Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks

Low Latency Byzantine Agreement Using RDMA

Markus Becker, 30. August 2019

Institute of Operating Systems and Computer Networks

Byzantine Agreement

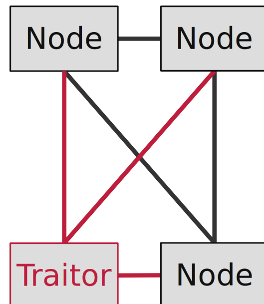
- Goal: reach consensus
- Application: distributed systems

Byzantine Failures

- Malicious party exhibits arbitrary behaviour

Fault Tolerant Systems

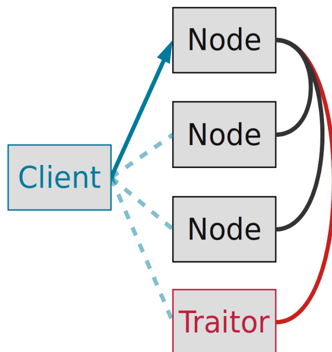
- Resilient against Byzantine Failures
- Operates correctly despite faults



Byzantine Fault Tolerant (BFT) Protocols

Example protocol: PBFT [M. Castro and B. Liskov, 1999]

1. Replicated nodes act as servers
2. Client sends request
3. Communication rounds between replicas take place
   - Fast paced data transfer occurs
4. Client receives responses
5. Client performs voting on results
   - Requires $n \geq 3f + 1$ for f faults



Reptor [J. Behl, T.Distler, R. Kapitza, 2015]

- BFT Framework
- Written in Java
- Implements multiple BFT protocols
- Highly customizable layer structure

Features

- Highly optimized and parallelized
- Based on asynchronous operations
- Very performant but limited by the network

Possible bottlenecks

Latency	$\Rightarrow$	Already optimized for TCP
Bandwidth	$\Rightarrow$	Add network cards and connections
CPU Busy	$\Rightarrow$	Cause fewer interrupts

Reptor [J. Behl, T.Distler, R. Kapitza, 2015]

- BFT Framework
- Written in Java
- Implements multiple BFT protocols
- Highly customizable layer structure

Features

- Highly optimized and parallelized
- Based on asynchronous operations
- Very performant but limited by the network

Possible bottlenecks

Latency	$\Rightarrow$	Already optimized for TCP
Bandwidth	↯	Add network cards and connections
CPU Busy	↯	Cause fewer interrupts

Remote Direct Memory Access (RDMA)

Using RDMA[1] offers:

Low latency

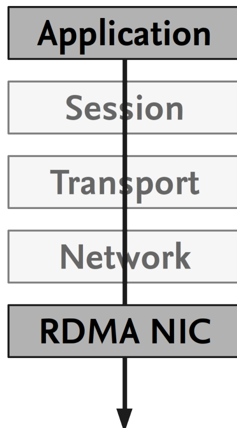
- Around 1 µs latency
- Grows with message size

High bandwidth

- Up to 56 Gb/s per port

CPU bypass

- Accessing remote memory
- Skip OS to network buffer copy
- Generates notifications



[1] Mellanox Whitepaper (2015)

RUBIN [S. Rüsch, I. Messadi, R. Kapitza, 2018]

- Network layer for Reptor
- Generalized to work for Java applications with minimal redesign

Java NIO Selector

- Multiplexes channels
- Supports blocking and non-blocking access
- Aware of channel state and capabilities
- Used by Reptor

Problem

- Buffer copies can be avoided
  - RUBIN performs unnecessary buffer copies
- RDMA requires flow control
  - RUBIN is unstable

Thesis Goals

Goals

1. Implement copy free memory management
   - Additional data safeties
   - Performant management structure
2. Resolve flow control issue
   - Introduce scheme which does not exhaust available buffers

Design

Memory management

- Copy free send and receive
  - Requires new memory structure
  - Performance critical
  - Integration with RDMA

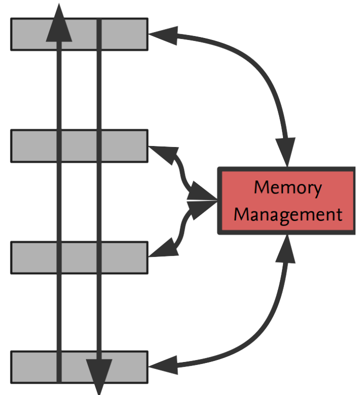
Flow control

- RDMA requires redesign
  - New flow control scheme
  - Preparing buffers in time
  - Never exhaust all buffers

