2023 IEEE International Performance, Computing, and Communications Conference (IPCCC) | 979-8-3503-0293-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPCCC59175.2023.10253859

Achieving Zero-copy Serialization for Datacenter RPC

Tianfan Zhang †, Huaping Zhou †, Chengyuan Huang †, Chen Tian †, Wei Zhang §, Xiaoliang Wang †,
Yi Wang ‡, Ahmed M. Abdelmoniem ¶, Matthew Tan ‖, Wanchun Dou †, and Guihai Chen†
†State Key Laboratory for Novel Software Technology, Nanjing University, China
‡Nanjing University of Posts and Telecommunications, China
§University of Connecticut, USA
¶Queen Mary University of London, UK
‖West Windsor Plainsboro High School South, USA

***Abstract*—Remote Procedure Call (RPC) is widely used in distributed systems and it usually needs to serialize data before transmission. Serialization accounts for a large proportion of the overhead in RPC and becomes a bottleneck for RPC communications. Because the size of the output serialized message cannot be predicted in advance, there could be multiple memory reallocations and copies in typical serialization libraries (e.g., FlatBuffers), which dominates the overhead. We propose the novel serialization library, zFlatBuffers, to eliminate these avoidable copies during the serialization process and realize zero copy during communication. Unlike the typical serialization library, FlatBuffers, the message generated by zFlatBuffers consists of multiple non-contiguous buffers due to its zero-copy nature. Moreover, we integrate zFlatBuffers with RDMA-based RPC systems. For RDMA Unreliable Datagram, we modify the message buffer of eRPC to enable it to transmit messages composed of multiple buffers. We also build the zRPC system based on RDMA Reliable Connection, which transmits the zFlatBuffers message by the scatter/gather function. Compared to the original FlatBuffers, zFlatBuffers improves the throughput of eRPC and zRPC by 11.2%-33.7% and 5.8%-53.6%, respectively.**

***Index Terms*—RPC, Serialization, Zero Copy, RDMA**

I. Introduction

Remote Procedure Call (RPC) [1] is a popular communication method for datacenter applications. For example, Hadoop, HDFS, and HBase all use RPC to exchange metadata information. The communication procedure of RPC is shown in Fig. 1. Initially, the client calls the local client stub to pack the parameters into a message and sends the message to the remote server over the network. After receiving the message sent by the client, the server stub unpacks the message and sends the parameters to the corresponding server procedure. When the corresponding procedure is executed, the steps for the server to return the results back to the client are similar to the sending process, but in the opposite direction. The process of packing and unpacking messages is called serialization (or marshalling) and deserialization (or unmarshalling), respectively. RPC systems usually rely on dedicated serialization libraries. For example, as an open-source, high-performance and cross-platform RPC framework, gRPC [2] uses Protobuf [3] or FlatBuffers [4] to serialize messages.

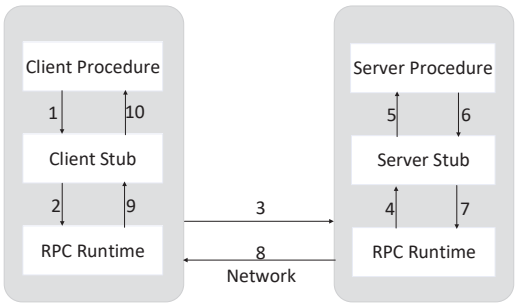


Fig. 1: RPC communication procedure.

Typically, serialization accounts for a large proportion of the overhead for applications with RPC in datacenters. The work in [5] indicated that serialization accounted for nearly 5% of total CPU cycles in Google datacenters. Moreover, the measurements in [6] show that serialization and deserialization account for nearly 30% of all execution time of Spark jobs. With the advances in networking hardware, network speeds continue to increase. Consequently, it is expected that the contribution of serialization to the overall overhead will become even higher, making it the main bottleneck in the RPC.

We observe that current mainstream serialization libraries can have the multi-copy problem. Taking the typical serialization library, FlatBuffers, as an example, the size of the message after serialization cannot be predicted in advance. Therefore, the initial buffer size allocated by FlatBuffers may not be enough to hold the serialized message. In this case, it needs to reallocate a larger buffer, copy the original data to the new buffer, and then release the previously allocated buffer. The subsequent serialized data will be placed after the original data in the new buffer. Hence, FlatBuffers suffers from multiple memory reallocations and copies, which greatly increases the overhead of serialization. We use cpp-serializer [7] to measure the performance of FlatBuffers and count the proportion of memory reallocation and copy in the total serialization time. We demonstrate that when the serialized message sizes are 17,632B and 104,032B, memory reallocation and copy account for 38.4% and 57.7% of the time, respectively. We also find that, as the message size grows, memory reallocation and copy dominate the overhead, so we should minimize the reallocation

and copy.

The key idea of this paper is eliminating memory copies during the serialization. When the buffer space is insufficient, instead of copying the original data to the new buffer, we reallocate a new buffer, write the subsequent serialized data directly to it and splice these physically fragmented buffers logically, achieving the zero copy in the serialization. We implement this idea and name the optimized serialization library as zFlatBuffers. However, there are two key challenges when designing zFlatBuffers:

- The zero-copy serialization involves multiple non-contiguous buffers, while the existing methods can only handle a contiguous one. So to overcome this, we re-designed the memory hierarchy of the serialization library.
- The output of zFlatBuffers consists of non-contiguous buffers, but the data needs to be stored in continuous memory when deserialized. In order to avoid the overhead caused by copying these buffers together, we send these non-contiguous buffers as one message directly.

We integrate zFlatBuffers with RDMA-based RPC systems. However, the existing RPC systems generally only support the transmission of messages composed of a single buffer and do not support the zFlatBuffers messages composed of multiple non-contiguous buffers.

