sRPC Design and Evaluation

March 2024

1 Design

sRPC follows a very simple design and uses prevalent techniques such as Kernel Bypass, RSS, lock-less rings to achieve better performance. It provides transparency in RPC call and the semantics are similar to calling a function. Transparency is often overlooked when RPC framework are specialised for a hardware or use niche techniques like RDMA.

1.1 Standard RPC Services

If one looks at standard RPC frameworks, all their components can be broadly belong to the three components listed below. Figure 1 outlines how each component interact.

- **RPC Library:** A library which implments network protocols, application use this library to create a RPC server , clients use it to connect to server and communicate.
- **Stub:** These are ususally some encapsulation of a RPC requests which hides away details of argument marshalling, unmarshalling and provide a transparent way of calling RPC as if called on a local machine.
- **Protocol Compiler:** A simple compiler which takes as input a contract / definitions of RPCs that a server runs and clients call.

1.2 Overview

1.2.1 sRPC API

I will be adding a hello world example here to explain the API.

1.2.2 Transport

sRPC at a high level uses a transport layer abstraction to hide network related implementation. Figure 2 draws a bird eye's view of how a client-server setup looks in sRPC.

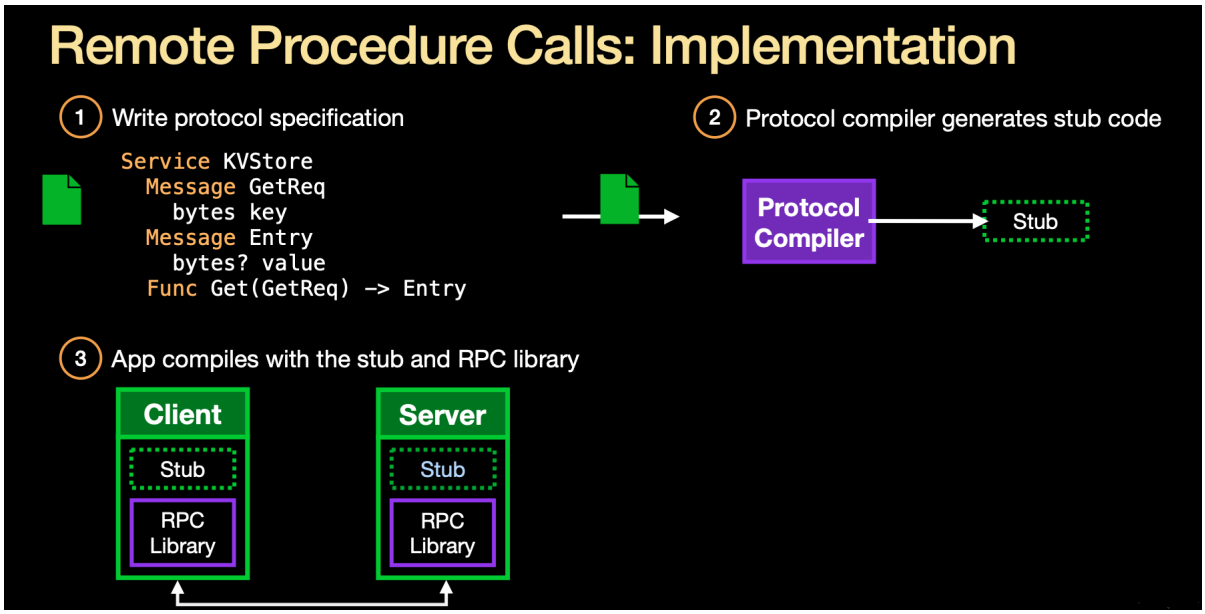

Figure 1: A Generic RPC framework

- The DPDKTransport layer provides a clean (probably not the cleanest) way to deal with low level details of userspace networking without much overhead.
- The transport layer provides abstractions such as connections and marshals which are easier to deal than pure network pakcets and NIC rx / tx queues.

1.3 Abstractions

In the previous section we talked about how transport layer provides abstractions which are easy to deal.

sRPC clients and servers both uses TransportConnection and TransportMarshals to communicate with each other. In next section we describe how a RPC requests flows throughouts lifetime.

