RAMCloud RPC Performance Over 10G Ethernet Fabric

Behnam Montazeri, Stanford University

1. INTRODUCTION

RAMCloud [Ongaro et al. 2011] is a distributed, low latency, in memory storage system for datacenter applications that provides scalability and high availabilty by means of replication and fast crash recovery. The data model offered by RAMCloud storage is either key-value pairs or secondary indexes. The high level architecture for RAMCloud is as follows: there are three types of nodes in a RAMCloud cluster. A node can be a coordinator, a master or a client or any combination of these. At any point of time there is only one active coordinator in the cluster and any number of masters or clients. Coordinator is the node that assigns the ranges of keys to the masters and keeps track of which master is holding which part of client application data. The masters are essentially the storage nodes of the system that keep the client application data on themselves. Clients of the RAMCloud are the nodes that keep the client library and applications can talk to RAMCloud storage through them. When a client wants to read or write data in RAMCloud, it first issues an RPC the coordinator to get the address of the master server that keeps the data for that client. The coordinator will respond to client RPC with the address of the server that keeps the range of the keys client is interested in. The client at this point caches that address information for that particular master server for future references to that server. The client can then issue RPCs to that server to write a key-value pair on the RAMCloud or read a previously written value object associated with a key from that server. Figure 1 shows the high level architecture of the RAMCloud system.

2. MOTIVATION

RAMCloud RPC system is one of the lowest latency RPC systems designed up until today. The low latency characteristics in RAMCloud's RPC has been achieved using a dispatch-worker threading architecture and highly optimized code path for critical operations when an RPC in being serviced. Other than that, one crucial part of the

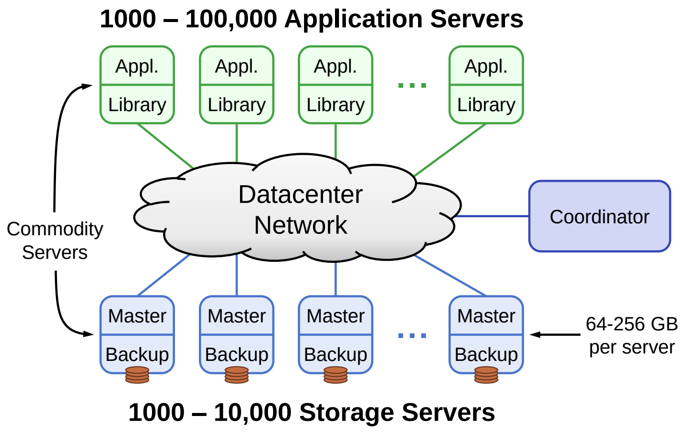


Fig. 1. RAMCloud high level architecture

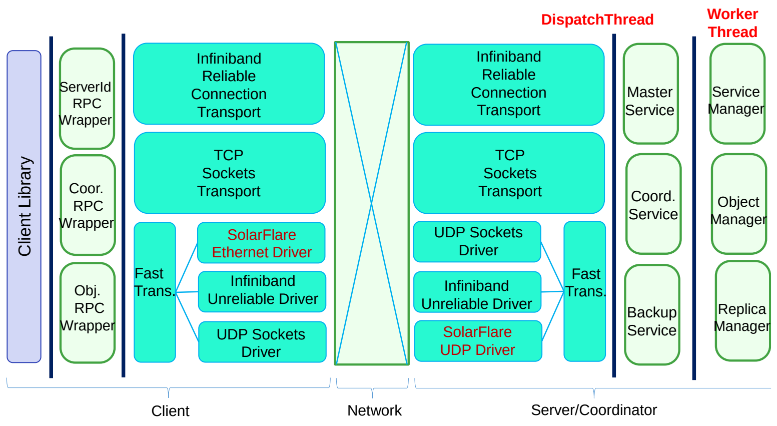


Fig. 2. RAMCloud RPC Architecture

RAMCloud RPC system is the Melenox Infiniband transport which has been the main transport system for RAMCloud over the last couple of years. Infiniband fabric are designed based on RDMA protocols and provide low latency and high throughput that are ideal for High Performance Computing applications but they are traditionally expensive and not compatible with commodity Ethernet base networks. Even after more than a decade that Infiniband has been around, most people still prefer to use Ethernet fabric in datacenters.

As a part of RAMCloud team, we are mostly involved in the design and improvement of RAMCloud RPC system and one of the challenges that We've been facing is design of a low latency transport protocol over Ethernet physical layer that matches the latency of RAMCloud Infiniband transport. While this is still an ongoing project, over the course of this quarter I've been mainly working on a low latency driver code and some parts of transport protocol for SolarFlare 10G Ehternet network interface (one the commodity 10G Ethernet NICs that is used by industry for low latency applications). Initially for this project we started very ambitious and wanted to show the first version of an end to end design and implementation of the low latency and scalable user space transport protocol for RPC systems but gradually it turned out to be much more work to be done over a quarter time period. Therefore here we will only discuss the design and implementation of a low latency user space 10gig Ethernet driver code that could directly plug into RAMCloud RPC system. We will show the latency and bandwidth measurements of this design for SolarFlare NIC in RAMcloud RPC system and compare the latency with the Infiniband base transport.

