Utilizando a RFT para a Detecção de Falhas de Microsserviços do Controlador O-RAN

Alexandre Huff[1], **Matti Hiltunen**[2], **Elias P. Duarte Jr.**[3]

[1]Universidade Tecnológica Federal do Paraná – Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

[2]AT&T Labs Research
1 AT&T Way, Bedminster, NJ 07921 – USA

[3]Universidade Federal do Paraná (UFPR) – Programa de Pós-Graduação em Informática
Caixa Postal 19081 – CEP: 81531-980 – Curitiba – PR – Brasil

alexandrehuff@utfpr.edu.br, hiltunen@research.att.com, elias@inf.ufpr.br

Abstract. *The O-RAN (Open Radio Access Network) Alliance is defining a new open source communication interface (E2) to customize and control RAN elements. The RIC (RAN Intelligent Controller) platform allows implementing RAN control functions through microservices called xApps. This work describes a fault tolerance strategy for microservices (xApps) that run on the RIC controller. The proposed strategy consists of state partitioning techniques with partial replication in groups of xApps and context-aware rerouting of messages. A library called RFT (RIC Fault Tolerance) was implemented and is publicly available for the development of fault-tolerant xApps. This paper describes experimental results that demonstrate the detection of faults of RIC microservices with the RFT.*

Resumo. *A O-RAN (Open Radio Access Network) Alliance está definindo uma nova interface de comunicação (E2) de código aberto para customizar e controlar o comportamento da RAN. A plataforma RIC (RAN Intelligent Controller) permite implementar funções de controle da RAN por meio de microsserviços chamados xApps. Este trabalho descreve uma estratégia de tolerância a falhas para os microsserviços (xApps) que executam no controlador RIC. A estratégia proposta consiste de técnicas de particionamento de estado com replicação parcial em grupos de xApps e re-roteamento de mensagens com ciência de papel. Uma biblioteca chamada RFT (RIC Fault Tolerance) foi implementada e disponibilizada para o desenvolvimento de xApps tolerantes a falhas. Resultados experimentais apresentados neste artigo demonstram a detecção de falhas de microsserviços do controlador RIC com a RFT.*

1. Introdução

A Rede de Acesso por Rádio (RAN - *Radio Access Network*) tem promovido uma verdadeira revolução na forma com que pessoas e máquinas comunicam e interagem [Olwal et al. 2016, Parvez et al. 2018, Habibi et al. 2019]. A RAN é parte fundamental das redes móveis, fornecendo conectividade para dispositivos sem fio dos mais diversos tipos, denominados UE (*User Equipment*). A tecnologia 5G [Shayea et al. 2020] corresponde a mais recente geração da RAN e visa fornecer serviços móveis ubíquos com alta qualidade (QoS - *Quality of Service*).

A RAN está evoluindo como parte de um conjunto de novas tecnologias (incluindo 5G, IoT - *Internet of Things*, entre outras) e tem a tarefa não trivial de determinar a melhor forma de utilização e gerenciamento do espectro limitado da rede sem fio para possibilitar a conectividade de equipamentos de usuário (UEs). Em redes 5G densas (*i.e.*, muitas estações base e UEs em uma área geográfica limitada) torna-se ainda mais desafiador alocar recursos de rádio, implementar *handovers* (*i.e.*, transferir a conexão de um UE de uma estação base para outra), gerenciar interferências, balancear a carga entre células da rede, entre outras tarefas que a RAN deve executar. Uma célula consiste em uma determinada área geográfica em que há cobertura de sinal da RAN [Gudipati et al. 2013, Habibi et al. 2019].

Embora as interfaces de comunicação entre os equipamentos de usuário e os elementos da RAN sejam definidas por padrões abertos, a maioria das implementações são normalmente soluções proprietárias fornecidas por fabricantes de equipamentos [Olwal et al. 2016, Parvez et al. 2018, Habibi et al. 2019]. Nesse contexto, provedores de telecomunicação enfrentam diversos desafios para realizar aprimoramentos bem como desenvolver novos serviços e, ao mesmo tempo, torna-se complexo e lento para a comunidade de pesquisa contribuir com essa importante área das redes de telecomunicações.

Recentemente, a [O-RAN Alliance 2021] e a *O-RAN Software Community* [O-RAN SC 2021b] vêm coordenando esforços para apoiar o desenvolvimento de software de código aberto para a RAN considerando requisitos de desempenho, escalabilidade e alinhamento com os padrões especificados pelo projeto 3GPP (*3rd Generation Partnership Project*) [3GPP 2021]. Uma nova interface de comunicação chamada E2 está sendo definida pela [O-RAN Alliance 2021] para permitir que elementos da RAN sejam capazes de expor suas funcionalidades e reportar notificações ao controlador O-RAN. A [O-RAN SC 2021b] vem liderando esforços para a implementação e disponibilização de um controlador O-RAN de código aberto denominado RIC (*RAN Intelligent Controller*). O RIC [O-RAN SC 2021a] pode ser visto como uma plataforma para executar microsserviços, chamados xApps, que usam a interface E2 para acessar e controlar os elementos da RAN.

A alta disponibilidade é considerada requisito fundamental para a RAN [Habibi et al. 2019]. Um xApp falho pode impedir que milhares de dispositivos sem fio conectem-se à rede em uma determinada área geográfica. Além de alta disponibilidade, as redes 5G demandam requisitos rígidos de baixa latência e alta vazão para as funções de controle da RAN [Parvez et al. 2018, Habibi et al. 2019]. A alta disponibilidade pode ser alcançada por meio de técnicas de replicação e tolerância a falhas. Entretanto, as estratégias empregadas com as técnicas de replicação tradicionais para fornecer tolerância a falhas não são capazes de suportar os requisitos de baixa latência e alta vazão exigidos pelas redes 5G no contexto do controlador RIC [O-RAN Alliance 2020c, O-RAN Alliance 2020a].