1.4 RPC Flow

- A TransportConnection object has lock-less rings which are polled by dpdk-threads to enqueue incoming requests/replies (inrings) or look for fresh outgoing requests/replies (outrings).
- A client through a stub (Proxy class) calls a wrapper RPC method and internally the this wrapper creates a Transport Marshal object with all

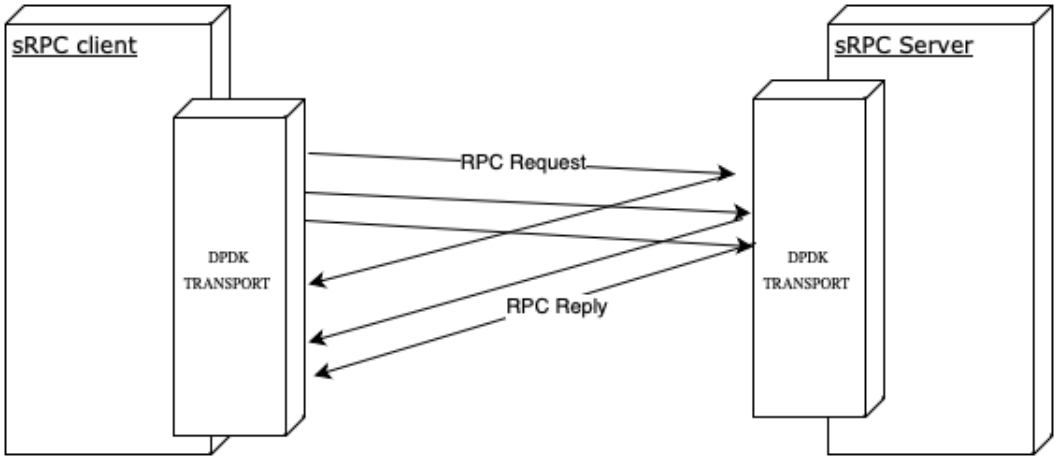

Figure 2: sRPC high level design

argument marshalled into a packet. Finally the packet which this TransportMarshal uses is enqueued in one of the outrings.

- DPDK Transport layer threads pick this packet from outrings and put them on NIC queues to be transmitted.
- On the server side DPDK Layer receives a packet, it either enqueues this packet directly in on of the inrings or handles it itself depending on how the server is setup.
- The server thread responsible for handling RPC requests gets the packets from inrings, creates a RPC Request object and passes it to a RPC handler wrapper method from the stub service class.
- The stub service class uses the RPCRequest object to unmarshal the arguments from underlying packets as a TransportMarshal and calls the actual RPC function.
- After the the end of the call the wrapper method does similar process and prepares a reply using TransportMarshal and finally enqueues it on outrings.
- DPDK Threads picks up this reply as they constantly poll for new entries in the buffer and put them on NIC queues to network.
- When a reply is received a similar process takes place to handle reply.

1.5 Scalability

I will be explaining how sRPC scales with multiple client connections and server threads.

