Some content adapted from Matei’s NSDI talk.
Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.
Some material taken/derived from MIT 6.824 by Robert Morris, Franz Kaashoek, and Nickolai Zeldovich.
Lowering Networking Overheads

• Kernel TCP & UDP
  • RTT today between two hosts largely dominated by kernel time and abstractions
    • Inherent copying in (sockets) interface
    • Privilege boundary crossings
    • Scheduler interactions (in both directions)
      • Possible context switch, page table switch, TLB flush
    • Interrupts on receive side
      • ISR and softirq handlers – coherence traffic
      • Wake core from C-state, schedule process, more context switch

• Turns < 1 µs wire time into > 40 µs RTT
• TCP processing at > 100 Gbps uses all CPU resources
  • 15 ns of wall time per packet even at MTU
Kernel-bypass
Kernel-bypass Frameworks/APIs

• DPDK
  • Work with raw packet buffers for Ethernet frames
  • NIC control structures mapped into app
  • App “posts” buffer descriptors into tx and rx queues
  • App has to deal with frames, reorder, loss, etc.
    • Designed for “dataplanes”; Ethernet focus
    • No bells and whistles

• ibverbs
  • Work with messages
  • NIC control structures mapped into app
  • App “posts” buffer descriptors into tx and rx queues
  • Both reliable (RC) and unreliable modes
    • Designed for Infiniband; low-latency messaging focus
    • High-level API, extra features like one-sided RDMA
## Fast Networks

- **Ethernet**
- **Infiniband**

<table>
<thead>
<tr>
<th>Gbps</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2002</td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>2010</td>
</tr>
<tr>
<td>100</td>
<td>2010</td>
</tr>
<tr>
<td>200</td>
<td>2017</td>
</tr>
<tr>
<td>400</td>
<td>2017</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gbps</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDR</td>
<td>8</td>
</tr>
<tr>
<td>DDR</td>
<td>16</td>
</tr>
<tr>
<td>QDR</td>
<td>32</td>
</tr>
<tr>
<td>FDR</td>
<td>54</td>
</tr>
<tr>
<td>EDR</td>
<td>100</td>
</tr>
<tr>
<td>HDR</td>
<td>200</td>
</tr>
<tr>
<td>NDR</td>
<td>400</td>
</tr>
<tr>
<td>XDR</td>
<td>1000</td>
</tr>
</tbody>
</table>
Verbs usually work directly with NIC control structures in app
Different interface than sockets, but still usually “connected” client-server
Why the hype?

• Remote Direct Memory Access
  • Similar to local DMA in some respects
• Schedule NIC for remote xfers, asynchronous without remote host disruption
• Low-latency (low as µs one-sided); no kernel time
• Zero-copy; no copying in/out of buffers
• Hardware handles transport, app works at message/op level, saving more CPU
• End result: low-latency, higher-throughput at finer-message granularity
Verbs

• Two sided: send and recv
• One sided: read and write and atomics

• Unreliable Datagram mode
  • Only send and recv, and only up to 2 to 4 kB messages
  • No integrity, ordering, or delivery guarantee
  • Supports multicast

• Reliable Connected mode
  • Supports send, recv, read, write, atomics
  • At-most once on connection (ack when in-memory)
  • Up to 2 GB message size
  • Scaling issues with many QPs; problematic past 100s of QPs
High-level RC Workflow

• Call kernel module to map NIC control registers
• Register regions of process address space
• Create a “completion queue”
  • NIC fills completion queue with notifications of completed “work requests”, e.g. send has completed
• Create an RC “queue pair” with the other end
  • Need to exchange a QP number, LID, ...
• Receiver: post receive buffer descriptors
• Send: post send buffer descriptors
• Both: “poll” CQ to find incoming events
Memory Registration

• NIC must do address translation
  • App knows virtual addresses, NIC DMAs physical
• OS cannot swap out/evict registered memory
  • If buffer posted in send/recv, NIC may be accessing it!

