A desvantagem é a complexidade introduzida para manter o caminho de execução o mais curto possível e manter a consistência do sistema quando as dependências são modificadas. Em cada componente, toda dependência no fluxo abaixo (i.e., *downstream*) precisa ser avaliada para remover as versões desnecessárias no caminho de execução. Além disto, caso uma reconfiguração insira uma nova dependência, todos os componentes no fluxo acima (i.e., *upstream*) devem ser notificados para que atualizem a lista de dependências, e as tuplas em trânsito devem ser especialmente manipuladas pelo sistema. Na Figura 15, caso o administrador do sistema adicione o *Componente E2* que têm dependência com o *Componente A2*, e considerando que não há dependência entre os componentes *Componente A1* e *Componente E1*, as dependências *downstream* dos componentes *Componente B1*, *Componente C1* e *Componente D1* devem ser atualizadas. Assim, as tuplas em trânsito, as quais não têm a versão *A1* no caminho de execução, devem ser processadas pela versão antiga do *Componente E* (i.e., *Componente E1*). Após a atualização das dependências (Figura 16), os componentes passam a manter no caminho de execução a versão *A2*.

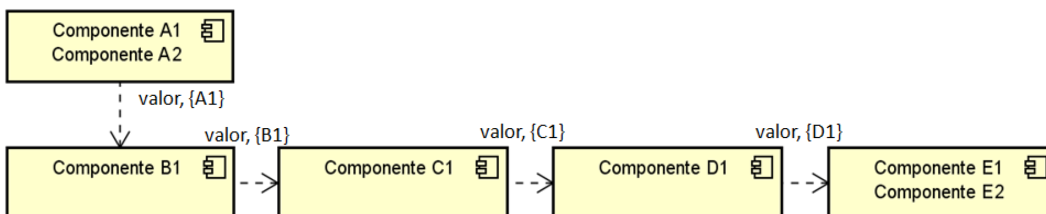

Figura 15. Caminho de execução de uma tupla em trânsito

PUC-Rio - Certificação Digital Nº 1312421/CA

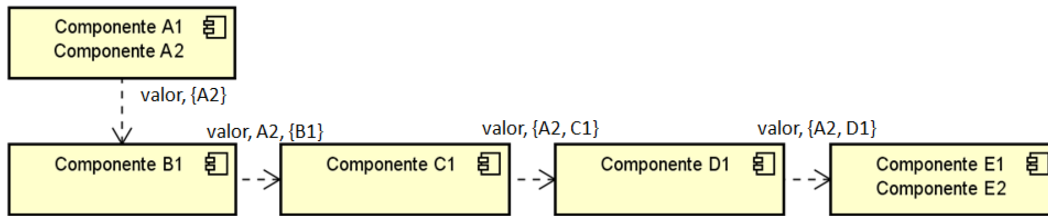

Figura 16. Caminho de execução de uma tupla gerada após a modificação das dependências e enquanto o sistema está parcialmente reconfigurado

4.2 Gerenciamento de estado

Na maioria das abordagens, quando uma função $f^{atualização}$ C é reconfigurada (i.e., atualizada), a abordagem de reconfiguração deve atualizar as instâncias C_N para C_{N+1} (i.e., as versões antiga e nova, respectivamente). Para componentes (i.e., funções $f^{atualização}$) com estado, a abordagem deve transferir o estado de C_N para C_{N+1}. Entretanto, a coexistência de C_N e C_{N+1} requer um mecanismo de sincronização para garantir que a execução concorrente de C_N e C_{N+1} não leve o sistema para um estado inconsistente, como discutido por Chen *et al.* [81]. Para evitar que existam estados duplicados entre C_N e C_{N+1} (i.e., cada versão mantendo o seu estado próprio) e simplificar o mecanismo de sincronização, ambas as versões compartilham o mesmo estado (Figura 17 b) durante a reconfiguração do sistema, de modo semelhante aos trabalho de Ajmani, Liskov e Shrira [44] e Ajmani [85].

Quando o componente C_{N+1} possui o mesmo tipo de estado (i.e., a definição do estado) do componente C_N, a reconfiguração de C_N para C_{N+1} não requer que o estado seja reconfigurado. Deste modo, o processo de reconfiguração é feito como mostrado na Figura 17, onde antes da reconfiguração (a) apenas o componente *C1* utiliza o estado, durante a reconfiguração (b) ambos os componentes *C1* e *C2* compartilham o mesmo estado, e após a reconfiguração (i.e., após *C1* ser removido) apenas o componente *C2* tem acesso ao estado (c). Por outro lado, quando o componente C_{N+1} utiliza uma definição de estado diferente do componente C_N, como no exemplo da Figura 4, o desenvolvedor do sistema (i.e., o desenvolvedor do software construído com uso da abordagem proposta) deve fornecer a nova versão da definição do estado (Figura 4), a função de transformação do estado *v1* para *v2* (vide Figura 5), e um adaptador de versão (AV), vide Figura 18.

PUC-Rio - Certificação Digital Nº 1312421/CA

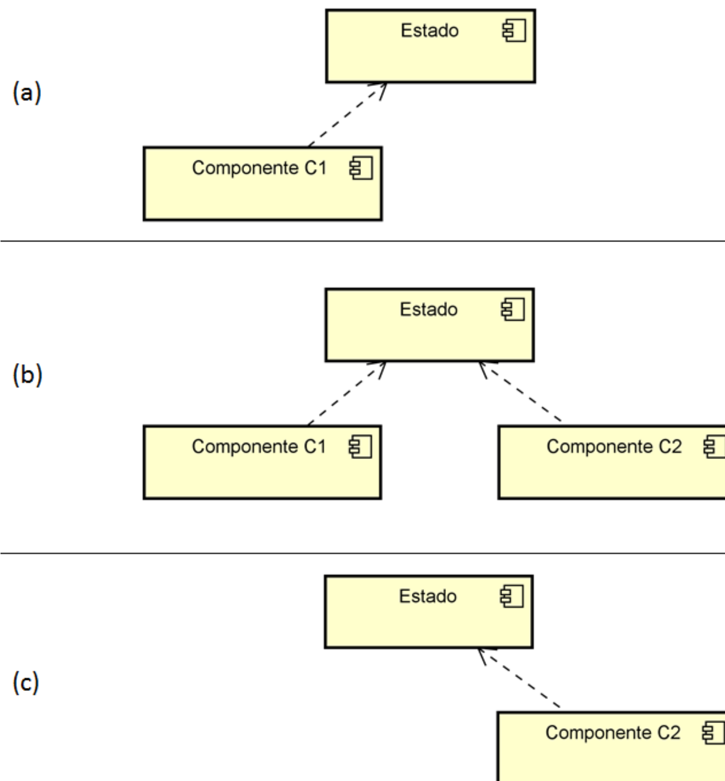

Figura 17. Exemplo de estado compartilhado antes (a), durante (b) e depois (c) da reconfiguração de um componente

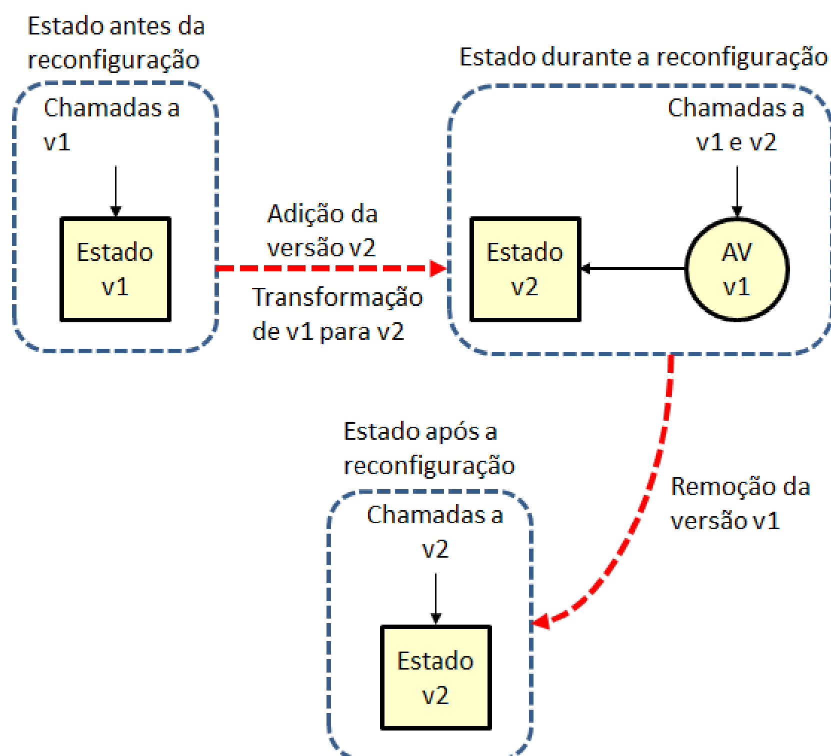

Figura 18. Ciclo para reconfiguração de um estado v1 para v2

Quando um estado *E* é reconfigurado de uma versão *v1* para *v2*, de modo análogo à reconfiguração de um componente, o estado *E* pode estar parcialmente reconfigurado (i.e., deve atender chamadas para as versões *v1* e *v2*) ou

PUC-Rio - Certificação Digital Nº 1312421/CA

completamente configurado (i.e., deve atender somente chamadas à versão *v2*). Assim que a reconfiguração do estado *E* na sua versão v1 é iniciada, a abordagem proposta aplica a função de transformação fornecida pelo desenvolvedor do sistema para que o estado *E* na versão *v1* possa ser convertido (i.e., transformado ou atualizado) na versão *v2*. Após ser aplicada a função de transformação, o estado *E* versão *v2* é então inicializado com o estado convertido pela função. Neste momento em que o sistema está parcialmente reconfigurado, isto é, alguns componentes ainda têm funções $f^{atualização}$ que funcionam com a versão *v1* de *E*, enquanto outros já utilizam a versão *v2*, a abordagem proposta utiliza o adaptador de versão para que as chamadas provenientes de funções $f^{atualização}$ que ainda não foram atualizadas, e deste modo utilizam o estado *E* na versão *v1*, sejam mapeadas para a versão *v2* de *E*. Após concluída a reconfiguração dos componentes (i.e., remoção das funções $f^{atualização}$ que utilizavam *v1* de *E*), a abordagem proposta remove o adaptador da versão *v1* e assim as chamadas realizadas ao estado *E* são diretamente atendidas por *v2*, como mostrado na Figura 18. O processo de reconfiguração do estado proposto nesta tese é uma adaptação da abordagem de atualização software proposta por Ajmani, Liskov e Shrira [44] e Ajmani [85].

PUC-Rio - Certificação Digital Nº 1312421/CA

4.3
Reconfiguração distribuída com múltiplas instâncias

Até o momento, só foi discutido como realizar reconfiguração coordenada dos componentes localmente, sem considerar a natureza distribuída dos sistemas de processamento de fluxo de dados, como na Figura 10. A abordagem proposta deve manter as instâncias relativas às versões antiga e nova em execução enquanto houver algum nó com uma instância da abordagem antiga. Além disto, a reconfiguração deve ser iniciada em sentido oposto ao fluxo (i.e., a reconfiguração é iniciada do vértice consumidor paro o vértice produtor). Deste modo, o processo de reconfiguração do sistema deve ser coordenado pelo *Gerente de Reconfiguração*. Sempre que o administrador do sistema precisar substituir algum componente, ele usa o Gerente de Reconfiguração para iniciar o processo da reconfiguração dinâmica de software.

Na Figura 19, se parte do fluxo de dados do PNB *M* (*Processing Node B* instância *M*) for encaminhado para o PNC *M*, o sistema atinge um estado inconsistente uma vez que o PNC *M* é incapaz de processar corretamente este

fluxo de dados. Assim, a Figura 20 mostra que os servidores PNC precisam ter ambas as versões (i.e., componentes *D1* e *D2*) enquanto o sistema está parcialmente reconfigurado, uma vez que alguns servidores PNA e PNB não foram reconfigurados ainda. Assim que os servidores PNA e PNB são reconfigurados, e não existem mais tuplas em trânsito dos componentes *B1* e *C1*, as instâncias dos componentes *D1* e E1 são removidas dos servidores PNC e dos nós NC, concluindo então a reconfiguração, como mostrado na Figura 21.