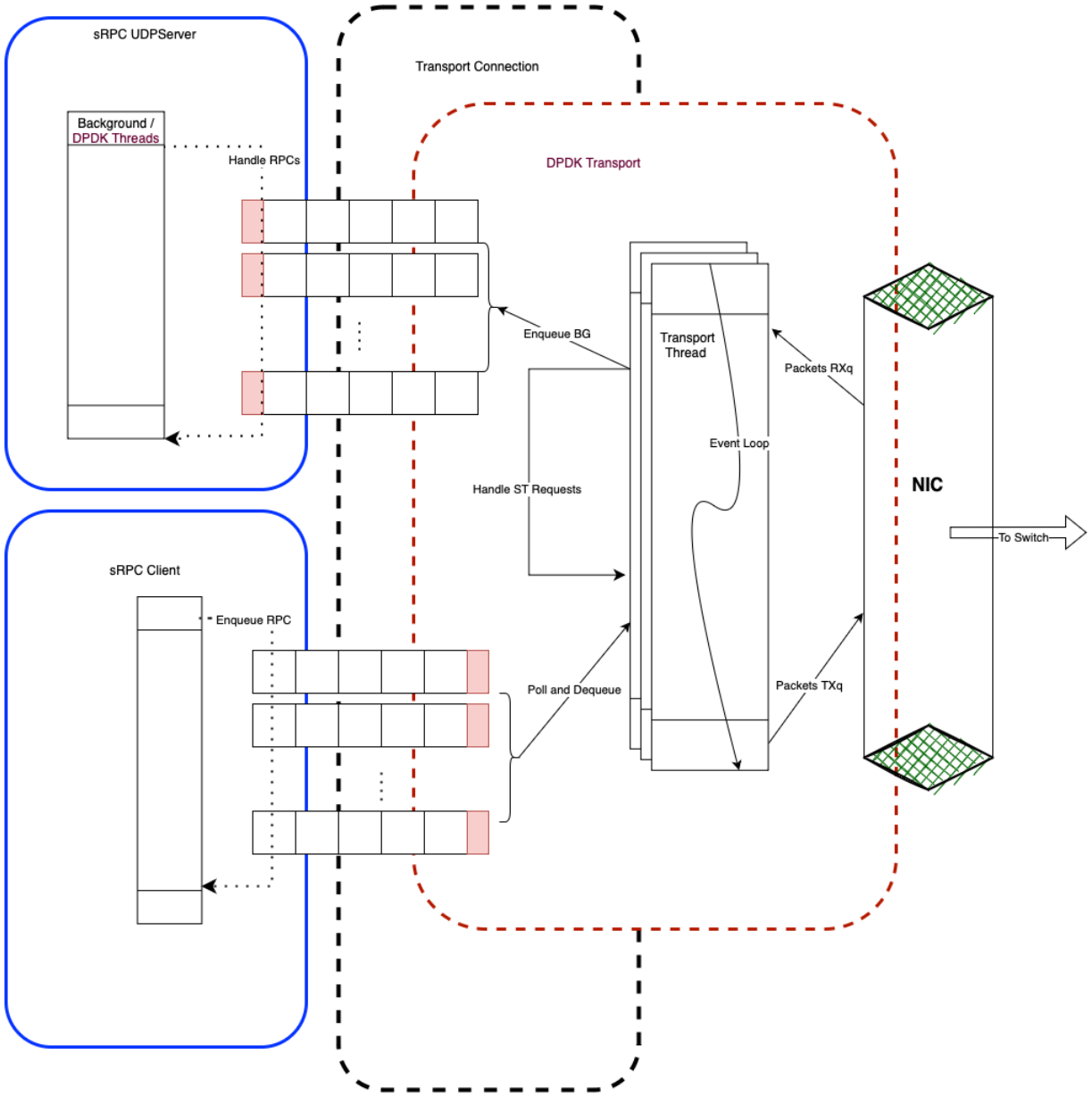

Figure 3: Interctions between different sRPC components. The pink colored boxes represent head of inrings and outrings. ST is single thread and BG is background.

2 Evaluation

sRPC is implemented in CPP along with a compiler written in python which generates stub class. In this section we evaluate the performance of sRPC in different settings and answer several question:

- How does latency and RPC rate look like in a simple scenario where a single server thread handles requests from a single client thread?
- How does sRPC scale ?
- What is the cost of submitting requests to background threads?
- What is the contribution of optimizations used in the designing and implementation of sRPC?
- How efficient is sRPC, how many threads does it take to saturate NIC bandwidth.
- How does sRPC performs when it is used to implement a complex distributed protocol like Raft?

2.1 Micro Benchmarks

2.1.1 Small RPC Latency

In this experiment we measure how much latency does sRPC server adds while handling requests. We run a client simulator which can bombard the server with requests at arbitrary rate and we measure the Round Trip Time (RTT) latency against different rates. The server in this case receives 97B sized requests and responds with 97B reply, the requests carry a 64B sized string as an argument for the call along with some metadata such as RPC ID requests size, a timestamp to measure latency etc. The server replies with a 64B string as a reply along with some metadata. Figure 4 plots the latency observed against the RPC requests submitted per second.

2.1.2 Cost of submitting a request to background thread

With this experiment we try to find cost of submitting a request to background thread instead of handling it in the same thread which receives requests (which runs the event loop).We use the drop in peak throughput for a single server thread and latency characteristics to provide comparison between single thread handling vs submitting to background thread. Figure 5 shows the drop and latency and throughput for a server. When requests are handled by a background thread, the peak throughput drops and the transfer of request adds significant latency.

