Routing in Fat Trees: a protocol analyzer for debugging and experimentation

Maximiliano Lucero, Agustina Parnizari, Leonardo Alberro, Alberto Castro, Eduardo Grampín
Facultad de Ingeniería, Universidad de la República (UdelaR)
Montevideo, Uruguay
Email: {mlucero, agustina.parnizari, lalberro, acastro, grampin}@fing.edu.uy

***Abstract*—In the last decade, the Internet has been experiencing a gradual transition from a hierarchical network of networks to a content-driven network. This refactoring of the Internet's architecture is heavily based on the deployment of massive data centers whose basic services are computing, storage, and replication. This type of data centers may scale up to hundreds of thousands of servers, therefore, demanding a communication scale and bandwidth much higher than the historical Internet WAN traffic. Specific data center routing and forwarding solutions are needed to cope with the stringent communication requirements in this context. We are experimenting with different routing protocols to understand each one's characteristics, advantages, and disadvantages, focused on measuring convergence time and behavior under different failure scenarios. A key requirement for experimentation is to have the ability to decode protocol messages, typically analyzing control-plane traffic captures. While this is straight forward for established protocols such as BGP and IS-IS, it is challenging for newer, under development ones, such as RIFT (Routing in Fat Trees). In this work, we present an implementation of a RIFT packet dissector for Wireshark, which permits to analyze control-plane traffic under different scenarios. We tested the dissector with two different RIFT implementations, and it successfully dissected them both. Moreover, our analyzer can be easily extended to parse any RIFT generated Thrift code.**

***Index Terms*—data center, routing protocol, network-traffic analyzer, dissector.**

I. Introduction

Internet has been experiencing a gradual transition from a hierarchical network of networks, with strong Tier-1 and Tier-2 Transit Providers, to a content-driven network [1], given Content Providers' capacity to create user-targeted content and deploy the needed infrastructure to distribute it. There are different roles in this content-driven Internet; namely: *i*) *Content Delivery Networks* (CDNs), which are built over a mix of public and private infrastructure, seeking to replicate content as close to the user as possible for improving their Quality of Experience; *ii*) *Over The Top* (OTT) providers, which run media applications such as TV streaming, voice, and video calls; and *iii*) *Cloud Providers*, which provide virtualized resources for deploying online applications. These actors (which frequently play more than one role) have Points of Presence all over the world. They exchange two types of traffic: *i*) public traffic with their users; and *ii*) private traffic, mostly related to replicating content and application data storage. This refactoring of the Internet's architecture is heavily based on the deployment of massive data centers with similar basic functions. These functions include computing, storing, and replicating, using message exchange among servers over the supporting communication infrastructure. It is worth noticing that this type of data center may scale up to hundreds of thousands of servers. Therefore, conveying packets in these infrastructures (or among them and the Internet) demands a vast communication scale and bandwidth much higher than the historical Internet WAN traffic. Consequently, it is necessary to design and develop specific data center routing and forwarding solutions to meet such communication requirements.

978-3-903176-32-4 © 2021 IFIP

Data-center traffic is usually classified as East-West and North-South. East-West traffic refers to traffic between server racks, namely, the result of internal applications requiring data transfers. In contrast, North-South traffic refers to traffic as a result of external requests from the Internet.

A fundamental requirement for state-of-the-art data centers is to guarantee constant bisection bandwidth (i.e., the same capacity available for any-to-any communication among servers), leading to the resurgence of non-blocking Clos networks [2]. These networks are built up from multiple stages of switches, where each switch in a stage is connected to all the switches in the next stage, which provides extensive path diversity. A Fat-tree data-center topology is a particular case of a Clos network, where high bisection bandwidth is achieved by interconnecting commodity switches. The Fat-tree topology idea was originally proposed in [3] for supercomputing and has been adapted by [4] for data center networks. Further information can be found in [5] and [6].

Different routing algorithms for Fat-tree topologies have been proposed, namely BGP in the data center [7], Openfabric (IS-IS with flooding reduction) [8], and a couple of routing protocols under active development by the IETF. These new proposed protocols, Routing in Fat Trees (RIFT) [9] and Link State Vector Routing (LSVR) [10], seek to combine the valuable features from both link-state and distance-vector algorithms. Furthermore, many solutions based on Software Defined Networking (SDN) have been explored [11]–[13].

While a couple of implementations of RIFT are accessible [14], [15], ongoing LSVR implementations are still unavailable [16]. Trying to understand the different alternatives, we have been experimenting with the routing protocols mentioned above, mainly measuring convergence time and behavior under different failure scenarios.