3. RAMCLOUD RPC ARCHITECTURE

Figure 2 shows the current RPC architecture in RAMCloud. In general after clients make their requests, the client library will marshal their requests and pass the request to the transport software stack where it will be raliably transmitted to the coordinator or master (depending on the type of the request) to handle the request. On the other side the request will be received in the transport stack. Once all packets are put together and the request is complete and the type of the request is known (might be an rpc for Master, Coordinator, or even a backup node), the Dispatch thread will pass it to the worker threads to handle the request rpc and prepare an appropriate response for it. For example if the rpc request is of type “read” or “write”, then the worker thread will handle it using the Object Manager and if the request is recovery related one, then it'll be handled by the replica manager. Once the response is ready, the reply rpc will be passed to the Dispatch where it will be transmitted to the client using the transport stack protocol and eventually the client will receive its response back.

4. RPC TRANSPORT AND UDP DRIVER

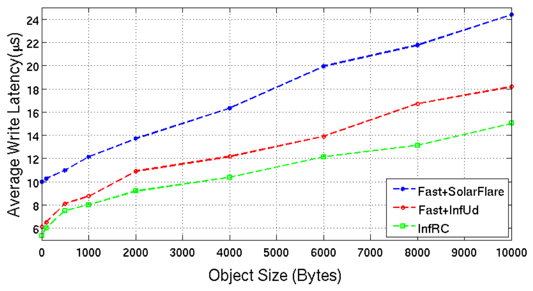
Traditionally most of the current RPC systems use a reliable delivery transport like TCP or some kind of low weight transport based on UDP protocol. However when we are optimizing for sub 10us round trip latencies this traditional transport protocols will have fundamental limitations. First of all most of the available implementations of TCP and UDP are done in the Kernel and their code path in the Kernel is somewhat hairy and complex with lots of unnecessary overheads. Therefore we think the right low latency transport protocol must be implemented in the userspace not in the Kernel. On the other hand, although there are implementations of the TCP and UDP available in the userspace, still TCP does not seem to be right protocol because of the over heads in protocol design and implementations (eg. Scalability issues) and UDP is too shallow of transport protocol (eg. no congestion control or flow control mechanism in the protocol). Therefore the ideal transport protocol for RAMCloud type of applications and low latency RPC systems in datacenters must be implemented in userspace. A transport that provides low latency and scalability (in the order of millions of connections per server) and also would react properly to possible congestions happening in datacenter network (ie. Congestion aware protocol).

Although this is still an ongoing project, in this paper we are more focussed on the userspace UDP driver over the 10G Ethernet for our RPC system and less focussed on the transport protocol itself. The driver code is the software part that talks to the NIC and provides the abstractions to send and receive packets from the NIC. Here we discuss the design considerations for the UDP driver in the RPC system:

— In general there are two approaches to receive packets from NIC. We could either configure the driver to fire interrupts when a packet is received or we can continuously poll on the NIC's notification queue and find out the received packets as soon as possible. Although event and interrupt based driver will save the cpu from unnecessary idle polling, it has significant delay and jitter penalties on the end to end latency therefore we prefer to use active polling rather than interrupt/event based design. Furthermore, in today's datacenter it's very common that a commodity machine has 16 or even 32 cores that are not even fully utilized. So having one of the cores actively polling the NIC does not seem to be cause a problem.

— SR-IOV and IO-MMU: Most NICs these days send and receive the packets directly from machine's main memory by DMAing into RAM. This exposes a problem that is the NIC needs to have a mapping from packet buffers in its TX/RX ring to the memory addresses. NIC's default mapping is done based on an address table on the NIC hardware but usually the on chip memory for this table is limited in size and you wont be able to address the whole machine memory. One work around for this is to use SR-IOV virtual functions to setup an address map to a larger chunk of machine's memory address space. We initially used this approach but we observed 1.7us overhead in each way in our RPCs and eventually we took it out.