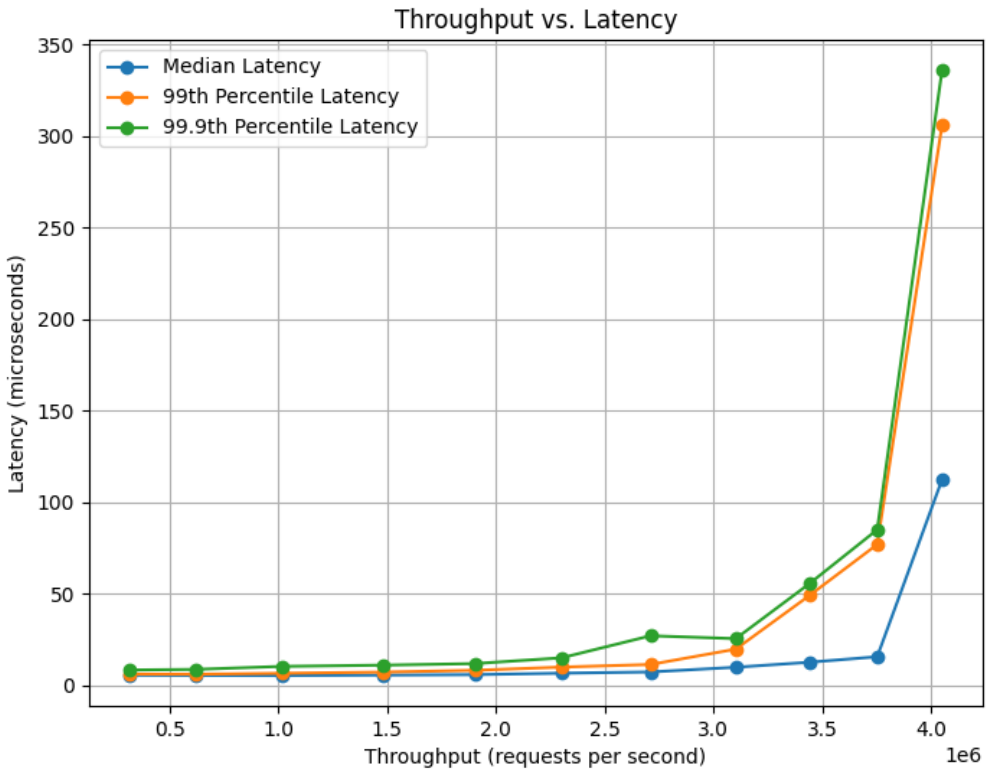

Figure 4: Small RPC latency vs RPC rate

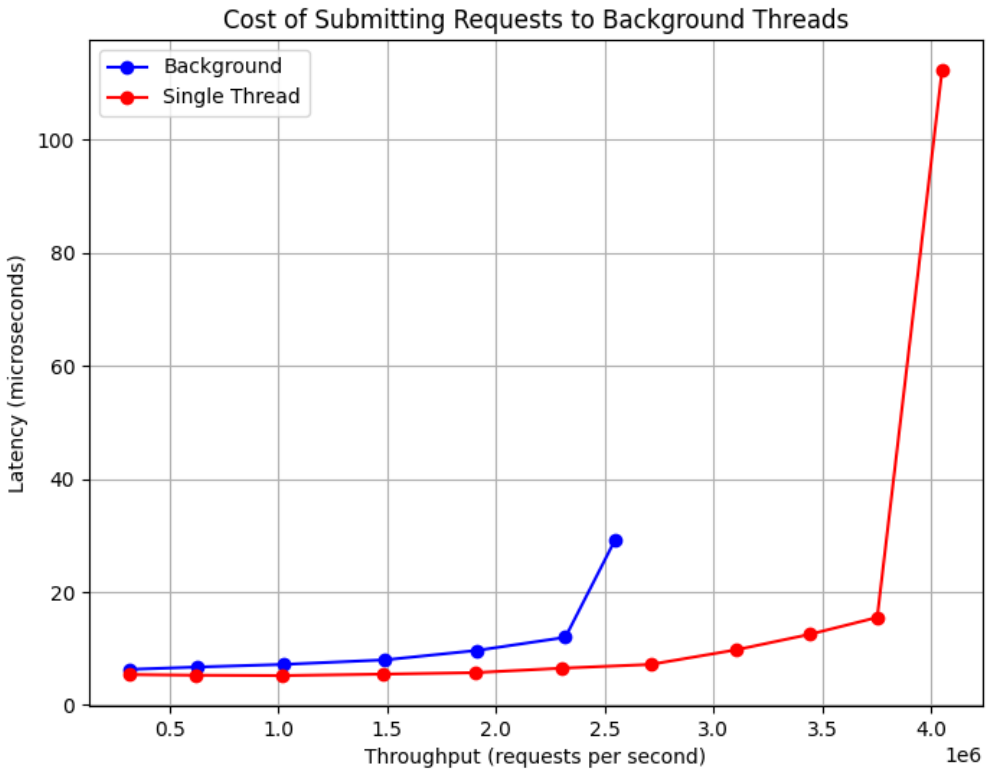

Figure 5: Cost submitting a request to background thread

Technique	Peak Throughput (Mrps)
Base	0.6
Zero Copy Marshals	1.9
Single thread	3.6

Table 1: Peak throughput handled by a minimal server with given optimisation technique

2.1.3 Contribution of optimizations

This figure tries to approximate contribution of each optimization technique we have implemented in sRPC server.

- Kernel Bypass Technique (DPDK).
- Zero Copy Transport Marshals.
- Single thread requests handling.

We use peak throughput for a single thread as a metric to measure the impact of each optimization. Figure 6 shows the gains in peak throughput as a we add optimisation techniques. 1 summarizes the gains with each technique.

2.1.4 Scalability benchmarks

We perform three experiments to measure scalability

- **How does it scale with request size?**: we increase request size from 64B to 1280B and use a single client against a single thread server. Figure 7 shows how throughput scales as we increase requests size.
- **How does server handles requests from multiple clients?** We run several clients on a single machine , each client is handled by a different thread in the transport layer. The server handles all these client through a single thread. Figure 8 shows the observed throughput at server as we increase clients from 1 to 6.
- **How does client utilises the NIC bandwidth available to it, when they share a transport layer?** in this experiment we run several clients sharing the same transport layer, each client is handled by single dpdk thread. On the server side each thread handles 2 clients. Figure 9

  Shows how bandwidth utilisation increases at first (till 10 clients) and then starts dropping as, client contend on transport layer transmit rings, access to future table available mbuf structures.

2.2 Full System benchmarks

Time taken to reach consensus on single log entry: in N=3 nodes. Through put on consensus. Number of appends that can happen in a second.