Memory Management

Requirements

- Integrate into Reptor's existing design
  - Global memory management
  - Replacing lambda object
  - Used to allocate new memory on demand
- Provide safeties against overwriting
- Allow use with RDMA
  - Memory location and size is fixed



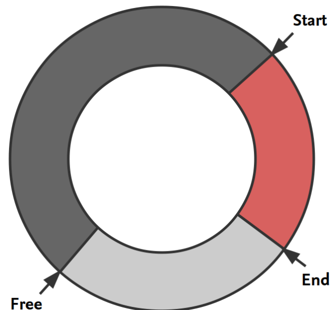
Memory Management: Ring Buffer

Pro

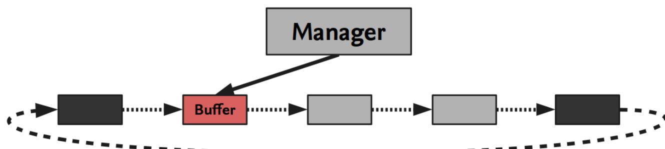
- Resizing buffers
- Better utilization of allocated memory

Con

- Tracking usage is complex
- Overhead when using with RDMA
  - Data structure for specific part of buffer
- Pointer usage in Java unidiomatic



Memory management: Buffer Ring



Pro

- Simple and robust
- Easy integration in Reptor
- Allows tracking of usage
- Map to send and receive with RDMA

Con

- Sets maximum message size

Reason for Implementing Flow Control

Back Pressure

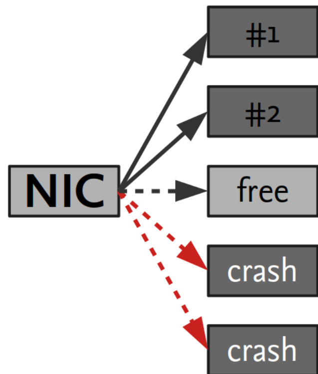
- Receiving more messages than can be dealt with
- Dealing with excess incoming messages:
  - Dropping messages
  - Buffering
  - Preparing buffers faster
  - Sharing information and waiting

Flow Control Problem

- Not consuming messages in time
  1. Data may get overwritten
  2. RDMA buffers may be exhausted

Using two-sided operations like RUBIN:

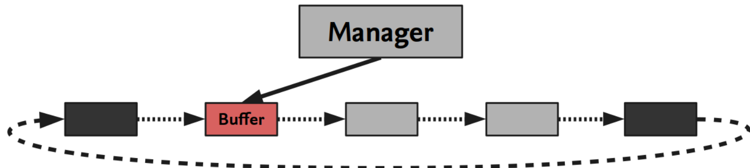
- Exhausting buffers causes crash
- No checks in place if data consumed



Receive Window

- Similar to sliding window
- Communicate local capabilities
- Send update to remote on change
- Only send if remote is capable to receive

  ⇒ Back pressure can be completely avoided
- Buffer ring can be utilized



Flow Control

Additional requirements

- Minimize overhead
  - No additional communication rounds
  - Same communication channel
  - Piggyback on normal messages
- Avoid starvation
  - Allow *emergency* flow control messages
- Application agnostic

Not needed by using RDMA

- Sequencing
- Negative acknowledgements
  - Error correction
  - Replaying messages

Implementation

- Based on `rdma-reptor` project
- Implementation in Java 8 under Ubuntu 14.04
- Using Java RDMA library *DiSNI*
  - RUBIN was built on top of DiSNI 1.4
  - Since then DiSNI 2.0 and 2.1 have been released

Upgrading DiSNI

- Large changes from 1.4 to 2.0
- RUBIN patched directly on top of DiSNI

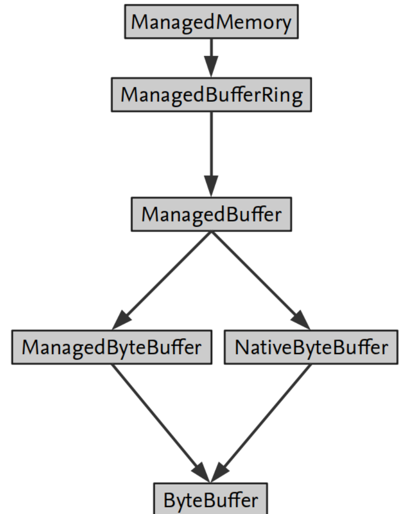
Attempts

1. Patching DiSNI directly again
2. Creating a wrapper for DiSNI 2.0

Implementation continued with RUBIN using DiSNI 1.4.