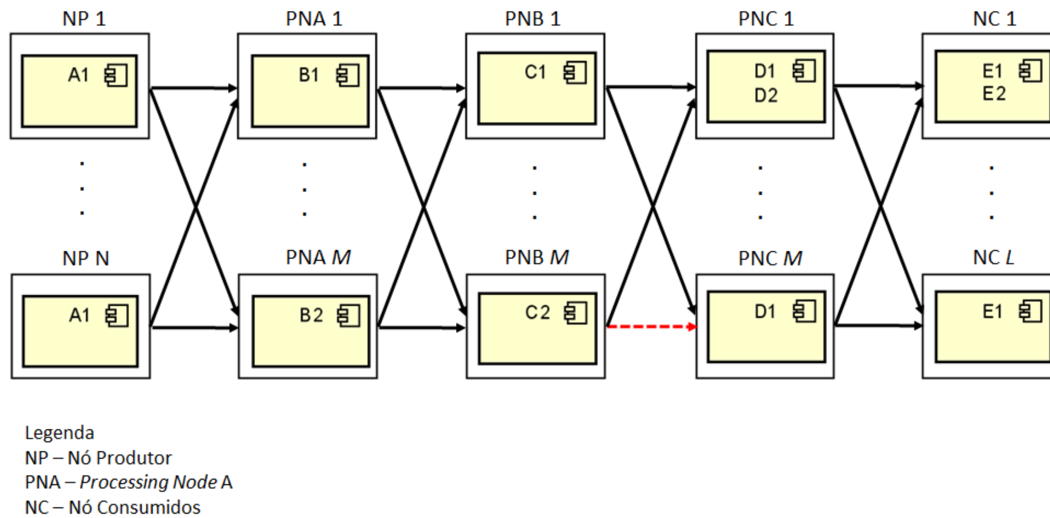

Figura 19. Reconfiguração parcial inconsistente no fluxo tracejado em vermelho

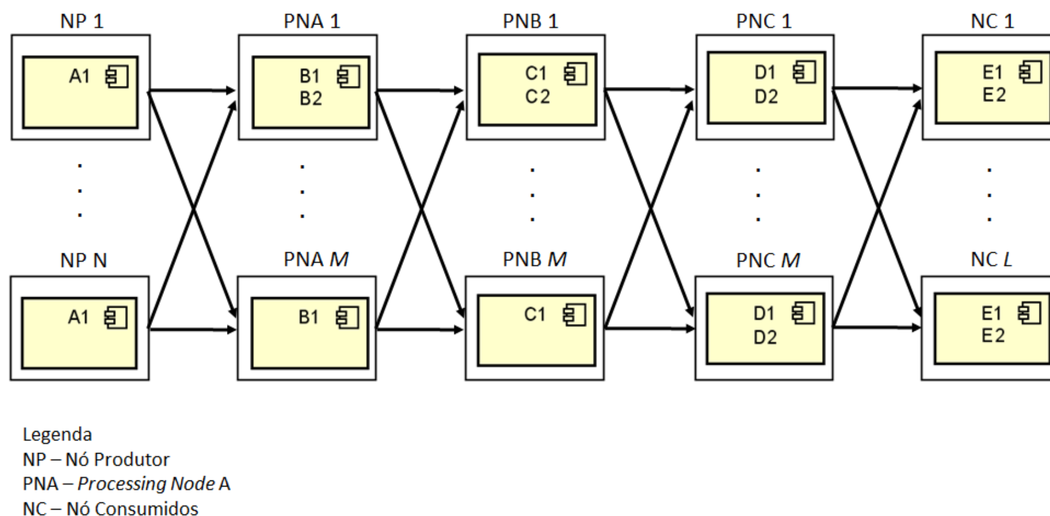

Figura 20. Reconfiguração parcial consistente

Para remover com segurança um componente *R* e garantir que não existam tuplas em trânsito em direção a *R*, a abordagem proposta é inspirada nos trabalhos seminais de Lamport e Chandy [93] [94] para saber quando é seguro remover um componente. A ideia é adicionar uma mensagem especial, chamada de marcador, no fluxo de dados para marcar um tempo específico *T* no qual um componente

PUC-Rio - Certificação Digital Nº 1312421/CA

pode ser removido com segurança. De modo semelhante ao trabalho de Ertel e Felber [7], assim que *R* recebe os marcadores de todas as instâncias de componentes dependentes, *R* é removido do sistema. Esta abordagem funciona como esperado pois as tuplas são entregues em ordem FIFO. Assim, quando o componente *R* recebe todos os marcadores, não há mais tuplas em direção a *R*.

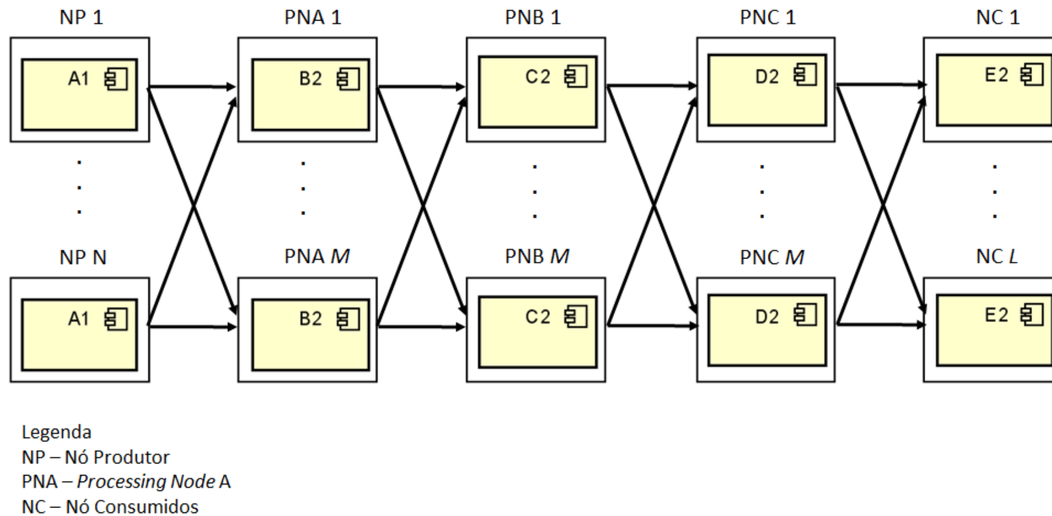

Figura 21. Sistema reconfigurado

Deste modo, para remover o componente *D1*, sempre que um servidor *PNB* remove o componente *C1* por conta de uma requisição do Gerente de Reconfiguração, o componente *C1* adiciona um marcador em todos os seus *M* canais de fluxo (i.e., *stream channels*) para as instâncias do componente *D1*, e assim que os *M* marcadores (i.e., um marcador de cada instância de *C1*) chegam em uma instância do componente *D1*, a instância *D1* é removida. Assim, as instâncias do componente *D1* são removidas com segurança e gradualmente de todo o sistema. Para tal, é assumido que cada componente conhece suas dependências fluxo a cima (i.e., *upstream*). Cada marcador carrega a informação sobre a versão do componente que foi previamente removida de um nó. Além disto, o Gerente de Reconfiguração informa a cada nó a quantidade de marcadores que ele deve receber antes de remover a sua instância do componente. Por exemplo, sempre que o componente *C1* for removido, o Gerente de Reconfiguração informa aos servidores *PNC* que existem *M* servidores *PNB* (i.e., cada servidor *PNC* deve receber *M* marcadores dos servidores *PNB*, notificando que o componente *C1* foi removido do sistema, para que o componente *D1* seja removido com segurança).

PUC-Rio - Certificação Digital Nº 1312421/CA

Como é esperado que muitos dos nós produtores sejam dispositivos móveis usando redes sem fio (e.g., Wi-Fi, 3G ou 4G), eles podem perder a conexão a qualquer momento. Sendo assim, a abordagem proposta lida com nós que podem aparecer e desaparecer do sistema como segue. Enquanto o sistema está sendo reconfigurado, como na Figura 10, se o remetente que tem a versão *Compressão C1* ficar indisponível antes de atualizar sua instância para a versão *Compressão C2*, o nó destinatário deve adiar a remoção do componente *Descompressão D1* até que o remetente fique disponível novamente e termine sua reconfiguração. Como este nó remetente pode ter ficado permanentemente indisponível (e.g., o dispositivo quebrou ou o usuário decidiu não mais utilizar a aplicação de bate-papo), é possível informar ao Gerente de Reconfiguração o *timeout* da reconfiguração. Após o *timeout*, é considerado que o nó saiu permanentemente do sistema para que o processo da reconfiguração possa ser concluído. Caso o timeout não seja informado, o Gerente de Reconfiguração esperará indefinidamente que o nó retorne. Outras políticas de reconfiguração podem ser implantadas no Gerente de Reconfiguração para cobrir diversos casos, como por exemplo dispositivos legados que não podem mais ser atualizados por algum tipo de restrição [95].

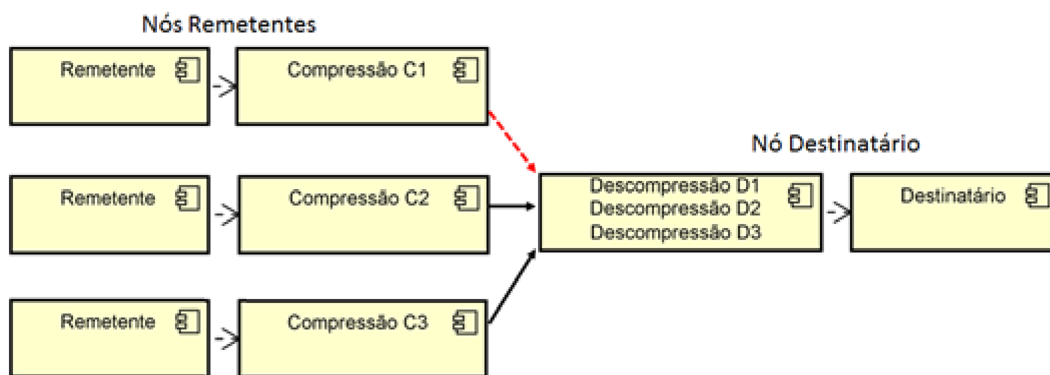

Figura 22. Sistema parcialmente reconfigurado com três versões em paralelo

Apesar da saída momentânea de um nó adiar a conclusão da reconfiguração, o sistema pode continuar sendo reconfigurado mesmo que alguns nós não tenham concluído ainda suas reconfigurações. No exemplo da Figura 22, caso o remetente que tem a versão *Compressão C1* fique indisponível por algum tempo, o qual é indicado com a seta tracejada em vermelho, o sistema é capaz de evoluir para as versões *Compressão C3* e *Descompressão D3*. Isto é possível pois cada componente pode ter várias funções $f^{\text{atualização}}$. Assim, os componentes

PUC-Rio - Certificação Digital Nº 1312421/CA

Compressão e *Descompressão* podem ter as funções $f^{atualização}$ *C1*, *C2* e *C3*, e *D1*, *D2* e *D3*, respectivamente. Na Figura 23, após a reconexão do nó, ambas as reconfigurações são concluídas e as versões *C1*, *C2*, *D1* e *D2* são removidas do sistema.

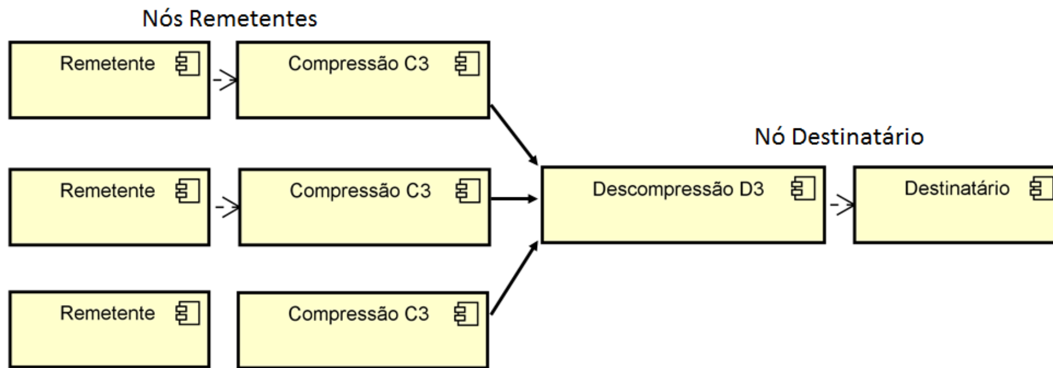

Figura 23. Sistema após reconexão do nó e conclusão das duas reconfigurações

4.4 Implementação

A implementação protótipo da abordagem proposta por esta tese foi nomeada de *D-Joseph* (*Dynamic Joseph*). Como prova de conceito, D-Joseph (i.e., o *middleware* que implementa a abordagem proposta) foi desenvolvido usando a linguagem de programação Java para as plataformas Java e Android. Apesar do Android usar a linguagem de programa Java, ele executa sobre a máquina virtual *Android runtime* (ART), enquanto a plataforma Java usa a máquina virtual *HotSpot*. Outro ponto importante é que o *bytecode* gerado para aplicações Java e Android são incompatíveis, onde no Java são gerados arquivos *.class* e no Android arquivos *.dex*. Além disto, o Android não provê toda a API (*Application Program Interface*) Java e tem algumas APIs específicas, como por exemplo para manipulação do *classloader*.

Alguns recursos de reconfiguração foram implementados usando reflexão computacional Java/Android para permitir a adição de JARs (*Java Archives*) no *classloader* em tempo de execução (i.e., dinamicamente) e para iniciar os componentes. Em cada nó, seja ele um nó cliente ou de processamento, é executado um serviço de reconfiguração do *middleware* D-Joseph que é responsável por executar localmente a reconfiguração. O Gerente de Reconfiguração envia os comandos de reconfiguração (e.g., atualização de um componente) para este serviço, o qual é responsável por realizar de fato a

PUC-Rio - Certificação Digital Nº 1312421/CA

reconfiguração no nó e informar ao Gerente de Reconfiguração se a reconfiguração foi realizada com sucesso ou não.