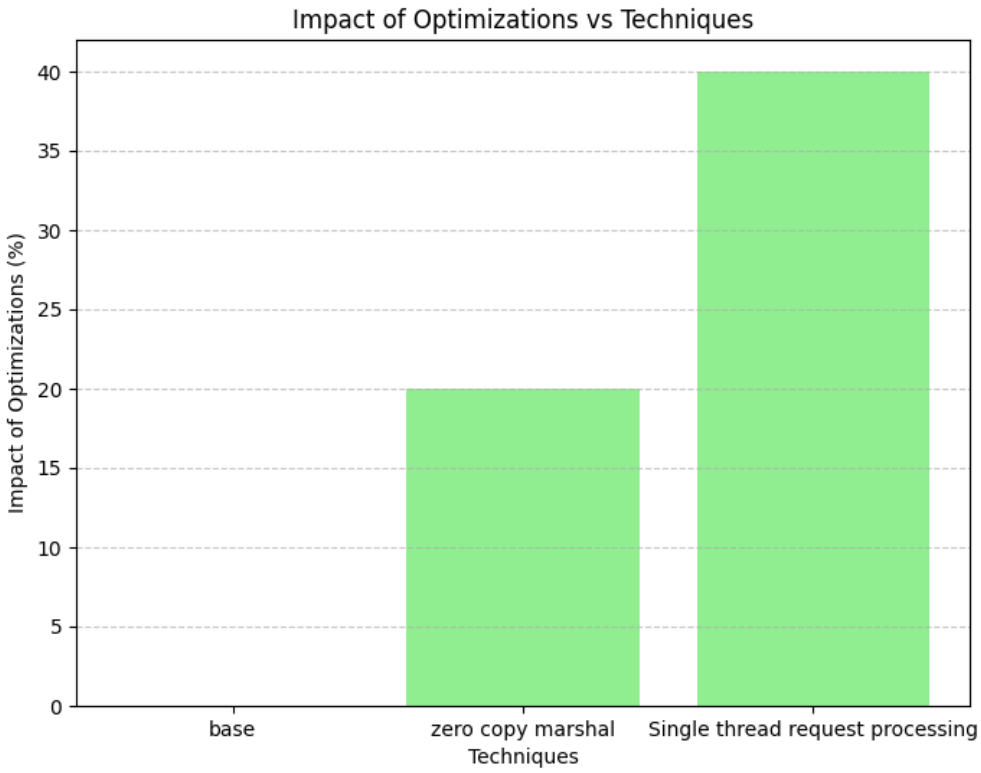

Figure 6: impact of optimization techniques

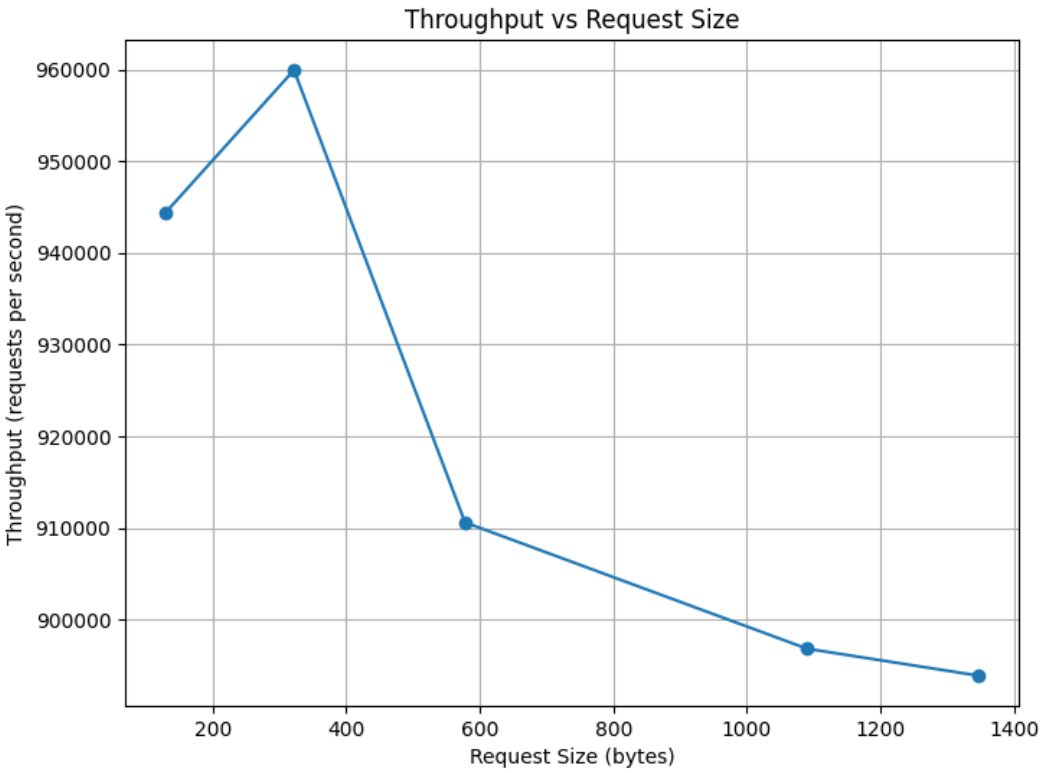

Figure 7: RPC rate vs Request size

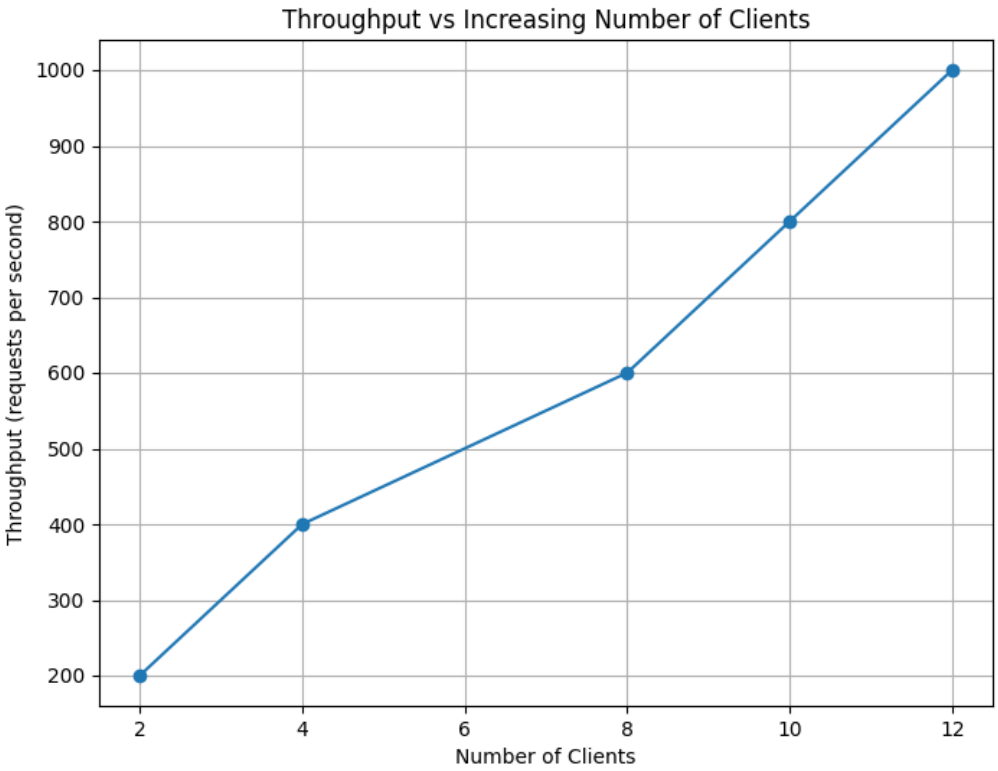