Este trabalho descreve uma estratégia de tolerância a falhas para os microsserviços (xApps) que executam no controlador RIC. São utilizadas técnicas de particionamento de estado com replicação parcial em grupos de xApps e re-roteamento com ciência de papel para garantir a disponibilidade e, ao mesmo tempo, atender aos requisitos de desempenho impostos pela RAN. O objetivo é permitir a implementação de xApps tolerantes a falhas capazes de suportar os requisitos de baixa latência e alta vazão na plataforma RIC

desenvolvida pela [O-RAN SC 2021b]. Uma biblioteca chamada RFT (*RIC Fault Tolerance*) foi implementada e disponibilizada como software livre para o desenvolvimento de xApps tolerantes a falhas [Huff 2021, Huff et al. 2021a, Huff et al. 2021b]. Neste artigo são apresentados resultados experimentais que demonstram que a RFT detecta falhas de microsserviços com baixa latência.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 descreve uma visão geral do controlador RIC e dos xApps. A RFT é descrita na Seção 3. A avaliação empírica da RFT é descrita na Seção 4 seguida dos trabalhos relacionados na Seção 5. As considerações finais e trabalhos futuros são apresentadas na Seção 6.

2. xApps Tolerantes a Falhas

Esta seção descreve uma visão geral dos componentes do controlador RIC que são fundamentais para construir xApps tolerantes a falhas.

2.1. O RIC e os xApps

A arquitetura da O-RAN [O-RAN Alliance 2020a] define que o RIC monitora e controla os elementos da RAN através da interface E2, que é utilizada pelos xApps para receber eventos específicos da RAN (*e.g.*, um novo UE tentando se conectar na rede) e para emitir mensagens de controle (*e.g.*, rejeitar um pedido de conexão ou mover um UE de uma célula para outra).

A lógica de controle do RIC é implementada por meio de microsserviços chamados de xApps. Os xApps têm como principal objetivo fornecer funções de controle específicas e bem definidas que permitem customizar e otimizar o comportamento da RAN. As funções de controle fornecidas pelos xApps são consideradas funções adicionais e que, portanto, complementam as funcionalidades que já são fornecidas originalmente pelos elementos da RAN. Cada elemento da RAN exporta as suas funcionalidades para o RIC através da interface E2, a qual é utilizada pelos xApps para receber notificações da RAN (*i.e.*, monitorar), emitir mensagens de controle e atualizar as políticas que definem o comportamento padrão dos respectivos elementos da RAN [O-RAN Alliance 2020a, O-RAN Alliance 2020b]. No contexto deste trabalho, um elemento da RAN é qualquer componente da RAN que troca mensagens com os xApps através da interface E2.

Cada instância do RIC pode gerenciar um número potencialmente grande de elementos da RAN em uma determinada região geográfica. Os elementos da RAN por sua vez, devem ser capazes de continuar provendo os seus serviços para a RAN mesmo em uma eventual falha do controlador RIC, resultando apenas na interrupção temporária das funções adicionais de customização e otimização fornecidas pelo RIC [O-RAN Alliance 2020a, O-RAN Alliance 2020b]. Exemplos de funcionalidades adicionais que o RIC pode fornecer através de xApps incluem prevenção de DDoS, transferência de UE entre células (*i.e.*, *handover*), previsão de congestionamento de células e otimização de QoS (*Quality of Service*) baseada na categoria de UEs (*e.g.*, equipamentos usados por policiais, dispositivos da IoT e veículos conectados). Cada xApp interage com os componentes da plataforma RIC e outros xApps usando duas interfaces principais:

- RMR (*RIC Message Router*): uma biblioteca utilizada pelos componentes do controlador RIC para trocar mensagens em uma mesma instância do RIC; e,

- SDL (*Shared Data Layer*): fornece uma abstração de armazenamento *chave-valor* e permite aos xApps armazenar/recuperar informações fornecidas por outros componentes da plataforma RIC e também por outros xApps.

A Figura 1 ilustra os componentes do controlador RIC que são fundamentais para construir xApps tolerantes a falhas. O componente *E2 Term* implementa a interface para os elementos da RAN utilizando o protocolo de transporte SCTP (*Stream Control Transmission Protocol*). O *payload* das mensagens é especificado utilizando ASN.1 (*Abstract Syntax Notation 1*). O componente *Subscription Manager* implementa o padrão de troca de mensagens, que ocorre de acordo com o paradigma *Publish-Subscribe*. O *Subscription Manager* permite que xApps assinem tópicos de interesse que representam eventos específicos que ocorrem na RAN. O *Routing Manager* é responsável por atualizar dinamicamente as políticas de roteamento de mensagens nos componentes da plataforma RIC. A figura assume que xApps podem ser replicados; as n réplicas de um determinado xApp são referenciadas como $xApp_{1..n}$. É importante destacar que o RIC inclui vários componentes adicionais e que não são apresentados na Figura 1, estando incluídos apenas os componentes essenciais para descrever a RFT.

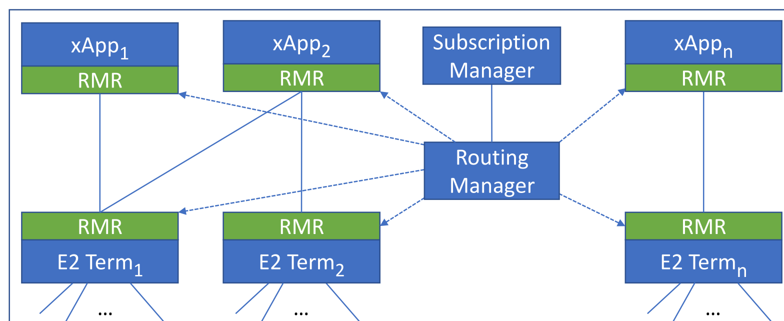

Figura 1. Componentes do controlador RIC.