To this end, we design methods to transmit zFlatBuffers messages using Unreliable Datagram (UD) and Reliable Connection (RC) transport modes of RDMA, respectively. For UD, we extend the message buffer of the existing eRPC system so that it can represent multiple buffers, thus supporting zFlatBuffers messages. For RC, we build the zRPC system, which uses the scatter/gather function of the NIC to directly read multiple non-contiguous buffers and transmit them as a message. With the novel design and implementation, we successfully send zFlatBuffers messages to the contiguous memory of the receiver.

In conclusion, our contributions are:

1) We observe that there are multiple memory reallocations and copies in serialization libraries, which decreases the performance of serialization.
2) We design zFlatBuffers to achieve zero copy during serialization. zFlatBuffers reduces the serialization time by 7.6% to 72% compared to the original FlatBuffers.
3) In order to transmit the messages generated by zFlatBuffers, we modify the UD-based eRPC system and build the RC-based zRPC system. Then we integrate zFlatBuffers with eRPC and zRPC. Our experiments show that compared to the original FlatBuffers, zFlatBuffers improves the throughput of eRPC and zRPC by 11.2%-33.7% and 5.8%-53.6% respectively.

II. Related Work

NIC scatter/gather. The scatter/gather function is widely used and well researched in high-performance computing [8]. One of our basic ideas is to take advantage of the NIC scatter/gather function to optimize the serialization library. The scatter/gather function is not free because the NIC needs to read data from multiple buffers through PCIe requests and merge them into a single packet. [9] explores the trade-off between the scatter/gather function and memory copy on CX-5. Scattering/Gathering a large number of small entries hurts performance, so they recommend that the scatter/gather operations should be applied to entries that are at least 512 Bytes large.

Memory reallocations and copies in serialization. Previous work has also studied the problem of reallocation and copy in serialization. RPCoIB [10] uses the locality of message size to solve this problem. The message size of the same type of RPC is similar, so they allocate the different sizes of the initial buffer for different types of RPC, so as to reduce the number of reallocations and copies. However, not all data have good locality. As shown in [10], the size of the heartbeat RPC in Hadoop JobTracker fluctuates, which may lead to reallocation and copying of memory (if the initial buffer size is too small) or waste of memory (if the initial buffer size is too large).

Protobuf [3] and Cap'n proto [11] adopted the arena allocation to improve the efficiency of memory allocation, which enhances performance by aggregating allocations into larger blocks and freeing allocations all at once. However, NIC cannot directly access the memory allocated by these serialization libraries, so one additional copy is still required.

Completely zero-copy serialization during entire RPC. [12] designed a hardware Zerializer and integrated it into NIC to achieve zero-copy serialization and improve application performance. [9] proposed building a zero-copy general serialization library with NIC scatter/gather. A complete zero-copy serialization is exciting, but it requires the application memory to be registered on the NIC. It is impractical to register all the memory of the application on the NIC, which significantly increases the amount of memory footprint and causes a big waste. The memory on the NIC is quite limited and it's like a cache of the host's main memory. Registering too many memory regions will increase the probability of cache misses, thereby affecting performance. Our solution has a clear boundary between application memory and registered memory (between ① and ② in Figure 3(c)), so these issues do not exist in our solution.

III. Background

Remote Procedure Call. Remote Procedure Call (RPC) has become an important part of distributed systems since the 1980s [1]. RPC is widely used in many distributed systems, such as Hadoop, HDFS, and HBase. For instance, the recent distributed file system, Pangu [13], [14] which was developed by Alibaba, uses RPC for all its data communication.

With the large-scale deployment of high-performance RDMA networks [15]–[19], many recent studies have focused on improving the performance of data communication and RPC systems in datacenter [20]–[29]. DaRPC [21] is an RPC library optimized based on RDMA. FaSST [22] is an RPC-based, fast, scalable distributed transaction system. eRPC [23]

TABLE I: Performance of Protobuf and FlatBuffers

Serialization Library	Message Size (Byte)	Time (ms)
FlatBuffers	17,632	3,250
Protobuf	16,116	29,855

is a high-speed, general-purpose RPC library for datacenters. eRPC can run on RDMA, DPDK and raw Ethernet.

However, all of these works did not take the serialization process into consideration. In most cases, the application needs to use a third-party serialization library to serialize the message, and then write it into the RDMA-supported message buffer allocated by the message transmission mechanism. When combined with serialization, the performance of these message transmission mechanisms will decrease due to the need for additional memory copy operations.

Serialization Overhead. The overhead of serialization and deserialization operations occupies a large proportion of the end-to-end data exchange, especially for systems that use RDMA. A recent study [9] indicates that the serialization for a 1024-byte string by Protobuf adds 43% overhead of eRPC. In the industrial system [14], it has been observed that using Protobuf for serialization/deserialization incurs 30% of CPU overhead. According to Skyway [6], serialization/deserialization accounts for 30% of the execution time in Spark jobs. Naos [30] shows that when transferring an array of 1.28M objects on 1Gbps, 10Gbps and 100Gbps networks, the time spent on CPU for serialization/deserialization accounts for 71.4%, 96.4%, and 99.9% of the total transfer time, respectively. To make matters worse, as the network becomes faster, the overhead due to the serialization/deserialization process increases. A report [5] in 2015 showed that Protobuf accounted for 5% of total CPU cycles in Google datacenters and we believe that this proportion is higher nowadays. As network capacity continues to grow, serialization will easily become the main bottleneck of RPC in datacenters.

