Serialization Analysis of Data Transmission in Radio Telescope Control Systems (Postprint)

Authors: Li Jun, Liu Zhiyong, Wang Na, Song Yining, Yang Lei, Wang Jili

Date: 2021-03-04T00:00:00+00:00

Abstract

The control system can interface, integrate, and manage the software and hardware systems of radio telescopes. Serialization tools within the control system can encode and decode information transmitted among different devices, operating systems, programming languages, and networks in radio telescopes, thereby enhancing data transmission efficiency between systems. This paper analyzes and compares the encoding principles and characteristics of three binary serialization tools—Msgpack, Protobuf, and Flatbuffers, and evaluates their serialized data size, serialization time, and CPU utilization through a case study. The results demonstrate that Msgpack' s overall performance is superior to that of Protobuf and Flatbuffers, making it suitable for encoding and decoding information transmitted between radio telescope systems with long development cycles and evolving requirements.

Full Text

Serialization Analysis of Data Transmission in Control Systems of Radio Telescopes

Li Jun[12], Liu Zhiyong[13], Wang Na[13], Song Yining[12], Yang Lei[12], Wang Jili[1]

[1] Xinjiang Astronomical Observatory, Chinese Academy of Sciences, Urumqi 830011, China
[2] University of Chinese Academy of Sciences, Beijing 100049, China
[3] Key Laboratory of Radio Astronomy, Chinese Academy of Sciences, Nanjing 210008, China

Abstract: The control system serves as a critical component that connects, integrates, and manages the software and hardware systems of radio telescopes. Serialization tools within the control system encode and decode information

transmitted between different devices, operating systems, programming languages, and networks in radio telescopes, thereby enhancing data transmission efficiency between systems. This paper analyzes and compares the encoding principles and characteristics of three binary serialization tools—Msgpack, Protobuf, and Flatbuffers—and tests their serialized data size, serialization time, and CPU utilization through a practical example. The results demonstrate that Msgpack offers superior overall performance compared to Protobuf and Flatbuffers, making it suitable for encoding and decoding transmission information between radio telescope systems with long development cycles and evolving requirements.

Keywords: radio telescope; binary serialization tool; control system; encoding; decoding

0 Introduction

Radio telescopes are the cornerstone of radio astronomy research, comprising systems such as antennas, receivers, terminals, monitoring, and control. Among these, the control system with connection, integration, and management functions constitutes an essential component of radio telescopes. Data exchange represents a fundamental function of control systems, which must ensure stability and reliability while maintaining efficiency and universality in telescope control and multi-terminal data exchange. For the QTT (Qitai 110-meter Radio Telescope) to be constructed in Qitai County, Xinjiang, the communication data size between devices depends on the observation band and mode. The single-communication data volume between them is generally less than 1KB. The antenna servo control has the most frequent data communication, with an exchange frequency of approximately 20Hz and a single data exchange size of about 200B. Other devices have a data exchange frequency of about 1Hz. When the active surface is operational, its data communication volume is approximately 10KB, while electromagnetic monitoring typically generates 10KB–100KB of data. The QTT control system plans to adopt a distributed architecture, with data exchange between subsystems encompassing multiple approaches, such as information transmission between Linux, Windows, VxWorks, Unix, and embedded systems; information transmission between network big-endian mode and machine little-endian mode; and information exchange between C++ and Python.

When machines with big-endian and little-endian architectures transmit information, the bytes of long-type data are swapped. To address data exchange format issues between systems, serialization tools encode transmission information between radio telescope systems into a unified format. Therefore, serialization tools serve as the foundation for transmission information formats in control systems, enabling information transfer between software and hardware systems of radio telescopes.

Existing radio telescope control systems predominantly employ serialization

technology. For instance, the ALMA control system architecture combines the text serialization tool XML as the foundation for encoding and decoding control system transmission information. The ASKAP monitoring and control system utilizes serialization technology provided by the Internet Communications Engine (ICE) to handle transmission information formats between different radio telescope hardware and software systems. The GMRT monitoring and control system is based on the Tango control system, combined with XML to implement system transmission information encoding and decoding. To address information transmission format issues between telescope control systems, Deng Hui et al. proposed a communication framework for telescope automatic control systems based on the communication middleware ZeroMQ and the text serialization tool JSON.