O RIC é executado em um cluster Kubernetes [Kubernetes 2021] e seus componentes são implantados por meio de *Pods* que consistem de um ou mais contêineres. O cluster Kubernetes é geralmente executado em um cluster de servidores de processamento, e seus componentes são replicados em diferentes servidores para evitar pontos únicos de falhas. A versão atual do RIC (*i.e.*, *Dawn*) não oferece suporte à replicação de xApps.

2.2. Informações de Estado dos xApps

Esta subseção descreve como as informações de estado dos xApps são mantidas, com a perspectiva da replicação de xApps na RFT descrita mais a frente no artigo. Réplicas de um determinado xApp podem executar em máquinas distintas. Cada xApp é inicializado no RIC com um estado inicial. Os xApps podem ser classificados como *stateless* ou *stateful*. Um xApp *stateless* requer apenas as informações contidas em cada mensagem recebida para executar sua função. Deste modo, réplicas de xApps *stateless* podem processar qualquer mensagem em qualquer momento sem qualquer informação de estado anterior.

Um xApp *stateful* depende de informações de estado para processar cada nova mensagem. O estado de uma réplica individual consiste de um ou mais *contextos*. Um

contexto inclui informações sobre um elemento específico da RAN. As informações específicas de cada contexto estão diretamente relacionadas com a lógica de cada xApp, e são determinadas pelo desenvolvedor do xApp. Exemplos de contexto incluem: o identificador de um UE, de um grupo de UEs, uma célula específica da RAN, uma estação base (*i.e.*, *Cell Site*), uma categoria de UE (*e.g.*, dispositivos IoT, telefones celulares), entre outros. Por sua vez, o estado de um xApp consiste do conjunto de contextos mantido por todas as réplicas do xApp. Cada réplica do xApp pode manter informações de contexto referentes a vários elementos da RAN. Réplicas distintas podem manter contextos distintos.

Cada contexto tem um identificador único. As informações mantidas em um contexto são estruturadas por meio de pares *chave-valor*. Como exemplo, considere um contexto de um determinado xApp referente a uma célula. Neste caso, o identificador do contexto é o próprio identificador daquela célula na RAN. O contexto pode ter informações diversas sobre a célula, tais como o nível de sinal e a utilização, que são armazenados como pares *chave-valor*. Diversos xApps requerem informações de contexto para processar uma mensagem e, portanto, se encaixam na classe *stateful*. Exemplos de funcionalidades implementadas por xApps *stateful* e as respectivas informações de contexto incluem:

- medir a duração da conexão de um UE: é necessário armazenar tanto o instante de estabelecimento como de encerramento da conexão daquele UE;
- controlar a admissão e alocação de recursos na RAN para prover conectividade aos UEs: requer o conhecimento dos recursos disponíveis de uma determinada célula ou de uma região que é coberta por várias células para determinar a admissão do UE na rede;
- prever a utilização de uma determinada célula: a partir da utilização é possível por exemplo evitar sobrecarga, movendo determinados UEs para células vizinhas (*i.e.*, *handover*). Informações de contexto da célula de destino e da células vizinhas são necessárias.

Os contextos de um xApp podem ser replicados de diferentes maneiras. Por exemplo, todos os contextos poderiam ser replicados em todas as réplicas do xApp. Entretanto, dependendo da quantidade de informação dos contextos e da frequência com que itens diferentes de contextos diferentes são atualizados, esta solução pode ter custo muito alto em termos de desempenho. Assim, a estratégia proposta é *particionar* o estado de um xApp em contextos e *replicá-los parcialmente* entre as instâncias do xApp. O particionamento permite duas réplicas distintas de um xApp manter conjuntos diferentes de contextos daquele xApp. Particionar o estado dessa forma permite reduzir a quantidade de mensagens que uma instância de um xApp precisa processar. Diminui também o tempo que a instância precisa para atualizar seu estado a cada nova requisição recebida.

Além do particionamento, que limita a quantidade de contextos que uma réplica de um xApp mantém, a *replicação parcial* de contextos permite que um xApp tenha um conjunto configurável de réplicas. Antes, uma definição: uma réplica primária é responsável por manter e atualizar um contexto enquanto uma réplica *backup* é responsável apenas por manter uma cópia do contexto. Neste trabalho, os termos *primário* e *backup* também são usados como sinônimos para réplica primária e réplica *backup*, respectivamente. Na replicação parcial, cada réplica do xApp pode ser primária para determinados contextos

e pode ter um número configurável de *backups* daquele contexto entre as demais réplicas do xApp.

Embora a replicação garanta que o serviço provido por um xApp esteja disponível mesmo após a ocorrência de falhas, a replicação por si só não assegura que o RIC é capaz de processar e responder uma mensagem com uma requisição referente a um contexto que chega logo após sua réplica primária ter falhado. O *re-roteamento com ciência de papel* tem essa funcionalidade, ou seja, possibilita que a mensagem seja direcionada rapidamente para uma réplica *backup*. Especificamente, caso a réplica primária de um determinado contexto não esteja disponível para processar a mensagem de um elemento da RAN, a mensagem é imediatamente encaminhada para uma das réplicas *backup* que possui uma cópia aproximada daquele contexto. O re-roteamento ciente de papel é usado para re-encaminhar mensagens após falhas. A réplica *backup* que recebe a mensagem da RAN estará ciente de sua função como *backup* para aquele contexto e pode usar esta "ciência" para processar a mensagem considerando que suas informações de contexto podem não estar totalmente atualizadas.

Mesmo a replicação parcial também pode ser cara em termos de latência caso seja realizada de maneira síncrona (*i.e.*, qualquer atualização de contexto é imediatamente processada pela réplica primária e pelas réplicas *backup*). Visando reduzir a latência dos xApps, é proposta a replicação assíncrona de contextos, em que inicialmente, apenas a réplica primária atualiza o contexto, o que é feito posteriormente pelas réplicas *backup*. A replicação assíncrona permite que a réplica primária possa modificar as informações dos contextos e responder as requisições da RAN sem ter que aguardar confirmações das réplicas *backup*. Na estratégia de replicação proposta, após a falha de uma réplica primária para um determinado contexto, uma das réplicas *backup* daquele contexto é eleita como sua nova primária.

