RFT: Scalable and Fault-Tolerant Microservices for the O-RAN Control Plane

Alexandre Huff
Federal University of Paraná, Curitiba, PR, Brazil
& *Federal Technological University of Paraná*
Toledo, PR, Brazil
alexandrehuff@utfpr.edu.br

Matti Hiltunen
AT&T Labs - Research
Bedminster, NJ, United States
hiltunen@research.att.com

Elias P. Duarte Jr.
Federal University of Paraná
Curitiba, PR, Brazil
elias@inf.ufpr.br

***Abstract*—The Open Radio Access Network (O-RAN) Alliance is opening up traditionally closed RAN elements by defining a new open communication interface (E2) that allows the behavior of a RAN element to be customized and controlled in real time. The RAN Intelligent Controller (RIC for short) is a platform for implementing RAN control functions as microservices called xApps. In this work, we propose and evaluate techniques to enable xApps in the RIC platform to be fault-tolerant while preserving high scalability. The key premise of our work is that traditional replication techniques cannot sustain high throughput and low latency as required by RAN elements. We propose techniques that use state partitioning, partial replication, and fast re-route with role awareness to decrease the overhead. We implemented the fault tolerance techniques as a library, called RFT (RIC Fault Tolerance), that xApp writers can employ to easily make their xApps fault-tolerant. We present performance results which show that RFT meets latency and throughout requirements as the number of replicas increases.**

***Index Terms*—5G, RAN, fault tolerance, high throughput, low latency, partial replication**

I. Introduction

The Radio Access Network (RAN) is one of the key elements of cellular networks, providing wide-area connectivity to wireless devices (User Equipment - UE). The RAN has the non-trivial task of managing the limited spectrum available to make UE connectivity possible even with a very large number of users. In dense 5G networks, it becomes even more challenging to allocate radio resources, implement handovers, manage interference, balance load between cells, among several other tasks the RAN is supposed to execute [1]. Despite the fact that standards have been defined for the RAN interfaces between network elements and wireless devices, most implementations are typically vendor provided closed solutions [2]–[4]. This makes it a challenge for telco providers to develop novel services, and at the same time makes it hard for the research community to contribute to this important area. Projects such as srsLTE [5] and Open Air Interface (OAI) [6] have started to address this problem by releasing open source implementations of the 3rd Generation Partnership Project (3GPP) standards [7]. Recently, the Open Radio Access Network Software Community (O-RAN SC) [8] has been leading efforts to support the development of open source software for the RAN, while addressing challenges in terms of performance, scalability, and 3GPP alignment.

The O-RAN alliance [9] splits the RAN into components such as the Radio Unit (RU), Distributed Unit (DU), Centralized Unit (CU), the near-real-time RAN Intelligent Controller (RIC), plus other higher level non-real time components. The focus of this work is on the RIC. The purpose of the RIC is to provide a platform for customizing the behavior of the RAN [10]. O-RAN is also defining a new communication interface, E2, between the RIC and the RAN elements, allowing a RAN element to expose its functionalities and report notifications to the RIC. The RIC provides a platform that runs microservices, called xApps, which use the E2 interface to access and control the RAN elements. The interaction between RIC xApps and RAN elements has strict requirements in terms of performance [10]–[13]. As an example, consider a scenario in which an xApp controls UE admission into a network to protect from intentional or accidental DDoS attacks from IoT devices [14], [15]. In that scenario the RIC platform and the xApps must present low latency since valid operations should not be delayed or the user experience can be degraded. For the same reason, fault tolerance is also a requirement, as a faulty xApp should not prevent a legitimate UE from joining the network. Actually, as the failure of a RIC application can have an impact on the reliability of the network itself, fault-tolerance is very much needed. However, the solution must be feasible in terms of cost: a single RIC instance must present high throughput to handle thousands of requests per second.

The problem addressed in this work is how to implement fault-tolerant, high throughput, and low latency xApps in the open-source RIC platform [16] provided by the O-RAN SC. The required approach for achieving fault tolerance is the replication of xApps. However, even existing replication solutions that focus on low latency and high throughput [17] cannot meet the goal of keeping the control loop latency at the RIC at a maximum of 2ms, while keeping the scalability required to support tens of thousands of requests per second [12], [13]. The 2ms limit for the control loop latency at the RIC is derived from the assumption that the RAN scheduler loop must operate below 10ms and the communication latency

This work was partially supported by CAPES Finance Code 001 and CNPq grant 311451/2016-0.

978-3-903176-32-4 ©2021 IFIP

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

between the RIC and the RAN element may be as high as 4ms in each direction.

In order to meet the challenges of high throughput and low latency required in the RAN problem domain, we propose to employ *state partitioning with approximate partial replication* and *fast re-route with role awareness* techniques to meet those RAN challenges. State partitioning resembles traditional sharding in the sense that different subsets of data are maintained by different replicas. However, in our context, data sharding is based on the semantics of the data, which differs from traditional sharding techniques that rely solely on key hashes. In the approximate partial replication strategy adopted, data items maintained by a primary xApp replica are replicated to one or more xApp backup replicas and consistency guarantees can be customized (e.g., replication frequency). Fast re-route with role awareness allows a message to be quickly re-routed to one of the backup replicas when the primary replica is unreachable or overloaded. The backup replica is then able to process the message with the awareness that it is a backup and may not have the most up to date state.