Regarding serialization tools, XML and JSON are text-based serialization tools widely used in internet software systems and for data exchange between application and service layers in early radio telescope control systems. However, in radio telescope control system applications, XML and JSON exhibit certain limitations, such as high memory usage, potential loss of data type precision, and difficulty in achieving data exchange between 底层 device drivers and services. Consequently, binary serialization tools such as Msgpack, Protobuf, and Flatbuffers have gradually replaced text-based tools for communication between 底层 and service layers in experimental physics facilities and radio telescopes, as they better address issues like data precision loss and exchange efficiency between 底层 and service layers. This paper focuses on analyzing the encoding principles and characteristics of three binary serialization tools—Msgpack, Protobuf, and Flatbuffers. Through testing and comparison of their serialized data size, serialization time, and CPU utilization, and considering 底层, service layer, and application layer requirements, we aim to select a suitable serialization tool for radio telescope control systems to improve information transmission efficiency and ensure uniformity and compatibility of information transmission formats across radio telescope systems.

1 Serialization Tools

Serialization tools consist of encoding (also called serialization) and decoding (also called deserialization). Serialization encodes structured data (or objects) into byte streams, while deserialization restores byte streams to their original structured data (or objects).

When building radio telescope control systems using serialization tools, it is necessary to analyze their encoding principles and characteristics, as different encoding methods affect serialized data size, serialization time, and CPU utilization. The subsequent sections of this chapter will analyze the encoding principles of Msgpack, Protobuf, and Flatbuffers using the JSON data shown in Figure 1 [Figure 1: see original paper] as an example.

1.1 Msgpack

Msgpack (also known as MessagePack) is a multi-language, cross-platform binary serialization tool with dynamic compilation capabilities. The byte stream after encoding objects (or structured data) is compact and concise, consisting of header bytes, prefix bytes, and data bytes. Header bytes indicate the data type and number of types that follow; prefix bytes indicate the subsequent data type; and data bytes represent the content of objects, such as basic types (bin, float, uint), structured types (str, array, map), and extended types (ext, fixext). Notably, the string type (str) does not use any markers (or escape characters) to indicate content termination. Msgpack employs two encoding methods:

The first method encodes key-value pairs using Msgpack (abbreviated as MSGP-M), which requires encoding the key first, followed by the value.

Figures 2 [Figure 2: see original paper] and 3 [Figure 3: see original paper] respectively illustrate the logical diagram and byte stream diagram after MSGP-M encodes the JSON data from Figure 1. The logical diagram represents the data structure after encoding, while the byte stream diagram shows the sequential order of bytes after encoding. For example, 83 is the first byte. The byte stream comprises seven parts: 1) The first byte (83) has its first four bits (1000) indicating the data type as map, and the last four bits (0011) indicating three map objects follow; 2) The second byte is the prefix byte for the key of the first map object (A5), indicating five string objects follow. Bytes 3–7 represent the key value "names" in ASCII; 3) Byte 8 (A5) indicates five strings follow. Bytes 9–13 use ASCII to represent the string "zhang"; 4) Byte 14 represents the prefix byte for the key of the second map object (A3), indicating three strings follow. Bytes 15–17 represent the string "num" in ASCII; 5) Byte 18, the prefix byte for the value of the second map object (CD), indicates an unsigned integer of two bytes follows. Bytes 19–20 represent the number 1331 in big-endian mode; 6) Byte 21 represents the prefix byte for the key of the third map object (A8), indicating eight strings follow. Bytes 22–29 use ASCII to represent the string "descript"; 7) Byte 30 (AF) is the prefix byte for the value of the third map object, indicating 15 strings follow. Bytes 31–45 represent the string "inthefirstnicks".

The second method uses Msgpack to encode only the values in JSON format, with serialization results represented as arrays. This Msgpack variant is abbreviated as MSGP-D.

After MSGP-D encodes the JSON format from Figure 1, the logical diagram and byte stream diagram correspond to Figures 4 [Figure 4: see original paper] and 5 [Figure 5: see original paper], respectively, comprising four parts: 1) The first byte (0X93) indicates three array objects follow; 2) The second byte (0XA5) indicates five strings follow. Bytes 3–7 use ASCII to represent the string "zhang"; 3) Byte 8 (0XCD) indicates a 16-bit unsigned integer follows. Bytes 9–16 represent the number 1331 in big-endian binary form; 4) Byte 11 (0XAF) indicates 15 strings follow. Bytes 12–26 use ASCII to represent the string "inthefirstnicks".