• Solution: register and “pin” memory
  • NIC knows VA to PA mappings for registered region
  • Implements on-card address translation
  • Scaling issues if registered region is large and physically discontinuous, so use huge pages
  • Applies to both send/recv and read/write
  • Too expensive to do this on the critical path
Two-sided “RDMA”
One-sided RDMA
One-sided vs Two-sided

- One-sided
  - No CPU use at target
  - Messaging scales at NIC, then scales indep. of CPU
  - Constrained semantics

- Two-sided
  - Busy-wait plus CQ/RX interaction costs
  - Hw DMA/transport still offloads most of work
  - Recvr is active, so can implement RPC-like ops
Key Question

• Use RDMA reads to offload the target?
  • Or implement richer functionality at target and reduce number of ops that it needs to handle?
  • Different systems have taken different approaches

• Considerations beyond CPU scaling?
RDMA Reads vs RPC

Figure 1: Per-machine RDMA and connected RPC read performance.

(a) 1 NIC (network-bound)

(b) 2 NICs (CPU-bound)

Using unreliable datagrams for RPC has some performance benefits over using connected transport on current hardware \[7\]. The main advantage is that UD does not use connections and is, therefore, not impacted by connection caching issues and the overheads of connection sharing described in Section 2. In addition, UD RPC can exploit application-level semantics to send fewer messages than RC RPC. For instance, an RPC response can be treated as the ack to the request meaning that a separate ack message does not need to be sent. RC RPC cannot do this as these application-level semantics are not available to the NIC which is responsible for sending acks. UD also allows more batching when issuing sends and receives to the NIC than RC. UD can send any group of requests as a batch whereas RC can batch only requests on the same connection, which amortizes costs of PCI transactions more efficiently.

The disadvantage of using UD is that, unlike RC, it does not implement congestion control, reliable delivery, and packet fragmentation and re-assembly in hardware, so all of this has to be done in software, which introduces CPU overheads. Of these, congestion control is the most challenging to support. Several promising approaches for congestion control in RoCE exist \[15, 24\], but they do not work with UD. Without congestion control, UD cannot be used in data centers. Packet re-transmissions, fragmentation and re-assembly can be implemented in software, but this would introduce overheads e.g. because they might require additional message copies.

Work on FaSST \[7\] has shown that UD RPC that does not implement any of the above features can even outperform RDMA reads in some specific scenarios. To achieve maximum performance, FaSST RPC exposes a very basic abstraction— unreliable send/receive of messages that can fit in a network packet (2 kB for RoCE). If messages are lost, FaSST crashes the machine that observes the loss. FaSST also does not implement any congestion control. This design was motivated by the setup the authors used—a single-switch cluster with an expensive Infiniband network and simple benchmarks that only send small messages. This is very different from large RoCE deployments. For instance, the authors have observed almost no message loss in their experiments.
Tight client-coupling challenges

• Key issue: lack of indirection, lack of single central canonical reference

• Look up by key?
• Variable length objects?
• Compacting live space?
  • How does the system determine if an object has a reference somewhere?
• Fine-grained data migration and rebalancing?
• Eviction of data to cold tier?
5.1 Overview

RAMCloud was the first storage system to keep all data in main memory. It exposes a key-value store with distributed transactions to the users. RAMCloud is optimized for storing objects of several hundred bytes. It keeps only the primary copy of data in main memory. Backups are written to secondary storage. It relies on a small amount of non-volatile memory on backups to batch writes to disk to make logging efficient. RAMCloud achieves latency of 4.7 microseconds for reads and 13.4 microseconds for small object updates \[10\]. Backups of data are spread across the cluster to make failure recovery fast. RAMCloud recovers 35 GB of data from a failed server in 1.6 seconds \[15\].

5.2 Memory management

RAMCloud organizes its memory as a log both in memory and on disk (Figure 2a). Log is split into 8 MB segments. RAMCloud stores each object's address only on the primary, in a hashtable that maps the identifier of the table the object belongs to and its key into object's address. When an object is inserted or updated, a new value of the object is appended to the log and its address is updated in the hashtable.

RAMCloud de-fragments the logs to achieve high memory utilization. It uses two-level de-fragmentation to ensure both high memory space utilization and low disk bandwidth utilization. In-memory de-fragmentation is done at the primary, without communicating with backups or accessing disk. The primary picks a segment and copies live objects from it into a number of 64 kB seglets. This saves memory while keeping the contents of the logical segment unchanged and thus does not require communication with backups. Using small fixed-size seglets instead of allocating a block of memory that can fit all the objects keeps fragmentation low. Full de-fragmentation is done at the backups to ensure high memory utilization and low disk bandwidth utilization.

(a) RAMCloud

(b) FaRM
Questions

- RDMA-based B-tree?
- RDMA-based MVCC?
- Are real in-memory data center systems CPU bound?