Thus the main contribution of this work consists of defining a set of techniques to implement fault-tolerant xApps that can support the requirements of low latency and scalable high-throughput in the RIC platform developed by O-RAN SC. High throughput and low latency are possible because of the techniques described in the paper to build fault-tolerant xApps: state partitioning with partial replication as well as fast re-route with role awareness. These techniques were implemented as a library called RFT (RIC Fault Tolerance) that can be used by developers to build fault-tolerant xApps. Experimental results show that RFT meets the RIC latency requirements as well as provides scalable throughput up to hundreds of thousands of requests per second.

The rest of the paper is organized as follows. Section 2 gives an overview of xApps and a classification based on the state they maintain/require to perform their functions. Section 3 describes the proposed RFT library. The empirical evaluation is presented in Section 4. Section 5 presents an overview related work. Finally, Section 6 describes future work and provides concluding remarks.

II. Fault-tolerant xApps

In this section, we give a brief overview of the RIC and xApps, and then drill down into the problem of how to provide fault-tolerant high-throughput low-latency xApps.

A. RIC and the xApps

The RIC controls RAN elements via the E2 interface through which it can subscribe to specific RAN events (e.g., a new UE is attempting to connect the network) and to issue control messages (e.g., reject an attach request or hand-off a UE from one cell to another). Each RIC instance manages a potentially large number of RAN elements within a geographic region.

The control logic in the RIC is implemented by microservices called xApps. An xApp implements a well-defined function to access, monitor, and control RAN elements. Examples of xApp functionalities include DDoS prevention, Cell Selector Handover, Cell Congestion Prediction, and QoS optimization based on the UE class (e.g., police/firefighter devices, IoT devices, and connected vehicles). Each xApp interacts with the RIC platform components and other xApps using two main interfaces:

- RMR (RIC Message Router) - a messaging library that RIC components use to send and receive messages within a RIC instance. Message routing is policy-driven and based on message attributes such as its type, subscription id (described below), source and target RAN node. However, an xApp simply populates the message attributes and sends the message without needing to specify the target for the message. The RFT library takes advantage of this feature to provide transparent failover services.
- SDL (Shared Data Layer) provides a key-value store abstraction that allows xApps to read information provided by the RIC platform components and other xApps.

Fig. 1 illustrates the components of the RIC platform that are relevant for this work. The *E2 Term* module implements the interface to the RAN elements using the SCTP (Stream Control Transmission Protocol) transport protocol and ASN.1 (*Abstract Syntax Notation 1*) message payloads. The *Subscription Manager* implements the Publish-Subscribe model and allows xApps to subscribe for specific RAN events. The *Routing Manager* updates message routes on the RIC components. A specific xApp may be replicated and its n replicas are referred to as $xApp_{1..n}$. Note that the current RIC platform includes several other components but they are omitted here for clarity. The RIC is implemented as a Kubernetes [18] cluster and its components are deployed as pods consisting of one or more containers. The current release of the RIC (*Cherry*) does not support xApp replication.

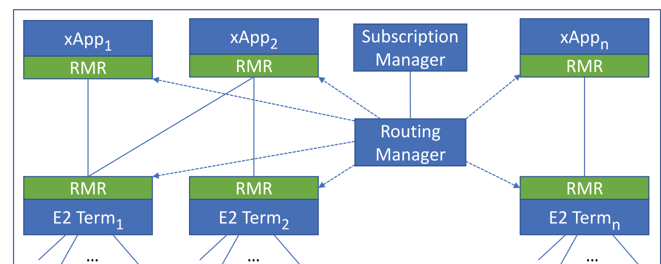

Fig. 1. RIC architecture: components.

B. xApp State Information

To simultaneously achieve its goals of fault tolerance, high throughput, and low latency, an xApp can be replicated and the multiple instances can run on separate hosts. A new xApp starts up in the RIC with a default initial state. The xApp will keep changing its state as it receives messages and executes its corresponding actions. In order to define a replication strategy, it is essential to identify which kind of state information an xApp needs to maintain to implement its function.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

xApps can be classified as stateless or stateful. A *stateless xApp* only requires information it receives in a single individual message to perform its function. This means that if the xApp is replicated, any replica can process any message without prior state information (i.e., all the required information is in the received message itself). As a result, these types of xApps can be replicated in a straightforward fashion and any load balancing algorithm can be applied to distribute traffic among the replicas. The basic requirement is that a faulty replica should be detected quickly so that traffic can then be re-routed to the remaining correct replicas.

A *stateful xApp* relies on state information to process a new message. The state of an individual replica is called its *context*. The state of an xApp consists of the contexts of all replicas of that xApp. A context includes information about the network elements managed by the particular xApp replica. Each context is xApp-logic dependent and is defined by the xApp writer. Examples of context information include: the id of a User Equipment (UE), of a group of UEs, a particular Cell, a Cell Site, a UE class (e.g., IoT device, cellphone), the total number of some type of device connected to the network, among others. The information kept in a context is structured through key-value pairs (e.g., consider a Cell context is for a Cell; signal strength and cell utilization can be the keys with their corresponding values).