Para sua comunicação, o *middleware* D-Joseph utiliza o SDDL *(Scalable Data Distribution Layer*), o qual é um *middleware* para comunicação escalável com baixa latência [96] [97]. O SDDL conecta nós móveis (e.g., *smartphones* ou *tablets*) com alguma conexão sem fio (e.g., 3G/4G ou Wi-Fi) a nós servidores estacionários em uma nuvem, denominada de *SDDL Core*, ou núcleo do SDDL. Ele utiliza dois protocolos de comunicação, o *Data Distribution Service* (DDS) *for Real Time Publish/Subscribe* para a comunicação cabeada dentro do núcleo SDDL, e o *Mobile Reliable UDP* (MR-UDP) para troca de mensagens entre os nós móveis e o núcleo SDDL [98]–[101]. O DDS é um padrão que especifica uma infraestrutura de comunicação *Publish/Subscribe* totalmente descentralizada com intuito de prover alto desempenho em sistemas distribuídos. O MR-UDP, por sua vez, é um UDP (*User Datagram Protocol*) confiável melhorado por mecanismos para tolerar conectividade intermitente, mudança de endereço IP (*Internet Protocol*) dos nós móveis. Entre outras funcionalidades do SDDL, ele informa sempre que um nó cliente se conecta ou desconecta e tem um serviço de persistência de mensagens. Este serviço salva as mensagens que não foram entregues ao nó por conta de uma intermitência na conexão. Sendo assim, ele é útil quando durante uma reconfiguração, um CN fica temporariamente indisponível e não pode receber as mensagens de reconfiguração. Ao restabelecer sua conexão, o SDDL automaticamente encaminha as mensagens que ficaram pendentes e o CN pode concluir sua reconfiguração.

D-Joseph foi implementado como um serviço do *middleware* ContextNe*t*. Assim sendo, ele pode ser utilizado em outras aplicações que necessitem evoluir em tempo de execução. D-Joseph, além de implementar a abordagem proposta por esta tese, fornece outras funcionalidades para reconfiguração dinâmica de software que são discutidas nos trabalhos [36] [54] [102].

PUC-Rio - Certificação Digital Nº 1312421/CA

5 Avaliação de desempenho

Foram realizados dois experimentos na avaliação desta tese. No primeiro experimento, foi avaliado o desempenho de D-Joseph em termo de vazão, latência, tempo de atualização e sobrecarga. Já no segundo experimento, foi realizada uma comparação entre D-Joseph, a abordagem quiescente proposta por Kramer e Magee [15], e Upstart [44] [85], o qual é uma abordagem não quiescente proposta por Ajmani, Liskov e Shrira. O trabalho de Kramer e Magee [15] com o algoritmo de quiescência foi escolhido por ser um trabalho seminal da área, não só relativo a abordagens quiescentes, como em reconfiguração dinâmica no geral. Já o Upstart foi escolhido por sua relevância científica com mais de 115 citações, e por ser semelhante ao modelo implementado por D-Joseph. O código fonte de D-Joseph, bem como o da avaliação, está disponível em http://www.inf.puc-rio.br/~rvasconcelos/tese/codigo.zip.

Os testes foram realizados com uso de 10 computadores rodando Windows 7 64 bits e interconectados através de um switch gigabit, além de um smartphone LG G4 rodando Android 6.0 conectado através de uma rede Wi-Fi 802.11n. Cada computador era equipado com Intel i5, 4GB DDR3 e placa de rede gigabit Ethernet. Três computadores e um smartphone foram utilizados para emular os CNs, e outros quatro computadores foram utilizados para executar os PNs, como mostrado na Figura 24. Além disto, foram utilizados dois computadores para executar os Gateways e um para executar o Gerente de Reconfiguração. Cada teste foi emulado cinco vezes e o nível de confiança de todos os resultados é de 95%.

PUC-Rio - Certificação Digital Nº 1312421/CA

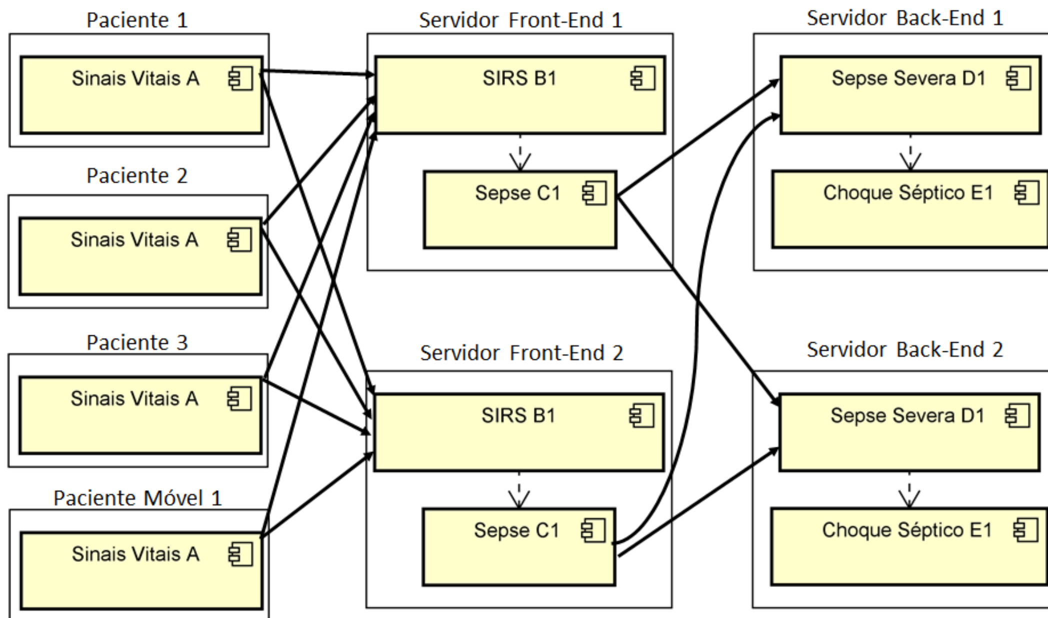

Figura 24. Implantação física utilizada nos testes. O Gerente de Reconfiguração e o Gateway foram omitidos por questões de legibilidade

PUC-Rio - Certificação Digital Nº 1312421/CA

5.1 Cenário de teste

Assim como no cenário motivador do monitoramento de pacientes apresentado na seção 1.2, o cenário de teste consiste de um hospital que monitora pacientes. Cada paciente tem um equipamento móvel dotado de sensores que monitoram continuamente os sinais vitais (e.g., temperatura, pressão arterial, e frequência cardíaca e respiratória) do paciente. O equipamento móvel envia os sinais vitais do paciente para os servidores do hospital a cada um segundo.

Neste cenário, o sistema da Figura 25 define a severidade da sepse e do choque séptico usando os critérios propostos por Bone *et al.* [103] e por Balk e Medlej [104]. O componente *Sinais Vitais* gera os sinais vitais – i.e., temperatura corporal, frequência cardíaca, frequência respiratória, pressão arterial e contagem de células brancas – que são processados pelo sistema do hospital. O restante dos componentes (*SIRS – Systemic Inflammatory Response Syndrome*, *Sepse*, *Sepse Severa* e *Choque Séptico*) verifica se os sinais vitais do paciente atendem aos critérios que indicam que o paciente está sofrendo um choque séptico.

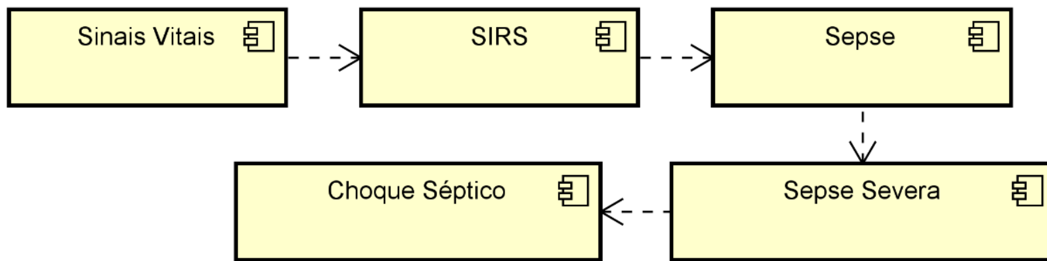

Figura 25. Abordagem para identificar choque séptico

Apesar da sua utilização no passado, desde 23 de Fevereiro de 2016, o método de detecção representado pela Figura 25 não é mais recomendado para o diagnóstico de choque séptico [104]. A nova abordagem recomendada para identificar choque séptico, mostrada na Figura 26, usa outros critérios, o qSOFA (*quick Sequential [Sepsis-related] Organ Failure Assessment*) e o SOFA (*Sequential [Sepsis-related] Organ Failure Assessment*) para determinar se o paciente está com choque séptico [105]. Portanto, devido a essa mudança de critério, a maior parte do sistema de monitoramento de pacientes precisa ser reconfigurada para usar a nova abordagem recomendada para identificar esta grave condição médica.

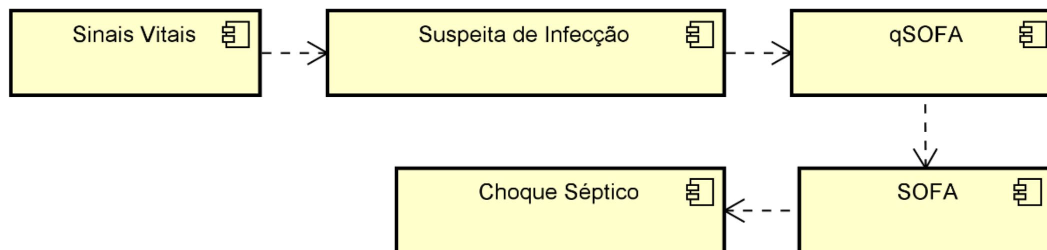

Figura 26. Nova abordagem recomendada para identificar choque séptico

Como o hospital tem uma quantidade variável de pacientes e de servidores que processam os dados provenientes dos pacientes, cada componente da Figura 25 e da Figura 26 terá tipicamente várias instâncias implantadas em diferentes nós servidores, assim como mostrado na Figura 24. Entretanto, o único componente executado no nó móvel é o *Sinais Vitais*, sendo os demais executados em servidores, como ilustrado na Figura 24. Como discutido anteriormente, para substituir um componente por outro, a plataforma de reconfiguração deve garantir a consistência do sistema. A Figura 24 mostra os tipos de componentes implantados em cada nó e que cada tipo de componente tem várias instâncias distribuídas pelos nós da rede. Cada etapa no processamento do fluxo dos dados tem um número arbitrário de servidores que dividem a carga de trabalho, Além disso, por questões de balanceamento de carga, o dado enviado por um paciente

PUC-Rio - Certificação Digital Nº 1312421/CA

pode ser redirecionado para qualquer servidor *front-end*, assim como o dado enviado por um servidor *front-end* pode ser redirecionado para qualquer servidor *back-end*, como encontrado em *middlewares* para processamento de fluxo de dados [3] [21] [106]. A nomenclatura *front-end* e *back-end* foi utilizada apenas indicar nós de processamento que estão localizados no início e no fim do processamento do fluxo de dados.

No momento em que o sistema precisar ser reconfigurado para os novos critérios de detecção de choque séptico, os componentes *SIRS*, *Sepse* e *Sepse Severa* serão substituídos pelos componentes *Suspeita de Infecção*, *qSOFA* e *SOFA*. Além disto, o componente *Choque Séptico* também deve ser substituído por sua nova versão, uma vez que o seu protocolo para identificar um choque séptico também foi alterado nos novos critérios [105]. Esta reconfiguração deve ser feita de tal modo que os sinais vitais de um paciente sejam integralmente processados (i.e., percorram todas as etapas da análise) ou pelo critério antigo ou pelo critério novo, mas nunca apenas parcialmente por uma delas, garantindo assim a consistência do monitoramento dos pacientes.

Aplicação protótipo implementada possui duas versões, a primeira versão é ilustrada pela Figura 25 a segunda versão pela Figura 26. A versão implantada originalmente foi a primeira, como ilustrado na Figura 24, a qual deveria ser posteriormente substituída em tempo de execução pela segunda versão. O número de CNs na avaliação levou em consideração a capacidade de hospitais reais. Dentre os maiores hospitais do mundo estão o *Chris Hani Baragwanath Hospital* com capacidade para 3.200 leitos e o *Clinical Centre of Serbia* com capacidade para 3.500 leitos [34] [107]. O JAR de cada componente implantado tem aproximadamente 4KB (*kilobytes*).

5.2 Métricas

Alinhado com a avaliação realizada em trabalhos como [6], [7], [19], [26], [44], [85], [87], [108], [109], as métricas utilizadas na avaliação desta tese foram tempo de atualização (i.e., *update time*), vazão, latência, e o tempo de processamento de todas as tuplas. O tempo de atualização, mensurado em milissegundos (ms), é medido a partir do instante de tempo que o Gerente de Reconfiguração inicia a reconfiguração até o momento que ele recebe a

PUC-Rio - Certificação Digital Nº 1312421/CA

confirmação que todos os nós finalizaram suas reconfigurações. A vazão (i.e., *throughput*) do sistema é medida em tuplas por segundo (tuplas/s). A latência, avaliada é milissegundos, é medida a partir do momento em que uma tupla é gerada até o momento que ela é processada pelo último componente do fluxo de dados. Por fim, o tempo de processamento informa o tempo necessário para que o sistema produza e processe uma determinada quantidade de tuplas.