Current Serialization Library. Both Protobuf [3] and FlatBuffers [4] are currently among the most commonly used serialization libraries. For example, Google's open-source RPC framework gRPC supports these two serialization libraries. But compared to FlatBuffers, Protobuf can cause higher performance overhead. FlatBuffers stores the serialized data into a flat binary buffer so that it can access the data without parsing or unpacking and require no additional memory allocation. But Protobuf requires converting the message to an object before accessing the object, and it usually allocates memory for each object during the deserialization. Therefore, the deserialization performance gap between these two libraries is quite large. To verify this, we use the benchmark cpp-serializer [7] to measure the performance of both Protobuf and FlatBuffers. The serialized source data is a struct Record made up of a string vector and an integer vector. As shown in Table I, although the data serialized by Protobuf is smaller, it costs significantly more time (nearly 10×). Therefore, FlatBuffers is a better choice for latency-sensitive applications in datacenters. For the aforementioned reasons, in this work, we choose FlatBuffers as a typical serialization library and propose novel designs to improve its performance.

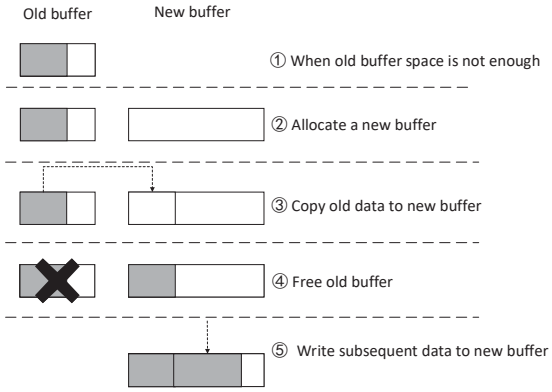


Fig. 2: FlatBuffers reallocation and copy.

FlatBuffers multi-copy problem. Despite being one of the best-performing serialization libraries, FlatBuffers still have some problems in the serialization process. Because FlatBuffers cannot predict the size of messages after serialization in advance, FlatBuffers may require multiple memory reallocations and copies. When FlatBuffers serializes data, by default, it will first allocate a 1024-byte buffer. If the serialized message is larger than 1024-byte, it will reallocate a larger buffer, copy the data from the old buffer to the new buffer, and then deallocate the old buffer. If the newly allocated buffer is still not large enough, it will repeat this process until it finally meets the memory needs of the message. The process of memory reallocation and copy is illustrated in Figure 2.

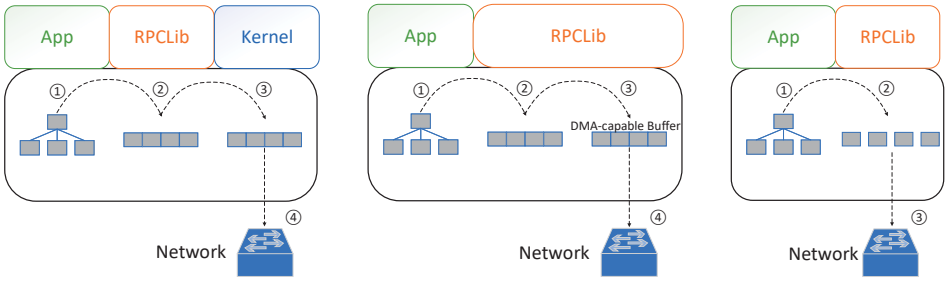
IV. zFlatBuffers

Figure 3(a) shows the multiple memory copies occurring in the RPC processing with the traditional network stack [12]. The data is first stored in the application's memory (①), and then the RPCLib (②) calls the serialization library to pack the data into a message. Later, the message is copied to the kernel socket buffers (③), and finally sent out by the NIC (④). We also show the memory copies of an RPC on a network that bypasses the kernel (such as RDMA) in Figure 3(b). The process is similar to Figure 3(a). Although the kernel is bypassed, the memory allocated by the serialization library is usually not registered on the NIC, so an additional copy to a DMA-capable buffer (such as the message buffer of eRPC) is still required.

As mentioned earlier, in the serialization process of FlatBuffers, there are multiple memory reallocations and copies, which will have a great impact on the performance of serialization. Therefore, in this part, we elaborate on the design of zFlatBuffers, which eliminates the memory copy during the serialization process. It efficiently reduces the total amount of memory allocation and realizes a zero-copy serialization library.

A. The Key Idea

The key idea of zFlatBuffers is: when the size of the object to be serialized is larger than the remaining size of the buffer, we will still allocate a new buffer; but unlike other



(a) RPC in Traditional Network (b) RPC in Kernel-Bypassing Network (c) RPC using zFlatBuffers

Fig. 3: Memory copies in RPC.

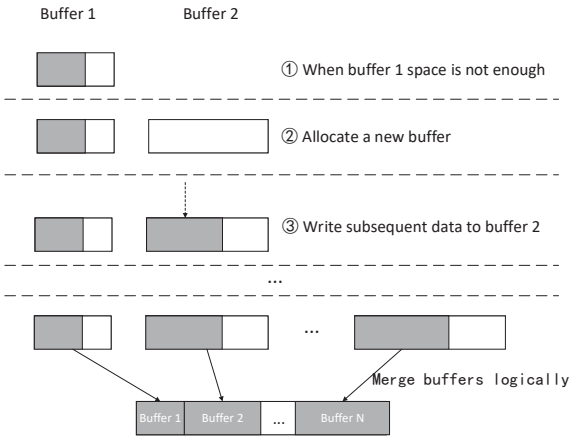


Fig. 4: zFlatBuffers buffer reallocation.

existing serialization libraries, the newly allocated buffer does not directly replace the current one. Instead, zFlatBuffers first truncates the payload of the current buffer and then splices the newly allocated buffer after the current buffer logically. The subsequent serialized data will be written into the newly allocated buffer. This process is shown in Figure 4. Since the newly allocated buffer is an expansion rather than a replacement for the current buffer, this avoids the overhead of copying data from the old buffer to the new buffer during the serialization process.

Therefore, unlike FlatBuffers which gets a continuous buffer after serialization, the final output of zFlatBuffers is composed of multiple non-contiguous buffers. During the deserialization process, data needs to be stored in a continuous memory space, so the non-contiguous output of zFlatBuffers needs to be moved to a continuous buffer. To achieve this, these non-contiguous buffers are sent as one message to a continuous buffer of the receiver.

