MRT#: a Fast Multi-Threaded MRT Parser

Lorenzo Ariemma, Mariano Scazzariello, and Tommaso Caiazzi
Roma Tre University – Rome, Italy

***Abstract*—BGP is the inter-domain routing protocol of the Internet. BGP routers exchange BGP Updates, and adjust their routing table to reflect changes in the network. A wide variety of research and operational projects leverage on massive processing of BGP Updates, so it is crucial to analyse such data in the most efficient way. Hence, different MRT parsers have been developed. Most of them are unsuitable for big data analyses due to various limitations. In this paper, we present MRT#, a multi-threaded MRT parser library written in C#. We show its architecture, a performance comparison with other MRT parsers, and its possible integration into a data processing pipeline.**

***Index Terms*—BGP, MRT, Packet analysis.**

I. Introduction

BGP [1] is the inter-domain routing protocol of the Internet. BGP routers exchange *BGP Updates*, and adjust their routing table to reflect physical or administrative changes in the network. A wide variety of research and operational projects leverage on massive processing of BGP data (e.g. [2], [3]). Such data could reflect network operators' internal policies and configurations, hence they are rarely published. However, there are at least four large-scale BGP archival projects freely accessible: (i) the University of Oregon Route Views Project [4], (ii) the RIPE NCC Routing Information Service (RIS) [5], (iii) the Isolario Project [6], and (iv) the Packet Clearing House (PCH) [7]. All these projects share similar architecture. They create a BGP session from each router they want to monitor to a so called *Route Collector*. Route Collectors are servers that mimic a BGP router and, instead of computing the best route, dump all the received BGP messages. BGP Updates, together with other control-plane messages, are stored using the MRT format [8].

In order to analyze this huge amount of information, several MRT parsers have been developed. Based on [9], there are eight currently available open-source MRT parsers. Most of them are not able to directly manipulate parsed data as software objects, or do not expose default mechanisms for multi-threading, making them unsuitable for big data analyses.

In this paper, we present MRT#, a multi-threaded MRT parser library written in C#. We illustrate its architecture and show a performance comparison with other existing parsers. After that, we demonstrate how MRT# can be easily integrated into a data processing pipeline.

II. MRT# Overview

MRT# is a library written in C#, following the Object-Oriented Programming approach. All the code is open source and available at [10]. Its architecture, depicted in Fig. 1, is heavily based on the *Pipes and Filters* [11] architectural pattern. This pattern defines an architecture for processing a stream of data in which each step is encapsulated in a different component. Each step takes an input, computes a partial result and outputs it to another component. The output of the last filter is the final result of the entire computation.

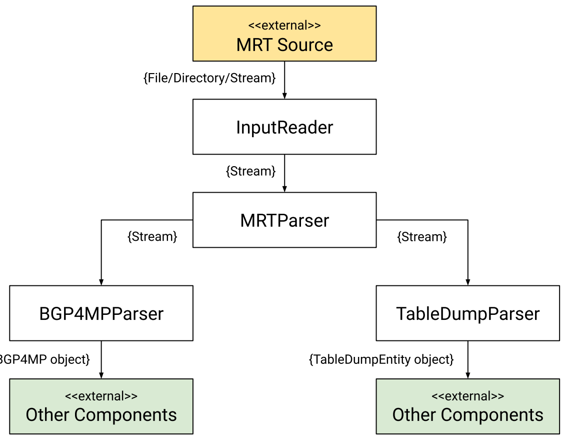

Fig. 1. MRT# Architecture.

MRT# architecture consists of four main components: (1) InputReader; (2) MRTParser; (3) BGP4MPParser; (4) TableDumpParser. The input of the system is a file, a folder or a stream containing one or more MRT records. The input enters the InputReader component, that can read different MRT formats, either compressed (GZip or BZip2) or uncompressed. The InputReader component, based on the type of the received input, reads MRT records and passes it to the MRTParser component. It parses the MRT header and, based on its type, delegates the computation to either the BGP4MPParser or the TableDumpParser component. The output of these components is the final result: a software object representing an MRT record. This object can be handled by other components specified by the user for further processing.

The MRT# architecture enables a multi-threaded execution, since the MRTParser component independently parses a set of MRT records for each thread. Furthermore, the architecture presents a high level of modifiability. In fact, each parsing task (e.g. dissecting the BGP Header) is implemented in a different class. Hence, it is easy to add functionalities to the library, extending the set of parsable fields (e.g. adding the BGP EVPNs SAFI support).

Finally, MRT# is a software library and not a CLI tool. So, it can be used in any project to parse MRT records and to directly feed other platforms, such as big data analysis frameworks like

978-3-903176-32-4 © 2021 IFIP

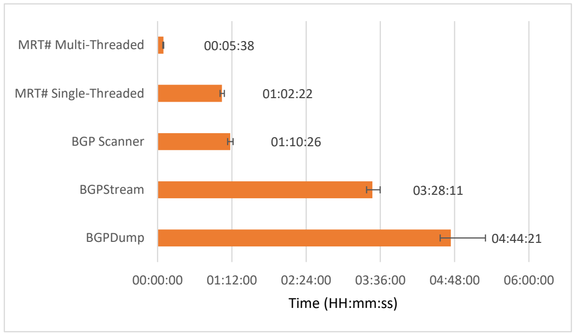

Fig. 2. Benchmark Average Processing Time

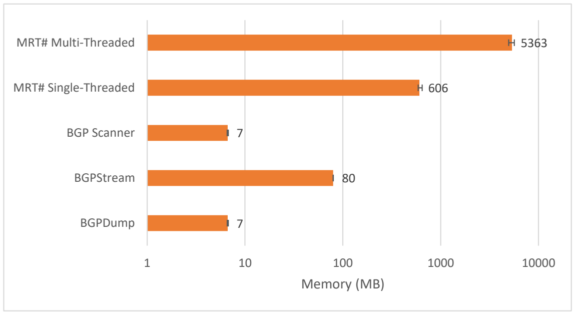

Fig. 3. Benchmark Maximum Memory Usage

Apache Spark, Hadoop or Kafka.