5.3
Avaliação de D-Joseph

Na avaliação de desempenho de D-Joseph, variou-se o número de CNs de 3 até 300 e a taxa de produção de tuplas de 150 tuplas/s até 15.000 tuplas/s, usando tanto o gerenciamento estático de dependência quanto o gerenciamento dinâmico. Para tanto, foi utilizada a aplicação ilustrada na Figura 25.

Na aplicação originalmente implantada nesta avaliação de desempenho, cada componente processa os sinais vitais de um paciente e os encaminham para o próximo componente do fluxo. Assim, mesmo que os sinais vitais de um paciente não satisfaçam os critérios do componente SIRS e não possam satisfazer mais nenhum outro critério, os sinais vitais são desnecessariamente repassados para o próximo componente, e assim sucessivamente. Deste modo, a primeira reconfiguração realizada foi uma otimização para melhorar o desempenho do sistema. Para tal, assim que uma tupla (i.e., sinais vitais) de um paciente não satisfaça os critérios de um componente, a tupla é descartada (i.e., não é repassada para o próximo componente). Esta alteração no comportamento do sistema diminui a carga do sistema, já que muitas tuplas são descartadas ao longo do fluxo de processamento. A segunda reconfiguração realizada foi a mudança da unidade de temperatura utilizada em todo o sistema de Fahrenheit para Celsius. Apesar destas reconfigurações talvez não serem realizadas na prática, elas foram utilizadas apenas com intuito de avaliar o desempenho de D-Joseph.

Tabela 1. Parâmetros dos cenários de avaliação

# CNs	Taxa de Produção de Tuplas (tuplas/s)	Gerenciamento de Dependência
3	150, 1.500 e 15.000	Estático e Dinâmico
30	150, 1,500 e 15.000	Estático e Dinâmico
300	150, 1.500 e 15.000	Estático e Dinâmico

PUC-Rio - Certificação Digital Nº 1312421/CA

A Tabela 1 mostra os parâmetros de configuração que foram aplicados para avaliar D-Joseph. A taxa de produção de tuplas informa a taxa de produção de todo o sistema, não a taxa de produção de cada CN. No cenário com 30 CNs e 150 tuplas/s, por exemplo, cada CN produz cinco tuplas por segundo (i.e., a taxa de produção de tuplas de cada CN é de 5 tuplas/s).

Em relação à consistência de D-Joseph, todas as reconfigurações foram realizadas preservando a consistência dos fluxos de dado. Isto significa que todas as tuplas foram *processadas precisamente uma vez* pela função $f^{atualização}$ correta. Portanto, D-Joseph foi capaz de manter a consistência global do sistema enquanto o sistema era reconfigurado.

Tabela 2. Tempo de atualização para cada cenário avaliado

# CNs	**Taxa de Produção de Tuplas (tuplas/s)**	**Gerenciamento Estático de Dependência**		**Gerenciamento Dinâmico de Dependência**	
		Tempo de Atualização (ms)	**Intervalo de Confiança (ms)**	**Tempo de Atualização (ms)**	**Intervalo de Confiança (ms)**
3	150	24,29	± 7,61	24,07	± 5,98
30	150	24,18	± 6,34	25,20	± 4,45
300	150	24,88	± 3,22	24,75	± 3,74
3	1.500	25,18	± 3,78	25,2	± 4,43
30	1.500	24,60	± 5,74	21,36	± 3,74
300	1.500	25,62	± 3,63	23,62	± 2,72
3	15.000	25,05	± 4,60	25,63	± 4,61
30	15.000	26,87	± 5,99	26,27	± 4,32
300	15.000	26,69	± 6,27	26,48	± 5,68

PUC-Rio - Certificação Digital Nº 1312421/CA

Analisando os dados da Tabela 2, é possível notar que o tempo de atualização de D-Joseph é consideravelmente estável. Esta característica se deve ao fato de D-Joseph não precisar esperar alcançar um estado seguro para proceder com a reconfiguração. O tempo de atualização, mostrado na Tabela 2, variou de 24,29ms no cenário com três CNs e taxa de produção de 150 tuplas/s para 26,69ms no cenário com 300 CNs e 15.000 tuplas/s, ambos usando o gerenciamento estático de dependência. Por outro lado, com o gerenciamento dinâmico, o tempo de atualização variou de 24,07ms para 26,48ms nos mesmos cenários. Analisando os dados da Tabela 2, é notório que o parâmetro que tem maior influência no tempo de atualização é a taxa de produção de tuplas. Este

fenômeno é explicado pois quanto maior for a carga de trabalho nos nós, maior será o tempo necessário para que um nó consiga concluir sua reconfiguração por conta do maior uso do processador.

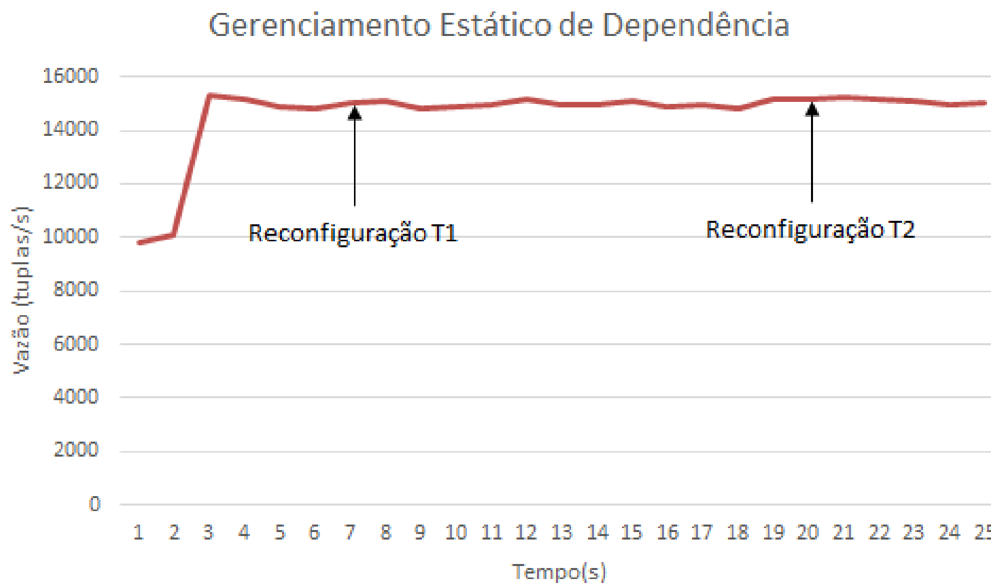

Figura 27. Vazão no cenário com 3.000 CNs, 15.000 tuplas/s e gerenciamento estático

O experimento realizado para avaliar como D-Joseph interfere na vazão do sistema indica que a perda de vazão é desprezível (vide Figura 27 e Figura 28), assim como mostrado na seção 5.4. Por conta disto, são mostrados apenas os gráficos referentes aos testes realizados no cenário com a maior quantidade de CNs e maior taxa de produção de tuplas (i.e., 300 CNs e 15.000 tuplas/s). A vazão usando o gerenciamento estático de dependência (Figura 27) teve um pequeno aumento no tempo de reconfiguração *T* (i.e., momento no qual a reconfiguração foi realizada) quando comparado com o tempo $T - 1$ (i.e., um segundo antes da reconfiguração acontecer), de 14.795 tuplas/s para 15.019 tuplas/s na reconfiguração *T1*, e de 14.869 tuplas/s para 14.924 tuplas/s na reconfiguração *T2*. Usando o gerenciamento dinâmico de dependência (Figura 28), a vazão variou de 15.060 tuplas/s para 15.030 tuplas/s na reconfiguração *T1* e de 15.073 tuplas/s para 15.043 tuplas/s na reconfiguração *T2*. Em ambos os gerenciamentos de dependência, a vazão não foi significantemente afetada pelas reconfigurações. Assim, os experimentos indicam que D-Joseph causa uma redução marginal (menor que 0,2%) na vazão do sistema no momento de uma reconfiguração.

PUC-Rio - Certificação Digital Nº 1312421/CA

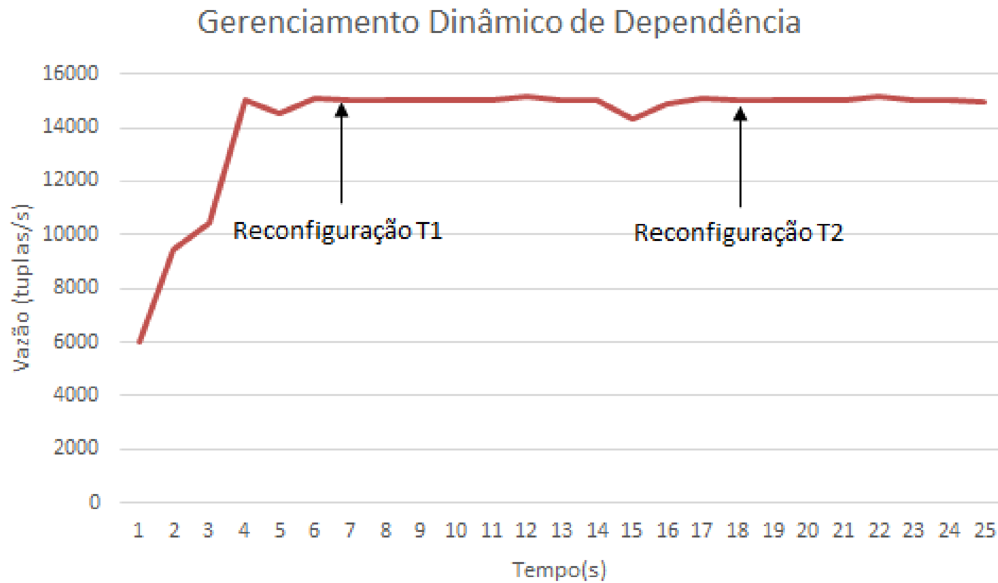

Figura 28. Vazão no cenário com 3.000 CNs, 15.000 tuplas/s e gerenciamento dinâmico

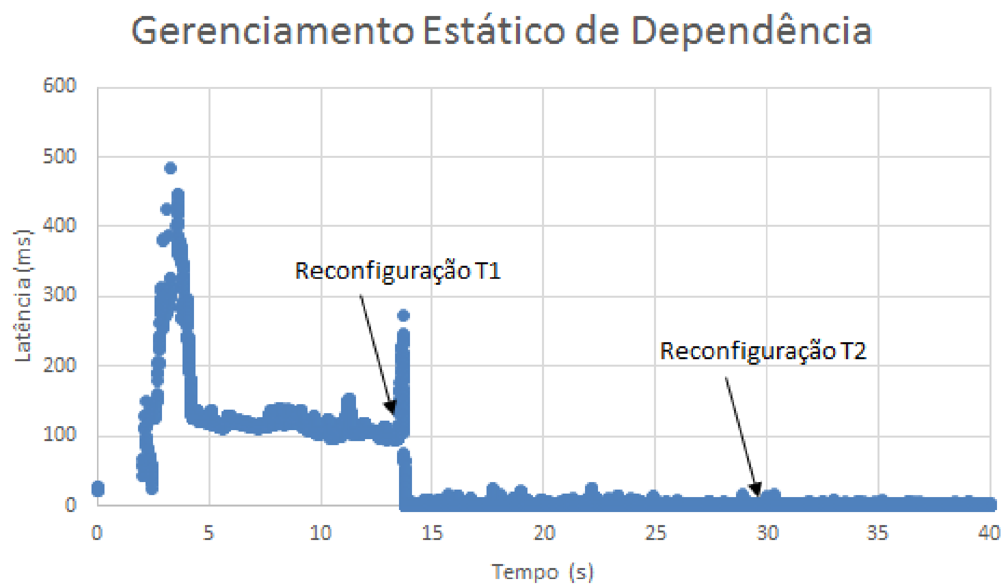

Figura 29. Latência no cenário com 300 CNs, 15.000 tuplas/s e gerenciamento estático de dependência

PUC-Rio - Certificação Digital Nº 1312421/CA

De acordo com os experimentos (vide Figura 29 e Figura 30), a reconfiguração pode afetar a latência quando o sistema tem uma carga de trabalho consideravelmente alta (i.e., os nós têm altas taxas de utilização do processador). Em ambos os gerenciamentos de dependência, a reconfiguração *T1* de *v1* para *v2*, a qual reduz a carga de trabalho do sistema por descartar assim que possível as tuplas que não satisfazem os critérios de um componente, interferiu na latência das tuplas por um curto espaço de tempo. Apesar disto, depois de otimizar o

sistema e assim reduzir sua carga de trabalho, a reconfiguração *T2* teve um pequeno impacto na latência (≈ 2ms) em ambos os gerenciamentos de dependência.