A context can be stored and maintained in different ways. For example, it could be stored in the SDL of the RIC or replicated across all xApp instances. We argue that both of these solutions are prohibitively expensive for xApps. Replication techniques that rely on stateless xApps and externalize their state to a non-local store (e.g., SDL) cannot meet the challenges of high throughput and low latency in this problem domain. The reason is that the state needs to be fetched from the external store through the network, modified locally, and then committed in the external store. This increases the latency when compared to maintaining the state locally in the xApp. The extra latency is caused by the network round-trip times as well as the natural overhead of running the additional application to store the state. The latency can be even worse if it is required to lock resources to access and modify the externalized state. Full state replication would be equally expensive, since an overhead is imposed on the latency if it is necessary to wait for a majority of the replicas to apply their state updates.

Therefore, we propose to *partition* the state of an xApp in contexts and *partially replicate* them among a group of xApp instances. For example, for the Cell context state partitioning can be achieved very naturally by having each xApp instance responsible for processing all the messages from a given set of RAN elements. State partition allows each xApp instance to process each message it receives by using (reading and updating) only its local context information stored on local memory. To address fault tolerance, we propose partial replication where each replica X has a dedicated set of backup replicas (say Y and Z) where Y and Z maintain a copy of X's contexts and if X fails, either Y or Z can take over processing messages targeted to X, until X recovers and can resume its work. The xApp instance responsible for managing a given context is the *primary replica* (e.g., X), while the xApp instances that maintain copies of that context are called *backup replicas* (e.g., Y and Z).

Even partial replication can be too expensive for xApps if it is all done synchronously (i.e., any update is immediately processed by the primary and a majority of backup replicas). Thus, we propose asynchronous replication for xApp contexts (i.e., only the primary is updated and backup replicas are updated later). The design and implementation of the proposed partial replication strategy is described in Section III.

While replication ensures that the service is available even if an xApp instance fails, it does not ensure that the RIC will provide a timely response right after the failure. Therefore, we propose *fast re-route with role awareness* for xApps. Specifically, if the primary replica of a given context is not available to process a message from the RAN, the message is immediately re-routed to one of the backup replicas that maintain an approximate copy of that context. The backup replica that receives the request will be aware of its role as a backup for this particular request, and can use this awareness to process the message considering that its context information may not be fully up-to-date. In case the primary has crashed, one of its backup replicas is elected as the primary for that context.

III. The Proposed Solution

In this section we describe the proposed RFT (RIC Fault Tolerance) library that can be linked by any xApp that requires fault tolerance support. RFT employs group membership and a message routing strategy that adapts to the current membership. RFT is used in the same way as other RIC libraries, such as the RMR, SDL, and logging libraries. Note that simply relying on Kubernetes to add fault tolerance to xApps is not an alternative. While Kubernetes monitors pod health using health checks, these checks are not comprehensive in the sense that they do not allow fine-grained monitoring to detect changes on the composition of a group of xApp replicas. After a membership change is detected, replica roles (i.e., primary or backup) and the routing rules for the RAN messages that are to be sent to primary and backup replicas also have to be updated.

A. Group Membership and Routing Management

RFT relies on a group membership subsystem that is based on the Raft consensus algorithm [19]. Raft was employed to implement stateful replication of membership changes. Group Membership maintains information about both the group composition and request assignments, i.e., which xApps have been assigned to which contexts. Each group is implemented as a list of all replicated instances of a given xApp running on a single RIC instance. Raft maintains the list consistent on all members. We implemented Raft within the RFT library instead of using an external tool such as Zookeeper [20] due to performance concerns: a single RIC instance can potentially

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

run hundreds of different xApp groups, all with very strict latency and throughput requirements.

Fig. 2 shows how a new member is added to a group and the corresponding routing policies are updated. We assume that this particular context is for the RAN element. We also assume that the group initially consists of the RFT replicas $xApp_1$ and $xApp_2$. Replica $xApp_3$ is then added. Note that each xApp runs its corresponding RFT instance. Obviously an RFT group can also be initialized with a single replica, i.e., with no replication.

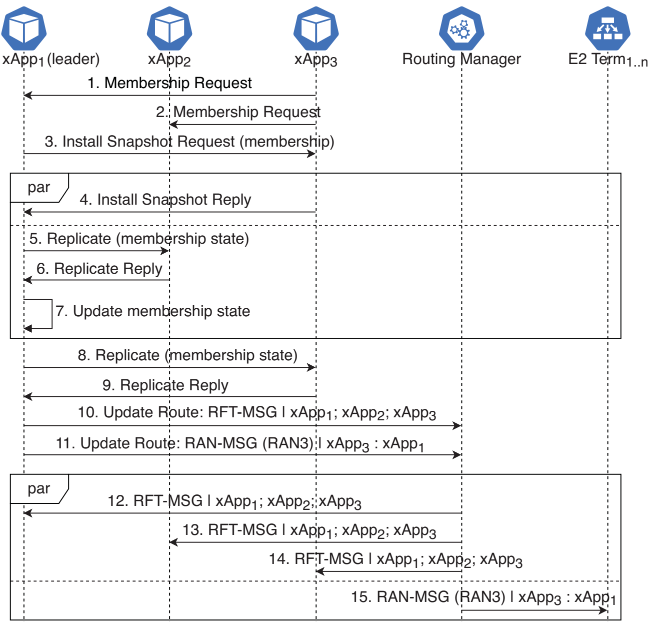

Fig. 2. A new replica is added to a group and the routing policies are updated.