3. RFT: *RIC Fault Tolerance*

A RFT (*RIC Fault Tolerance*) foi definida de forma a permitir a implementação de xApps não apenas tolerantes a falhas, mas também capazes de sustentar baixa latência e manter a escalabilidade necessária para suportar as dezenas de milhares de requisições por segundo da RAN. Como mencionado na seção anterior, a solução consiste de técnicas de particionamento de estado com replicação parcial e re-roteamento com ciência de papel. A RFT é implementada e disponibilizada por meio de uma biblioteca [Huff et al. 2021a] que pode ser vinculada à implementação dos xApps da mesma forma que outras bibliotecas da plataforma RIC, tais como RMR, SDL e bibliotecas de *logging*.

A RFT é baseada em uma estratégia de *group membership* que permite que os membros do grupo tenham uma visão consistente sobre a composição do grupo. O consenso é usado para que haja um acordo sobre a composição, *i.e.*, o conjunto de réplicas corretas em um determinado instante de tempo.

xApps tolerantes a falhas podem ser construídos de forma transparente por meio da RFT, *i.e.*, o fato de um xApp ser replicado fica transparente para o restante do sistema. Vale ressaltar que simplesmente depender do Kubernetes para adicionar tolerância a falhas aos xApps não é uma alternativa viável. Embora o Kubernetes faça monitoramento de *Pods* (*i.e.*, um grupo de um ou mais contêineres executando em um *cluster*), este monitoramento não é suficiente para implementar toda a lógica de replicação de xApps. Por

exemplo, após a detecção da mudança de estado de um membro do grupo, pode ser necessário modificar os papéis (*i.e.*, primário ou *backup*) de determinadas réplicas do xApp. Também são necessárias atualizações nas políticas de roteamento para encaminhar as mensagens da RAN corretamente às novas réplicas primária e *backup*.

A RFT implementa a gerência de grupos de réplicas de xApps utilizando o algoritmo de consenso Raft [Ongaro and Ousterhout 2014]. O algoritmo Raft permite manter a sequência do *log* da máquina de estado consistente em cada uma das réplicas. Uma máquina de estado consiste em variáveis que caracterizam o estado e de comandos, que executados modificam o estado [Schneider 1990]. A replicação de uma máquina de estado é geralmente implementada por meio da replicação de um *log*, que consiste de uma sequência de comandos que é executada pela máquina de estado. Cada réplica armazena um *log* e deve manter os mesmos comandos na mesma ordem, permitindo que cada réplica execute a mesma sequência de comandos das outras réplicas que resulta no mesmo estado.

Um dos componentes principais do Raft é referente à eleição de um líder entre os processos que executam o consenso. Assim, a consistência do *log* é garantida através do processo líder que gerencia a replicação de comandos do *log* para os demais processos. O fluxo de mensagens de replicação do Raft é sempre executado do processo líder para os demais processos que participam do consenso. O líder também aceita comandos de processos cliente que são adicionados em sequência no *log* e replicados para os demais processos do consenso. Os comandos somente são aplicados (*i.e.*, executados) em cada máquina de estado após o líder confirmar que o comando foi devidamente replicado na maioria dos processos corretos que participam do consenso.

Cada comando que é executado pela máquina de estado é armazenado no *log* de cada uma das réplicas. Para evitar que o *log* cresça indefinidamente e permitir a manipulação de seus dados de forma eficaz, o Raft define o conceito de *snapshot*. Um *snapshot* consiste do conjunto serializado de valores de todas as variáveis que compõem a máquina de estado em um dado instante. A estratégia de serialização de *snapshot* pode diferir de acordo com a implementação do algoritmo de consenso [Ongaro and Ousterhout 2014]. Ao ser recebido no destino, o *snapshot* é desserializado, processo que neste trabalho é chamado de "instalação de um *snapshot*".

3.1. Arquitetura da RFT

A Figura 2 apresenta a arquitetura da RFT, organizando todos os seus componentes. Também é apresentada a relação em alto nível entre a RFT, o RMR e os xApps (*xApp Logic*). Observa-se que ambos o xApp e a RFT compartilham a mesma instância do RMR para enviar e receber mensagens na rede.

O componente *Tasks* consiste de uma fila de tarefas que a RFT deve executar em ordem. As tarefas são representadas nesta fila por meio de mensagens recebidas pela rede e que foram adicionadas por uma réplica do xApp. As tarefas desencadeiam diferentes operações na RFT, tais como adicionar um novo membro ao grupo de xApps ou processar uma mensagem de replicação. A fila de tarefas é assíncrona, no sentido de que as mensagens que estão nesta fila não bloqueiam o processamento de outras mensagens na réplica do xApp. Novas mensagens (*i.e.*, tarefas) da RFT podem ser adicionadas nessa fila mesmo que as mensagens anteriores não tenham sido completamente processadas pela RFT.

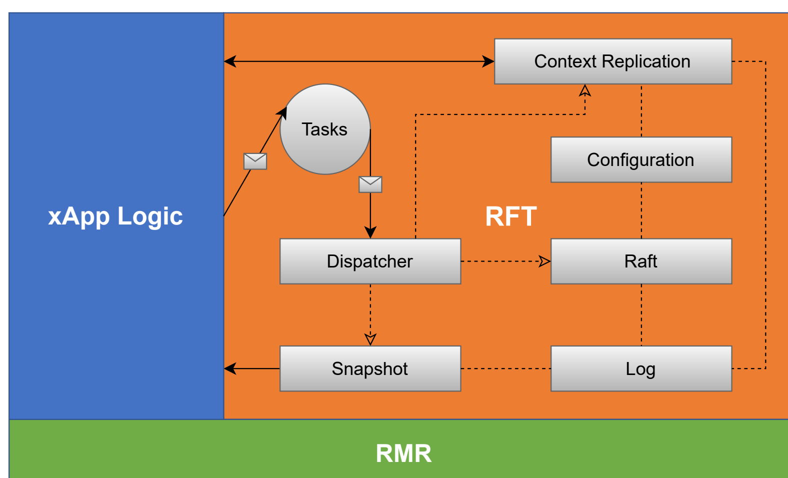

Figura 2. Arquitetura da RFT.