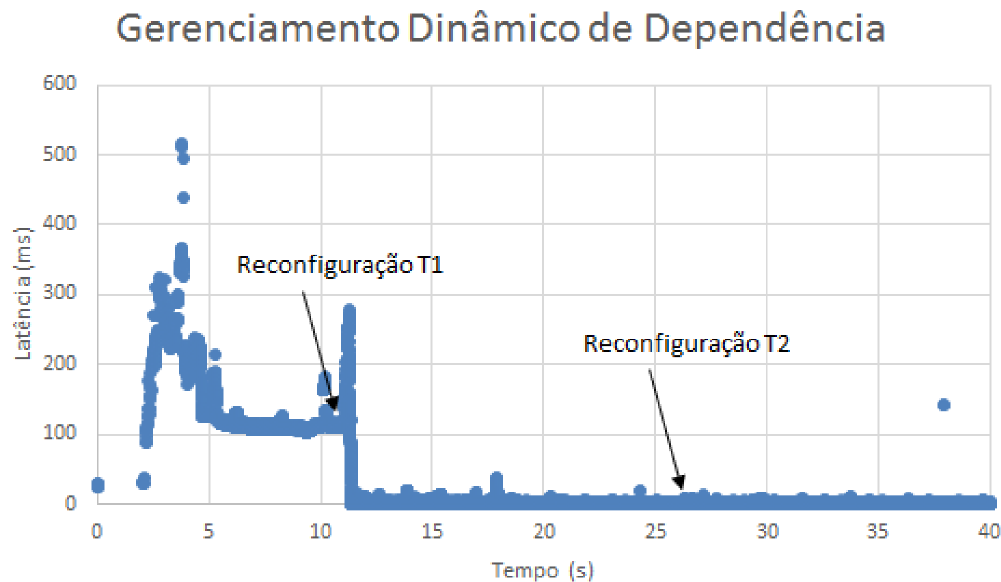

Figura 30. Latência no cenário com 300 CNs, 15.000 tuplas/s e gerenciamento dinâmico de dependência

Para avaliar a sobrecarga que D-Joseph impõe na aplicação protótipo, quando não são realizadas reconfigurações no sistema, foi avaliada a vazão, a latência e o tempo requerido pelo sistema para gerar e processar 330.000 tuplas. A primeira execução foi realizada na aplicação sem uso de D-Joseph, a segunda usando gerenciamento estático de dependência de D-Joseph, e por fim, a terceira execução foi realizada com o gerenciamento dinâmico de dependência de D-Joseph. Neste experimento foi utilizado apenas um computador para rodar todos componentes da aplicação já que o objetivo era avaliar o comportamento da aplicação com e sem o uso de D-Joseph em um cenário onde não havia a realização de reconfigurações dinâmicas.

Em relação ao tempo necessário para processar todas as tuplas (Figura 31), o gerenciamento estático de dependência resultou em um aumento de 3,83%, ao passo que o gerenciamento dinâmico resultou em um aumento de 8,98%. Já a vazão, mostrada na Figura 32, foi reduzida em 2,38% usando o gerenciamento estático e em 2,84% usando o gerenciamento dinâmico. A latência, como mostrado na Figura 33, sofreu um aumento de 6,57% e 12,50% com o gerenciamento estático e o gerenciamento dinâmico, respectivamente. Assim, para

PUC-Rio - Certificação Digital Nº 1312421/CA

a aplicação do cenário de teste, a melhor escolha em termos de desempenho é o gerenciamento estático de dependência. A seção 5.3.1 apresenta uma comparação entre o gerenciamento estático e o gerenciamento dinâmico que pode ajudar o desenvolvedor a escolher a opção mais adequada.

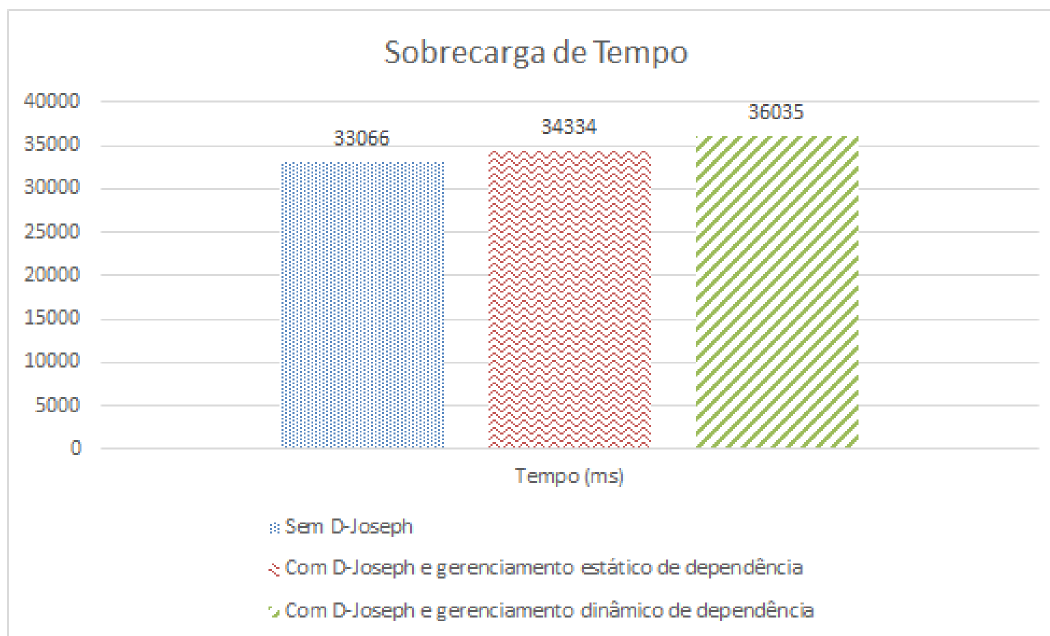

Figura 31. Sobrecarga em relação ao tempo de processamento imposta por D-Joseph

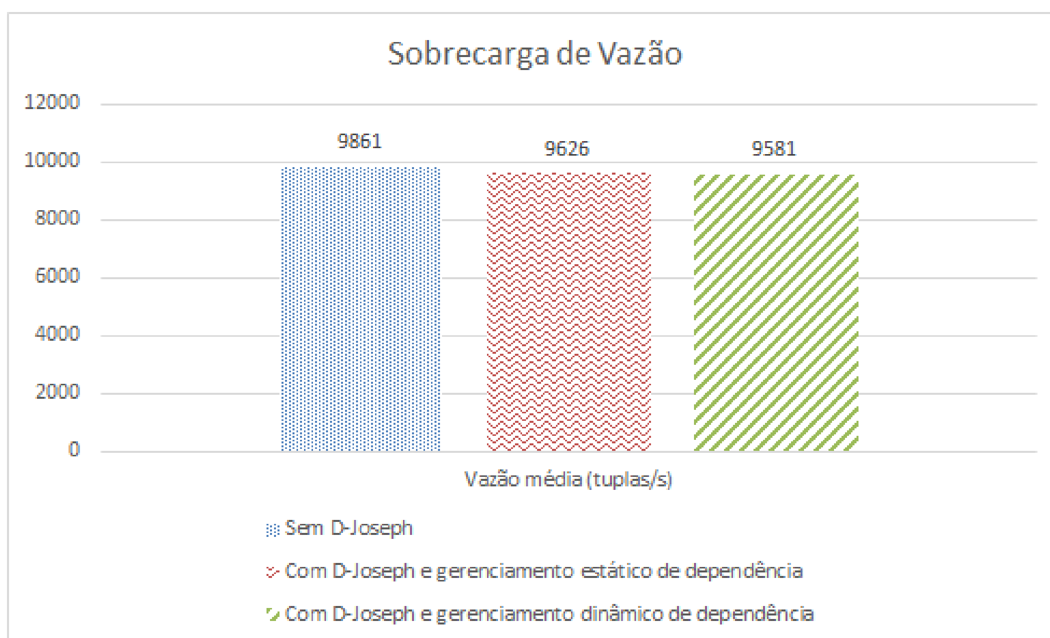

Figura 32. Sobrecarga em relação à vazão imposta por D-Joseph

PUC-Rio - Certificação Digital Nº 1312421/CA

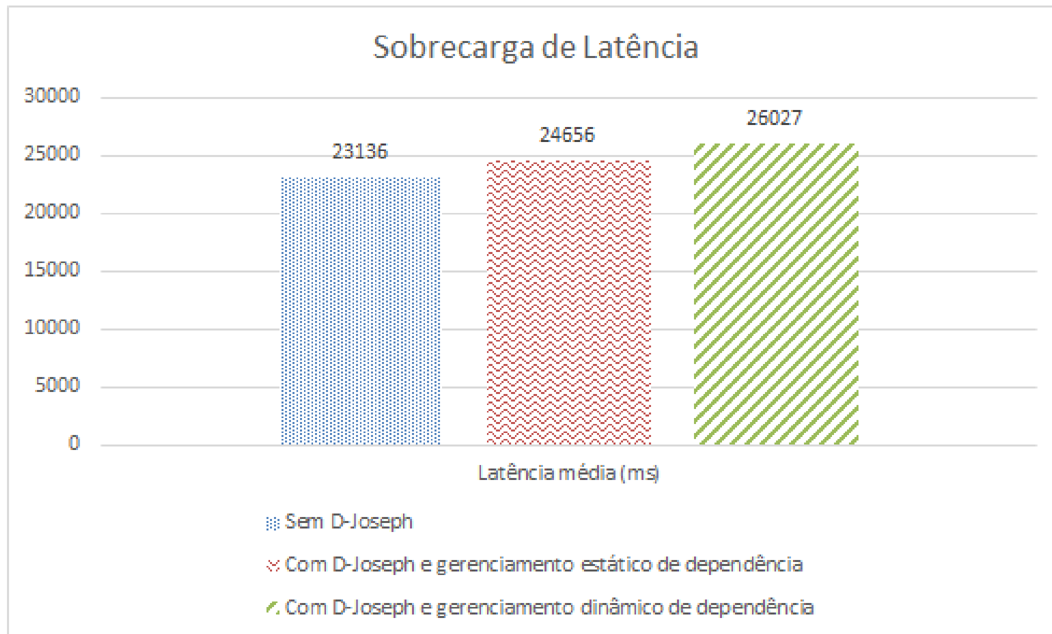

Figura 33. Sobrecarga em relação à latência imposta por D-Joseph

5.3.1 Comparação entre o gerenciamento estático e o gerenciamento dinâmico de dependência

Com o objetivo de verificar a relação entre o tamanho do fluxo de dados (i.e., a quantidade de componentes entre o vértice consumidor e o vértice produtor) e o desempenho de D-Joseph com o gerenciamento estático e o gerenciamento dinâmico, foi realizado um experimento onde o tamanho do fluxo foi variado artificialmente de 10 até 40 componentes, onde em todos os cenários foram geradas 30.000 tuplas com uma taxa de produção de 1.000 tuplas/s. Foram utilizados dois computadores, onde cada um tinha implantado metade dos componentes, sendo o primeiro computador apenas com os componentes de ordem ímpar e o segundo apenas com os componentes de ordem par. O intuito desta divisão na implantação era fazer com que o próximo componente no fluxo estivesse implantado em outro nó, forçando assim que a tupla sempre tivesse que ser transmitida pela rede.

Para simular as dependências entre os componentes, foi utilizado o seguinte critério. Um componente i tem dependência com os componentes $i+1$ e $i+2$ (i.e., com os próximos dois componentes no fluxo), como ilustrado na Figura 34. Deste modo, quando uma tupla T for entregue a um componente n, o caminho de execução usando o gerenciamento dinâmico de dependência terá informação apenas dos componentes n – 1 e $n-2$. Por outro lado, usando o gerenciamento

PUC-Rio - Certificação Digital Nº 1312421/CA

estático, o caminho de execução terá informação sobre *n – 1* componentes (i.e., o tamanho do caminho de execução será *n - 1*). Usando o exemplo da Figura 34, quando a tupla *T* for entregue ao *Componente 4*, o caminho de execução terá informação sobre *Componente 2* e *Componente 3* caso seja utilizado o gerenciamento dinâmico de dependência, caso contrário terá informação sobre todos os componentes anteriores ao *Componente 4*, ou seja, *Componente 1*, *Componente 2* e *Componente 3*.

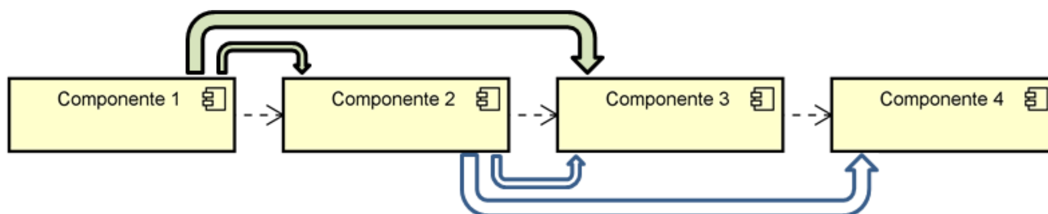

Figura 34. Dependências entre os componentes

Os resultados da Tabela 3 indicam que até 10 componentes no fluxo a diferença entre o gerenciamento estático e o gerenciamento dinâmico é desprezível. Com 20 componentes no fluxo de dados, a latência média das tuplas usando gerenciamento dinâmico foi 59,89% menor que a latência média com gerenciamento estático, a vazão foi 2,47% maior e o tempo para concluir o processamento de todas as tuplas (i.e., a duração do teste) foi 4,76% menor. No cenário com um fluxo de tamanho 30, a latência do gerenciamento dinâmico foi 36,23% menor, a duração do teste foi 17,02% menor e a vazão teve uma variação desprezível. No último teste, com um fluxo de tamanho 40, a vazão voltou a ter um ganho desprezível, ao passo que a latência e a duração do teste foram reduzidas em 34,18% e em 30,56%, respectivamente.