We assume in Fig. 2 that the Raft leader ($xApp_1$) had been previously elected among the group members, and is monitoring the liveness of the other members, which are Raft followers. Liveness is monitored with heartbeat messages, which are omitted in the figure for brevity. When $xApp_3$ starts running, it sends membership request messages to all current members of the group (messages *1* and *2*) requesting to join the group. Upon receiving this message, the leader sends an *Install Snapshot Request* to $xApp_3$ with group state information (i.e., information that currently $xApp_1$ and $xApp_2$ are in the group). Whenever a new member joins a group it receives a *snapshot* from the leader, which consists of the current group state serialized. After installing the snapshot, $xApp_3$ replies to the leader acknowledging that the snapshot is installed and indicating that the leader can start monitoring liveness (message *4*). Meanwhile, the leader also replicates membership information to the rest of the group, informing the other members (in this case, just $xApp_2$) that $xApp_3$ has requested to join the group, as shown in message *5*. After the leader has received replies from a majority of the members (message *6*) it updates the local membership state (message *7*). Next, the leader will also replicate membership updates to $xApp_3$, which learns that it has joined the RFT group. Messages *8* and *9* are employed to replicate membership updates to $xApp_3$.

After $xApp_3$ has fully joined the cluster, the leader has to update the RMR routing policies to allow messages from RAN elements as well as RFT multicast messages to be routed to $xApp_3$. The new multicast route is set up by sending an *Update Route* message to the *Routing Manager* (message *10*). This update informs the RMR that all messages of type *RFT-MSG* should be delivered using multicast (i.e., indicated with the semicolon) to the RFT replicas running on $xApp_1$, $xApp_2$, and $xApp_3$. The pipe is used in this rule to separate the message type field from the group field. The RFT group membership subsystem employs this policy to send multicast messages from an RFT instance to *all* other RFT replicas of the same group.

Next, the leader also sends an *Update Route* message to the *Routing Manager* (message *11*). This policy update specifies that all messages of type *RAN-MSG* coming from RAN element *RAN3* should be sent to primary replica $xApp_3$, and in case it is unreachable, then the message should be re-routed (i.e., to backup replica $xApp_1$). Upon receiving those update route messages, the *Routing Manager* replaces the updated routing policies on the corresponding RFT members and RIC components. After the multicast policy is installed (messages *12-14*), copies of every message of type *RFT-MSG* are sent to $xApp_1$, $xApp_2$, and $xApp_3$. Upon the delivery of message *15* to the corresponding *E2 Term* components, all messages of type RAN-MSG from RAN3 are delivered to the primary replica $xApp_3$, they can instead be delivered to backup replica $xApp_1$ if $xApp_3$ is overloaded or unreachable.

After receiving a message with the new membership configuration, each RFT instance updates both the local membership and the information about which backup replicas keep which of its contexts. A primary selects the backup replicas for a given context based on the updated membership information. The next m available instances of the list of replicas are selected as the backup replicas, where m is the number of backup replicas.

In addition to group membership management, the Raft leader is also in charge of determining which replica is the primary for each new context. A new context corresponds for instance to a new UE joining the network. This new UE sends a message requesting to connect to some RAN element, which needs to invoke the corresponding admission control xApps. When a RAN element needs to communicate with xApps it publishes its messages through the *E2 Term* component. In order to select a primary for a new context, the *E2 Term* sends a message to all xApps that have subscribed to that type of message. After an xApp instance receives a message, it invokes RFT function *get_role* to determine its role with respect to the corresponding context. If the outcome is *None*, the xApp sends a message to the leader claiming to become the primary for the context. The leader, which keeps a counter for the number of times each instance is a primary, selects the one with the lowest counter as the primary for the new context.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

Then the leader updates and replicates the new membership information with information on the new primary to all RFT replicas. Upon receiving this information, the chosen replica becomes the primary for that context, selects the corresponding backup replicas and starts the context replication. The leader also requests the *Routing Manager* to update the routing policies for the corresponding RAN element to enforce that its RAN messages are delivered to the chosen primary replica (see messages *11* and *15* in Fig. 2).

B. Partial State Replication

Keeping fully replicated stateful xApps can be expensive and prevent xApps built with RFT to meet the performance requirements of the RIC. Thus RFT employs a partial xApp replication strategy that allows each primary to replicate its assigned contexts on a set backup replicas previously selected based on RFT membership information. The number of replicas and the replication frequency are configuration options.