O componente *Dispatcher* distribui as tarefas entre os diferentes componentes da RFT com base no tipo da mensagem recebida. O tipo da mensagem identifica se a mesma é referente a replicação de contexto, uma solicitação de adição de membro no grupo, uma mensagem de monitoramento da RFT, entre outras. Outra funcionalidade do *Dispatcher* é desserializar o *payload* das mensagens recebidas antes de serem encaminhadas para os devidos componentes.

As mensagens referentes ao monitoramento de membros do grupo de xApps são encaminhadas ao componente chamado *Raft*, que implementa o algoritmo de consenso Raft e mantém uma visão consistente dos membros corretos do grupo de réplicas de um determinado xApp. O *Raft* replica os comandos da máquina de estado que modificam a composição do grupo de réplicas de um xApp e também realiza a eleição de líder entre as réplicas do grupo. O componente *Raft* implementado na RFT permite adicionar e remover membros do grupo de xApps automaticamente, enquanto que a versão original do algoritmo Raft requer intervenção manual [Ongaro and Ousterhout 2014]. O componente *Configuration* é essencialmente uma extensão do componente *Raft*. O componente *Configuration* mantém as informações sobre a composição de um grupo de réplicas de um xApp e é responsável por selecionar as réplicas *backup* de uma determinada réplica primária do xApp. Uma lista com as réplicas *backup* é disponibilizada para o componente *Context Replication*.

O componente *Context Replication* basicamente possui duas responsabilidades: replicar as atualizações de contextos das réplicas primárias para as respectivas réplicas *backup*; e, aplicar as atualizações de contextos nas réplicas *backup* assim que foram recebidas da réplica primária. O componente *Context Replication* atualiza as informações de contexto nas réplicas *backup* através de uma função de *callback* que é implementada na lógica do xApp (*xApp Logic*). A RFT é agnóstica com relação às informações mantidas nos contextos e processadas pelos xApps. Cada xApp possui contextos diferentes com informações distintas e, que são processadas de formas específicas.

O componente *log* mantém em ordem os comandos executados e replicados pela máquina de estados, incluindo aqueles para gerenciar o grupo de réplicas de um xApp. Os comandos das máquinas de estados são mantidos em duas filas independentes. Uma

fila mantém os comandos da máquina de estado dos contextos dos xApps que é replicada pelo componente *Context Replication*. Outra fila mantém os comandos da máquina de estados do grupo de réplicas do xApp que é replicada pelo componente *Raft*.

Os componentes *Context Replication* e *Raft* implementam estratégias diferentes de replicação. O componente *Context Replication* replica os comandos da máquina de estados dos xApps empregando replicação parcial assíncrona sem aguardar confirmações das réplicas *backup* para responder as requisições de clientes. O componente *Raft* implementa replicação síncrona e aguarda as confirmações da maioria das réplicas do grupo do xApp, para então responder as requisições de clientes. O componente *log* também é responsável pela serialização do *payload* das mensagens de replicação para ambas as estratégias de replicação. Por fim, o componente *Snapshot* é responsável coordenar o processo de serialização/desserialização dos *snapshots* das máquinas de estado. Em seguida, é apresentada uma avaliação empírica da RFT.

4. Avaliação Empírica

Uma avaliação empírica foi realizada para avaliar o desempenho da RFT no tocante à detecção de falhas. Especificamente, foi avaliado o tempo necessário para a RFT detectar que uma determinada réplica está falha, resultando em modificações na composição do grupo de réplicas e alterações nas políticas de roteamento do RMR. Um xApp seletor de células foi implementado em linguagem C para resolver um problema prático de seleção de células da RAN. O objetivo deste xApp é selecionar a melhor célula de destino em operações de *handover* solicitadas por diferentes UEs e melhorar o desempenho geral da rede. Especificamente, assume-se que na operação de *handover* a RAN irá gerar solicitações de transferência de UE sempre que um UE observar uma célula vizinha com intensidade maior de sinal do que a sua célula atual.

No entanto, o fato de uma célula ter a maior intensidade de sinal não a torna necessariamente a melhor escolha para transferir o UE, pois a mesma pode estar congestionada devido, por exemplo, a um grande número de UEs conectados. O xApp seletor de células considera também a utilização de recursos das células que pode ser acompanhada por meio das operações de conexão e desconexão dos UEs. Um contador simples é usado para controlar o número de UEs conectados em cada célula. Outras métricas poderiam ser utilizadas para o cálculo do critério de seleção de células, mas para o propósito deste experimento elas não são realmente necessárias.

A RFT utiliza replicação parcial e o estado do xApp seletor de células é particionado em contextos, cada um representando uma determinada célula. No experimento, cada réplica do xApp acompanha a utilização de um conjunto de células, diferentes réplicas do xApp acompanham diferentes conjuntos. Assim, cada réplica mantém informações atualizadas sobre o grupo correspondente de células. A RFT foi utilizada para gerenciar a replicação dos contextos entre cada réplica primária do xApp e outra réplica *backup*. Também foi implementado um gerador de tráfego em linguagem C. O gerador de tráfego troca mensagens com as réplicas primárias do xApp usando o RMR e implementa a combinação das funcionalidades de um elemento da RAN e de um componente *E2 Term* da plataforma RIC.

