Java Communications Faster than C++

by Guillermo L. Taboada, Ph.D.

Introducing myself

+10 years R&D in Java Communications for High Performance Computing

Now CEO/Co-founder of TORUS, the high-performance comms company

Multiple solutions in key sectors:

Finance / Trading

Telco / IT (Big Data)

Energy

Defense / Space

Torus 2013: Strong Debut

Torus Software Solutions wins UKTI Spain Technology Competition

IBM SmartCamp

Looking for global entrepreneurs who want to build a Smarter Planet

Torus has been selected as finalist in the IBM SmartCamp competition

Torus Big Data Projects

Torus technology is being used at the NASA Langley Research Center, 16x speedup

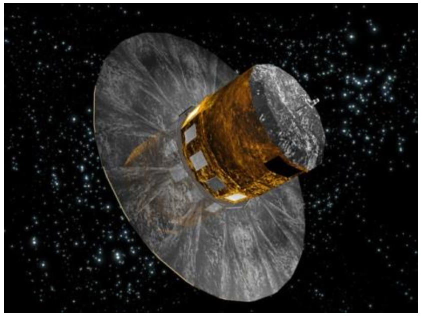
The amount of in-memory data handled surpasses 8TB, running on 8192 cores

Paper reference: http://dx.doi.org/10.1016/j.jcp.2012.02.010

Torus software is being used by the European Space Agency, 12x speedup

The developed software, MPJ-Cache, handles up to 100TB

Paper reference: http://dx.doi.org/10.1117/12.898217



The Context

High Performance Computing

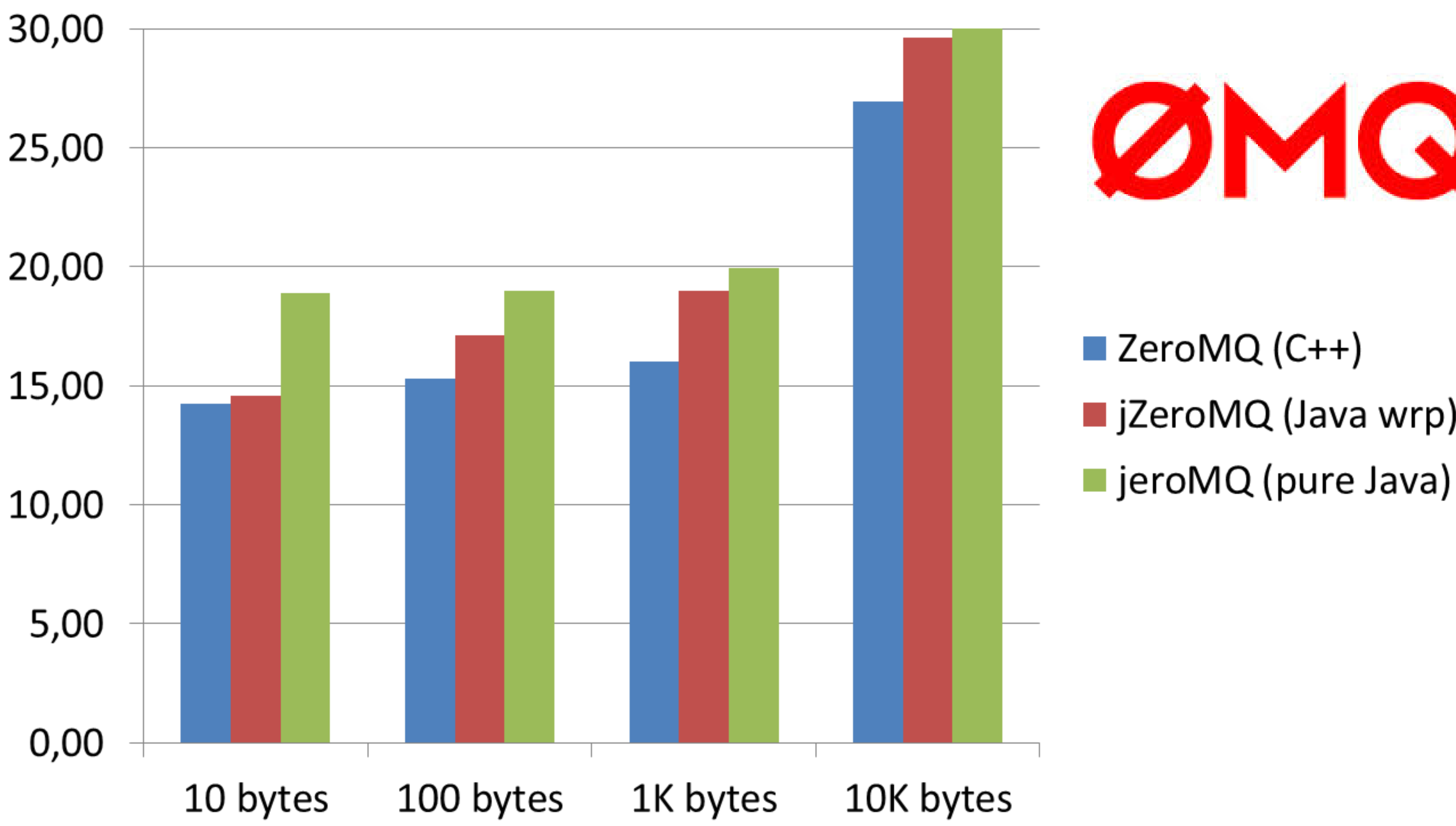
Software is not able to take advantage of high performance hardware