The basic unit of data that is replicated is called a *log entry*, that corresponds to an operation on a context maintained by the primary. The primary maintains all log entries sequentially in memory using a structure which is called the *log*. A log entry consists of the following fields: sequence, command, context, key, value, and length. The sequence is an integer that is increased monotonically each time the primary adds a new log entry to the log. The sequence identifies each log entry that is replicated: both the primary and backup replicas keep track of the replicated log entries based on the sequence field.

A partial state replication scenario is illustrated in Fig. 3, where messages are exchanged among three RFT instances running on the corresponding xApps. In this scenario, we note that while the RFT instance in $xApp_1$ is replicating contexts to $xApp_2$, it is also a replica of $xApp_3$.

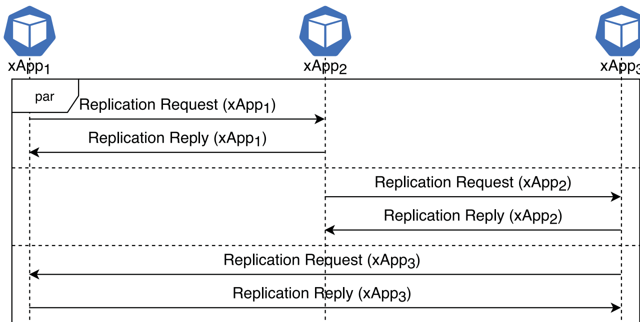

Fig. 3. Partial state replication: $xApp_2$ is a replica of $xApp_1$ and $xApp_3$ is a replica of $xApp_2$.

Each replication message can transport multiple log entries to reduce the overhead of replication messages, and to decrease the number of acknowledgments. After receiving a *Replication Request* message from the primary ($xApp_1$), a backup replica ($xApp_2$) checks the lowest sequence number of the log entries received in that message and the highest sequence number of the log entries it has received so far. This allows the backup replica to decide whether or not it needs to apply the log entries and update the corresponding contexts. After applying all log entries, the backup replica sends a *Replication Reply* message back to the primary acknowledging the replication. This information is then used by the primary to determine the next log entry to replicate to the backup. In case a backup replica receives a *Replication Message* with an unexpected sequence number, it discards the message and replies to the primary replica the sequence number of the next log entry it expects to receive.

The primary replica stores log entries in local memory. As the primary contexts are updated, more space is required to store more log entries. Eventually, if nothing is done, this may end up consuming all the available memory. Therefore, it is necessary to remove stale log entries and this is done with *log cleaning*. As soon as the combined sizes of log entries reaches a threshold, log entries are serialized in a so called *snapshot*. Basically, the idea of log cleaning is to delete all log entries that are no longer required to build the state of the last snapshot. Once a snapshot has been taken, all the prior updates – thus all log entries up to that point – can be deleted from the log. Furthermore, once a new snapshot is created the primary deletes the previous one. New log entries that have been created after the snapshot, are replicated to the backup as described above.

RFT implements state machine replication entirely in main memory, to minimize the latency to handle state operations. Despite the fact that all context state is lost in case either the whole RIC and all the replicas fail, state information becomes stale very quickly for RAN control applications, and thus in this case we claim it is better to reconstruct a fresh state rather than to use a stale one retrieved from persistent storage.

IV. Empirical Evaluation

The performance of RTF[1] was evaluated using a Cell Selector xApp that solves a practical problem in real cellular networks. The objective of this xApp is to pick up the best handover target cell for a User Equipment (UE) that improves the overall network performance. Specifically, we assume the RAN will generate UE handover requests when a UE is observing a neighboring cell with a stronger signal strength than its current serving cell. However, the fact that a cell has the strongest signal strength may not automatically make the cell the best handover destination for the UE. For example, the cell may be congested because it is serving a large number of UEs. Therefore, the Cell Selector xApp makes cell selection also considering cell utilization. This is done by having the Cell Selector keep track of cell attach and detach operations. A simple counter is used to keep track of the number of UEs connected to each cell. Note that a full Cell Selector could easily use additional information such as the throughput of the connected UEs, interference within the cell, and total transmit/receive power, which for our evaluation purposes are not really necessary.

[1]The implementation is available at https://github.com/alexandre-huff/rft

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

RFT is based on partial replication, which can be achieved in this case by having each replica keep track of the utilization of a subset of the cell sites. A cell site comprises the radio base stations and the network equipment installed at a given facility to transmit and receive mobile signals. Thus, the global state of the Cell Selector xApp can be partitioned into contexts, each representing a given cell site.

In the evaluation, we use the RFT library to replicate the cell site contexts an xApp primary replica maintains to another xApp backup replica. Due to fast re-route, the backup replica may receive messages related to the contexts managed by the primary, however, the backup is aware of its role and that cell utilization information may not be fully up-to-date. Because of this, the backup replica assigns a heavier weight to signal strength over cell utilization.

The Cell Selector xApp using RFT with an alternative based on the Redis database [21]. We employed Redis in the experiments due to two main reasons: Redis is currently used in the RIC to implement the Shared Data Layer (SDL), and it implements a highly available in-memory data structure store with low latency for data access, high throughput, built-in replication, as well as automatic state partitioning.