General header	Packet Header	Packet Data	Packet Header	Packet Data	...

Fig. 1. Pcap file format

In order to enable experimentation with the RIFT protocol, in this article, to the best of our knowledge, we present the first implementation of a RIFT traffic analyzer.

The rest of the paper is organized as follows: in Section II, we briefly review traffic analysis tools. Then, in Sections III and IV, we describe the implementation of a RIFT dissector for Wireshark [17]. Finally, in Section V, we discuss some aspects of the implementation and possible future work.

II. Traffic analysis tools

Network protocols are designed for communication between network devices, and they define mechanisms to identify and establish connections, and formatting rules and conventions specified for data transfer. Traffic analyzers (also known as packet sniffers) are specific software tools that intercept and log network traffic traversing a network link by means of packet capturing. The captured packets can then be analyzed by decoding their raw data and visualized via displaying various fields to interpret the content, using, for example, the Wireshark tool [17].

Network packet analysis is usually helpful to gather and report network statistics, debug application behavior, and network forensics, typically using traffic aggregations based on the TCP/IP network headers; inspecting the packet payload may contribute to further investigate application behavior, and is particularly useful to identify (and intercept) security threats, such as DDoS (Distributed Denial of Service). Deep Packet Inspection (DPI) techniques are implemented in middleboxes such as Intrusion detection systems (IDSs).

The de facto standard capture format is *pcap*, implemented by the libpcap API, originally developed by the tcpdump team [18]. Pcap is a binary format, which general structure comprises a global header that contains the magic number (to identify the file format version and byte order), the GMT offset, the timestamp precision, the maximum length of captured packets (in octets), and the data link type. This information is followed by zero or more records of captured packet data. Each captured packet starts with the timestamp in seconds, the timestamp in microseconds, the number of octets of the packet saved in the file, and the actual length of the packet. The general structure is shown in Figure 1.

A comprehensive guide on practical usage of Wireshark and Tcpdump, including discussion of traffic formats and other tools, can be found in [19], [20].

III. RIFT dissector design and implementation

The RIFT dissector was designed as a Wireshark plugin packet dissector and implemented in C language following the API provided by this tool.

A. Design

RIFT packets are transported over UDP. The design of our dissector is guided by the RIFT packet security envelope, shown in Figure 2, which led us to define three stages in order to dissect the RIFT packet content:

1) *Outer Security Envelope Header*: here the dissector must identify a set of specified and static fields.
2) *TIE Origin Security Envelope Header*: as well as the previous stage, here the dissection must identify a set of specified and static fields. The only difference between this stage and the previous one is that this set of fields is present if and only if the RIFT packet type is a Topology Information Element (TIE).
3) *Serialized RIFT Model Object*: in this stage, the dissector must process a set of dynamic fields that follows the Thrift Binary protocol encoding[1] defined for RIFT.

The first two stages are designed with a classical approach, i.e., the dissector must follow a static definition that assigns a range of bytes to a field. The third stage represents a change of paradigm. Thrift is an interface definition language and binary communication protocol used for defining and creating services for numerous languages [21]. Thrift allows defining service interfaces and data-types in a specification file. The latter permits a dynamic operation of the services and applications, letting the modeler extend the data-types in a simple manner by modifying the definition file. This mechanism provides a rapid process for aggregating or modifying data.

Hence, with this new approach, a RIFT packet needs to be decoded knowing in advance how the data is specified in the Thrift definition files and how the data-types and structures are encoded. It is important to mention that there is a Thrift compiler that generates a Thrift decoder based on a given model. Consequently, we can identify two options to design the dissection of the Serialized RIFT Model Object: *i*) passing this part of the binary packet to a Thrift back-end; or *ii*) write the C code following the encoded defined in Thrift for the data-types involved. The latter needs to be done following the schema for information elements, whose Interface Definition Language (IDL) is Thrift.

Wireshark's user interface lets the user highlight some particular fields in the decoded packet. Furthermore, it highlights the corresponding bytes in the hex dump of the binary packet. To that end, the Thrift decoder that is used in Wireshark is required to (*a*) have knowledge of the precise order in which the fields were encoded in the binary message, that could potentially not be the same sequence as in the model; and (*b*) have knowledge of the correspondence between the bytes in the binary message and the fields in the decoded message. Currently, the generated code by the Thrift compiler does not produce this information, so the dissector presented on this paper follows option (*i*).

