Optimising Serialisation for Cloud Applications

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

M.Tech

BY Siddharth Nayak MT22128

Computer Science and Engineering

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI
NEW DELHI, 110020

May 21, 2024

Thesis Certificate

This is to certify that the thesis titled **Optimising Serialisation for Cloud Applications**, submitted by **Siddharth Nayak**, to the Indraprastha Institute of Information Technology, Delhi, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Rinku Shah
Thesis Supervisor
Assistant Professor
Dept. of Computer Science
IIIT Delhi, 110020

Place: New Delhi

May 21, 2024

Acknowledgement

I express heartful gratitude to my supervisor, Dr. Rinku Shah. I want to thank my group members Vishesh, Rajorshi, Kartikay, and Maryam. Also, thanks to Shambhavi, Sumit, Arjun, Alvin and Neeraj for making my lab days happy and memorable.

Abstract

Serialisation latency is a significant concern in modern cloud applications that leverage the microservice paradigm. A cloud service request typically traverses a sequence of microservices across nodes, increasing latency due to (de) serialisation overhead at every hop. The serialisation process comprises memory allocation, data encoding, and data copy. Observations from existing benchmarking results show that the data copy operation dominates the overall (de) serialisation cost.

Existing serialisation libraries follow a two-copy technique — (1) the application copies the encoded data into a serialised buffer, and (2) the serialised data is copied to the NIC's device memory. To reduce serialisation latency, researchers have proposed (1) kernel bypass techniques that eliminate data copy, (2) use of hardware acceleration solutions, and (3) wire format optimisations. However, kernel bypass solutions have security concerns and cannot be deployed in public cloud networks, and hardware acceleration solutions depend on specialised hardware.

We propose designing and implementing a one-copy serialisation library, which leverages the scatter-gather I/O technique provided by the standard POSIX library for data movement. Our solution does not require special hardware support or any specialised network stack. Our design relies on the Linux network stack; there are no security concerns, making it usable in public and private clouds.

Contents

Chapter 1

Introduction

Modern cloud applications have strict latency requirements. Some applications require an end-to-end delay of tens of microseconds, and some require hundreds of milliseconds. These latency requirements force developers to minimise almost all the overheads possible on the critical path. To minimise latency, applications like[1, 2] leverage hardware capabilities to minimise the latency and deploy mainly in the private cloud. General microservices applications deployed over the public cloud try to minimise end-to-end latency by minimising software components, i.e. the libraries.

In cloud applications, a request goes through multiple services. At every service, the request is processed, which includes deserialising the message, generating results based on the message and the business logic and then serialising and sending it. The (de) serialisation process happens on the critical path. Google studied[3] that its data centre almost spends 9.5% of CPU cycles serialising and deserialising the message. A request in a microservices environment spends most of the time in the (de) serialisation process [4].

[5, 6] found that the bottleneck step in the serialisation process is the *copy*. Multiple works have been done to address the performance of the copy step in different ways, like optimising the serialisation library, leveraging I/O techniques and building hardware. Some of these optimisations need specialised hardware and software stacks, which sometimes make these optimisations costly and raise security questions.

To optimise without the help of a particular HW/SW stack and without any security concerns, we explored μSer, a one-copy serialisation library. μSer, leverages the Linux kernel's scatter-gather API to reduce one copy overheads. For message sizes above 30KB, μSer, performs (de) serialisation orders of magnitude faster than the state-of-the-art serialisation libraries.

Chapter 2

Background

2.1 Cloud Applications

The state-of-the-art application designs break applications into small services. Each service is treated as an independent application and is built, managed and deployed independently. These applications have different characteristics and requirements. We broadly categorised the application into micro-second scale and microservices applications based on the latency requirements. Microservice applications are built on the microservice design paradigm, where an extensive application is logically broken into small services. Many of our day-to-day applications are based on this architecture. A request in a microservice architecture goes under multiple services. At each service, it is processed and forwarded to some other service, depending on the business logic. Generally, the services are deployed over containers, over the public cloud and orchestrated through Kubernetes[7]. Here, the applications need an end-to-end latency of hundreds of milliseconds.