A. Experimental Setup

To run the experiments we set up a lab environment in which containers running xApps were executed on two hosts (called *server1* and *server2*) connected on a Gigabit Ethernet. We used Docker Compose to set up and run containers on each server. *Server1* ran Linux Ubuntu 18.04 on an Intel(R) Core(TM) i7-6700HQ @ 2.6 GHz processor with 4 cores, 12 GiB DDR4 RAM and 6144K L3 cache. *Server2* ran Linux Ubuntu 16.04 on an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor with 4 cores, 8 GiB DDR4 RAM and 8192K L3 cache. Five containers were executed, two on *server1* and three on *server2*. A single xApp executed on each container. Backup replicas are automatically selected by RFT as described in Section III-A.

In the Redis experiment, 5 xApps were distributed in the same way (2 xApps/containers on *server1* and 3 xApp/containers on *server2*). Now each xApp stores its state (i.e., context) on a Redis master replica, which executed locally with each xApp on the same host. Besides the 5 Redis master replicas, 5 backup replicas were executed on the remote host with respect to the original xApp instance.

We implemented a workload generator to quantify and compare RFT versus Redis in terms of latency and throughput. We ran five instances of the workload generator, each on the same physical machine as each xApp. Each workload generator directly sends and receives RMR messages to the corresponding xApp replica, and implements the equivalent combination of a RAN element and the *E2 Term* component. 50 rounds of 100K messages were sent both at (i) the maximum rate supported by the RMR on each machine and (ii) at a 1 microsecond interval.

B. Experimental Results

We evaluated the throughput of the Cell Selector xApp and results are shown in Fig. 4. xApp instances 1 and 2 were executed on *server1*, while xApp instances 3 to 5 were executed on *server2*. The reason to run 2 instances on *server1* and 3 instances on *server2* is that *server2* has a better CPU. For the RFT baseline measurement, we considered a single xApp instance (non-replicated) and one workload generator and we observed a throughput close to 125K msg/sec. The equivalent baseline xApp implemented using Redis reached a throughput of slightly over 48K msg/sec. Just to be clear, we recall that Redis is not saving data on disk, but the xApp had to access the context information on the remote Redis container.

We then measured the performance of the two systems in the context of partial state replication and compared the throughput of RFT and Redis. Fig. 4 shows that in both cases as the number of xApp instances increases, the overall system throughput also increases. Note that there is a slightly higher increase on the throughput after the third replica is added. The reason is that the third replica runs on *server2* which has better hardware. This behaviour happens for both RFT and Redis. Compared to the corresponding baseline, the RFT throughput increases by 91.16%, 238.4%, 336.42%, and 467% as the number of xApps increases to 2, 3, 4, and 5, respectively.

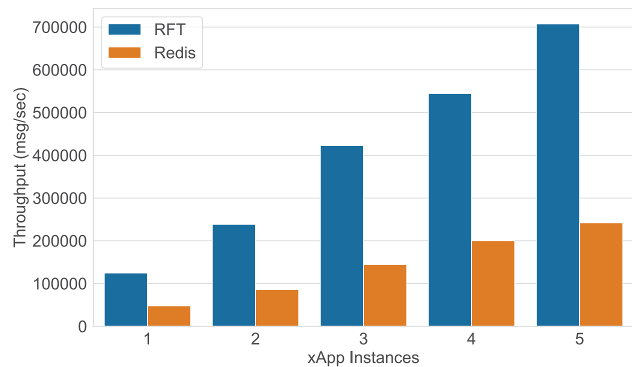

Fig. 4. Cell Selector throughput: RFT vs Redis.

Next, we ran an experiment to measure the round-trip latency perceived by an xApp user. We measured the time interval since a message is sent from the workload generator to the xApps, until the corresponding reply arrives back. We initially measured this latency by having each workload generator send messages in a slow pace: a single message is sent per microsecond. Fig. 5 shows the average latency with the 95% confidence interval computed after results were measured for 50 rounds during which 100K messages were sent, one per microsecond. It is clear that RFT presented lower average latency than Redis (RFT: 100 microseconds & Redis: 240 microseconds) but both can be considered low enough. Redis takes longer because it involves exchanging network messages. The figure also shows a slight decrease of the average latency when new replicas running on *server2* – which has better CPU – are added to the experiment. In the next experiment we show the results obtained when the message rate increases.

Fig. 6 shows the average latency obtained when message

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

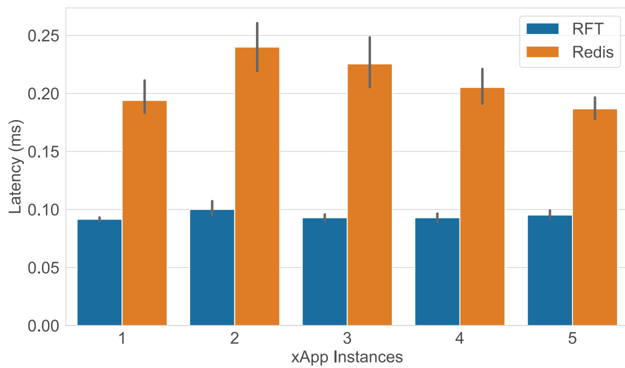

Fig. 5. Cell Selector average latency with a load of 1 message per microsecond.

rates increase to the maximum rates supported by the RMR on both *server1* and *server2*. Results are averages computed for 50 rounds of 100K messages and are shown with the 95% confidence interval. As can be seen in the figure, RFT was able to keep the latency low as the number of replicas increases. The latency measured for a single xApp was 0.944ms, as the number of xApps increases to 5 the average increases to 1.2ms.