— Zero-Copy: When a packet is received or being transmitted the NIC will DMA into the memory address for the buffer that contains the packet. Ideally we would like to hand over the responsibility of that buffer to the higher level code and use a different buffer and this save us from an extra memcopy in the critical path. But usually the problem is that NIC's buffers are limited (because the size of on chip address table is limited) and there's always the possibility that NIC runs out of the buffers and



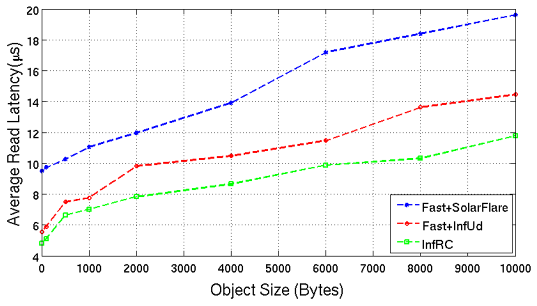


Fig. 3. Read and Write latency for different objects over different transport softwares.

higher level code will not return NIC buffer in a timely manner. So although we implemented zero-copy feature but we ended up not using it.

— Task offloading: These days the network interfaces provide many ways to assist the transport software and accelerate some inevitable tasks by doing them in hardware. Example of these offloaded tasks are checksum calculations and connection offloads. Although we are not using these features now, but we will definitely use them in near future.

— Layer 3 addressing: The user space driver must be capable of sending and were thinking to just query the Kernel cache each each time but this was causing unpredictable and long delays so we ended up implementing a userspace ARP cache for RAMCloud. We query the Kernel ARP cache once and cache the translation in our cache. If the translation is not present in Kernel cache, we trigger the kernel ARP module to resolve that translation for us. This operation is expensive but in only happens once for every new IP address and is tolerable.

5. EVALUATIONS AND CONCLUSION

Although the design is genral to any 10G Ethernet but the implementation depends on the manufacturer's provided low level driver codes. We have implemented the driver for SolarFlare 10G NICs and tested the basic read and write operations on 6 node cluster. Figure 3 shows read and write RPC latencies for various object sizes with three different transports. Since the Ethernet driver itself sends and receives UDP packets, in order to provide reliable packet delivery, we use a RAMCloud's low overhead transport protocol called FastTransport. FastTransport provides reliability by sending SACKs for every window of packets but does not provide any congestion control mechanism. InfRC on fig.3 corresponds to the RPC latency when the underlying transport is using infiniband reliable queue pair connections. Fast+InfUd means that the underlying transport is combination of FastTransport and unreliable Infiniband datagrams and Fast+SolarFlare stands for 10G Ethernet driver that we have developed. As we can can see InfRC transport provides 5us latency for every few bytes RPC while the 10G UDP driver has 9.7 us latency. Table 1 shows where all the latency is going for small 100Bytes read request RPC from the moment that RPC is issued in the client until the client will receive the response back and object is ready. As the object size gets larger, then the latency will be a function of NIC transmission rate and since Inifiniband NICs has 48gb/s rate, they perform considerably better than SolarFlare 10gb/s Ethernet NICs.

The measurement in table are the delays in nanoseconds of different parts of the RPC systems as the request is being processed in the system's critical path pipeline. The table compares the two cases of InfRC transport and SolarFlare Transport. The

Table I. Delay of the critical path modules when an RPC is being processed

InfRcTransport		Fast+SolarFlareDriver	
Client	**Server**	**Client**	**Server**
Marshaling: 103ns	—	Marshalling: 147ns	—
Transport: 170ns	—	Transport: 109ns	—
NIC communication: 273ns	—	NIC communication: 100ns	—
—	NIC communication: 521ns	—	NIC communication: 782ns
—	Transport: 118ns	—	Transport: 378ns
—	Preparing Response: 814ns	—	Preparing Response: 969ns
—	Transport: 220ns	—	Transport: 181ns
—	NIC communication: 228ns	—	NIC communication: 121ns
NIC communication: 485ns	—	NIC communication: 715ns	—
Transport: 144ns	—	Transport: 144ns	—
Unmarshalling: 73ns	—	Unmarshalling: 81ns	—

measurements are done on two different but very close cluster hardwares. The CPUs of the two clusters are both 8 core Intel Xeon processors and the SolarFlare test cluster is within 10% to 20% slower than InfRc test bed. The InfRc values in the table are calculated based on the measurements reported in [Ousterhout et al. 2014].

Moreover, we have measured the network latency for our networks and we know that the round trip latency is somewhere between 1.2 to 1.3us for the our 10G Ethernet network and 200 to 300ns for our Infiniband network. From these latencies values and and end to end latencies of fig.3 we can conclude that although our 10G transport stack still needs to be improved, most of the small RPC latency overhead in 10G case is coming from 10G NIC. Almost 4.7us of the total 9.7 us latency of small reads is the overhead of NIC transmit and receive architecture that we don't have any control over.

REFERENCES

Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 29–41.

John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seojin Park, and Henry Qin. 2014. RAMCloud Storage System. (2014). https://ramcloud.atlassian.net/wiki/download/attachments/6848571/RAMCloudPaper.pdf