Micro-second scale applications, such as caching, replication, etc., are generally part of microservice applications. These applications require a latency of tens of microseconds or below. They are developed using specialised networks and hardware like [1, 2, 8, 9] and deployed in a private cloud.

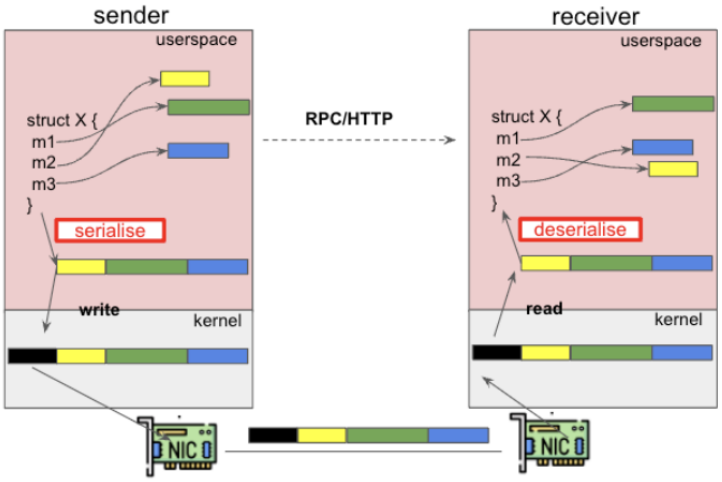

Figure 2.1: Communication between two applications

The communication between two services mostly happens using remote procedure calls (RPCs) or HTTP. The communication involves sending and receiving a message, an in-memory data structure between two remote devices. To send the in-memory data structure, the data pointed out by the structure's members must be gathered in a single place. This process is called serialisation. After the serialisation process, a system call like write or send is invoked, which copies the serialised

message from the user space to the kernel space. In the kernel space, the serialised message is added with some headers required for networking and eventually copied to the Network Interface Card (NIC). The NIC transmits the data through the connected wire. At the receiver end, the data is received at the NIC. The NIC copies the data to the kernel. After some checks, the data is delivered to the receiving application. After receiving the data, the data is deserialised, and an in-memory data structure is created.

The serialisation and deserialisation processes are two steps on the critical path of any request. These steps are necessary to make the communication happen and add significant costs to the sender and receiver. To reduce the end-to-end application latency, the (de) serialisation overheads must be minimised as much as possible.

2.2 Serialisation

The (de) serialisation process is one of the inevitable steps while remote devices communicate. These processes increase the request's latency. It also increases the CPU and memory consumption of the application. The state-of-the-art serialisation latency spans multiple microseconds.

The serialisation process consists of four steps: initialisation, encoding, copy and deallocation. An empty buffer, large enough to hold the serialised message, is allocated in the initialisation. Along with allocation, the library may perform extra operations depending on the implementation. The data is encoded in the encoding step following the serialisation library standards. The encoding happens either during the initialisation step or after the copy phase. Protobuf does this after creating the buffer, and Cap'n Proto does it during the initialisation step. Then, in the copy phase, the data is copied from their source location to the allocated buffer. The message is now serialised and is ready to be transmitted over the network. After the buffer usage is over, it is deallocated.

Like the serialisation process, deserialisation has the same steps in the opposite direction.

Steps	Protobuf	Cap'n Proto
Initialisation + Copy	201 ns	488 ns
Encode to Wire Format	351 ns	53 ns
Decode from Wire Format	491 ns	78 ns
Total	1043 ns	619 ns

Figure 2.2: Breakdown of steps while (de) serialising a message with a single 1024-byte-sized string field[5]

[5, 6] micro-benchmarked the serialisation libraries [10, 11] to find the bottleneck step. [5] showed that the cumulative cost of initialisation and copy contributes ma-

jorly towards the overall overhead. The encoding and decoding costs are enormous as it is They found that the *copy* takes the most time among all the steps.