1.2 Protobuf

Protobuf (PB) is an open-source, multi-language, cross-platform binary serialization tool that provides an Interface Description Language (IDL). When Protobuf encodes transmission information, it is necessary to define keys and fields (data types) in the IDL to generate code for specific programming languages such as C++ and Python. After encoding data with Protobuf, the resulting byte stream consists of keys, prefix bytes, and data bytes. Keys are divided into field numbers and wire types. Field numbers mark frequently used (or repeated) elements as 1–15 and less common elements as 16–2047. Wire types include string, float, etc. Keys uniquely mark data types in IDL files, and marked data types cannot be changed.

Figures 6 [Figure 6: see original paper] and 7 [Figure 7: see original paper] respectively show the logical diagram and byte stream diagram after Protobuf encodes the JSON data from Figure 1. The byte stream occupies 27 bytes and consists of three steps: 1) Bits 2–5 of the first byte (0A) (0001) represent the field number, while the last three bits (010) indicate the wire type as string. The second byte (05) indicates five string objects follow. Bytes 3–7 represent the string "zhang" in ASCII; 2) Byte 8 (10) indicates an integer follows. Bytes 9–10 represent the 16-bit unsigned integer 1331 in little-endian mode; 3) Byte 11 (10) indicates a string follows. Byte 12 (0F) indicates 15 strings follow. Bytes 13–27 represent the string "inthefirstnicks" .

1.3 Flatbuffers

Flatbuffers (FB) is a multi-language, cross-platform binary serialization tool that provides IDL. Flatbuffers offers excellent compatibility: when new features are added, new fields can only be appended to the end of the IDL file, and old fields remain readable; the data format in memory is consistent with the encoded format; and the deserialization process supports "zero-copy" for rapid data reading. Flatbuffers serialized byte streams include scalars (int, string) and vectors (struct, table). Scalars consist of fixed-length integers (8-bit to 64-bit) and floating-point numbers represented in little-endian mode. Vectors consist of strings and arrays, beginning with a 32-bit VECTOR SIZE indicating the vector length—excluding '\0' and its own space. The only difference between strings and arrays is that strings include a terminator '\0' .

Figure 8 [Figure 8: see original paper] shows the result of Flatbuffers encoding the data from Figure 1. The byte stream occupies 62 bytes and consists of seven parts: 1) Bytes 1–4 represent the root offset (10 00 00 00), which offsets 16 bytes to the encoded data. Bytes 5–6 (00 00) serve as alignment padding; 2) Bytes 7–8 (0A 00) represent the vtable size in bytes, including space occupied by vtable size, object size, offset num, offset descript, and offset names. Bytes 9–10 (10 00) represent the object size offset, indicating the space offset for storing data in the table, including vtable offset, int offset, 1331, and string offset. Bytes 11–12 (04 00) represent the offset for num, indicating that offset only needs to move 4

bytes to find num' s offset. Bytes 13–14 (08 00) represent the offset for descript, which can be found through vtable offset and int offset. Bytes 15–16 (0C 00) represent the offset for name, indicating the name object follows after vtable offset, int offset, and string offset; 3) Bytes 17–20 (0A 00 00 00) represent the vtable offset, which has the same size as vtable size, except the latter occupies 2 bytes. Bytes 21–28 represent the prefix for num and the number 1331 in little-endian mode. Bytes 29–32 (0F 00 00 00) indicate the subsequent type is string. Bytes 33–36 (0F 00 00 00) indicate 15 strings follow. Bytes 37–52 contain 15 ASCII characters representing "inthefirstnicks" and a terminating character "\0" . Bytes 53–62 follow the same encoding principle as bytes 33–52.

1.4 Analysis and Comparison of Encoding Principles

Among the three binary serialization tools, Msgpack does not use IDL to predefine data structures and allows manual field writing with two encoding methods. Protobuf and Flatbuffers define transmission information fields in IDL, use IDL compilers to generate corresponding programming language interfaces, and have only one encoding method. The three tools employ different encoding principles, resulting in different byte stream sizes after encoding. Msgpack-encoded byte streams contain only header bytes and corresponding prefix and data bytes. Protobuf byte streams contain corresponding keys, prefix bytes, and data bytes, where only data types use keys and data bytes. Flatbuffers serialized byte streams are consistent with data storage formats in memory. Flatbuffers byte streams include not only prefix and data bytes but also non-content bytes such as root offset, object size, and vtable offset. When encoding the same data format, MSGP-M serializes all data in JSON format, resulting in larger space occupation. MSGP-D produces the most compact byte stream, with only header and prefix bytes not representing information. Protobuf occupies slightly more space, with non-information content comprising only corresponding keys and prefix bytes. Flatbuffers occupies the most space, with byte streams containing substantial non-data information.