The memory copy in the RPC process using zFlatBuffers as the serialization library is shown in Figure 3(c). It shows that all unnecessary memory copies are eliminated and only the single essential copy from application memory to RPCLib is required.

B. Design

To realize the idea discussed above, we modify FlatBuffers and redesign its buffer hierarchy. Specifically, we divide the buffer into three levels, i.e., *Block*, *Slice* and *Message*. Figure 5 depicts one example of buffer hierarchy.

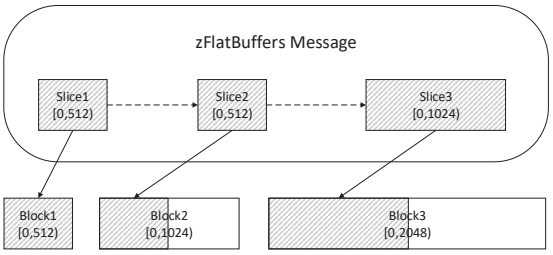


Fig. 5: The buffer hierarchy.

The block is located at the bottom of the buffer hierarchy. It is a direct encapsulation of the memory block allocated by the memory allocator and is registered on the NIC. A block is an abstraction of a fixed-length buffer.

The slice is located above the block. Since zFlatBuffers needs to truncate the payload of the current buffer when inserting a newly allocated buffer, the library must have the flexibility of dynamically adjusting the length and first address of the buffer. However, the block is a fixed-length buffer and cannot be adjusted directly. Therefore, we introduce slices as views of blocks. The slices are flexible since their lengths and first addresses can be dynamically adjusted by the Resize() function. The slice also manages the life cycle of the underlying block through a smart pointer: the underlying block will be released when no slice refers to this block. Therefore, the slice is an abstraction of the variable-length buffer.

The message is located at the top of the buffer hierarchy, and it is the final output of zFlatBuffers. The output message from zFlatBuffers contains a list of slices, which combine multiple non-contiguous variable-length buffers to form a complete message.

The hierarchical relationship between block, slice and message is shown in Figure 5. There are 3 blocks, 3 slices and 1 message. This message contains 3 slices, slice 1 refers to all 512 bytes of block 1, slice 2 only refers to the first 512 bytes of block 2 and slice 3 refers to the first 1024 bytes of block 3. Because slices may only use part of the block, there is a waste of memory space allocated by zFlatBuffers. Yet, the total memory allocation of zFlatBuffers is still less than the original FlatBuffers. We discuss this in more detail in § VI-A.

C. Implementation

FlatBuffers serializes data through the FlatBufferBuilder class, and FlatBufferBuilder uses the *vector_downward* class as a buffer for storing data. *vector_downward* implements the basic function of *std::vector*, but *vector_downward* expands from high address to low address.

The next three functions in *vector_downward* constitute the key operations of the buffer: 1) *make_space()* expands the buffer by *len* bytes and returns a pointer to the first address; 2) *pop()* shrinks *bytes_to_remove* bytes from the end of the buffer; and 3) *data_at()* returns the address of the *offset*. *vector_downward* will reallocate the buffer when the remaining space in the buffer is insufficient, and then copy the data to the newly allocated buffer.

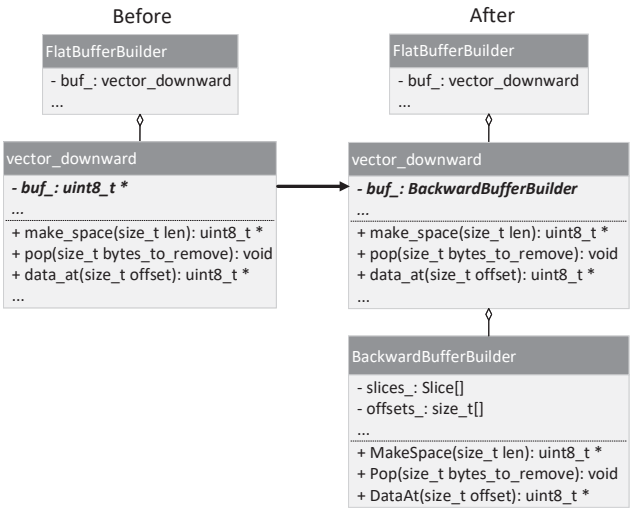


Fig. 6: The class graph.

We need to re-implement the above functions to make sure the serialization is zero-copy when constructing the message buffer. We develop BackwardBufferBuilder class to build the message buffer. The member variable of the BackwardBufferBuilder class contains an array of slices, which can represent multiple pieces of non-contiguous memory. The BackwardBufferBuilder class implements the above three functions on multiple buffers. We use BackwardBufferBuilder to replace the original continuous buffer in the *vector_downward* and re-implement the *make_space*, *pop* and *data_at* operations based on it. The overall relationship is shown in Figure 6.

These three operations of BackwardBufferBuilder may involve multiple buffers, so they are different from the original implementation of *vector_downward*.

Expand/Shrink Buffers. BackwardBufferBuilder member function *MakeSpace()* implements the function of *make_space()* to expand buffers. *MakeSpace()* follows the basic idea of zFlatBuffers. If the remaining space of the current buffer is less than the number of bytes that needs to be expanded, then *MakeSpace()* will truncate the current buffer and expand the space on the newly allocated buffer. *BackwardBufferBuilder::Pop()* shrinks buffers and is the reverse process of *MakeSpace()*. If the current buffer is less than the number of bytes that need to be removed, then *Pop()* function will continue to pop the buffer from *slices_* array until enough bytes are removed.