[1] https://github.com/apache/thrift/blob/master/doc/specs/thrift-binary-protocol.md

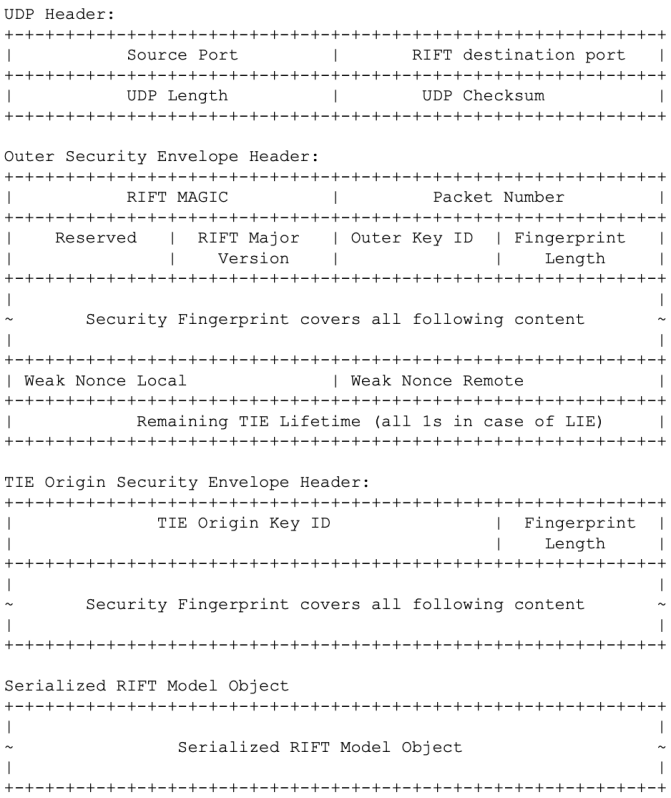

Fig. 2. Security Envelope, extracted from [9]

B. Implementation

The dissector is implemented in C language. Additionally, we coded a partial dissector for the outer security envelope header in Lua language following the Wireshark's Lua API reference. The implementation is open source and available at [22].

All the RIFT packets are conveyed on UDP. Hence, the presented implementation inherits the UDP header dissection and builds a complete dissection for the UDP payload.

First, the dissection of the security envelope header is processed. This implementation follows a static representation of the bits mapped to each field, e.g., the first four bytes represent the *RIFT Magic* value for the packet followed by the other four bytes that represent the *Packet Number* value.

Considering that the protocol was not associated with any particular range of ports at the time of the development of the dissector, it was implemented as a Wireshark heuristic dissector, i.e., the dissector recognizes a packet as RIFT if the field *RIFT Magic* contains the appropriate value defined as the hexadecimal `0xA1F7` in the current draft of the protocol.

After the *RIFT Magic* value check, the dissector is in charge of decoding all the fields of the security envelope header in the established order: *Packet Number*, *Reserved bytes*, *RIFT Major version*, and so on. Then, following the presented design, if the *TIE Origin Security Envelope Header* fields are included in the packet, they are identified.

At the end of this stage, the dissection process continues with the *serialized RIFT Model Object*, encoded with Thrift, is divided in a (header, payload) structure too. In the Thrift encoding, this tuple is defined as `PacketHeader` and `PacketContent` respectively. By dissecting the former, fields such as the protocol major and minor version, or the sender identification can be obtained. The `PacketContent` has the content of a RIFT packet, i.e., this structure can be filled with one of the packets defined in the protocol specification: Link Information Element (LIE), Topology Information Element (TIE), Topology Information Description Element (TIDE), or Topology Information Request Element (TIRE). It is worth remarking that the implemented dissector identifies and performs a complete dissection of all these types of RIFT packets.

TIE messages are exchanged between RIFT nodes to advertise the network topology (e.g., links and address prefixes). For instance, with the identification and dissection of all the TIEs exchanged between RIFT nodes, we can study the protocol convergence in a given topology.

LIE messages are equivalent to HELLOs in IGPs, and permit to observe the exchanged messages over all the links between systems running RIFT to form three-way adjacencies, as defined by the protocol draft.

Finally, RIFT nodes exchange TIDE and TIRE messages, which are equivalent to CSNP and PSNP (Complete and Partial Sequence Number PDU) in IS-IS, respectively.

Our dissector can fully decode all the aforementioned messages, which enables both to have a complete analysis of the behavior of the protocol and, since the implementation is work in process, serve as a debug tool for the RIFT developers.