2.3 Data Transmission Approaches

Transmitting data from a contiguous location is straightforward. The data address can be used to make an appropriate system call. While sending non-contiguous data, there are multiple ways provided by the operating system. One way is to serialise the data in the user space and then use the write system call to transmit it. This method is called the *two-copy* approach. It is called so because the data is copied two times, one during the serialisation process and then from the userspace to the kernel space by the write system call. Network packets are created by attaching networking headers once the data is copied to the kernel space. Eventually, the whole packet is transmitted via the NIC. The state-of-the-art serialisation libraries follow a library approach, similar to the two-copy approach, along with some serialisation headers. The serialisation headers keep extra information about the data, such as bitmaps, offset, etc. These two approaches incur extra serialisation costs, which occur in the user space.

The above two-copy approach is widespread and is used by almost all serialisation libraries. Apart from this two-copy approach, there are two other approaches where the user space copy is avoided. In the one-copy approach, the data is copied directly from multiple scattered locations in the user space to a contiguous area in the kernel space. Similarly, in the zero-copy approach, the non-contiguous data is copied directly from the user space to the kernel space without intermediate steps. The one-copy and the zero-copy approaches are serialisation-free. Here, the serialisation is not done explicitly; instead, it happens inherently due to the mechanism.

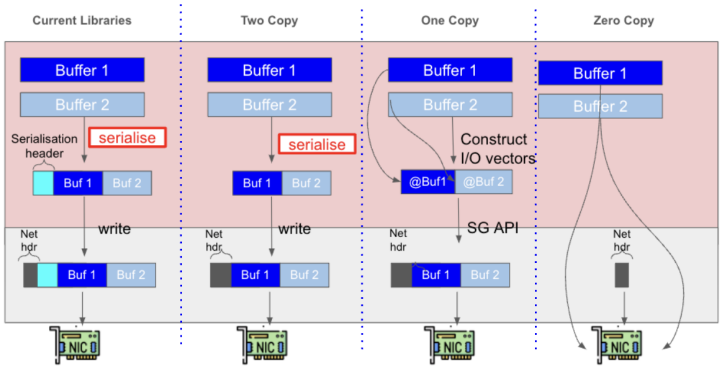

Figure 2.3: Non-contigous data transmission approaches[12]

In the one-copy approach, instead of serialisation, an I/O vector is constructed. An I/O vector consists of a vector of I/O segments. An I/o segment consists of a pointer to the data and the length of the data. These I/O vectors are passed to the appropriate one-copy API, discussed in the section provided by the operating system. During the invocation of the one-copy API, the I/O vector is copied to the kernel. From the vector, the total amount of memory required is computed, and memory for storing the data is allocated. After allocation, the I/O vector is iterated sequentially,

and data, pointed by the iterator, is copied from user space to kernel space. As it happens sequentially, the data gathered in the kernel space is aligned sequentially in a contiguous location. In this approach, there is only one copy cost between the user and kernel spaces, which is inevitable in the operating systems. Instead of the copy cost, the cost of the I/O vector creation is introduced.

The other approach to transmitting non-contiguous data is the zero-copy approach. In the zero-copy approach, data from the user to kernel space is copied directly. As the name states, this approach has zero copy cost. In this approach, the application has to satisfy some system requirements, as the data must lie on the pinned memory.

Chapter 3

Related work

3.1 Serialisation optimisation

The serialisation cost comes from all the components: initialisation, copy, and encoding/decoding.

The Protobuf[10] is a general-purpose serialisation library. It is usable on almost every platform and supports all the commonly used languages. It doesn't need any special hardware or software support. It is used as the baseline for all the other serialisation libraries.