High Performance Communications

Bridge the gap between network capacity and applications performance

The Typical (expected?) Scenario

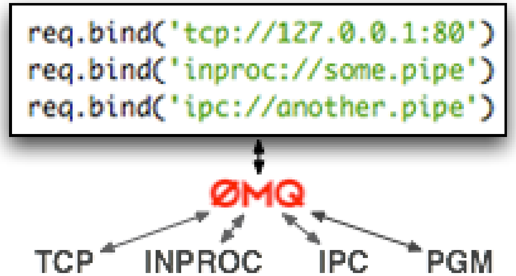
ZeroMQ Ping-Pong Latencies (in microseconds) over TCP loopback



The Typical WRONG! Reasons

- Java is slow, everybody knows this
- Java communications are even slower
- The best approach is to wrap Java on top of C++ via JNI
- Lots of JNI improves performance
- You are trading off performance for portability
- Bypassing TCP/IP breaks portability
- No one uses TCP for localhost,

  ZeroMQ has inproc/IPC support:



Some Arguably RIGHT! Reasons

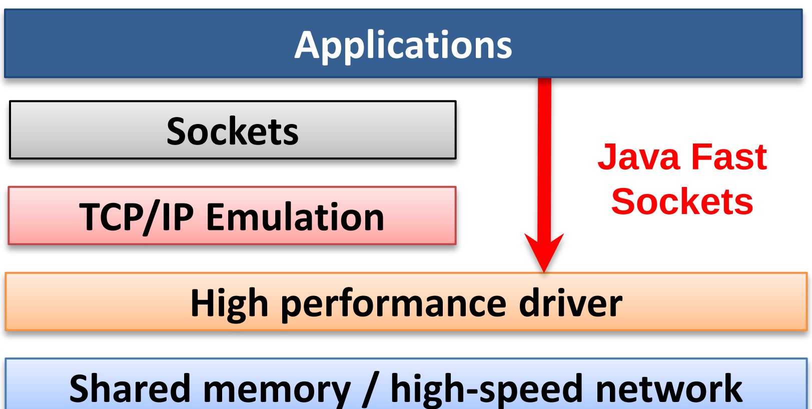
- No reason for Java being slower than natively compiled code.
  Even dynamic recompiling (JITC) makes code run faster.
- TCP/IP slows down Java communications
  (shy attempts for alternatives like SDP).
- Excessive wrapping is not the best option, JITC not possible,
  loses portability, memory conflicts, “bipolar” behaviour…

The torus® Approach

- Fully transparent TCP/IP-bypass, fully portable
- Use fast communication protocols for performance and TCP/IP for portability
- 1 JVM per server wastes resources and presents higher GC penalties, the best approach is multiple JVMs per server
- TCP loopback is quite popular, think in distributed applications over multicore servers, or multiple JVMs per server
- Low-latency networks and low-latency JVMs are key for scalability