Tabela 3. Comparação entre o gerenciamento estático e o gerenciamento dinâmico de dependência

Tamanho do Fluxo	**Vazão Média (tuplas/s)**		**Latência Média (ms)**		**Tempo de Processamento (s)**	
	Estático	**Dinâmico**	**Estático**	**Dinâmico**	**Estático**	**Dinâmico**
10	999,79	999,69	65,30	65,77	30,01	30,01
20	975,50	999,55	2.043,68	820,43	31,52	30,02
30	696,74	702,32	13.079,38	8.341,08	52,34	43,43
40	577,00	580,96	24.603,55	16.194,28	74,89	52,00

Como esperado, os experimentos realizados indicam que a partir de 10 componentes, o gerenciamento dinâmico de dependência tem um desempenho melhor que o gerenciamento estático em todas as métricas avaliadas, com destaque para a latência. A partir destes resultados, o desenvolver da aplicação

PUC-Rio - Certificação Digital Nº 1312421/CA

tem um referencial para decidir qual gerenciamento de dependência é mais adequado para a sua aplicação.

5.4 Comparação entre D-Joseph e outras abordagens

Para comparar D-Joseph com outras abordagens e assim mostrar os benefícios da abordagem proposta por esta tese, a aplicação protótipo descrita na seção 5.1 foi implementada e avaliada usando D-Joseph, Upstart e quiescência. Para comparar D-Joseph com Upstart e quiescência [15], foi realizada uma implementação simplificada baseada nas informações fornecidas pelos autores. D-Joseph foi avaliado usando o gerenciamento estático de dependência. Nos testes, variou-se o número de CNs (i.e., de pacientes) de 334 até 3.000.

Tabela 4. Tempo de atualização dos trabalhos comparados

# CNs	Tempo de Atualização (ms)		
	Quiescência	**D-Joseph**	**Upstart**
334	583,46 ± 40,13	25,72 ± 5,98	76,89 ± 27,82
1.000	629,62 ± 43,28	26,92 ± 4,45	115,62 ± 21,41
3.000	737,28 ± 49,71	27,30 ± 3,74	183,47 ± 18,16

A Tabela 4 mostra que o tempo de atualização do sistema usando quiescência é consideravelmente maior que usando Upstart e D-Joseph. Enquanto o tempo de atualização com quiescência variou de 583ms até 737ms, com Upstart o tempo necessário para reconfigurar o sistema variou de 77ms até 183ms, enquanto com D-Joseph o tempo de atualização é consideravelmente estável, ficando entre 26ms e 27ms. Isto indica que o número de CNs tem pouco impacto no tempo de atualização do sistema usando D-Joseph.

Como esperado, a Figura 35 mostra que a quiescência causa um grande impacto na vazão do sistema quando é realizada uma reconfiguração, já que o sistema precisa ser bloqueado antes de ser reconfigurado. Mais especificamente, a quiescência causa reduções de 88,70% (de 1.000 tuplas/s para 113 tuplas/s) e 66,55% (de 2.831 tuplas/s para 947 tuplas/s) na vazão nos cenários com 1.000 e 3.000 CNs, respectivamente. Já Upstart, que é menos intrusivo que a quiescência, causou reduções na vazão de 27,20% (de 1.000 tuplas/s para 728 tuplas/s) e

PUC-Rio - Certificação Digital Nº 1312421/CA

19,73% (de 2.924 tuplas/s para 2.347 tuplas/s) nos mesmos cenários com 1.000 e 3.000 CNs, respectivamente. Por outro lado, por não precisar bloquear o fluxo para reconfigurar os componentes, D-Joseph causa uma pequena redução na vazão de 0,37% (de 2.997 tuplas/s para 2.986 tuplas/s) apenas no cenário com 3.000 CNs.

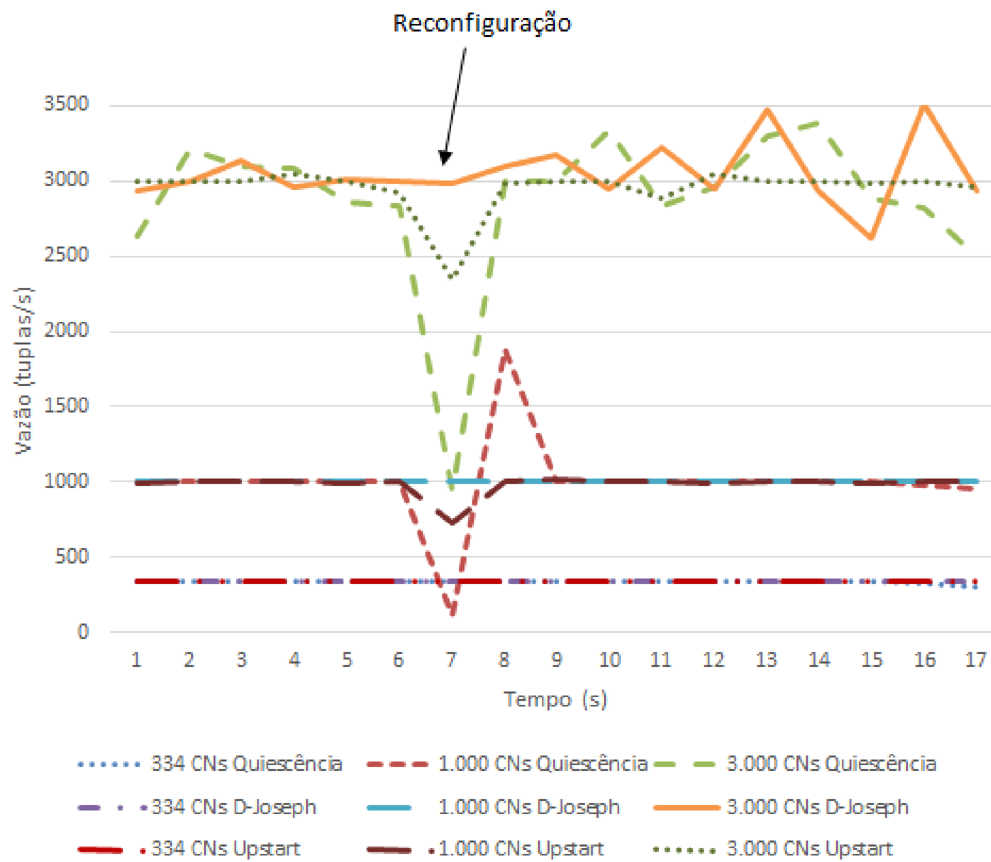

Figura 35. Vazão do sistema usando quiescência, D-Joseph e Upstart

PUC-Rio - Certificação Digital Nº 1312421/CA

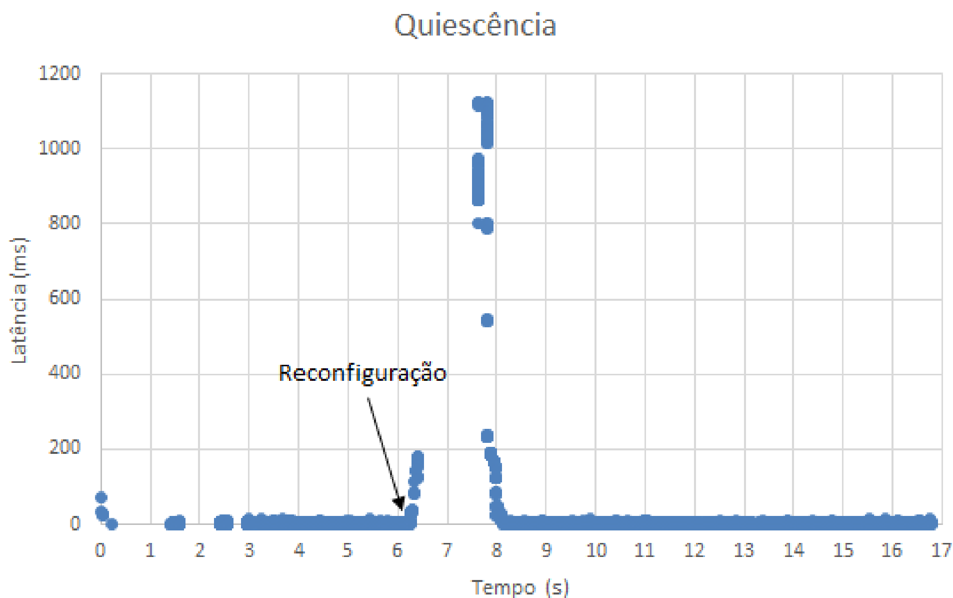

Figura 36. Latência no cenário com 3.000 CNs usando quiescência

Analisando a latência das tuplas usando os três trabalhos, a quiescência afeta a latência das tuplas por quase 2 segundos e requer até 1.126ms para processar uma tupla no cenário com 3.000 CNs (Figura 36). Já Upstart, mostrado na Figura 37, afeta a latência por aproximadamente um segundo e meio e requer até 331ms para processar uma tupla. Por outro lado, D-Joseph (Figura 38) afeta a latência por menos de meio segundo e requer no máximo 75ms para processar uma tupla no momento de uma reconfiguração, o que é 93,64% e 77,34% menor que a latência usando quiescência e Upstart, respectivamente.

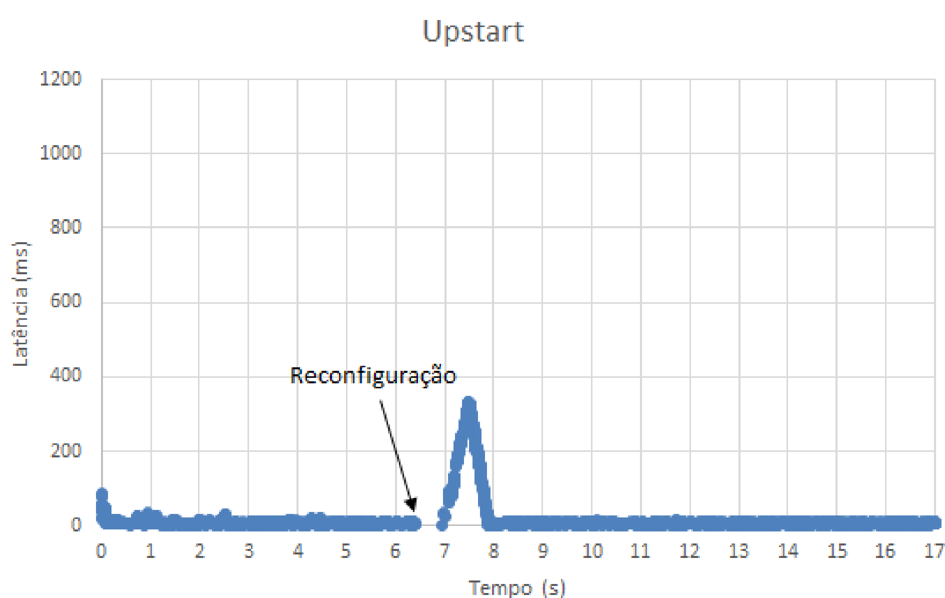

Figura 37. Latência no cenário com 3.000 CNs usando Upstart

PUC-Rio - Certificação Digital Nº 1312421/CA

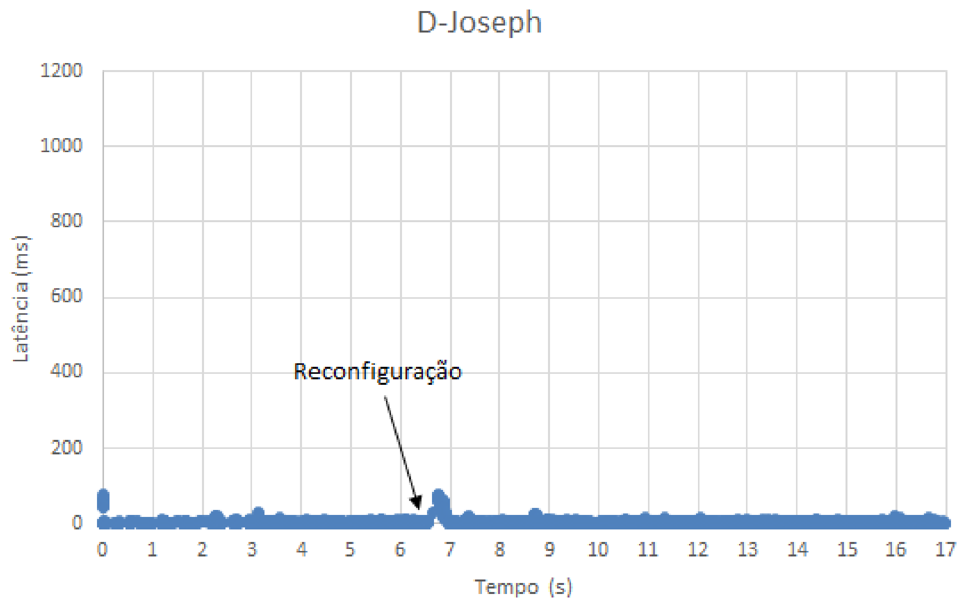

Figura 38. Latência no cenário com 3.000 CNs usando D-Joseph

PUC-Rio - Certificação Digital Nº 1312421/CA

6 Conclusão

Esta tese propõe e valida uma abordagem eficiente para reconfiguração dinâmica de software de sistemas de fluxo de dados distribuídos, que preserva a consistência global. Diferentemente de outros trabalhos que precisam ser movidos para um estado seguro ou serem “silenciados” temporariamente as componentes envolvidas com a reconfiguração, a abordagem proposta permite que o sistema evolua (i.e., seja reconfigurado) gradual e suavemente, sem precisar interromper o processamento do fluxo de dados. Além da preservação da consistência do fluxo de dados, a abordagem proposta suporta a substituição de componentes com estado (i.e., “stateful”) e é capaz de lidar com nós que podem se desconectar e reconectar a qualquer momento. Portanto, as principais contribuições desta tese são (i) um modelo para reconfiguração dinâmica que permite o sistema evoluir sem a necessidade que seus nós alcancem um estado seguro (e.g., quiescência), e (ii) um protótipo do modelo implementado em Java que permite reconfiguração dinâmica consistente e de modo não quiescente em sistemas de fluxo de dados.