Serialisation optimisation is divided into three categories: Library Optimisation, Zero-Copy I/O, and HW acceleration.

Methods	Details	Existing Solutions	Cons	Public cloud	Private cloud
Base line	General Purpose	Protobuf	Compute intensive	✓	✓
Library optimisation	Compute & memory efficient	FlatBuffers[13]	Huge serialised buffer size (~2x protobuf)	✓	✓
	Wire Format	Cap'n Proto[11]	Not maintained	✓	✓
Zero-copy I/O	Leverage scatter-gather I/O	Cornflakes[12]	spec. net. stack (DPDK)	✖	✓
HW acceleration	On-CPU	Optimus Prime[14]	Build HW + multi-tenancy	✓	✓
	Fixed funcion HW	Cereal[4] Protobuf Accel[16]		✓	✓
	NIC-enhance	Zerializer[15] Cerebros[16]		✓	✓

Figure 3.1: Serialisation libraries with optimisations

3.1.1 Library Optimisation

In the library optimisation, the optimisations are added at the library level, which includes the library implementation, wire format, encoding techniques, etc.

FlatBuffers[13] and Cap'n proto[11] serialisation libraries are optimised versions of the Protobuf[10]. The objectives of optimisations are different for both libraries.

The FlatBuffers[13] is designed for mobile devices and optimised for computed and memory. It provides lazy deserialisation, which offers zero-cost deserialisation. In lazy deserialisation, the whole message is not deserialised immediately after the receive. Instead, the elements are deserialised on demand. The wire format is also designed to provide on-demand deserialisation. To avoid additional cost, the wire format is in little-endian, as most of the devices in the world are little-endian, unlike Protobuf, which has a wire format in big-endian.

The Cap'n proto[11] is another optimised version of Protobuf. It dramatically reduces the encoding/decoding cost by merging the overheads at the initialisation time, which reduces the overall cost as described in 2.2. Also, it minimises the copy overhead by initialising the data structure to avoid some copies. Although it is pretty fast compared to Protobuf, the issue with Cap'n Proto is that it has meagre development resources and documentation.

These library-optimised libraries have less latency than Protobuf, but they still have the serialisation cost in terms of microseconds.

3.1.2 Zero-copy I/O

As the name says, the serialisation library voids the copy cost by leveraging the zero-copy I/O technique. This method is proposed by Cornflakes[12]. It leverages the NIC's scatter-gather feature to directly copy the message between the application and the NIC. It is built over DPDK[9], the kernel-bypass stack; it can raise security concerns in the public cloud. Although there are different ways to implement kernel-bypass in the public cloud, the security issue still arises because of less isolation.

To leverage the zero-copy, the message must reside in a pinned memory. It is the requirement of the DMA. So, to use zero-copy I/O, the application needs a memory manager over the pinned memory, which has its complexity.

3.1.3 Hardware Acceleration

The third approach to accelerate the (de) serialisation is to build customised hardware for the processes. The HW can be placed at different places in the system. [14] is an on-cpu HW accelerator. [15, 3] are fixed-function HW accelerator. [6, 16] proposed methods to perform (de) serialisation around the NIC.

Chapter 4

Design

4.1 Design of one-copy serialisation library

The zero-copy serialisation proposed by [12] has two negative points.

- It needs a specialised SW/HW stack.
- Memory management of the pinned memory.

To optimise serialisation without the help of unique requirements, we designed μSer, microsecond serialiser, a one-copy general-purpose serialisation.

μSer uses scatter-gather API provided by the Linux system call to implement the one-copy approach. It has no special requirements and works on Linux's network stack, which removes all the security concerns.

4.1.1 Linux Scatter-gather API

To implement one-copy, μSer uses the Linux kernel's writev[17] (gather output) system call. The API is the following.