O experimento foi realizado em ambiente de laboratório no qual os xApps foram executados em contêineres em uma máquina física. A máquina executou Linux Ubuntu

20.04 em um processador Intel(R) Core(TM) i7-6700HQ @ 2.6 GHz com 4 núcleos e 12 GiB de memória RAM. Foram utilizados cinco contêineres para a execução do experimento, cada um executou uma única réplica do xApp. Cada réplica primária selecionou uma réplica de *backup*, tal que, uma mesma réplica é primária para um conjunto de contextos e *backup* para outro conjunto de contextos.

No experimento o líder do grupo de réplicas nunca falha e envia mensagens de monitoramento periodicamente para os demais membros do grupo. A periodicidade de envio de mensagens de monitoramento e a quantidade de mensagens necessárias para detectar uma réplica falha são opções de configuração na RFT. Na configuração da RFT para este experimento considera-se que uma réplica está falha caso não responder a cinco mensagens consecutivas de monitoramento que são enviadas pelo líder do grupo. A falha foi simulada por meio de interrupções de software empregando as chamadas de sistema `signal` e `kill` do sistema operacional Linux. Ao inicializar, a RFT registra funções que tratam os sinais `SIGUSR1` e `SIGUSR2`. `SIGUSR1` sinaliza ao líder o instante de tempo que uma réplica falhou, enquanto `SIGUSR2` é lançado no instante de tempo que o líder detecta a falha. Os instantes de tempo e a quantidade de mensagens de monitoramento enviadas são coletadas pelas funções que tratam destas interrupções de software. As réplicas que falharam foram escolhidas aleatoriamente durante a execução do experimento, com distribuição de probabilidade uniforme.

A Figura 3 apresenta a quantidade de mensagens de monitoramento enviadas pelo líder do grupo entre o instante de tempo que uma das réplicas do grupo falhou e o instante de tempo que o líder detecta a falha. Mensagens de monitoramento foram enviadas em diferentes intervalos de tempo (10ms, 30ms, 50ms, 100ms e 150ms) com o objetivo de mensurar a quantidade de mensagens de monitoramento necessárias até a detecção da réplica falha em cada um dos intervalos. Uma única réplica aleatória falhou em cada rodada de execução do experimento que foi repetido 50 vezes. As médias e os respectivos intervalos de confiança de 95% também são apresentadas na Figura 3.

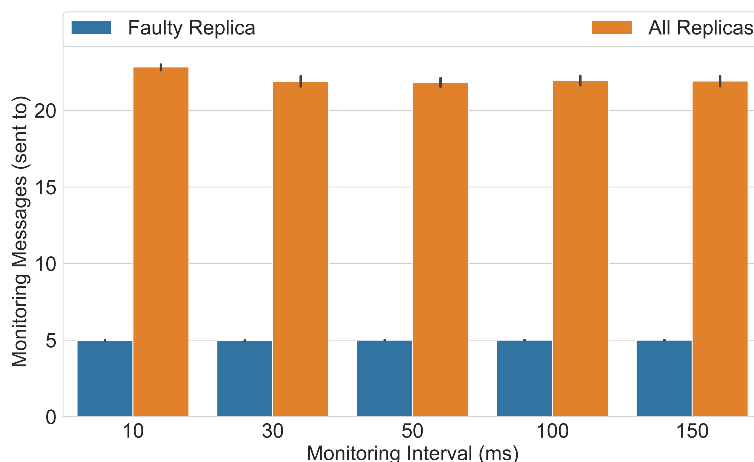

Figura 3. Número de mensagens de monitoramento enviadas.

Foi possível observar que no total (`All Replicas`), o líder precisou enviar em média pouco mais de 21 mensagens de monitoramento para as outras 4 réplicas do grupo até a réplica falha ser detectada. Este número de mensagens deve-se também ao

escalonamento das *threads* que realizam o monitoramento das réplicas de forma concorrente. A quantidade de mensagens de monitoramento enviadas em particular à réplica falha em cada rodada (`Faulty Replica`) foi em média de 5 mensagens conforme a configuração da RFT. Isto deve-se ao fato de que a *thread* que realiza o monitoramento e detecta a falha envia o sinal de interrupção `SIGUSR2` assim que a falha é detectada, direcionado a execução da thread de monitoramento para a rotina de tratamento da interrupção que foi sinalizada.

A Figura 4 apresenta a latência da detecção de falhas da RFT, considerando uma única réplica falha em cada rodada de execução. Vale ressaltar que o conjunto de dados desta figura foi obtido a partir do mesmo experimento da figura anterior. Assim, a figura apresenta a média da latência de detecção de falhas de 50 execuções e seus respectivos intervalos de confiança de 95%. Foi possível observar neste experimento que são necessários em média 55,5ms para detectar uma réplica falha com mensagens enviadas a cada 10ms e uma tolerância de até 5 mensagens não respondidas consecutivamente pela réplica monitorada. Para intervalos de monitoramento de 30ms a latência de detecção foi em média 164,9ms. Para os intervalos de 50ms, 100ms e 150ms a latência média foi respectivamente 276,4ms, 554,1ms e 812ms. De modo geral, o *overhead* aproximado da latência de detecção de falhas da RFT foi em média de 10,11% considerando a tolerância de 5 mensagens não respondidas pela réplica que está sendo monitorada.

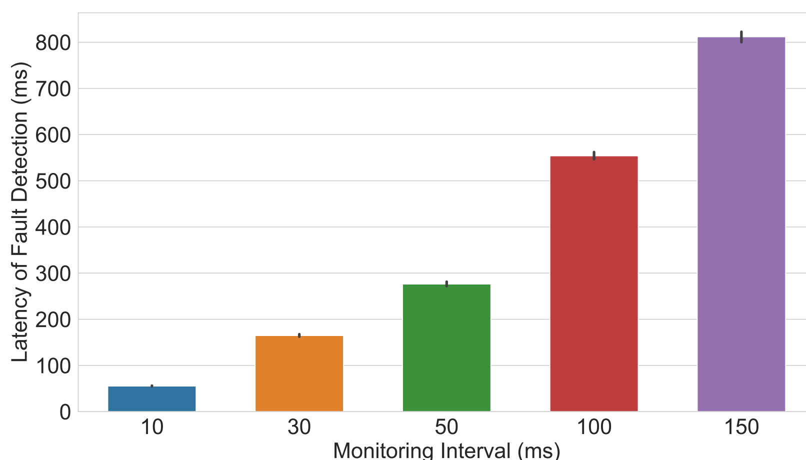

Figura 4. Latência da detecção de falhas da RFT.