For the same message rates, Redis reached on average more than 80ms of latency even when running a single xApp instance and its corresponding Redis master replica. This significantly higher latency is due to the fact that Redis was not able to sustain the rate of 125K requests per second per xApp. When Redis is used, the xApp instance must wait for Redis to reply before it sends a reply back to the workload generator. This causes both an increase of the latency and decreases the throughput in comparison with RFT. Note that this experiment is the same that was run to measure the throughput reported above. In this way it is possible to compare how throughput and latency fare as the number of replicas grows. Overall the results confirm that RFT meets the latency requirements of the RIC (less than 2ms) and provides scalable throughput, being able to process hundreds of thousands of requests per second.

V. Related Work

The problem of providing fault tolerance while guaranteeing low latency and high throughput has been addressed in different ways over the years. For example, LLFT (Low Latency Fault Tolerance) [17] is based on a leader-follower replication pattern. The system ensures strong replica consistency but with low overhead since the primary replica can reply without waiting for responses from backups, but message ordering established by the primary replica ensures the other replicas will reach the same state. While the system provides low latency, the throughput is limited to around 20K msg/sec. This somewhat low throughput can be explained by the fact that the state is fully replicated and the system cannot take advantage of concurrent processing by different replicas like in our approach where state is only partially replicated. Chain replication [22], achieves high throughput and high availability

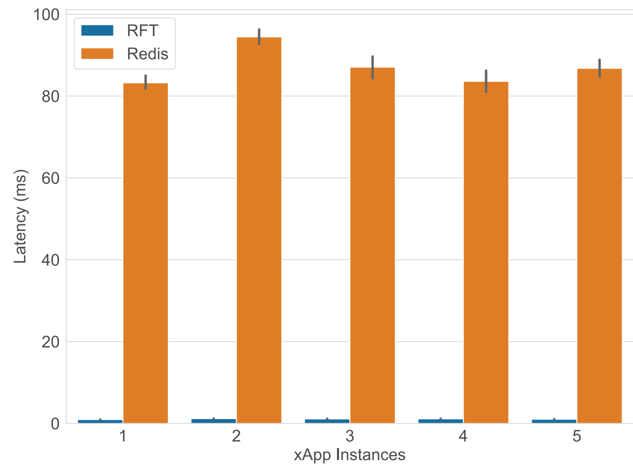

Fig. 6. Average latency for the maximum message rate supported.

at the same time reducing the number of messages required to ensure all replicas are updated. However, it requires the replicas in a chain to update one replica at a time and as a result the control loop latency can easily double even with 3 replicas.

The system described in [23] presents a high-availability solution for middleboxes. The strategy assumes stateful middleboxes that must have their state properly recovered after a failure. The system is based on classical rollback recovery, but in order to improve performance, it employs a lightweight logging strategy that ensures fast and correct recovery. Two loggers are employed, for both the input and output traffic. Periodic checkpoints are saved on the main memory of switches up- and downstream. Packets are not released until all the information needed for the retransmission of the packet is stored. As a checkpoint is taken, the function is frozen. The duration of the freeze, hence the impact on latency, is proportional to how much the state has changed between checkpoints, and is inversely proportional to the bandwidth required by storage.

In order to achieve scalable throughput, an alternative is to give up full replication and allow different replicas to process different subsets of messages and to store different states. In the database domain, sharding is an example of dividing the state of the database into different replicas [24]. The shards may be then replicated over many servers and can scale to very large numbers of concurrent reads as well as updates.

Slicer [25] allows data center applications to distribute their workload to a set of tasks through a sharding service. A task is an application process that runs concurrently with tasks from other applications on a multi-tenant host. Slicer uses keys chosen by applications to define how sharding is done, also to balance the workload across the tasks. The system is monitored to detect hot-spots as well as failures and does dynamic assignments in order to provide a highly available evenly distributed load. Slicer is more focused on load balancing than on high throughput and low latency,

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.

and sharding is done based on simple hash functions. RFT focuses on high throughput, low latency, and assigns requests from RAN elements based on the semantics of the problem. RFT also maintains the required state on local memory and replicates their updates, while Slicer stores the state in an external system such as Redis.

Related to the O-RAN effort in general, the OAI (Open Air Interface) and srsLTE are initiatives to provide open source implementations of the 3GPP standards for the RAN and the packet core [5], [6], [26]. The focus of those efforts is on the implementation of evolving standards and use cases, but to the best of our knowledge, in both projects there has been no work done on fault tolerance.

VI. Conclusions

In this work we addressed the problem of how to implement a fault-tolerant RAN controller that presents high sustainable throughput and low latency, under the 2ms threshold. Our solution relies on state partitioning with partial replication on groups of xApps and fast context-aware routing. The policy-based message routing strategy ensures that messages from the RAN elements are delivered to the right xApp replica that maintains the state needed to process the message. As a result, consensus is only needed for membership management. The group leader is responsible for requesting routing policy updates to the RIC. RFT was implemented and evaluated with a Cell Selector xApp. The solution was also compared with an alternative based on the Redis in-memory data store. Our experimental results show that RFT meets the RIC control loop latency requirements while sustaining scalable throughput, being able to process hundreds of thousands of requests per second.

