Poster: Reducing Data Movement Tax for Serialization in Microservices

Siddharth Nayak, IIIT-Delhi, India
Vishesh Rangwani, IIIT-Delhi, India
Kartikay Dubey*, IIIT-Delhi, India
Rajorshi Mondal*, IIIT-Delhi, India
Tushar Gupta*, IIIT-Delhi, India
Rinku Shah, IIIT-Delhi, India

Abstract

Small-scale public cloud providers cannot adopt serialization data-copy optimizations that leverage hardware acceleration or kernel-bypass zero-copy paradigm (e.g., DPDK) because of (1) accelerator deployment costs, (2) securing kernel-bypass zero-copy serialization communication relies on encryption at the NIC hardware, and (3) kernel-bypass solutions require developers to rewrite the business logic. We propose a serialization library that leverages the kernel scatter-gather communication primitives to reduce serialization data copy costs without using custom hardware or a custom network stack.

CCS Concepts

• **Networks** → **Programming interfaces**.

Keywords

serialization; scatter-gather; Linux API; microservices

ACM Reference Format:
Siddharth Nayak, Vishesh Rangwani, Kartikay Dubey, Rajorshi Mondal, Tushar, Gupta, and Rinku, Shah. 2024. Poster: Reducing Data Movement Tax for Serialization in Microservices. In *Proceedings of the 20th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '24), December 9–12, 2024, Los Angeles, CA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3680121.3699882

1 Introduction

Modern cloud applications (e.g., Uber and 5G RAN core) with μs-scale processing times have strict latency requirements (i.e., 10s to 100s of μs). A single client request involves traversing multiple microservices and forces per-hop data (de) serialization (e.g., more than 20% of the requests traverse > 50 hops [1]), resulting in serialization latency domination over request processing time.

The data serialization process involves three key steps: (1) *Initialization:* allocate an empty buffer to hold the serialized data, (2) *Encoding:* convert the application-specific data structures to wire format for transmission, and (3) *Data copy:* moving data from application data structures into the memory buffers for transmission. *"Data copy"* is identified as the root cause of the bottleneck [3].

*Student authors with equal contribution.

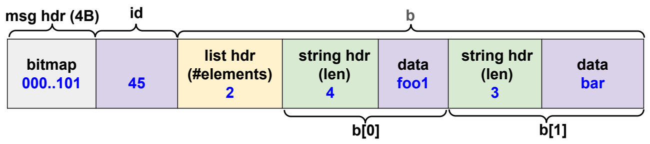

Figure 1: μSer wire format for a message with three fields, *id* (int) and *b* (repeated string), with values 45, "foo1", and "bar", respectively.

There are three ways to implement data movement: (1) *Two-copy:* (a) Data is copied from the application data structures to contiguous memory buffers allocated in the user space, (b) serialized buffers copied from the user space to kernel network buffers, and (c) copy from network buffers to the NIC; (2) *One-copy:* (a) The system uses a *scatter-gather* vector with data object addresses and length. The kernel copies data from memory regions specified in the scatter-gather array to kernel network buffers, (b) copy data from network buffers to the NIC, and (3)*Zero-copy:* Data is copied from application data structures directly to the NIC via zero-copy scatter-gather I/O.

Popular serialization libraries such as Protobuf, FlatBuffers, and Cap'n Proto use *two-copy* approach and incur high serialization latency. Existing works propose hardware accelerators for serialization, for example, on-CPU , fixed-function , and on-NIC [2] accelerators. However, such solutions warrant the need for hardware security accelerators for encrypted communication between microservices. Another solution approach [3] leverages kernel-bypass zero-copy I/O techniques such as DPDK to reduce data movement cost. However, this approach has two challenges: (1) it lacks secure microservice communication support, and (2) it requires changes to the business logic and the host networking stack. Our key idea is to reduce serialization latency by leveraging existing Linux scatter-gather communication primitives, *one-copy* and *zero-copy*, without relying on custom NIC or network stack changes. We build the prototype of the serialization library, μSer, to enable our idea.