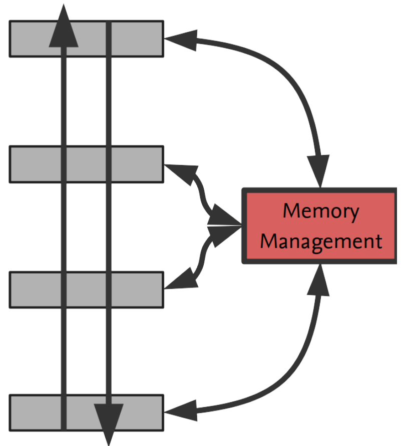
Memory Management Stages

1. **Management Classes**
   - ManagedRingBuffer
     + Interfaces and abstract classes
   - ManagedBuffer
     – Access safety
     – RDMA integration
   - NativeByteBuffer
     – Java's ByteBuffer *clone*



Memory Management Stages

1. Management Classes
2. **Integration in Reptor**
   - Replacing lambda object
   - Adapt Reptor classes
   - Provide compatibility
   - Change memory usage behaviours



Memory Management Stages

1. Management Classes
2. Integration in Reptor
3. **Tests and Benchmarks**

- Driving motivation
  - Microbenchmark everything
  - Tests for all new classes
  - Fine-tune for performance

Flow Control Specifics

- Limit changes to network layer[1]
  - In-line with RUBIN's original goals
  - Most changes in RDMA channel handler (1053 cloc[1])
  - Small changes to RUBIN (35 cloc[1])
  - A handful of lines in Reptor (15 cloc[1])
- Main obstacle
  - Rejection of messages at network layer not in Reptor's design

[1]Only including flow control implementation

Error State in Flow Control

Throw exception when illegal state reached

1. Message tried to be sent by application
2. Flow control algorithm *blocking*

- Exception allows compile time checking

Exception handling

- Buffering
- Handling flow control exceptions in Reptor
- Avoiding flow control exceptions by changing RUBIN

Error State in Flow Control

Throw exception when illegal state reached

1. Message tried to be sent by application
2. Flow control algorithm *blocking*

- Exception allows compile time checking

Exception handling

↯ Buffering

↯ Handling flow control exceptions in Reptor

$\Rightarrow$ Avoiding flow control exceptions by changing RUBIN

Changes to RUBIN

- RUBIN aware of *writable* channel status
  - Channel updates status when *blocked*
  - Channel determines own status by counting messages
- Parties append capabilities to each message
  - At most 8 bytes of flow control data
  - Capabilities of local channel
- RDMA channels interpret flow control data
  - Never exhaust remote capabilities
  - Send flow control message to avoid starvation

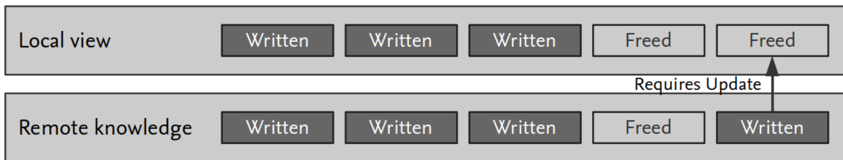
New state of RDMA Channel

Remote capabilities

- Updated when receiving a message

Remote’s knowledge of local capabilities

- Outdated by consuming and preparing buffers
- A flow control message can be sent to avoid imminent starvation
- Only send flow control message if necessary



Evaluation

Memory Management

- Microbenchmarks

Flow Control

- Distributed connection test
- Replicated system benchmark

Memory Management Benchmarks

Type	Iterations	Round [s]	Dev. [s]	GC calls	GC time [s]
Native	10^3	**0.09**	0.04	**10**	0.03
Ring	10^3	**0.08**	0.02	**12**	0.03

- Ring contains 1000 buffers
- Compare access times

Native: • Allocating ByteBuffer

Ring: • Iterating through Ring

• Allocate internal buffers lazily

Memory Management Benchmarks

Type	Iterations	Round [s]	Dev. [s]	GC calls	GC time [s]
Native	10^5	**9.62**	0.56	**1142**	3.21
Ring	10^5	**0.10**	0.02	**14**	0.05

- Iterating 100 ring lengths shows performance advantage
- Performance increases over longer runtime

Memory Management Benchmarks

Type	Iterations	Write	Round [s]	Dev. [s]	GC calls	GC time [s]
Native	10^5	No	9.62	0.56	1142	3.21
Ring	10^5	**No**	**0.10**	0.02	**14**	0.05
Native	10^5	Yes	10.6	0.72	1142	3.48
Ring	10^5	**Yes**	**0.10**	0.02	**14**	0.04