A avaliação da abordagem proposta sugere que a abordagem proposta por esta tese tem potencial para ser utilizada no desenvolvimento de sistemas de processamento de fluxo de dados reconfiguráveis. Quanto a sobrecarga que D-Joseph impõe ao sistema na ausência de reconfigurações, o gerenciamento estático de dependência contribuiu com somente 4% do tempo para processar as tuplas, ao passo que o gerenciamento dinâmico implicou em um aumento de meros 9%. Já a vazão foi reduzida em 2% e em 3% usando o usando o gerenciamento estático e o gerenciamento dinâmico, respectivamente. A latência, por sua vez, sofreu um aumento de 7% e 13% usando o gerenciamento estático e o gerenciamento dinâmico, respectivamente. Por fim, a avaliação realizada indica que o gerenciamento dinâmico de dependência começa a ter desempenho melhor que a versão estática para sistemas que apresentem fluxos com mais de 10 componentes.

O experimento que compara a abordagem proposta com a abordagem quiescente e com Upstart mostra que a abordagem proposta tem um desempenho

PUC-Rio - Certificação Digital Nº 1312421/CA

melhor em todos os cenários e em relação a todas as métricas utilizadas. Para concluir uma reconfiguração em todo o sistema, a abordagem proposta necessitou de no máximo 27ms. Em relação ao tempo de atualização, a abordagem proposta é 96% mais rápida que a abordagem de Kramer e Magee usando quiescência e 85% mais rápido quando comparado com Upstart. Quando comparada a perda de vazão no momento de uma reconfiguração, a abordagem proposta causou uma redução de menos de 1%, ao passo que quiescência e Upstart apresentaram reduções na vazão de até 89% e 28%, respectivamente. Do mesmo modo, a latência das tuplas no momento de uma reconfiguração usando a abordagem proposta foi 94% e 77% menor quando comparado com a latência da quiescência e Upstart, respectivamente.

Apesar dos benefícios da abordagem proposta, problemas como variabilidade paramétrica e *reasoning* de reconfiguração, o qual é responsável por decidir quando uma reconfiguração é necessária e qual a melhor solução (i.e., reconfiguração) para o sistema, não foram tratados nesta tese. Outro aspecto importante relacionado a adaptação dinâmica é segurança, particularmente para sistemas onde os nós estão potencialmente expostos a comunicação através da internet. Portanto, autenticidade, integridade e confidencialidade emergem como aspectos chaves. Assim, garantir que apenas os administradores do sistema, ou o próprio sistema, tenham a capacidade de realizar reconfigurações evitará a implantação de componentes não autorizados, como por exemplo vírus, nos nós. Entretanto, aspectos de segurança estão além do escopo desta tese.

Dentre alguns trabalhos futuros estão o estudo de técnicas que permitam o desenvolvimento de sistemas autônomos e auto reconfiguráveis, estudo e a incorporação de mecanismos de segurança (e.g., *sandbox*), a exploração de políticas de reconfiguração (i.e., estratégias de como reconfigurar o sistema), suporte a nós legados que não podem mais ser atualizados por conta de alguma limitação, e a incorporação de mecanismos que permitam a transferência de estado automatizada.

6.1 Resultados da tese

Esta tese deu origem a um projeto de iniciação científica no curso de Ciência da Computação da Universidade Tiradentes (UNIT) cujo o título foi

PUC-Rio - Certificação Digital Nº 1312421/CA

"Estudo teórico e prático sobre aplicações dinamicamente adaptáveis". Este projeto de iniciação científica concedeu ao aluno Victor Brito Villar uma bolsa do Programa Institucional de Bolsas de Iniciação Cientifica (PROBIC) da UNIT[1].

Esta tese também deu origem às seguintes publicações:

- R. O. Vasconcelos, I. Vasconcelos, and M. Endler, "Management of Mobile Dynamic Adaptation in Cyber-Physical Systems", in *10th International Conference on Network and Service Management (CNSM 2014)*, 2014, p. 4.
- R. O. Vasconcelos, I. Vasconcelos, and M. Endler, "A Middleware for Managing Dynamic Software Adaptation", in *13th International Workshop on Adaptive and Reflective Middleware (ARM 2014), In conjunction with ACM/IFIP/USENIX ACM International Middleware Conference 2014*, 2014, p. 6.
- R. O. Vasconcelos, L. Talavera, I. Vasconcelos, M. Roriz, M. Endler, B. de T. P. Gomes, and F. J. da S. e Silva, "An Adaptive Middleware for Opportunistic Mobile Sensing", in *11th International Conference on Distributed Computing in Sensor Systems (DCOSS 2015)*, 2015.
- R. O. Vasconcelos, I. Vasconcelos, M. Endler, and S. Colcher, "Supporting Dynamic Reconfiguration in Distributed Data Stream Systems", in *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2016)*, 2016, p. 14.
- R. O. Vasconcelos, I. Vasconcelos, and M. Endler, "Dynamic and coordinated software reconfiguration in distributed data stream systems", *Journal of Internet Services and Applications (JISA 2016)*, vol. 7, no. 1, p. 21, 2016.
- R. O. Vasconcelos, I. Vasconcelos, and M. Endler, "D-Joseph: An Efficient Approach for Dynamic Software Reconfiguration in Data Stream Processing Systems," in *The Thirteenth International Conference on Autonomic and Autonomous Systems (ICAS 2017)*, 2017.

1https://eventos.unit.br/sempesq/wp-content/uploads/sites/33/2016/10/03_Semina%CC%81rio-de-Iniciac%CC%A7a%CC%83o-Cienti%CC%81fica-PROBIC-Unit_2016.pdf

PUC-Rio - Certificação Digital Nº 1312421/CA

7
Referências bibliográficas

[1] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, Jun. 2012.

[2] M. Garofalakis, J. Gehrke, and R. Rastogi, *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[3] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic Scaling for Data Stream Processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, 2014.

[4] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," *ACM SIGPLAN Not.*, vol. 48, no. 4, p. 279, Apr. 2013.

[5] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, "Mutable Checkpoint-restart: Automating Live Update for Generic Server Programs," in *Proceedings of the 15th International Middleware Conference*, 2014, pp. 133–144.

[6] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent dynamic reconfiguration of component-based distributed systems," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - ESEC/FSE '11*, 2011, p. 245.

[7] S. Ertel and P. Felber, "A framework for the dynamic evolution of highly-available dataflow programs," in *Proceedings of the 15th International Middleware Conference on - Middleware '14*, 2014, pp. 157–168.

[8] S. Andova, L. P. J. Groenewegen, and E. P. de Vink, "Dynamic adaptation with distributed control in Paradigm," *Sci. Comput. Program.*, vol. 94, pp. 333–361, Nov. 2014.

[9] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State Transfer for Clear and Efficient Runtime Updates," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, 2011, pp. 179–184.

[10] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, General-purpose Dynamic Software Updating for C," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 249–264.

[11] F. Kon, "Automatic Configuration of Component-Based Distributed Systems," PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[12] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *IEEE*

PUC-Rio - Certificação Digital Nº 1312421/CA

Comput., vol. 42, no. 10, pp. 22–27, Oct. 2009.
[13] K. Kakousis, N. Paspallis, and G. A. Papadopoulos, “A survey of software adaptation in mobile and ubiquitous computing,” *Enterp. Inf. Syst.*, vol. 4, no. 4, pp. 355–389, Nov. 2010.
[14] W. Li, “QoS Assurance for Dynamic Reconfiguration of Component-Based Software Systems,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 3, pp. 658–676, May 2012.
[15] J. Kramer and J. Magee, “The evolving philosophers problem: dynamic change management,” *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, 1990.
[16] C. Escoffier, P. Bourret, and P. Lalanda, “Describing Dynamism in Service Dependencies: Industrial Experience and Feedbacks,” in *Proceedings of the 2013 IEEE International Conference on Services Computing*, 2013, pp. 328–335.
[17] C. Costa-Soria, “Dynamic Evolution and Reconfiguration of Software Architectures through Aspects,” PhD Thesis, Department of Information Systems and Computation, University of Politecnica de Valencia, 2011.
[18] M. Ghafari, P. Jamshidi, S. Shahbazi, and H. Haghighi, “An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems,” in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering - CBSE ’12*, 2012, p. 177.
[19] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum, “Automating Live Update for Generic Server Programs,” *IEEE Trans. Softw. Eng.*, vol. PP, no. 99, p. 21, 2016.
[20] J. Zhang and B. H. C. Cheng, “Model-based Development of Dynamically Adaptive Software,” in *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06)*, 2006, pp. 371–380.
[21] R. O. Vasconcelos, M. Endler, B. de T. P. Gomes, and F. J. da S. e Silva, “Design and Evaluation of an Autonomous Load Balancing System for Mobile Data Stream Processing Based on a Data Centric Publish Subscribe Approach,” *Int. J. Adapt. Resilient Auton. Syst.*, vol. 5, no. 2, p. 19, 2014.
[22] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, Dec. 2005.
[23] R. O. Vasconcelos, I. Vasconcelos, and M. Endler, “Dynamic and coordinated software reconfiguration in distributed data stream systems,” *J. Internet Serv. Appl. (JISA 2016)*, vol. 7, no. 1, p. 21, 2016.
[24] R. O. Vasconcelos, M. Endler, B. Gomes, and F. Silva, “Autonomous Load Balancing of Data Stream Processing and Mobile Communications in Scalable Data Distribution Systems,” *Int. J. Adv. Intell. Syst.*, vol. 6, no. 3&4, pp. 300–317, 2013.
[25] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, “Software Dependencies, Work Dependencies, and Their Impact on Failures,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 864–878, Nov. 2009.
[26] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D’Hondt,

PUC-Rio - Certificação Digital Nº 1312421/CA

"Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 856–868, Dec. 2007.

[27] L. Gatzoulis and I. Iakovidis, "Wearable and Portable eHealth Systems," *IEEE Eng. Med. Biol. Mag.*, vol. 26, no. 5, pp. 51–56, Sep. 2007.

[28] A. Pantelopoulos and N. G. Bourbakis, "A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis," *IEEE Trans. Syst. Man, Cybern. Part C (Applications Rev.*, vol. 40, no. 1, pp. 1–12, Jan. 2010.

[29] A. Lmberis and A. Dittmar, "Advanced Wearable Health Systems and Applications - Research and Development Efforts in the European Union," *IEEE Eng. Med. Biol. Mag.*, vol. 26, no. 3, pp. 29–33, May 2007.

[30] R. Paradiso, G. Loriga, and N. Taccini, "A Wearable Health Care System Based on Knitted Integrated Sensors," *IEEE Trans. Inf. Technol. Biomed.*, vol. 9, no. 3, pp. 337–344, Sep. 2005.

[31] R. Paradiso, A. Alonso, D. Cianflone, A. Milsis, T. Vavouras, and C. Malliopoulos, "Remote health monitoring with wearable non-invasive mobile system: The Healthwear project," in *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2008, pp. 1699–1702.

[32] R. G. Haahr *et al.*, "An Electronic Patch for Wearable Health Monitoring by Reflectance Pulse Oximetry," *IEEE Trans. Biomed. Circuits Syst.*, vol. 6, no. 1, pp. 45–53, Feb. 2012.

[33] E. Sardini and M. Serpelloni, "Instrumented wearable belt for wireless health monitoring," *Procedia Eng.*, vol. 5, pp. 580–583, 2010.

[34] Saturn, "Clinical Centre of Serbia – (CCS)," 2016. [Online]. Available: http://www.saturn-project.eu/about-saturn/consortium-partners/clinical-centre-of-serbia-ccs/. [Accessed: 15-Jul-2016].

[35] E. Jovanov *et al.*, "A WBAN System for Ambulatory Monitoring of Physical Activity and Health Status: Applications and Challenges," in *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*, 2005, pp. 3810–3813.

[36] R. O. Vasconcelos *et al.*, "An Adaptive Middleware for Opportunistic Mobile Sensing," in *11th International Conference on Distributed Computing in Sensor Systems (DCOSS 2015)*, 2015.