We present the following contributions in this paper. (1) Initial prototype of μSer serialization library, (2) Insights on kernel processing overheads and performance analysis for two-copy, one-copy, and zero-copy kernel communication primitives.

2 Design

μSer comprises three components: an interface definition language (IDL), wire format, and the μSer library.

IDL. The IDL describes the syntactic structures of the message schema, similar to Protobuf's IDL. Unlike state-of-the-art serialization libraries, μSer generates code to read and write structured data at runtime instead of compile-time. However, this approach introduces overheads required for runtime checks, which we plan to solve in our future work.

Wire format. Inspired by Cornflakes[3], the wire format (see Fig. 1) consists of a header with a 4-byte bitmap followed by data stored

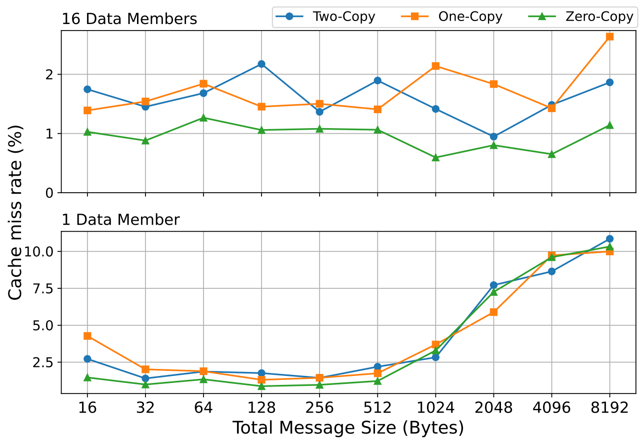

Figure 2: Cache miss rate for communication primitives

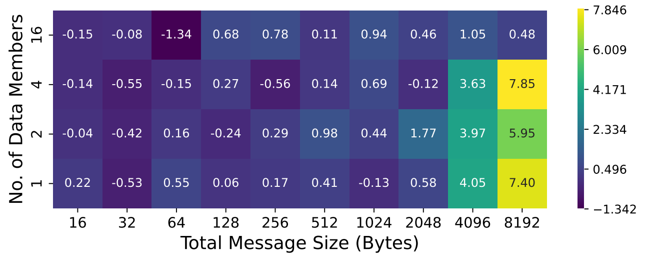

Figure 3: Data copy latency improvement (in μs): One-copy vs. Two-copy (baseline).

No. of Data Members	16	32	64	128	256	512	1024	2048	4096	8192
16	-0.15	-0.08	-1.34	0.68	0.78	0.11	0.94	0.46	1.05	0.48
4	-0.14	-0.55	-0.15	0.27	-0.56	0.14	0.69	-0.12	3.63	7.85
2	-0.04	-0.42	0.16	-0.24	0.29	0.98	0.44	1.77	3.97	5.95
1	0.22	-0.53	0.55	0.06	0.17	0.41	-0.13	0.58	4.05	7.40

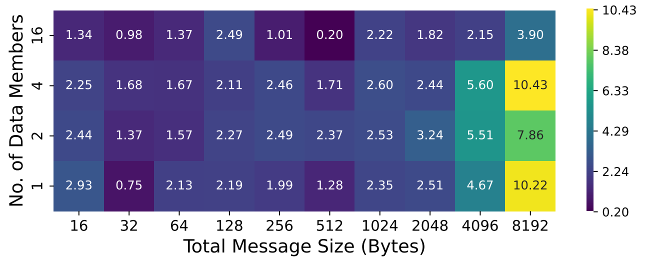

Figure 4: Data copy latency improvement (in μs): Zero-copy vs. Two-copy (baseline).

No. of Data Members	16	32	64	128	256	512	1024	2048	4096	8192
16	1.34	0.98	1.37	2.49	1.01	0.20	2.22	1.82	2.15	3.90
4	2.25	1.68	1.67	2.11	2.46	1.71	2.60	2.44	5.60	10.43
2	2.44	1.37	1.57	2.27	2.49	2.37	2.53	3.24	5.51	7.86
1	2.93	0.75	2.13	2.19	1.99	1.28	2.35	2.51	4.67	10.22

sequentially, in the ascending order of member identifier "ID". The bitmap indicates the presence of data values.