Listing 4.1: Linux Scatter-gather APIs[17]

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);

struct iovec {
    void *iov_base; /* Starting address */
    size_t iov_len;  /* Size of the memory pointed to by iov_base */
};
```

The writev[17] system call writes iovcnt buffers of data described by iov to the file descriptor fd ("gather output").

4.1.2 μSer Components

μSer's serialisation framework has four components: domain-specific language (DSL), data access library, runtime, and wire format. The DSL describes the syntactic structures of the message schema and is similar to the protbuf's DSL. Unlike state-of-the-art libraries, μSer doesn't generate code that can be statically linked with the application code; instead, it does it at runtime. We chose this design for simplicity because we want to demonstrate the power of scatter-gather I/O and its effect on reducing the copy cost. μSer's data access library provides standard APIs to access the data and can be statically linked with the application. It provides APIs similar to the setter/getter APIs offered by the Protobuf but differs slightly. At the beginning of the program, the schema files are fed to the program, and the runtime is initialised. The runtime is responsible for building the I/O vector for the message serialisation and it intelligently handles the lazy deserialisation.

4.1.3 μSer Features

μSer provides primitive standard data types like integers and floats of different sizes of both signed and unsigned variants. Apart from the primitive types, it provides complex types, such as string and byte. The complex types are described by the data along with the length of the data. It doesn't provide some auxiliary data types, such as enums, but these can be created by using the provided types. By default, all the fields are optional in μSer. It also implements repeated and nested messages. Currently, the maximum number of fields is limited to 32. If more than 32 fields are required, the nested message can be used to satisfy the requirement. There is no encoding and decoding involved in μSer.

4.1.4 μSer Data-access Library

To access (set/get) any message, μSer's data-access library provides appropriate APIs. This library is statically linked with the program. This library and the runtime are tightly connected. The schema files are provided at the beginning of the program, and the runtime is initialised. It creates a symbol table from the schema files and references it in the future. After that, the program allocates a message to send and sets the fields. After that, it calls the initAndSerialise function, which returns an I/O vector. This I/O vector contains the pointer to the address of the data bytes to be sent over the wire, arranged following μSer's wire format. This I/O vector is then passed to the scatter-gather-based transmission API, such as writev, to transmit the data.

Listing 4.2: A simple message schema

```
message M {
  int32 id = 1;
  repeated string a = 2;
  repeated string b = 3;
 }
```

Listing 4.3: Serialisation usage

```
user::Message *m =
  user::InitMessage::allocateMessage("M");
m->setval("id", 45);
m->addVal("b", "foo1");
m->addVal("b", "bar");
ssize_t bytes_sent =
  m->serializeAndWrite(sock, user::SerializationMethod::SG);
```

Listing 4.4: Deserialisation usage

```
Deserialiser *de_obj = InitMessage::getDeserialiserPtr("M", buf, buf_size);
std::optional<int32_t> id_opt = de_obj->getVal("id");
if(id_opt.has_value()) {
    id = *id_opt; // id = 45
}
std::optional<uint32_t> a_size = 0, b_size = 0;
a_size = de_obj->size("a");
b_size = de_obj->size("b");
for (uint32_t i = 0; a.has_value() && i < *a_size; i++) {
```