Access Data. *BackwardBufferBuilder::DataAt()* implements a modified version of *data_at()* function. FlatBufferBuilder sometimes needs to access the data in the buffer during the serialization. Hence, it uses the offset to refer to the data and *data_at()* converts the offset to the actual address. The data_at of vector_downward is an operation with a constant time complexity because it only needs to do one calculation according to the buffer address and the *offset* parameter. However, the *DataAt()* of BackwardBufferBuilder involves multiple buffers, which makes the management challenging. To overcome this, we store the offset of the tail of each slice, and then find the slice corresponding to the specified offset through binary search. However, this increases the algorithm complexity of the *DataAt()* operation from $O(1)$ to $O(logn)$, where n is the number of slices in the message buffer.

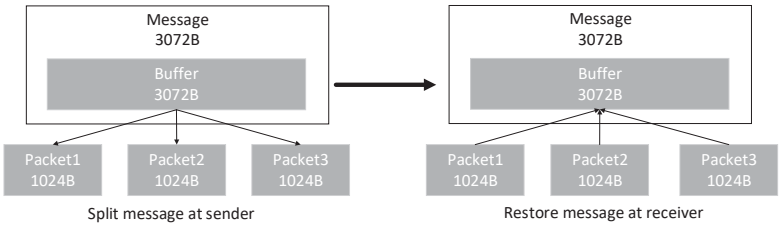
Optimization for Accessing Data. The complexity of *DataAt()* becomes $O(logn)$, which will increase the serialization overhead. We observe that the fundamental reason why FlatBufferBuilder uses *data_at()* to access the data is that when there is memory reallocation, the pointer to the data in the buffer may become invalid. On the contrary, in BackwardBufferBuilder, the allocated buffer will not be replaced by the newly allocated buffer, so the pointer to the data in the buffer is not likely to be invalid and there is no need to use offset. In FlatBufferBuilder, both vtables and shared strings are referenced using offsets. We replace these indirect references via offsets with direct accesses via pointers, which significantly reduced the need for using the *DataAt()* operation.

V. System Integration

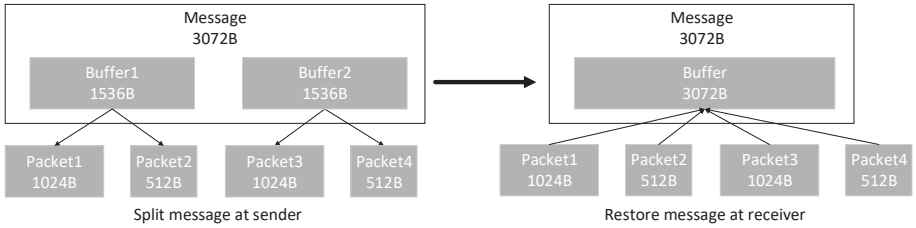
In this section, we describe how we integrate zFlatBuffers with the RPC systems. The key challenge is that the data needs to be stored in a contiguous memory space during deserialization while the output of zFlatBuffers consists of multiple non-contiguous buffers. To overcome this, we need to send the output of zFlatBuffers as one message to the contiguous memory on the receiver. Currently, most common RPC systems only support the transmission of messages containing one buffer, which makes the integration with zFlatBuffers rather challenging. In the following, we detail how we resolve these issues. Note that Unreliable Datagram (UD) and Reliable Connection (RC) are the two typical transport modes of RDMA. We take eRPC [23] and zRPC (the RPC system we built based on RC) as examples to introduce how to use UD and RC to transmit zFlatBuffers messages, respectively.

A. eRPC

eRPC [23] is currently one of the best open-source RPC frameworks in academia. eRPC uses UD to transmit messages and the UD message size cannot exceed the MTU, so the CPU needs to split the message into packets and send them to the network. The receiver restores them in order accordingly.



(a) The original eRPC



(b) The modified eRPC

Fig. 7: The message transmission process of eRPC.

We try to reuse the transmission mechanism of eRPC, but the problem is that the message buffer of eRPC contains only one contiguous buffer, so the message sent by eRPC can only have a contiguous buffer. Therefore, we modify the struct *msg_buffer* of eRPC and re-implement its member functions to enable it to contain multiple buffers.

The eRPC transmission process before and after our modification is shown in Figure 7. We demonstrate the process of transmitting a 3072B message, and the MTU is 1024 Bytes [1]. Originally, the message buffer of eRPC could only contain one buffer, and eRPC divides this single buffer into three packets, as shown in Figure 7(a). After our modifications, the message buffer of eRPC can have multiple buffers. As shown in Figure 7(b), the message buffer is composed of two non-contiguous buffers of size 1536 Bytes each. The sender divides the buffers one by one, and finally divides the message into four packets and sends them. After receiving these network packets, the receiver restores them to a message made up of only one continuous buffer. We enable eRPC to send non-contiguous buffers as one message through these modifications so that zFlatBuffers can be combined with eRPC.

B. zRPC

zRPC is an RPC framework we built to cooperate with zFlatBuffers. zRPC uses Boost Asio [31] to handle task scheduling, which is a cross-platform C++ library for network and low-level I/O programming. Different from eRPC, zRPC is based on RC, as RC has the advantages of high throughput, low latency and low CPU overhead. zRPC uses RDMA SEND/RECV verbs to send messages. The message size of RC is not limited by the network MTU, so the CPU does not need to split the message into packets and then restore it.

[1]The message buffer of eRPC contains packet headers, here we ignore this detail for brevity.

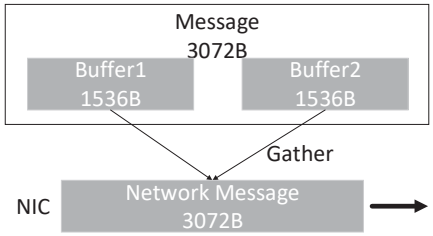


Fig. 8: Message Gathering in zRPC.