Figure 8: Throughput observed at server as it handles multiple clients

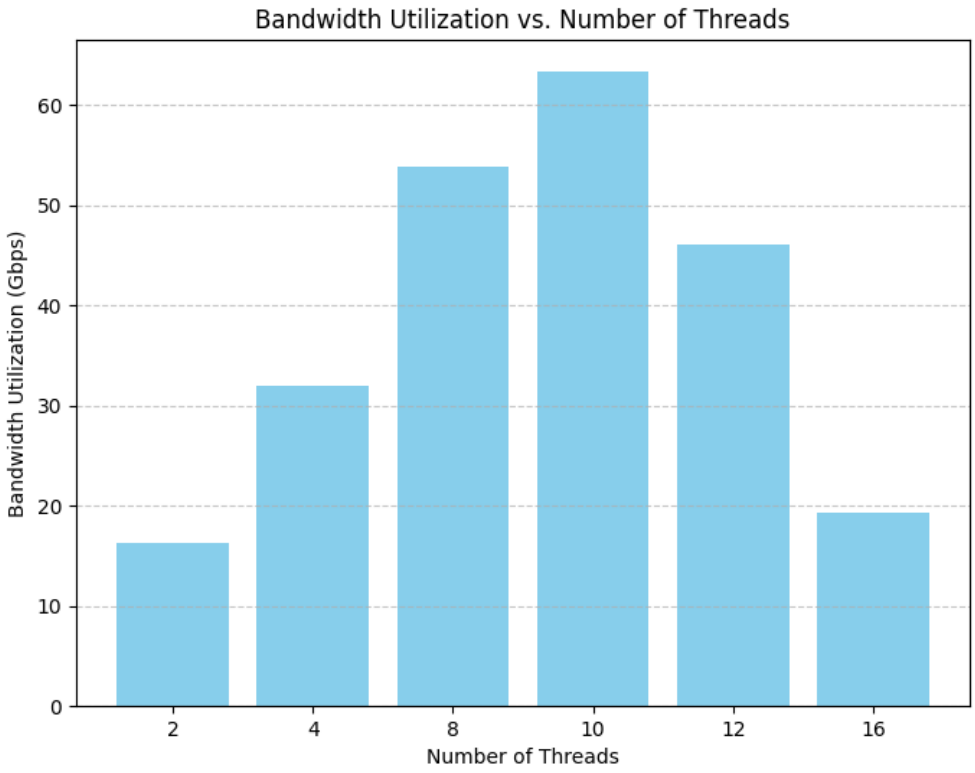

Figure 9: bandwidth utilisation by clients threads

3 Conclusion