```
    std::optional<byte_t> val = de_obj->getByte("a", i);
    std::cout << std::string(val->first, val->first + val->second)
              << std::endl;
  }
  for (uint32_t i = 0; b.has_value() && i < *b_size; i++) {
    std::optional<byte_t> val = de_obj->getByte("b", i);
    std::cout << std::string(val->first, val->first + val->second)
              << std::endl;
  }
```

The deserialisation doesn't use a scatter-gather array. The received buffer is forwarded. The deserialisation happens lazily like the FlatBuffers[13] and has zero cost.

μSer depends on the language's memory manager in an automatic memory-managed language like go. In the case of manual memory-managed language, the programmer has to call the destructor to delete the memory allocated explicitly.

4.1.5 Wire Format

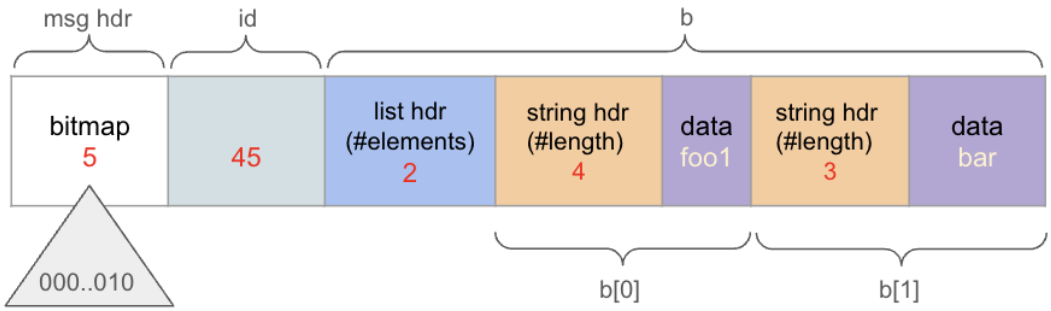

Figure 4.1: μSer Wire Format for 4.1.4

μSer's wire format is inspired by Flatbuffer, Cap'n proto, and Cornflakes[12]. The data is encoded in little-endian, as almost every machine supports little-endian. The wire format consists of the header and the data. The header comprises a 4Byte bitmap, limiting it to only 32 members inside a message. It can be extended to more than 32 members by adding an additional member to the header, which specifies the size of the bitmap. The header bitmap specifies which member is present in the message. After the header, everything in the wire format is data, in the wire format specified from the lower field to the higher field ID sequentially. A primitive type field is stored alone. A size field precedes complex data types. Fields with repeated type begin with a size field, specifying the number of times the field is repeated. These size fields are 4-byte unsigned integers. A nested message is also encoded as a message.

Chapter 5

Evaluation

This section discusses the empirical evaluation of one-copy and two-copy transmission APIs and compares the μSer library with other serialisation libraries.

5.0.1 API performance

The *write*[18] system call is used as the transmission API for the two-copy approach, and the *writev*[17] system call is used for the one-copy transmission API.

Figure 5.1: Steps in two-copy approach

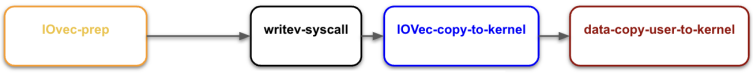

Figure 5.2: Steps in one-copy approach

Testbed setup : The testbed consists of client and server machines, over two Intel Xeon servers, running at 3.6GHz with 92GB RAM. The two servers are connected via 40Gbps QSFP cables through Netronome Agilio CX 40 Gbit/s dual port smartNIC (running in non-smartNIC mode).

For the experiment, we developed a simple TCP-based client and server program that is hosted over two servers. Both the client and servers are single-threaded and run for 10 seconds. At the server, a message is initialised with a fixed number of equal-sized members and is initialised with random bytes at different locations.

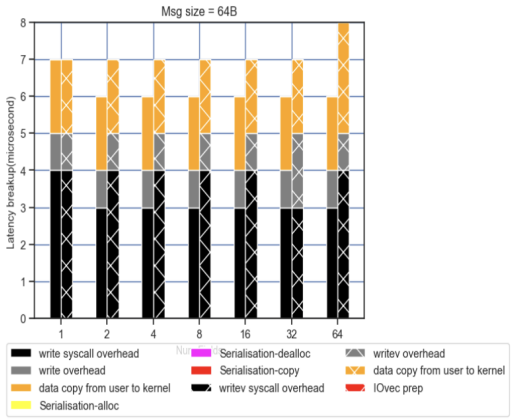