RDMA has good support for the scatter/gather functions, which enables the NIC to read data from multiple local buffers and write the data to a single remote buffer, or read data from a single local buffer and write the data to multiple remote buffers. RDMA uses *sge* to represent scatter/gather elements, which holds a pointer to a registered memory block. When transmitting zFlatBuffers messages, zRPC will construct *sge*s based on the slices of the zFlatBuffers message, and then use the scatter/gather function to directly transmit the message composed of non-contiguous buffers. As shown in Figure 8, when transmitting a message made up of two buffers, zRPC gathers these two buffers together by the NIC and sends the message to the network.

C. Memory Registration

The memory allocated by original serialization libraries is not registered on the NIC, so the message needs to be completely copied to the registered memory after the serialization (i.e., the copy from ② to ③ in Figure 3(b)). We directly transmit the buffers of zFlatBuffers, which requires the memory of zFlatBuffers to be registered on the NIC.

To overcome the time-consuming RDMA memory registration, RDMA-based RPC systems usually adopt some methods to reduce the overhead. For instance, memory is registered in batches to amortize the overhead and Linux HugePage [32] is also used to reduce page table entries. Both eRPC and

zRPC allocators use these approaches as well. The difference is that eRPC's *HugeAlloc* is just a rudimentary allocator, while zRPC's *rmalloc* is modified based on the general-purpose allocator *mimalloc* [33]. We replace the memory allocator of zFlatBuffers with the allocator used by the corresponding RPC system so that the output messages of zFlatBuffers can be directly transmitted through the RPC system.

VI. Evaluation

Configuration. The machine we use in the experiments is equipped with 2 Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 8 cores, 256GB of memory, and a Mellanox ConnectX5 100Gbps NIC. The RDMA protocol in the experiments is RoCEv2 and the MTU is set to 1024 Bytes.

Structure. § VI-A shows the performance of zFlatBuffers and the details of the memory allocation of FlatBuffers and zFlatBuffers. § VI-B shows the comparison between eRPC with zFlatBuffers and eRPC with the original FlatBuffers. § VI-C compares zRPC with zFlatBuffers and zRPC with the original FlatBuffers. § VI-D shows the performance of a real application integrated with zFlatBuffers.

Summary of results. When the size of the message to be serialized is large, the serialization performance of zFlatBuffers exceeds the original FlatBuffers by 7.6%-72%. After combining with zFlatBuffers, the throughput of both eRPC and zRPC has been improved. Specifically, the throughput of eRPC has increased by 11.2%-33.7% while the throughput of zRPC has increased by 5.8%-53.6%. For a real key-value store application, eRPC reduces the median latency by 8% and the 99^{th}-percentile latency by 5%.

A. zFlatBuffers

We use cpp-serializers [7] to test zFlatBuffers and FlatBuffers, which is a benchmark for comparing serialization libraries. We set the number of iterations as 1,000,000. The serialized source data is a struct Record containing a string vector and an integer vector. The size of the string and integer is equal, i.e., if the size of the source data is 1024 Bytes, the sizes of the string and integer are both 512 Bytes.

As shown in Figure 9, the serialization time of zFlatBuffers is obviously lower than that of FlatBuffers. As the message size increases, the gap between these two libraries becomes larger. This is because the overhead from memory reallocation and copy of FlatBuffers becomes higher with the increase in the message size. For instance, for the source data sizes of 1,024B, 16,384B and 65,536B, zFlatBuffers can decrease the serialization time by 7.6%, 22.3% and 72%, respectively.

Later, we zoom into the memory allocation process. Specifically, when the source data size is 16,384B and 65,536B, Figure 10(a) and Figure 10(b) reveal the number of memory allocations required during serialization. Note that we omit the case when the source data size is 1,024B because it only needs one or two allocations.

Finally, the statistics of allocation and copy are listed in Table II. The total allocation size of FlatBuffers is twice that of zFlatBuffers. The reason is the inefficient buffer usage of FlatBuffers and it requires more allocations to meet the memory requirements. When a new buffer is allocated, FlatBuffers needs to copy the old data to this new buffer before the remaining space can be used. The data in the previously allocated buffer becomes useless and needs to be released. In contrast, with zFlatBuffers, all the space in the newly allocated buffer can be used to write new data. It splices the newly allocated buffer directly after the current buffer, without the need to copy the old data into the new buffer. The experimental results show that zFlatBuffers reduces the serialization time by 7.6%-72% according to different message sizes.

B. zFlatBuffers with eRPC

We combine zFlatBuffers and the original FlatBuffers with the eRPC system, and compare the performance of these two systems. We use two machines for the experiments, one as a client and one as a server. The client will first serialize the message and then send it as a request to the server. When receiving the request, the server will respond with an ACK. After the client receives the ACK, it will re-serialize the message and send it again. This ping-pong process repeats continuously.

We adjust the number of concurrent requests so that the eRPC system achieves the maximum throughput or the minimum latency. The results of throughput and latency are shown in Figure 11(a) and Figure 11(b), respectively. Figure 11(a) shows that the throughput of eRPC first increases and then decreases. In theory, the throughput of eRPC should increase monotonically with the increase of message size until it encounters a network bottleneck. However, with the growing message size, the overhead of serialization will also increase, resulting in a decrease in throughput. Both zFlatBuffers and FlatBuffers conform to this trend. Compared to FlatBuffers, as the message size increases, the optimization effect of zFlatBuffers on eRPC becomes increasingly significant. The results show that when the message size is larger than 1K, zFlatBuffers increases the throughput of eRPC by 11.2% to 33.7% and reduces the latency of eRPC by 3% to 18%.

C. zFlatBuffers with zRPC

We redo the above ping-pong experiment but with zRPC system in this part. We give a detailed comparison of the system performance with zFlatBuffers and FlatBuffers.