2 Experimental Results and Analysis

Msgpack, Protobuf, and Flatbuffers operate not only on Linux, Windows, and other operating systems but also support programming languages such as C, C++, and Python. However, control system development often employs multiple programming languages, with C and C++ used for 底层 driver development and communication, and Python for server-side development and astronomer data processing. Therefore, for testing the three binary serialization tools, the test environment used a 2.0GHz Intel Core i7-4750 CPU, 8GB memory, and Ubuntu 16.04 operating system. The compilation environment used GCC version 5.3.1 and Python version 3.7.3. The versions of Msgpack, Flatbuffers, and Protobuf were 1.2.1, 1.1.0, and 3.7.1, respectively.

Figure 9 [Figure 9: see original paper] displays a data exchange format showing the encoding of radio telescope status information in the control system. "Tel-

Status" represents the radio telescope status information, primarily including antenna azimuth flag "AzFlag" , antenna elevation flag "ElFlag" , subsystem time flag "TimeFlag" , telescope status flag "ServerFlag" , and antenna position flag "PositionFlag". "Weather"represents meteorological information around the observatory site, such as meteorological equipment status, date, wind tower, meteorological instruments, and wind rose diagram. "WindTower"describes the five elements of meteorological instruments: temperature, pressure, humidity, wind speed, and wind direction. "Plot" represents the wind rose diagram for statistical analysis of wind direction and speed around the site over a period. The control system performs only one encoding/decoding operation on transmitted data. Since the transmitted data includes floating-point and double-precision data, the encoded data cannot be transmitted using ASCII encoding in the control system. The following tests use the data shown in Figure 9 to evaluate the serialized data size, serialization time, and CPU utilization of the three binary serialization tools using C++ and Python.

2.1 Serialized Data Size

Msgpack has two encoding/decoding methods, capable of encoding/decoding both key-value pairs and values in JSON format. Testing the data from Figure 9 with the three binary serialization tools yielded byte stream sizes of 713B, 460B, 520B, and 794B for MSGP-M, MSGP-D, Protobuf, and Flatbuffers, respectively. Therefore, serialized data size is closely related to encoding principles. MSGP-M occupies more space than MSGP-D because MSGP-D only encodes the values from Figure 1. The MSGP-D byte stream represents only a header byte and corresponding prefix and data bytes, while the Protobuf byte stream contains corresponding keys, prefix bytes, and data bytes. Flatbuffers occupies more space because it encodes not only values from key-value pairs but also non-data values such as root offset, int offset, and float offset. Consequently, MSGP-D produces a more compact output format and occupies less space than Protobuf and Flatbuffers.

2.2 Serialization Time

The different encoding/decoding principles of Msgpack, Protobuf, and Flatbuffers result in variations in serialization and deserialization time. Using Figure 9 as an example, after 100,000 iterations, the average single-operation serialization times are detailed in Figure 10 [Figure 10: see original paper]. MSGP-M (C++) has a serialization time of 22.425 microseconds and deserialization time of 52.491 microseconds; Python has a serialization time of 15.566 microseconds and deserialization time of 10.896 microseconds. In C++, serialization time is shorter than deserialization time, while in Python, serialization time is also shorter than deserialization time. This occurs because C++ has many basic data types, resulting in longer decoding time, whereas Python has fewer basic types and better matches key-value pairs, resulting in shorter decoding time. MSGP-M (C++) has a longer serialization time of 22.425 microseconds compared to

MSGP-D (C++) at 15.566 microseconds because MSGP-M needs to encode key-value pairs while MSGP-D only encodes values. The same principle applies to decoding. Protobuf (C++) has a serialization time of 10.514 microseconds and deserialization time of 17.163 microseconds; Python has a serialization time of 86.506 microseconds and deserialization time of 62.431 microseconds. The serialization and deserialization times for each programming language are similar, determined by Protobuf' s IDL. Flatbuffers (C++) has a serialization time of 5.446 microseconds and deserialization time of 0.344 microseconds; Python has a serialization time of 203.718 microseconds and deserialization time of 2.130 microseconds. Flatbuffers has longer serialization time than deserialization time because the encoded byte stream format is consistent with the in-memory data format, requiring virtually no time for deserialization—only I/O reading time.

As shown in Figure 10, for the same binary serialization tool across different programming languages, Flatbuffers (C++) serialization time is 40 times faster than Python' s. Protobuf (Python) serialization time is more than 8 times that of C++. The serialization and deserialization times for MSGP-M or MSGP-D using the same encoding method are similar between C++ and Python, avoiding imbalance due to programming language differences.