Figure 5.3: Performance of write vs writev for 64B size message

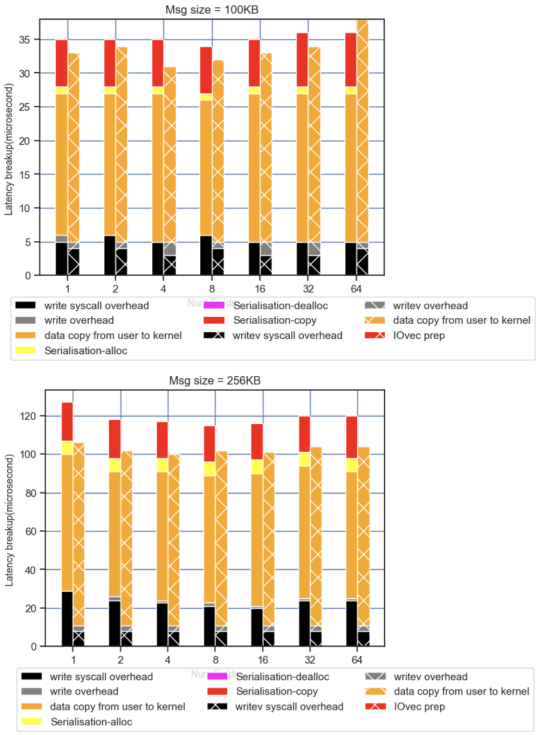

Figure 5.4: Performance of write vs writev for 100KB and 256KB size message

The client application sends 32-byte data to the server, and the server responds by serialising the non-contiguous bytes using either of the techniques set at the start.

The end-to-end steps in the two-copy approach are shown in 5.1. Similarly, 5.2 shows the steps in the one-copy approach.

We measured the latency of all the steps shown in 5.2 for the one-copy approach and 5.1 for the two-copy approach and also the extra overheads. For finding the in-kernel latency, we used [19].

Up to 100KB, the write performs better than *writev*. Beyond 100KB, the *writev* performs better than the *write*, increasing the latency difference.

5.0.2 Library Performance

This section discusses the performance of μSer with other libraries. We compared the cumulative serialisation and deserialisation latency of μSer one-copy, μSer two-copy, Protobuf[10] and FlatBuffers[13] libraries.

The testbed is similar to 5.0.1. The server is a single-threaded Echo Application. The client sends a serialised message to the server. The server deserialises it, then serialises it and sends it back. The message consists of two repeated string fields.

In a run, a single library performs the (de) serialisation operation, and the message size is set to a fixed value. Each run runs for ten seconds, and the average is calculated at the end. Strings of random length are set depending on the total size of the message.

5.5 shows the cumulative (serialisation + deserialisation) latency of various serialisation libraries. Above 30KB, both versions of μSer outperform the state-of-the-

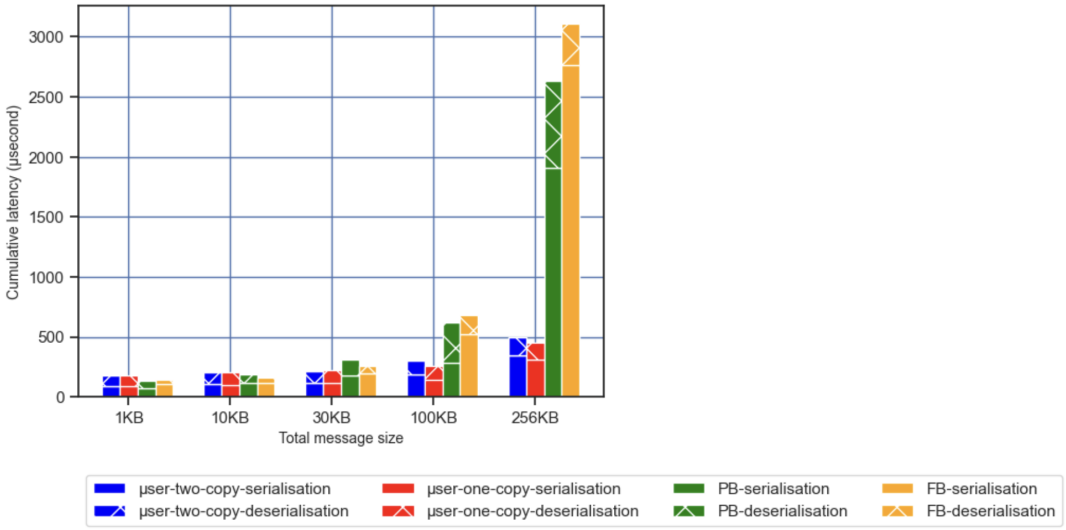

Figure 5.5: Latency comparison of μSer one-copy, μSer two-copy, Protobuf, FlatBuffers

art serialisation libraries. The difference increases with an increase in the size of the message. The difference majorly comes from the encoding/decoding step, as μSer doesn't perform any encoding/decoding operations like Protobuf or other serialisation libraries. It also has a simpler wire format header which again makes it less time consuming.

The μSer one-copy outperforms the two-copy after 30KB, and the difference also increases. In this case, the μSer one-copy has less copy cost (because of no serialisation) than the two-copy.

Chapter 6

Future work

- Identifying bottleneck component of scatter-gather API for small message sizes: Currently, the performance of scatter-gather API, *writev* is performing badly as compared to two-copy API, *write*. Our plan is to go deep into the kernel, identify the bottleneck point, and fix it, if possible.
- End-to-end performance: We will test the end-to-end performance improvement on the deathStar benchmark suite[20].