μSer library. Our library exposes APIs and implements functions such as (1) schema file compilation, (2) (de) allocation of the message, (3) setting/getting data to/from the message, (4) serialization of the message to wire format and vice-versa.

Workflow. The developer provides schema files with the program. At runtime, the symbol table is created, message is allocated, and its elements are initialized. The developer invokes the μSer library function, `serialize`, which returns an I/O vector. The I/O vector is passed to the scatter-gather API such as `sendmsg` to transmit data. Like FlatBuffers, μSer uses lazy deserialization, which is zero-cost.

Challenges. We see several challenges in designing μSer. We plan to solve these in our future work. (1) Microservices leverage TLS at the application layer to support secure communication. However, since scatter-gather primitives assemble the message within the kernel, TLS cannot be used; (2) μSer suffers from runtime check overheads because our current prototype does not generate code to write/read data at compile time; and (3) The developer must ensure safe memory access because the current μSer prototype deals with pointers but does not manage memory.

Table 1: Serialization latency across libraries for message with 1 key of 24 bytes and 10 data fields of 100 bytes each (YCSB workload).

Overheads (in μs)	Protobuf	FlatBuffers	μSer (with One-copy)
Wire format conversion	6.9 ± 0.8	8.8 ± 0.4	0.9 ± 0.3
Others (includes data copy)	35.5 ± 5	43.1 ± 7	12.9 ± 4
Total	42.4±5.8	51.9 ± 7.4	13.8 ± 4.3

3 Evaluation

We deployed an echo client-server application, written in C++, on two Intel Xeon 6403N machines connected using 100Gbps Intel 810 NICs. We use our observations to answer the following questions.

How do One-copy and Zero-copy primitives perform compared to Two-copy? We vary the cumulative message sizes from 16 bytes to 8KB with varying data members from 1 to 16.

Cache miss rate (Fig. 2). We observe that the zero-copy primitive has a lower cache miss rate across the number of data members, followed by one-copy and then two-copy. The cache miss rate is inversely proportional to the number of data members across primitives. The cache miss rate scales directly with increased message size for a smaller number of data members.

Data-copy latency improvement (Fig. 3 and Fig. 4). We observe that the zero-copy primitive outperforms two-copy by 6% (0.75 μs) to 43% (10.4 μs). The one-copy primitive outperforms two-copy by up to 32% (7.855 μs) for data field sizes >= 128 bytes.

How do existing serialization libraries perform compared to μSer with one-copy? Table 1 shows the split serialization latency, (1) data object to wire format conversion, and (2) data copy, syscall, and other overheads for a YCSB workload , across standard serialization libraries and μSer. We observe that μSer improves the overall serialization latency by 67% and 73% for Protobuf and FlatBuffers, respectively. This improvement is significant because a client request suffers serialization costs at each microservice (e.g., Alibaba's trace analysis shows 50 to 54000 calls per request [1]).

4 Conclusion and Future Work

Our initial prototype results motivate the role of kernel I/O primitives to improve serialization latency for cloud microservices. As part of future work, we plan to (1) integrate secure microservice communication (using kTLS), (2) μSer with adaptation across kernel I/O primitives based on object data type (`ptr` type or not), and data characteristics (message size and number of members), and (3) memory safety without developer intervention.

References

[1] Shutian Luo et al. 2022. An In-Depth Study of Microservice Call Graph and Runtime Performance. *IEEE TPDS* 33, 12 (2022). https://doi.org/10.1109/TPDS.2022.3174631
[2] Arash Pourhabibi et al. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO '21*. 407–420. https://doi.org/10.1145/3466752.3480055
[3] Deepti Raghavan et al. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of SOSP '23*. 200–215. https://doi.org/10.1145/3600006.3613137