- Writing to memory to simulate use with RDMA
- Ruling out compiler optimization invalidating results

System Configuration

Machines	`beagle1`, `beagle4` @ Ubuntu 14.04
CPU	Intel®Xeon®E5-2430 v2 @ 2.5GHz
RAM	16GB DDR3
OFED	v4.2-1.2.0
RNIC	Mellanox MT27520 Family ConnectX®-3

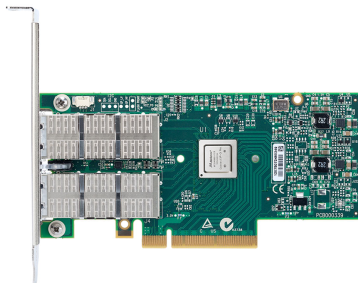


Image from mellanox.com

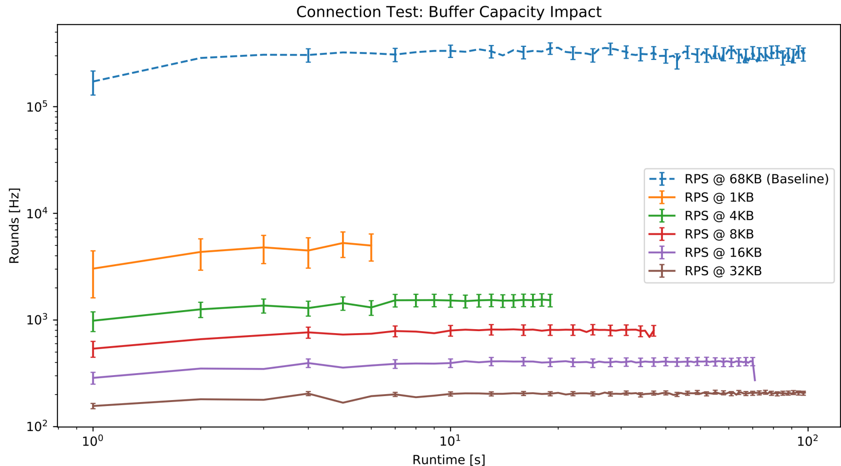
Flow Control Benchmarks

1. Distributed connection test
2. Replicated system benchmark

Flow Control Benchmarks

1. **Distributed connection test**
   - Ping-Pong application
   - Using Reptor's communication stack
   - Uses new flow control and memory management
   - Sending and checking counter value
   - Benchmarking flow control at different buffer sizes
2. Replicated system benchmark

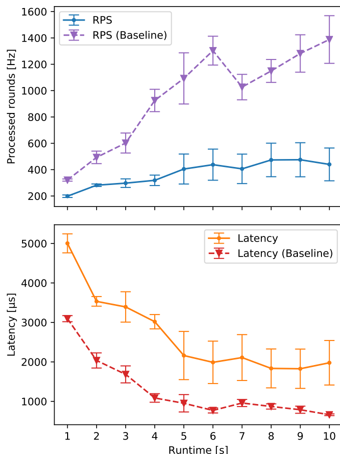
Flow Control Benchmarks



Flow Control Benchmarks

1. Distributed connection test
2. **Replicated system benchmark**
   - Using PBFT with client sending empty requests
   - Not distributed during this thesis
     - Problems with configuration across machines
     - Used 4 local replicas with 1 client
   - Using new memory management
   - Using new flow control scheme

Flow Control Replicated Benchmark



- Negative performance impact notable
  - Cost of added stability
  - Latency only increased slightly
- Baseline crashes frequently
- Buffer copies show low overall impact
- RDMA overall still not fully utilized
  - TCP implementation still superior

Conclusion

Memory Management

- Constant memory footprint management structure
- Integration in Reptor and RUBIN
  - Enables idiomatic usage with RDMA in Java
  - Stable, with safety features
- Very performant in microbenchmarks

Flow Control

- Network layer based flow control scheme implemented
- Eliminated crashes due to *back pressure*

Evaluation

- Increased stability
- Reduced throughput caused by flow control overhead

One-sided Operations

The *back pressure* problem could be solved by using one-sided operations, if it was not for:

- The Java NIO Selector functionality would be harder to provide
- No notifications would be generated
  - Memory locations could be polled
  - A different communication channel could be used for notifications
- The network layer would have to undergo large changes
  - Instead of sending messages it would write or read memory directly