- Exploring zero-copy I/O: The Linx provides support for zero-copy I/O. Currently, there are very few resources and usage of it. Our plan is to explore it more and find the performance, gaps, and simplicity of the technique.

Bibliography

[1] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter rpcs can be general and fast," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (USA), p. 1–16, USENIX Association, 2019.

[2] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, nov 2015.

[3] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for protocol buffers," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 462–478, Association for Computing Machinery, 2021.

[4] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, "Dissecting service mesh overheads," 2022.

[5] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang, "Breakfast of champions: Towards zero-copy serialization with nic scatter-gather," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, (New York, NY, USA), p. 199–205, Association for Computing Machinery, 2021.

[6] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: Towards zero-copy serialization," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, (New York, NY, USA), p. 206–212, Association for Computing Machinery, 2021.

[7] "Kubernetes 2023." Available online at `https://kubernetes.io/`.

[8] "A rdma protocol specification.." Available online at `https://www.rdmaconsortium.org/`.

[9] "Dpdk: Data plane development kit." Available online at `https://www.dpdk.org/`.

[10] "Protocol buffers 2023." Available online at `https://protobuf.dev`.

[11] "Cap'n proto 2023." Available online at `https://capnproto.org`.

[12] D. Raghavan, S. Ravi, G. Yuan, P. Thaker, S. Srivastava, M. Murray, P. H. Penna, A. Ousterhout, P. Levis, M. Zaharia, and I. Zhang, "Cornflakes: Zero-copy serialization for microsecond-scale networking," in *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, (New York, NY, USA), p. 200–215, Association for Computing Machinery, 2023.

[13] "Flatbuffers 2023." Available online at https://google.github.io/flatbuffers/.

[14] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 1203–1216, Association for Computing Machinery, 2020.

[15] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A specialized architecture for object serialization with applications to big data analytics," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 322–334, 2020.

[16] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the rpc tax in datacenters," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 407–420, Association for Computing Machinery, 2021.

[17] "Linux writev system call." Available online at https://linux.die.net/man/2/writev.

[18] "Linux write system call." Available online at https://linux.die.net/man/2/write.

[19] "bpf: Berkeley packet filter." Available online at https://docs.kernel.org/bpf/.

[20] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," ASPLOS '19, (New York, NY, USA), p. 3–18, Association for Computing Machinery, 2019.