Experimentos que avaliam a vazão e latência da RFT com o xApp seletor de células utilizado neste trabalho são apresentados em [Huff et al. 2021b].

5. Trabalhos Relacionados

O problema de fornecer tolerância a falhas e ao mesmo tempo garantir baixa latência e alta vazão foi abordado de diferentes maneiras ao longo dos anos. A tolerância a falhas é tipicamente alcançada usando replicação. Por exemplo, a LLFT (*Low Latency Fault Tolerance*) [Zhao et al. 2013] é baseada em replicação com líder primário. O sistema garante a consistência forte da réplica e baixa sobrecarga, pois a réplica primária pode responder sem aguardar pelas respostas das réplicas *backup*, mas a ordem das mensagens estabelecida pela réplica primária garante que as outras réplicas atingirão o mesmo

estado. Embora o sistema forneça baixa latência, a vazão é limitada, pois o estado é totalmente replicado e o sistema não pode utilizar as diferentes réplicas para tarefas distintas simultâneas. Na RFT, o estado é particionado e parcialmente replicado permitindo que o processamento das requisições seja realizado em paralelo, no sentido de que as requisições são encaminhadas para as réplicas que possuem as informações correspondentes a cada requisição.

A replicação em cadeia (*chain replication*) [van Renesse and Schneider 2004] atinge alta disponibilidade com a replicação de um objeto em t réplicas, sendo que t-1 réplicas podem falhar sem comprometer a disponibilidade do objeto replicado. Enquanto que as operações de leitura dos objetos são executadas na primeira réplica da cadeia, as atualizações são processadas pela última réplica da cadeia. A consistência é garantida pelo fato de que a alteração de estado deve passar por todas as réplicas até chegar à última da cadeia. O desempenho da leitura é beneficiado, pois essas operações envolvem apenas uma única réplica, enquanto que atualizações envolvem o processamento por todas as réplicas, e a latência pode crescer significativamente mesmo para poucas réplicas.

O sistema descrito em [Sherry et al. 2015] apresenta uma solução de alta disponibilidade para *middleboxes* usando máquinas virtuais. A estratégia assume que os *middleboxes* mantêm estado que deve ser devidamente recuperado após uma falha. O sistema é baseado na estratégia clássica de *rollback recovery*. Para aumentar o desempenho é empregada uma estratégia de *logging*. Máquinas virtuais executam uma réplica primária e outra de *backup* enquanto que dois registradores são utilizados para armazenar o tráfego de entrada e saída em um *switch* virtual. *Checkpoints* periódicos da réplica primária são salvos na memória principal do *switch* que são utilizados para inicializar o estado da réplica de *backup* caso a primária falhe. Os pacotes da rede somente são liberados pelo *switch* após todas as informações necessárias para a retransmissão do pacote terem sido armazenadas. Além disso, a função de rede implementada no *middlebox* não é executada enquanto um *checkpoint* está sendo processado. Portanto, o impacto na latência é proporcional à quantidade de alterações do estado da aplicação entre os *checkpoints* e é inversamente proporcional à largura de banda exigida pelo armazenamento.

Uma alternativa para atingir replicação com desempenho escalável é permitir que diferentes réplicas processem diferentes subconjuntos de mensagens e fragmentem o armazenamento do estado da aplicação. Original da área de banco de dados, a técnica de *sharding* permite a divisão de estado em diferentes réplicas [Cattell 2011]. Essas divisões de estado, denominadas *shards*, podem ser replicadas tantas vezes quanto necessário e podem ser configuradas para processarem um maior número de leituras ou atualizações simultâneas.

Em [Adya et al. 2016], é proposta uma estratégia que permite que aplicações de *datacenter* distribuam a carga de trabalho por meio de um serviço de fragmentação denominado Slicer. No Slicer, uma tarefa é um processo de uma aplicação que executa de forma simultânea com processos de outras aplicações em um hardware compartilhado. O Slicer utiliza, além da fragmentação, o balanceamento de carga na execução das tarefas. O sistema é monitorado para detectar congestionamento e falhas. O Slicer tem como maior objetivo realizar o balanceamento de carga uniforme. A fragmentação da carga de trabalho é feita utilizando funções *hash*. O Slicer armazena o estado das aplicações em um sistema externo, como o Redis.

Com relação aos esforços da O-RAN em geral, as iniciativas OAI (*Open Air Interface*) e srsLTE visam fornecer implementações de código aberto dos padrões 3GPP para a RAN [Gomez-Miguelez et al. 2016, Kaltenberger et al. 2019, OAI 2021, O-RAN SC 2021b]. A plataforma RIC está sendo desenvolvida sob a responsabilidade da [O-RAN SC 2021b] e, até onde se sabe, não existem propostas para tolerância a falhas de xApps e do controlador RIC.

6. Conclusão

Este trabalho apresentou uma estratégia para a tolerância a falhas do controlador RIC baseada na replicação de xApps. A estratégia consiste na definição de um conjunto de técnicas para permitir a implementação de xApps tolerantes a falhas capazes de suportar os requisitos de baixa latência e alta vazão impostos pela RAN, sobretudo em redes 5G. Foi proposta a utilização de técnicas de particionamento de estados com replicação parcial em grupos de xApps e re-roteamento de mensagens com ciência de papel para atender aos requisitos exigidos pelo controlador RIC. As políticas de roteamento são definidas tendo em vista a composição atualizada do grupo de réplicas correspondente, possibilitando que as mensagens sejam entregues para a réplica adequada para processar a mensagem.