2.3 CPU Utilization

CPU utilization during program execution affects performance. Testing the three binary serialization tools using the data shown in Figure 9 yielded the results in Table 1 . The table shows that whether using C++ or Python, Msgpack encoding/decoding CPU utilization is approximately 12.4%. Protobuf CPU utilization is also 12.4% for both C++ and Python. However, Flatbuffers encoding and decoding CPU utilization differs significantly. During encoding, Flatbuffers Python CPU utilization reaches 25.9%, far exceeding that of Msgpack and Protobuf during encoding in both C++ and Python. During decoding, Flatbuffers CPU utilization is slightly lower than Msgpack and Protobuf. Therefore, Msgpack and Protobuf are suitable for scenarios where both server and client have sufficient memory, while Flatbuffers can be applied where server memory is adequate but client memory is limited. However, in practical radio telescope control system applications, server and client memory configurations are similar. In terms of CPU utilization, the overall performance of Msgpack and Protobuf is significantly better than Flatbuffers.

Table 1 CPU utilization of three binary serialization tools

Tool	MSGP-M	MSGP-D	Protobuf	Flatbuffers
Python encode CPU usage	12.45%	12.44%	12.38%	25.9%
Python decode CPU usage	12.5%	12.37%	12.48%	12.21%
C++ encode CPU usage	12.4%	12.36%	12.37%	12.57%
C++ decode CPU usage	12.48%	12.4%	12.47%	11.82%

3 Conclusion

This paper analyzed and compared the encoding principles and characteristics of Msgpack, Flatbuffers, and Protobuf, and conducted tests and analysis. Msgpack does not require IDL; developers only need to write code to implement encoding/decoding functionality. Flatbuffers and Protobuf use IDL to encode/decode transmitted information. When encoding/decoding the same information, MSGP-D' s byte stream size and multi-language serialization time outperform Protobuf and are significantly better than Flatbuffers. MSGP-M offers advantages for encoding small data with significant requirement changes, as it can encode/decode key-value data in any order for the same data, whereas Protobuf and Flatbuffers cannot decode such approaches. According to the development requirements of radio telescope control systems, when communication data formats are determined, MSGP-D can be used; when communication data formats change frequently, MSGP-M can be employed. In summary, through analysis of the three binary serialization tools, Msgpack is more suitable for information transmission in radio telescope control systems, facilitating information exchange between hardware systems, software systems, operating systems, programming languages, and networks of radio telescopes, and providing good extensibility and portability.

References

[1] Wang Jian, Liu Jiajing, Tang Pengyi, et al. A study on generic models of control systems of large astronomical telescopes[J]. Publications of the Astronomical Society of the Pacific, 2013.

[2] A Götz, E.T. Taurel, P.V. Verdier. TANGO - Can ZMQ Replace CORBA?[C]. Proceedings of the 14th International Conference on Accelerator & Large Experimental Physics Control Systems. California. 2013.

[3] Heiko Sommer and Gianluca Chiozzi, et al. Transparent XML Binding using the ALMA Common Software (ACS) Container/Component Framework[J]. 2004.

[4] Juan C. Guzman, Ben Humphreys. The Australian SKA Pathfinder (ASKAP) software architecture[C]// Spie Astronomical Telescopes + Instrumentation. International Society for Optics and Photonics, 2010.

[5] Kodilkar J, Uprade R, Nayak S, et al. Developments of next generation monitor and control systems for radio telescopes[J]. Iop Conference, 2013, 44:012026.

[6] Deng Hui, Zhong Wenjie, Fu Yingxue, et al. The Design of Communication Framework for a New Generation of Telescope Autonomous Control System Based on ZeroMQ[J]. Astronomical Research & Technology, 2018, 15(03): 308-314.

[7] Daradkeh T, Agarwal A, Goely N, et al. Real Time Metering of Cloud Resource Reading Accurate Data Source Using Optimal Message Serialization and

Format[C]// 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018.

[8] Marassi A. Status of the Local Monitor and Control System of SKA Dishes[C]// ICALEPCS. 2015.

[9] Clarke M.J., Akeroyd F.A., Arnold O., et al. Live Visualisation of Experiment Data at ISIS and the ESS[C]// ICALEPCS 2017 - International Conference on Accelerator and Large Experimental Physics Control Systems. 2017.

Note: Figure translations are in progress. See original paper for figures.