As shown in Figure 12(a), zFlatBuffers increases the throughput of zRPC by 5.8% to 53.6% with different message sizes. We also show the multi-threaded scenario of zRPC when the message size is 131,072B, in Figure 12(b). The results demonstrate that the RPC system with zFlatBuffers can achieve better scalability. The throughput of zRPC with zFlatBuffers reaches the peak, which is mainly constrained by the NIC capacity, when the number of threads is 3. On the contrary, zRPC with FlatBuffers can only achieve comparable throughput when the number of threads is 7.

Later, we combine the normalized throughput of eRPC and zRPC with zFlatBuffers, and the results are depicted in Figure 13. We take the throughput of the RPC system with

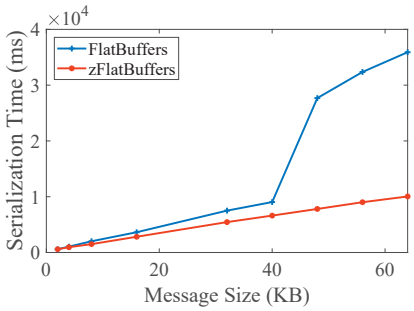
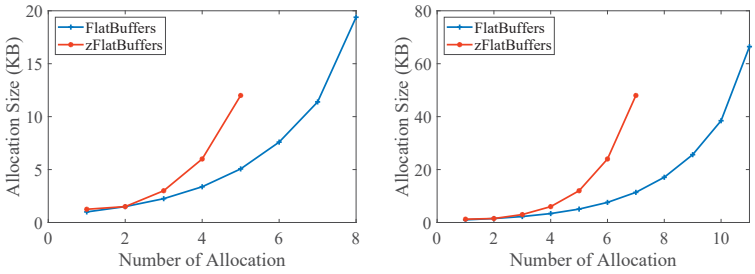


Fig. 9: Serialization time.

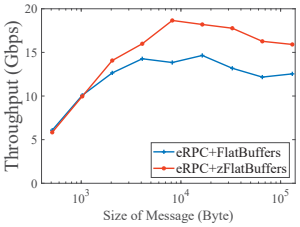


(a) Size = 16,384B. (b) Size = 65,536B.

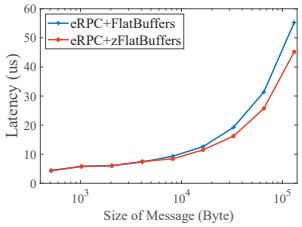
Fig. 10: Memory allocation during serialization.

TABLE II: Memory Allocation

Library	Source Data Size	Size after Serialization	Allocations Number	Total Allocation Size	Total Copy Size
FlatBuffers	1,024B	1,152B	2	2,560B	1,024B
zFlatBuffers	1,024B	1,152B	1	1,280B	0B
FlatBuffers	16,384B	17,952B	8	52,800B	32,944B
zFlatBuffers	16,384B	17,952B	5	24,320B	0B
FlatBuffers	65,536B	71,712B	12	184,112B	116,064B
zFlatBuffers	65,536B	71,712B	8	98,048B	0B

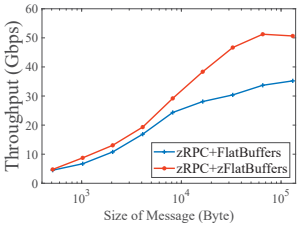


(a) Throughput with different message sizes

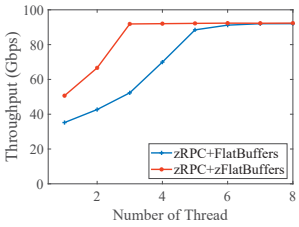


(b) Latency with different message sizes

Fig. 11: Throughput and Latency of eRPC.



(a) Throughput with different message sizes



(b) Throughput with multiple threads

Fig. 12: zRPC Throughput.

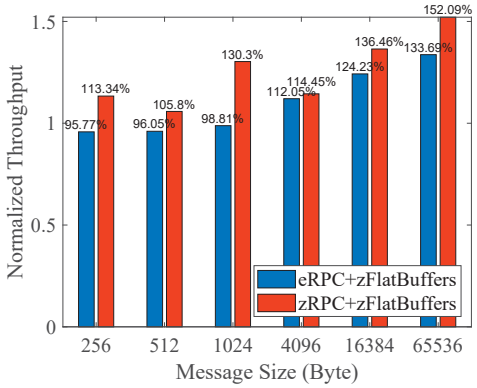


Fig. 13: Normalized Throughout.

the original FlatBuffers as the baseline. For example, when the message size is 256 Bytes, the normalized throughput of eRPC with zFlatBuffers is 95.77%, indicating that the throughput of eRPC with zFlatBuffers is 95.77% of the eRPC with the original FlatBuffers. The performance improvement of eRPC with a small message is negligible, mainly because the reallocation and copy do not occur yet with such a small memory footprint. As the message size increases, the improvement becomes more significant. When the source data size is 65,536 Bytes, the performance improvement on zRPC exceeds 50%. Moreover, we also find that the performance of zRPC is generally better than eRPC. We attribute it to the RC transport mode of RDMA in zRPC. With RC mode, NIC packetizes the message and reduces the CPU overhead compared to the UD transport of RDMA. In contrast, eRPC with UD transport consumes more CPU cycles during transmission and the CPU is apt to be overwhelmed, which degrades the overall performance.

D. Full-system benchmark

We apply zFlatBuffers to the real key-value store in this part. Raft [34] is an election-based distributed consensus protocol. A leader is elected between candidates to handle requests from clients. [23] combines eRPC with LibRaft [35](an open source implementation of Raft on GitHub) and implements a 3-way replicated in-memory key-value store [36]. We reproduce this experiment and add a serialization step to the communication between the client and server. A client randomly generates a 256 Bytes key and a 1024 Bytes value. Later, it serializes the key and value, and sends them out as a PUT request to the leader. After receiving the request, the leader deserializes it and obtains the contents of the key and value.