To translate into bytes a given struct defined with the Thrift IDL, it must be considered that for each field a byte indicating the type must be decoded. Afterward, the decoding continues with the two bytes indicating its identifier, and finally its value. Finally, a null-byte indicates that we finished traversing that struct.

IV. Tool testing

In this section, we will present how the RIFT dissector works in a set of selected scenarios set up on the Fat-tree topology presented in Figure 3. The available RIFT implementations are, on the one hand, an open-source python project led by Bruno Rijsman [14] (for now on "rift-python"), one of the RIFT draft co-authors, and on the other hand, a proprietary implementation under development by Juniper [15]. We run the simulated scenarios, capturing rift-python network-control traffic, using the Kathará framework [23], [24], while Juniper implementation was tested using their own simulation environment. We will show and explain the output of relevant packets useful for debugging the protocol implementation and helping to understand its behaviour.

A. Bootstrap

The first scenario is aimed at verifying the standard behavior of RIFT at bootstrap. In this sense, when the fabric starts, the protocol in the first place tries to set up the adjacencies

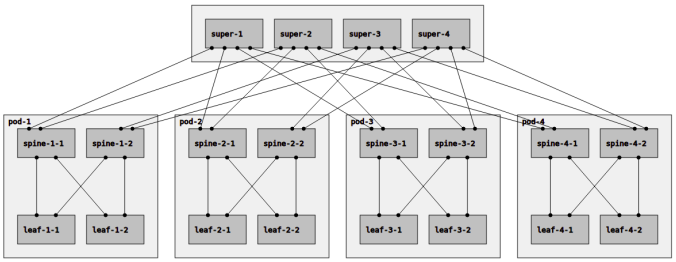

Fig. 3. Multi-plane Fat-tree topology

```
Routing in Fat Trees
  Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 2
      Reserved
      RIFT Major Version: 4
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 22812
      Weak Nonce Remote: 2221
      TIE Time-To-Live: 604630
    TIE Origin Security Envelope Header
        TIE Origin Key ID: 0
        TIE Security Fingerprint Length: 0
  Serialized RIFT Model Object
      Protocol major version: 4
      Protocol minor version: 1
      Sender: 102
      Level: 0
    Packet Type: Topology Information Element (4)
    Topology Information Element (TIE)
      Direction: North (2)
      Packet Originator: 102
      TIE Element Type: Prefix (3)
      TIE Number: 2
      Sequence Number: 1
    Prefix TIE
      Total prefixes: 1
        200.0.1.0/24
```

Fig. 4. RIFT TIE message dissection

between the nodes by interchanging LIEs packets, which have been completely dissected by our tool.

After the LIE interchanges, the nodes running RIFT are capable of start to share the routing information by sending TIE messages. The dissector recognizes the TIE security envelope and parses the serialized RIFT Model Object that contains the routing information (e.g., TIE prefixes).

With the dissection of this type of packets we are capable of debugging the implementation by, for instance, checking the correct values for the fields presented in Figure 2 or verifying the correctness of the prefixes distribution. The dissection of TIE messages is shown in Figure 4.

B. Link Failure

The Link Failure scenario permits to verify the ability of the RIFT protocol to converge after a single link failure. The link failure was particularly selected in order to cause the called "Fallen leaf problem". A "Fallen Leaf" is defined as a leaf that can be reached by only a subset, but not all, of Top-of-Fabric nodes due to incomplete connectivity. This particular scenario triggers a mechanism to prevent black-holing called "negative disaggregation". This feature was recently incorporated in the python-rift implementation and tested using our dissector.

```
Routing in Fat Trees
  Outer Security Envelope Header
  Serialized RIFT Model Object
      Protocol major version: 4
      Protocol minor version: 1
      Sender: 121
      Level: 24
      Packet Type: Topology Information Element (4)
    Topology Information Element (TIE)
      Direction: South (1)
      Packet Originator: 121
      TIE Element Type: Negative Disaggregation Prefix (5)
      TIE Number: 5
      Sequence Number: 110
    Prefix TIE
      Total prefixes: 3
        200.0.4.0/24
        200.0.2.0/24
        200.0.3.0/24
```

Fig. 5. RIFT negative disaggregation advertised prefixes

After a link failure that causes a fallen leaf, as shown in the Figure 5, we can observe how our dissector is capable of dissecting the negative-disaggregation messages. Notably, the negative disaggregation is advertised with a particular TIE Element in the TIE's header.

V. Closing Remarks

In this work, we presented an implementation of a RIFT Wireshark dissector, which was successfully tested with two different RIFT implementations.