Although we ran the RFT experiments in a lab environment with a limited number of replicas, we believe that state partitioning with partial replication and fast re-routing with role awareness as implemented in RFT also fits large-scale deployments. State partitioning allows a given xApp replica to be responsible for processing a subset of contexts, while partial replication allows keeping the replication overhead low as the number of replicas increases, even in large-scale deployments. Messages of a given context are delivered to the corresponding primary replica (or backup) by configuring particular routing policies in the RIC platform. RFT employs fast-reroute and is in charge of configuring the routing policies based on the current group composition.

Future work includes addressing xApp scale-out and scale-in, as well as dynamic context assignment employing improved load balancing techniques. RFT will also be evaluated with other xApps from the O-RAN community.

References

[1] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "Softran: Software defined radio access network," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2013, p. 25–30.

[2] M. A. Habibi, M. Nasimi, B. Han, and H. D. Schotten, "A comprehensive survey of ran architectures toward 5g mobile communication system," *IEEE Access*, vol. 7, pp. 70 371–70 421, 2019.

[3] T. O. Olwal, K. Djouani, and A. M. Kurien, "A survey of resource management toward 5g radio access networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1656–1686, 2016.

[4] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai, "A survey on low latency towards 5g: Ran, core network and caching solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3098–3130, 2018.

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

[6] F. Kaltenberger, G. de Souza, R. Knopp, and H. Wang, "The openairinterface 5g new radio implementation: Current status and roadmap," in *23rd International ITG Workshop on Smart Antennas (WSA)*, 2019.

[7] 3GPP, "The 3rd generation partnership project (3GPP)," 2020. [Online]. Available: https://www.3gpp.org/

[8] "O-RAN software community," 2020. [Online]. Available: https://wiki.o-ran-sc.org/

[9] "O-RAN Alliance," 2019. [Online]. Available: https://www.o-ran.org/

[10] C. Coletti, W. Diego, R. Duan, S. Ghassemzadeh, D. Gupta, J. Huang, K. Joshi, R. Matsukawa, L. Suciu, J. Sun, and et al, "O-RAN: Towards an open and smart RAN," O-RAN Alliance, White Paper, Oct. 2018. [Online]. Available: https://www.o-ran.org/s/O-RAN-WP-FInal-181017.pdf

[11] A. Akman, C. Li, L. Ong, L. Suciu, B. Y. Sahin, T. Li, P. Stjernholm, J. Voigt, A. Buldorini, Q. Sun, and et al, "O-RAN use cases and deployment scenarios: Towards open and smart RAN," O-RAN Alliance, White Paper, Feb. 2020. [Online]. Available: https://www.o-ran.org/s/O-RAN-Use-Cases-and-Deployment-Scenarios-Whitepaper-February-2020.pdf

[12] O-RAN Alliance, "O-RAN operations and maintenance architecture," O-RAN Alliance, Technical Specification O-RAN.WG1.OAM-Architecture-v03.00, Apr. 2020.

[13] ——, "O-RAN architecture description," O-RAN Alliance, Technical Specification O-RAN-WG1-O-RAN Architecture Description - v01.00.00, Feb. 2020.

[14] K. Bhardwaj, J. C. Miranda, and A. Gavrilovska, "Towards iot-ddos prevention using edge computing," in *Proc. HotEdge*, Jul. 2018.

[15] L. Liang, K. Zheng, Q. Sheng, and X. Huang, "A denial of service attack method for an iot system," in *Proc. 8th Int. Conf. on Information Technology in Medicine and Education (ITME)*, 2016, pp. 360–364.

[16] "O-RAN gerrit code repository," 2020. [Online]. Available: https://gerrit.o-ran-sc.org/r/admin/repos

[17] W. Zhao, P. Melliar-Smith, and L. Moser, "Low latency fault tolerance system," *The Computer Journal*, vol. 56, pp. 716–740, 06 2013.

[18] "Kubernetes," 2020. [Online]. Available: https://kubernetes.io/

[19] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX ATC*, Jun. 2014, pp. 305–319.

[20] Apache, "Apache Zookeeper," 2020. [Online]. Available: https://zookeeper.apache.org/

[21] "Redis," 2020. [Online]. Available: https://redis.io/

[22] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. OSDI*, 2004.

[23] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and et al., "Rollback-recovery for middleboxes," in *Proc. SIGCOMM*, 2015, p. 227–240.

[24] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[25] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari, "Slicer: Auto-sharding for datacenter applications," in *Proc. OSDI*, Nov. 2016, pp. 739–753.

[26] "OpenAirInterface: 5G software alliance for democratising wireless innovation," 2020. [Online]. Available: https://www.openairinterface.org/

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO PARANA. Downloaded on October 19,2021 at 17:43:10 UTC from IEEE Xplore. Restrictions apply.