[37] S. I. Lee *et al.*, "Remote patient monitoring: what impact can data analytics have on cost?," in *Proceedings of the 4th Conference on Wireless Health - WH '13*, 2013, pp. 1–8.

[38] Forbes, "4 Interesting Tech Trends In Patient Monitoring," 2014. [Online]. Available: http://www.forbes.com/sites/robertszczerba/2014/12/10/4-interesting-tech-trends-in-patient-monitoring. [Accessed: 14-Jul-2016].

[39] H. Catalyst, "The Year of Healthcare Data Analytics," 2014. [Online]. Available: https://www.healthcatalyst.com/2014-Year-Healthcare-Data-Analytics. [Accessed: 14-Jul-2016].

[40] Chalmers University of Technology, "Research and collaboration,"

PUC-Rio - Certificação Digital Nº 1312421/CA

2016. [Online]. Available: https://www.chalmers.se/en/areas-of-advance/ict/research/digital-sustainability/Pages/research.aspx. [Accessed: 07-Mar-2016].

[41] S. Cook, R. Harrison, M. M. Lehman, and P. Wernick, “Evolution in Software Systems: Foundations of the SPE Classification Scheme: Research Articles,” *J. Softw. Maint. Evol.*, vol. 18, no. 1, pp. 1–35, 2006.

[42] M. M. Lehman and J. F. RAMIL, “Is Evolution Intrinsic to All Real World Software? An Answer and Some Implications.” The “Software Ev olution and Evolutionary Computation Symposium” (EPSRC Network on Evolvability in Biology and Software Systems), Hatfield, U.K, 2002.

[43] Q. Wang, J. Shen, X. Wang, and H. Mei, “A component-based approach to online software evolution: Research Articles,” *J. Softw. Maint. Evol. Res. Pract.*, vol. 18, no. 3, pp. 181–205, 2006.

[44] S. Ajmani, B. Liskov, and L. Shrira, “Modular Software Upgrades for Distributed Systems,” in *ECOOP 2006 – Object-Oriented Programming*, D. Thomas, Ed. Springer Berlin Heidelberg, 2006, pp. 452–476.

[45] K. H. Bennett and V. T. Rajlich, “Software maintenance and evolution: a roadmap,” in *Proceedings of the conference on The future of Software engineering - ICSE ’00*, 2000, pp. 73–87.

[46] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, “Composing adaptive software,” *Computer (Long. Beach. Calif).*, vol. 37, no. 7, pp. 56–64, Jul. 2004.

[47] S. Corrêa and R. Cerqueira, “Computação autônoma: Conceitos, infra-estruturas e soluções em sistemas distribuídos,” in *Anais do 27o. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC’09)*, 2009, pp. 151–198.

[48] B. de T. P. Gomes, “AGST (Autonomic Grid Simulation Tool): uma ferramenta para modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores,” Dissertaçao de Mestrado, Programa de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão, 2012.

[49] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Pearson, 2017.

[50] R. H. Soares, “Gerenciamento Dinâmico de Modelos de Contexto: Estudo de Caso Baseado em CEP,” Dissertaçao de Mestrado, Instituto de Informática, Universidade Federal de Goiás, Orientador: Ricardo Couto Antunes da Rocha, 2012.

[51] L. Chen, “A Grid-Based Middleware for Processing Distributed Data Streams,” Ph.D. Thesis. Ohio State University, Columbus, OH, USA. Advisor(s) Gagan Agrawal, 2006.

[52] Q. Zhu, L. Chen, and G. Agrawal, “Supporting fault-tolerance in streaming grid applications,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–12.

[53] A. Deshpande, J. M. Hellerstein, and V. Raman, “Adaptive Query Processing: Why, How, When, What Next,” in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD ’06*, 2006, p. 806.

PUC-Rio - Certificação Digital Nº 1312421/CA

[54] R. O. Vasconcelos, I. Vasconcelos, and M. Endler, "A Middleware for Managing Dynamic Software Adaptation," in *13th International Workshop on Adaptive and Reflective Middleware (ARM 2014), In conjunction with ACM/IFIP/USENIX ACM International Middleware Conference 2014*, 2014, p. 6.

[55] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *J. Softw. Maint. Evol. Res. Pract.*, vol. 17, no. 5, pp. 309–332, Sep. 2005.

[56] R. Sterritt and D. Bustard, "Autonomic Computing-a means of achieving dependability?," *Eng. Comput. Syst. 2003. Proceedings. 10th IEEE Int. Conf. Work.*, pp. 247–251, 2003.

[57] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer (Long. Beach. Calif).*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[58] M. C. Huebscher and J. a. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, Aug. 2008.

[59] Y. Brun *et al.*, "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems*, vol. 5525, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg, 2009, pp. 48–70.

[60] M. Parashar and S. Hariri, *Autonomic Computing: Concepts, Infrastructure, and Applications*. CRC Press, 2006.

[61] M. Ghafari, P. Jamshidi, S. Shahbazi, and H. Haghighi, "Safe Stopping of Running Component-Based Distributed Systems: Challenges and Research Gaps," in *Proceedings of the 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2012, pp. 66–71.

[62] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lü, "Low-disruptive dynamic updating of Java applications," *Inf. Softw. Technol.*, vol. 56, no. 9, pp. 1086–1098, Sep. 2014.

[63] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster, "A study of dynamic software update quiescence for multithreaded programs," in *2012 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp)*, 2012, pp. 6–10.

[64] M. Hicks and S. Nettles, "Dynamic Software Updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, 2005.

[65] L. Pina, L. Veiga, and M. Hicks, "Rubah: DSU for Java on a stock JVM," *ACM SIGPLAN Not.*, vol. 49, no. 10, pp. 103–119, Oct. 2014.

[66] K. Makris and K. D. Ryu, "Dynamic and Adaptive Updates of Non-quiescent Subsystems in Commodity Operating System Kernels," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 327–340.

[67] C. Giuffrida and A. S. Tanenbaum, "Cooperative update: a new model for dependable live update," in *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp '09*, 2009, p. 6.

[68] D. Gupta and P. Jalote, "On-line software version change using state transfer between processes," *Softw. Pract. Exp.*, vol. 23, no. 9, pp. 949–964, Sep. 1993.

PUC-Rio - Certificação Digital Nº 1312421/CA

[69] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum, “Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer,” in *Proceedings of the 27th International Conference on Large Installation System Administration*, 2013, pp. 89–104.

[70] A. Baumann *et al.*, “Providing Dynamic Update in an Operating System,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, p. 32.

[71] T. Bloom and M. Day, “Reconfiguration and module replacement in Argus: theory and practice,” *Softw. Eng. J.*, vol. 8, no. 2, pp. 102–108, 1993.

[72] J. Adamek and F. Plasil, “Component Composition Errors and Update Atomicity: Static Analysis: Research Articles,” *J. Softw. Maint. Evol.*, vol. 17, no. 5, pp. 363–377, 2005.

[73] H. Schweppe, A. Zimmermann, and D. Grill, “Flexible On-Board Stream Processing for Automotive Sensor Data,” *IEEE Trans. Ind. Informatics*, vol. 6, no. 1, pp. 81–92, Feb. 2010.

[74] L. Golab and M. T. Özsu, “Issues in Data Stream Management,” *ACM SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[75] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS ’02*, 2002, p. 1.

[76] M. Cherniack *et al.*, “Scalable distributed stream processing,” in *2003 Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, 2003, p. 12.

[77] IBM, “Stream Computing Platforms, Applications, and Analytics,” 2015. [Online]. Available: http://researcher.ibm.com/researcher/view_group.php?id=2531. [Accessed: 14-Oct-2015].

[78] S. Chandrasekaran *et al.*, “TelegraphCQ: Continuous Dataflow Processing for an Uncertain World,” in *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD ’03*, 2003, p. 668.

[79] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, and V. Kumar, “Building user-defined runtime adaptation routines for stream processing applications,” *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1826–1837, 2012.

[80] A. J. Ramirez and B. H. C. Cheng, “Design patterns for developing dynamically adaptive systems,” in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS ’10*, 2010, pp. 49–58.

[81] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, “Dynamic Software Updating Using a Relaxed Consistency Model,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 679–694, Sep. 2011.

[82] S. Subramanian, M. Hicks, and K. S. McKinley, “Dynamic Software Updates: A VM-centric Approach,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 1–12.

[83] J. P. Morrison, *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. Paramount, CA:

PUC-Rio - Certificação Digital Nº 1312421/CA

CreateSpace, 2010.
[84] N. Nouali-Taboudjemat, F. Chehbour, and H. Drias, “On Performance Evaluation and Design of Atomic Commit Protocols for Mobile Transactions,” *Distrib. Parallel Databases*, vol. 27, no. 1, pp. 53–94, 2010.
[85] S. Ajmani, “Automatic Software Upgrades for Distributed Systems,” PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (MIT), 2004.
[86] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.
[87] M. Siniavine and A. Goel, “Seamless kernel updates,” in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.
[88] B. Gedik and H. Andrade, “A Model-based Framework for Building Extensible, High Performance Stream Processing Middleware and Programming Language for IBM InfoSphere Streams,” *Softw. Pr. Exper.*, vol. 42, no. 11, pp. 1363–1391, 2012.
[89] D. Turaga *et al.*, “Design principles for developing stream processing applications,” *Software—Practice Exp. - Focus Sel. PhD Lit. Rev. Pract. Asp. Softw. Technol.*, vol. 40, no. 12, pp. 1073–1104, 2010.
[90] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch, “Balancing load in stream processing with the cloud,” in *2011 IEEE 27th International Conference on Data Engineering Workshops*, 2011, pp. 16–21.
[91] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7–18, Apr. 2010.
[92] D. J. Cook and S. K. Das, “Pervasive computing at scale: Transforming the state of the art,” *Pervasive Mob. Comput.*, vol. 8, no. 1, pp. 22–35, Feb. 2012.
[93] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[94] K. M. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.
[95] A. Bakshi, A. Talaei-Khoei, and P. Ray, “Adaptive policy framework: A systematic review,” *J. Netw. Comput. Appl.*, vol. 36, no. 4, pp. 1261–1271, Jul. 2013.
[96] L. David, R. Vasconcelos, L. Alves, R. Andre, G. Baptista, and M. Endler, “A Communication Middleware for Scalable Real-Time Mobile Collaboration,” in *IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2012, pp. 54–59.
[97] L. David, R. Vasconcelos, L. Alves, R. André, and M. Endler, “A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes,” *J. Internet Serv. Appl.*, vol. 4, no. 1, p. 16, 2013.
[98] OMG, “The Real-time Publish-Subscribe (RTPS) Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS),” 2010.

PUC-Rio - Certificação Digital Nº 1312421/CA

[Online]. Available: http://www.omg.org/cgi-bin/doc?formal/10-11-01.pdf. [Accessed: 05-Oct-2012].

[99] L. D. Silva *et al.*, “A Large-scale Communication Middleware for Fleet Tracking and Management,” in *Salão de Ferramentas, Brazilian Symposium on Computer Networks and Distributed Systems (SBRC 2012)*, 2012, pp. 54–59.

[100] G. Pardo-Castellote, “Omg data-distribution service: Architectural overview,” *ICDCSW ’03 Proc. 23rd Int. Conf. Distrib. Comput. Syst.*, 2003.

[101] Ö. Köksal and B. Tekinerdogan, “Obstacles in Data Distribution Service Middleware: A Systematic Review,” *Futur. Gener. Comput. Syst.*, vol. 68, pp. 191–210, 2017.

[102] R. O. Vasconcelos, I. Vasconcelos, and M. Endler, “Management of Mobile Dynamic Adaptation in Cyber-Physical Systems,” in *10th International Conference on Network and Service Management (CNSM 2014)*, 2014, p. 4.

[103] R. C. Bone *et al.*, “Definitions for Sepsis and Organ Failure and Guidelines for the Use of Innovative Therapies in Sepsis,” *Chest*, vol. 101, no. 6, pp. 1644–1655, 1992.

[104] R. A. Balk and K. Medlej, “SIRS, Sepsis, and Septic Shock Criteria,” 2016. [Online]. Available: http://www.mdcalc.com/sirs-sepsis-and-septic-shock-criteria. [Accessed: 15-Mar-2016].

[105] M. Singer *et al.*, “The Third International Consensus Definitions for Sepsis and Septic Shock (Sepsis-3),” *J. Am. Med. Assoc.*, vol. 315, no. 8, pp. 801–810, 2016.

[106] A. Jain and A. Nalya, *Learning Storm - Create real-time stream processing applications with Apache Storm*. Packt Publishing Ltd, 2014.

[107] C. H. B. Hospital, “The Chris Hani Baragwanath Hospital, South Africa,” 2016. [Online]. Available: https://www.chrishanibaragwanathhospital.co.za/. [Accessed: 15-Jul-2016].

[108] E. C. M. Câmara, “Um estudo sobre atualização dinâmica de componentes de software,” M.Sc Thesis, Departamento de Informática, PUC-Rio - Pontifícia Universidade Católica do Rio de Janeiro, 2014.

[109] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, “Contextual effects for version-consistent dynamic software updating and safe concurrent programming,” *ACM SIGPLAN Not.*, vol. 43, no. 1, p. 37, Jan. 2008.

PUC-Rio - Certificação Digital Nº 1312421/CA