Using our dissector, we have been able to parse the whole security envelope for the RIFT messages. This allows us to identify all RIFT packet types, including the negative-disaggregation ones, which are relevant for some specific types of failures in the Fat-tree fabric.

One of our dissector's main advantages is that it was designed to facilitate the addition of new RIFT Thrift models.

The dissector is being used to perform scalable RIFT experiments over the Kathara emulated environment, being very helpful to understand and debug the RIFT protocol implementation. Moreover, we are performing live classification and statistical analysis of RIFT messages using the Python module pyshark[2], which permits to use wireshark functionalities from Python code.

Acknowledgment

We would like to thank Bruno Rijsman, Antony Przygienda, and the whole RIFT working group for their insightful comments. We would also like to thank our colleagues from the Computer Networks research group at Università Roma Tre, Italy, for their invaluable assistance during the development of this work.

[2]https://github.com/KimiNewt/pyshark/

REFERENCES

[1] The death of transit? — APNIC Blog. [Online]. Available: https://blog.apnic.net/2016/10/28/the-death-of-transit/

[2] C. Clos, "A study of non-blocking switching networks," *The Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, March 1953.

[3] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, Oct 1985.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967

[5] D. Medhi and K. Ramasamy, *Network Routing, Second Edition: Algorithms, Protocols, and Architectures*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[6] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 909–928, Second 2013.

[7] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for Routing in Large-Scale Data Centers," IETF, RFC 7938, Aug. 2016. [Online]. Available: http://tools.ietf.org/rfc/rfc7938.txt

[8] R. White, S. Hegde, and S. Zandi, "IS-IS Optimal Distributed Flooding for Dense Topologies," Internet Engineering Task Force, Internet-Draft draft-white-lsr-distoptflood-00, Nov. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-white-lsr-distoptflood-00

[9] T. Przygienda, A. Sharma, P. Thubert, B. Rijsman, and D. Afanasiev, "RIFT: Routing in Fat Trees," Internet Engineering Task Force, Internet-Draft draft-ietf-rift-rift-12, May 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rift-rift-12

[10] K. Patel, A. Lindem, S. Zandi, and W. Henderickx, "Shortest Path Routing Extensions for BGP Protocol," Internet Engineering Task Force, Internet-Draft draft-ietf-lsvr-bgp-spf-11, Aug. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-lsvr-bgp-spf-11

[11] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *Commun. ACM*, vol. 59, no. 9, p. 88–97, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2975159

[12] Segment routing - line data center networking with srv6. [Online]. Available: https://www.segment-routing.net/conferences/2019-09-20-SRv6-LINE-DC/

[13] A. Abouchaev, T. LaBerge, P. Lapukhov, and E. Nkposong. BrainSlug: A BGP-Only SDN Controller for Large-Scale Data-Centers. [Online]. Available: https://archive.nanog.org/sites/default/files/wed.general.brainslug.lapukhov.20.pdf

[14] B. Rijsman, "Routing In Fat Trees (RIFT)," https://github.com/brunorijsman/rift-python, 2019, accessed: May 2020.

[15] Juniper, "RIFT in JunOS," https://www.juniper.net/documentation/en_US/junos/topics/topic-map/rift-in-junos.html.

[16] P. Sarkar, K. Patel, J. Networks, and sajibasil@gmail.com, "BGP Shortest Path Routing Extension Implementation Report," Internet Engineering Task Force, Internet-Draft draft-psarkar-lsvr-bgp-spf-impl-00, Jun. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-psarkar-lsvr-bgp-spf-impl-00

[17] A. Orebaugh, G. Ramirez, J. Beale, and J. Wright, *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing, 2007.

[18] Tcpdump/libpcap public repository. [Online]. Available: http://www.tcpdump.org/

[19] L. F. Sikos, "Packet analysis for network forensics: A comprehensive survey," *Forensic Science International: Digital Investigation*, vol. 32, p. 200892, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287619302002

[20] C. Chapman, "Chapter 7 - using wireshark and tcp dump to visualize traffic," in *Network Performance and Security*, C. Chapman, Ed. Boston: Syngress, 2016, pp. 195 – 225. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978012803584900007X

[21] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Facebook, Tech. Rep., 4 2007. [Online]. Available: http://thrift.apache.org/static/files/thrift-20070401.pdf

[22] RIFT dissector. [Online]. Available: https://gitlab.com/fing-mina/datacenters/rift-dissector

[23] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Battista, "Kathará: A container-based framework for implementing network function virtualization and software defined networks," 04 2018, pp. 1–9.

[24] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A lightweight network emulation system," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2.