Uma biblioteca para a construção de microsserviços tolerantes a falhas para o RIC foi implementada e disponibilizada como software livre, denominada RFT (*RIC Fault Tolerance*). A RFT foi avaliada no tocante ao desempenho na detecção de falhas para microsserviços do controlador RIC. Um xApp seletor de células com 5 réplicas foi utilizado na avaliação. Resultados experimentais mostraram que a RFT é uma estratégia efetiva para a detecção de falhas dos xApps com baixa latência. A sobrecarga na latência de detecção de falhas da RFT foi em média de 10,11% considerando a tolerância de 5 mensagens de monitoramento não respondidas pela réplica monitorada.

Trabalhos futuros incluem a investigação de técnicas para elasticidade dos xApps tolerantes a falhas na plataforma RIC. Também será investigada a atribuição e o gerenciamento dinâmico das réplicas primária e de *backup* para os elementos da RAN, empregando técnicas de balanceamento de carga e posicionamento de xApps (*i.e.*, *placement*). Também é objetivo avaliar a RFT para outros xApps na medida em que sejam implementados e disponibilizados pela comunidade O-RAN.

Por fim, a abordagem de replicação de xApps da RFT está sendo proposta para a O-RAN SC, visando torná-la estratégia oficial da plataforma RIC em um futuro próximo.

Referências

3GPP (2021). The 3rd generation partnership project (3GPP). `https://www.3gpp.org/`. Acessado em junho de 2021.

Adya, A., Myers, D., Howell, J., Elson, J., Meek, C., Khemani, V., et al. (2016). Slicer: Auto-sharding for datacenter applications. In *Proc. OSDI*, pages 739–753.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec.*, 39(4):12–27.

Gomez-Miguelez, I., Garcia-Saavedra, A., Sutton, P., Serrano, P., Cano, C., and Leith, D. (2016). srsLTE: an open-source platform for LTE evolution and experimentation. In *Proc. 10th ACM Int. Workshop on Wireless Network Testbeds, Exp. Evaluation, and Characterization*, pages 25–32.

Gudipati, A., Perry, D., Li, L. E., and Katti, S. (2013). SoftRAN: Software defined radio access network. In *Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, page 25–30.

Habibi, M. A., Nasimi, M., Han, B., and Schotten, H. D. (2019). A comprehensive survey of RAN architectures toward 5G mobile communication system. *IEEE Access*, 7:70371–70421.

Huff, A. (2021). *Composição de Serviços Virtualizados de Rede sobre Múltiplos Orquestradores NFV & Tolerância a Falhas para Microsserviços do Controlador O-RAN*. PhD Thesis, UFPR.

Huff, A., Hiltunen, M., and Duarte Jr., E. P. (2021a). RFT: RIC fault tolerance. `https://github.com/alexandre-huff/rft`. Acessado em junho de 2021.

Huff, A., Hiltunen, M., and Duarte Jr., E. P. (2021b). RFT: Scalable and fault-tolerant microservices for the O-RAN control plane. In *2021 IFIP/IEEE Int. Symp. on Integrated Network Management*.

Kaltenberger, F., de Souza, G., Knopp, R., and Wang, H. (2019). The OpenAirInterface 5G new radio implementation: Current status and roadmap. In *23rd Int. ITG Workshop on Smart Antennas (WSA)*.

Kubernetes (2021). Kubernetes. `https://kubernetes.io/`. Acessado em junho de 2021.

O-RAN Alliance (2020a). O-RAN architecture description. Technical Specification O-RAN-WG1-O-RAN Architecture Description - v01.00.00, O-RAN Alliance.

O-RAN Alliance (2020b). O-RAN near-real-time RAN intelligent controller architecture & E2 general aspects and principles 1.01. Technical Specification O-RAN.WG3.E2GAP-v01.01, O-RAN Alliance.

O-RAN Alliance (2020c). O-RAN operations and maintenance architecture. Technical Specification O-RAN.WG1.OAM-Architecture-v03.00, O-RAN Alliance.

O-RAN Alliance (2021). O-RAN Alliance. `https://www.o-ran.org/`. Acessado em junho de 2021.

O-RAN SC (2021a). O-RAN SC projects: Near realtime RAN intelligent controller (RIC). `https://docs.o-ran-sc.org/en/latest/projects.html`. Acessado em junho de 2021.

O-RAN SC (2021b). O-RAN software community. `https://o-ran-sc.org/`. Acessado em junho de 2021.

OAI (2021). OpenAirInterface: 5G software alliance for democratising wireless innovation. `https://www.openairinterface.org/`. Acessado em junho de 2021.

Olwal, T. O., Djouani, K., and Kurien, A. M. (2016). A survey of resource management toward 5G radio access networks. *IEEE Communications Surveys & Tutorials*, 18(3):1656–1686.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proc. USENIX ATC*, pages 305–319.

Parvez, I., Rahmati, A., Guvenc, I., Sarwat, A. I., and Dai, H. (2018). A survey on low latency towards 5G: RAN, core network and caching solutions. *IEEE Commun. Surveys & Tutorials*, 20(4):3098–3130.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319.

Shayea, I., Ergen, M., Azmi, M. H., et al. (2020). Key challenges, drivers and solutions for mobility management in 5G networks: A survey. *IEEE Access*, 8:172534–172552.

Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., et al. (2015). Rollback-recovery for middleboxes. In *Proc. SIGCOMM*, page 227–240.

van Renesse, R. and Schneider, F. B. (2004). Chain replication for supporting high throughput and availability. In *Proc. OSDI*.

Zhao, W., Melliar-Smith, P., and Moser, L. (2013). Low latency fault tolerance system. *The Computer Journal*, 56:716–740.