III. Comparison with other MRT Parsers

We choose to compare MRT# with the main contributions of the ones cited in [9]: (i) BGPdump [12], from RIPE NCC [13], (ii) BGPStream [14], from CAIDA [15], and (iii) BGP Scanner [16], from the Isolario Project. These tools are able to parse the main MRT packet types and the main BGP Path Attributes [17], with no significant differences. However, they parse a small subset of the BGP Path Attributes with respect to MRT#, so their result could miss some crucial information.

The tools also differ for flexibility and resources usage. About flexibility, all of them expose a command line interface, that takes an MRT file as input (both compressed and uncompressed), and prints the MRT content directly to standard output. So, in order to manipulate extracted data, it is required to parse again the standard output result and load it into another software. Of course, it is possible to analyse data with classic Linux tools (e.g. grep, awk) but they do not provide all the features of a programming language. On the other hand, MRT#, being a library, outputs software objects that can be directly handled.

To analyse the resources usage, we compared MRT# against the other tools. For each parser, we measure the elapsed time and memory usage. The benchmark consists of one month (December 2020) of MRT files collected from RIPE RIS RRC00, for a total amount of about 1 963M BGP messages distributed into 8 928 compressed files (49 GB).

To be fair with the other tools, that are single-threaded, we execute two runs of MRT#, one with the default multi-threaded configuration and another with a single-threaded configuration. Additionally, in order to reduce possible performance gaps introduced by printing on standard output, we discard all the resulting output of other tools.

All tests are performed on a Debian Buster server with 24 CPUs and 64GB of RAM. Each test is repeated ten times and both Fig. 2 and Fig. 3 show the average results with their confidence intervals.

Fig. 2 depicts the elapsed time to parse all the MRT files, while Fig. 3 shows the maximum memory usage of the process. In Fig. 2, it is possible to observe that MRT#, in its multi-threaded configuration, is about twelve times faster than BGP Scanner, the fastest of the other tools. Instead, the single-threaded version shows comparable performance with it. However, MRT# has the highest memory consumption, since it loads the whole uncompressed MRT file into memory before parsing it. For example, an uncompressed MRT file containing BGP Updates from RIPE RIS is about 50MB. Besides that, this approach leads to better performance over higher memory usage, that is still reasonable for modern hardware.

IV. Demonstration

We will demonstrate how MRT# can be easily integrated into a data processing pipeline. We will show how to connect MRT# to an arbitrary source of MRT records. After that, we will illustrate the correctness of parsed software objects, comparing them with the result of the other tools. Then, we will demonstrate how it easily possible to process MRT records, manipulating the parsed objects with a sample of user-defined functions, such as simple queries for counting received BGP Updates for a specific prefix, or for finding the Updates containing a specific AS Number in their AS path.

References

[1] Y. Rekhter, T. Li, and S. Hares, “A Border Gateway Protocol 4 (BGP-4),” RFC 4271, IETF, Tech. Rep., 2006.
[2] M. Candela, G. Di Battista, and L. Marzialetti, “Multi-view routing visualization for the identification of BGP issues,” *Journal of Computer Languages*, 2020.
[3] L. Ariemma, S. Liotta, M. Candela, and G. Di Battista, “Long-lasting Sequences of BGP Updates,” in *International Conference on Passive and Active Network Measurement*. Springer, 2021.
[4] “Route Views Project,” http://www.routeviews.org/.
[5] “RIPE-RIS,” https://ris.ripe.net.
[6] “Isolario Project,” https://www.isolario.it/.
[7] “Packet Clearing House,” https://www.pch.net/.
[8] L. Blunk, C. Labovitz, and M. Karir, “Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format,” RFC 6396, IETF, Tech. Rep., 2011.
[9] “New MRT-BGP reader six times faster than its predecessors,” https://blog.apnic.net/2018/11/29/new-mrt-bgp-reader-six-times-faster-than-its-predecessors/.
[10] “MRT#,” https://gitlab.com/uniroma3/compunet/networks/mrtsharp.
[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2*. Wiley, 2000.
[12] “BGP Dump,” https://bitbucket.org/ripencc/bgpdump/src/master/.
[13] “RIPE Network Coordination Centre,” https://ripe.net.
[14] “BGP Stream,” https://github.com/caida/bgpstream.
[15] “CAIDA: Center for Applied Internet Data Analysis,” https://caida.org.
[16] “BGP Scanner,” https://gitlab.com/Isolario/bgpscanner.
[17] “BGP Path Attributes,” https://www.iana.org/assignments/bgp-parameters/bgp-parameters.xhtml#bgp-parameters-2.