Table III shows the request latency of the client with no serialization, FlatBuffers, and zFlatBuffers. We set the

TABLE III: Latency comparison for replicated PUTs

System	Median	99%
eRPC with no serialization	$15.47\mu s$	$21.53\mu s$
eRPC with FlatBuffers	$21.19\mu s$	$27.09\mu s$
eRPC with zFlatBuffers	$19.49\mu s$	$25.67\mu s$

eRPC performance with no serialization as the baseline. The latency with FlatBuffers and zFlatBuffers is increased by 37% and 26%, respectively. Compared to FlatBuffers, zFlatBuffers cuts the median and 99^{th}-percentile latency by 8% and 5%, respectively.

VII. Conclusion

The RPC serialization can dominate the communication time in large-scale distributed systems. To mitigate the problem, we propose a simple and efficient message serialization approach, zFlatBuffers, to eliminate expensive memory copies and achieve an ideal zero-copy transmission. Specifically, we redesign the buffer hierarchy in the RPC serialization library, which utilizes the three-layer hierarchy to support multiple non-contiguous buffers. Moreover, we propose and implement a series of novel memory management approaches to achieve fast memory allocation/deallocation/address location. We also integrate zFlatBuffers into the existing UD-based eRPC system and RC-based zRPC system. The experiments show that zFlatBuffers improves eRPC and zRPC throughput by up to 33.7% and 53.6%, respectively.

VIII. Acknowledgement

This project is partially supported by the National Key Research and Development Program of China under Grant Number 2022YFB2901502, the National Natural Science Foundation of China under Grant Numbers 62072228 and 62172204, and the Jiangsu Funding Program for Excellent Postdoctoral Talent.

References

[1] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM TOCS*, vol. 2, pp. 39–59, 1984.

[2] R. L, "The grpc blog grpc motivation and design principles[eb/ol]," https://grpc.io/blog/principles/, 2015.

[3] G. Inc, "Protocol buffers: Google's data interchange format," https://github.com/protocolbuffers/protobuf, 2008.

[4] W. V. Oortmerssen, "Flatbuffers: a memory efficient serialization library," https://opensource.googleblog.com/2014/06/flatbuffers-memoryefficient.html, 2014.

[5] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ACM/IEEE ISCA*, 2015, pp. 158–169.

[6] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu, "Skyway: Connecting managed heaps in distributed big data systems," *ACM SIGPLAN Notices*, vol. 53, pp. 56–69, 2018.

[7] S. K, "cppserializers:benchmark comparing various data serialization libraries (thrift, protobuf etc.) for c++[eb/ol]. 2019," https://github.com/thekvs/cppserializers.

[8] S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler, "Network-accelerated non-contiguous memory transfers," in *ACM SC*, 2019, pp. 1–14.

[9] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang, "Breakfast of champions: towards zero-copy serialization with nic scatter-gather," in *ACM HotOS*, 2021, pp. 199–205.

[10] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *IEEE ICPP*, 2013, pp. 641–650.

[11] K. Varda, "Cap'n proto," https://capnproto.org/.

[12] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: Towards zero-copy serialization," in *ACM HotOS*, 2021, pp. 206–212.

[13] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan *et al.*, "When cloud storage meets RDMA," in *USENIX NSDI*, 2021, pp. 519–533.

[14] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu, "More than capacity: Performance-oriented evolution of pangu in alibaba," in *USENIX FAST*, 2023, pp. 331–346.

[15] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *ACM SIGCOMM*, 2016, pp. 202–215.

[16] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 523–536, 2015.

[17] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *ACM SIGCOMM*, 2018, pp. 313–326.

[18] Y. Le, B. Stephens, A. Singhvi, A. Akella, and M. M. Swift, "Rogue: Rdma over generic unconverged ethernet," in *ACM SoCC*, 2018, pp. 225–236.

[19] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen, "Tagger: Practical pfc deadlock prevention in data center networks," in *ACM CoNEXT*, 2017, pp. 451–463.

[20] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *USENIX NSDI*, 2014, pp. 401–414.

[21] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "Darpc: Data center rpc," in *ACM SoCC*, 2014, pp. 1–13.

[22] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs," in *USENIX OSDI*, 2016, pp. 185–201.

[23] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in *USENIX NSDI*, 2019, pp. 1–16.

[24] S. K. Monga, S. Kashyap, and C. Min, "Birds of a feather flock together: Scaling rdma rpcs with flock," in *ACM SOSP*, 2021, pp. 212–227.

[25] B. Yi, J. Xia, L. Chen, and K. Chen, "Towards zero copy dataflows using rdma," in *ACM SIGCOMM*, 2017, pp. 28–30.

[26] Y. Chen, Y. Lu, and J. Shu, "Scalable rdma rpc on reliable connection with efficient resource sharing," in *ACM EuroSys*, 2019, pp. 1–14.

[27] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "Rfp: When rpc is faster than server-bypass with rdma," in *ACM EuroSys*, 2017, pp. 1–15.

[28] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir *et al.*, "Storm: a fast transactional dataplane for remote data structures," in *ACM SYSTOR*, 2019, pp. 97–108.

[29] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *USENIX ATC*, 2016, pp. 437–450.

[30] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler, "Naos: Serialization-free rdma networking in java," in *USENIX ATC*, 2021, pp. 1–14.

[31] C. Kohlhoff, "Boost.asio 1.72.0[eb/ol]," https://www.boost.org/doc/libs/1_72_0/doc/html/boost_asio.html, 2019.

[32] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *ACM SOSP*, 2017, pp. 306–324.

[33] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Asian Symposium on Programming Languages and Systems*, 2019, pp. 244–265.

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

[35] Willem, "C implementation of the raft consensus protocol," https://github.com/willemt/raft, 2018.

[36] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, 2014, pp. 429–444.