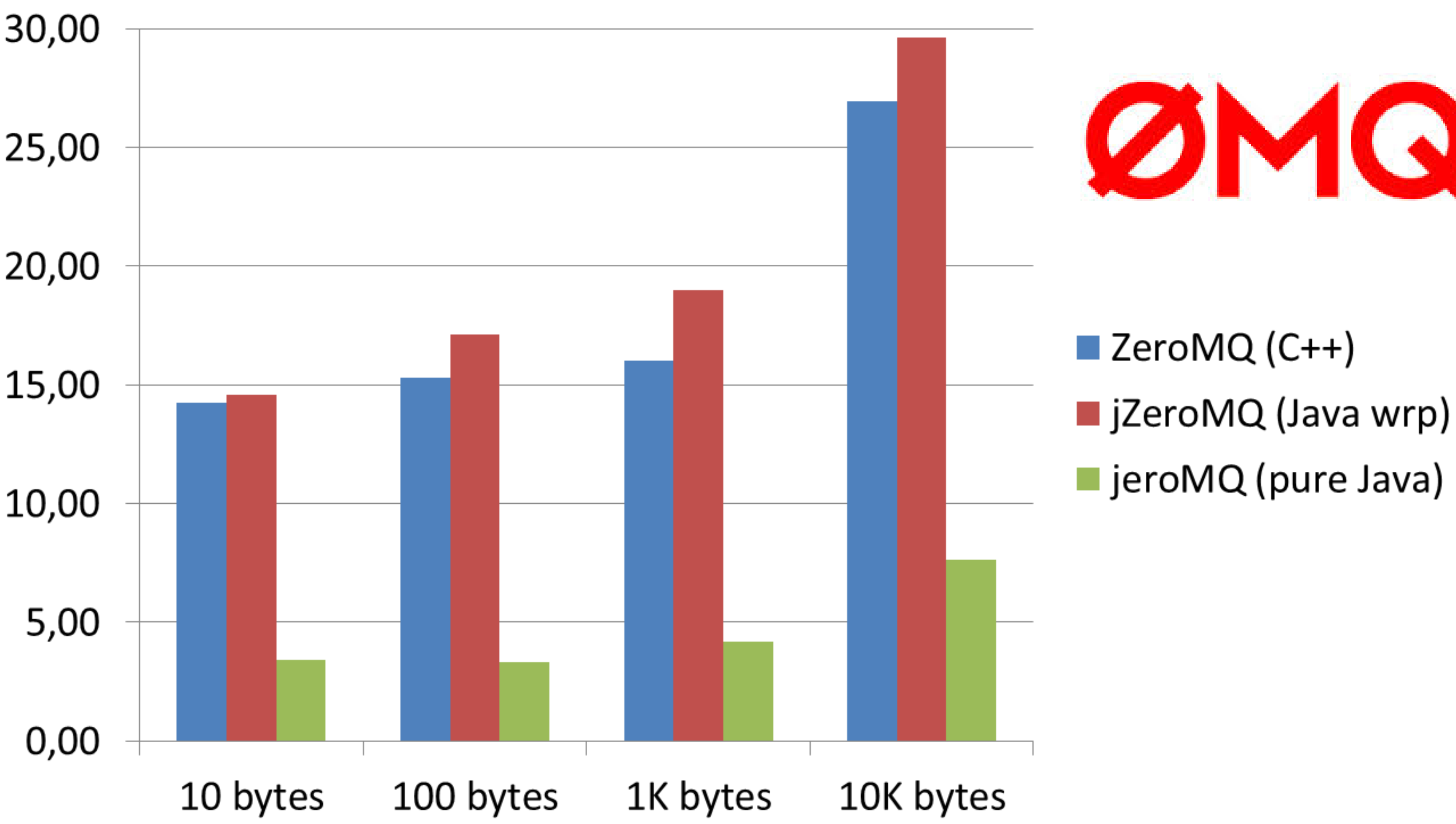
Java Fast Sockets

- JFS skips the TCP/IP processing overhead for shared memory and high-speed networks
- JFS is just plug&play, user and application transparent, without source code changes
- Further information and demo downloads at http://www.torusware.com



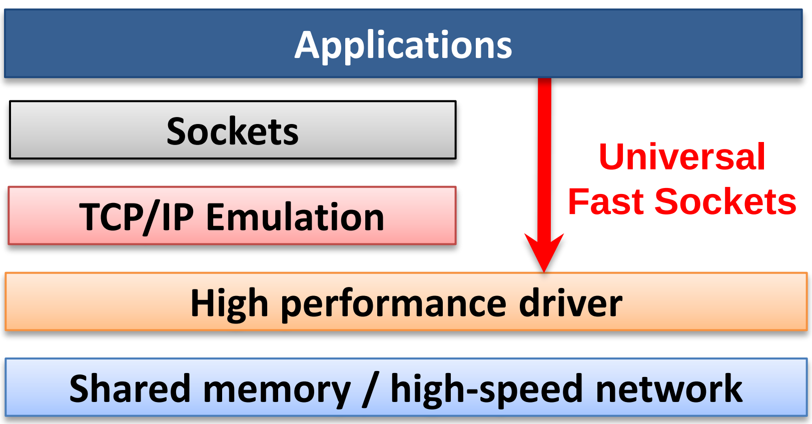
Accelerating JVM sockets (bypassing TCP/IP)

ZeroMQ Ping-Pong Latencies (in microseconds) over TCP loopback

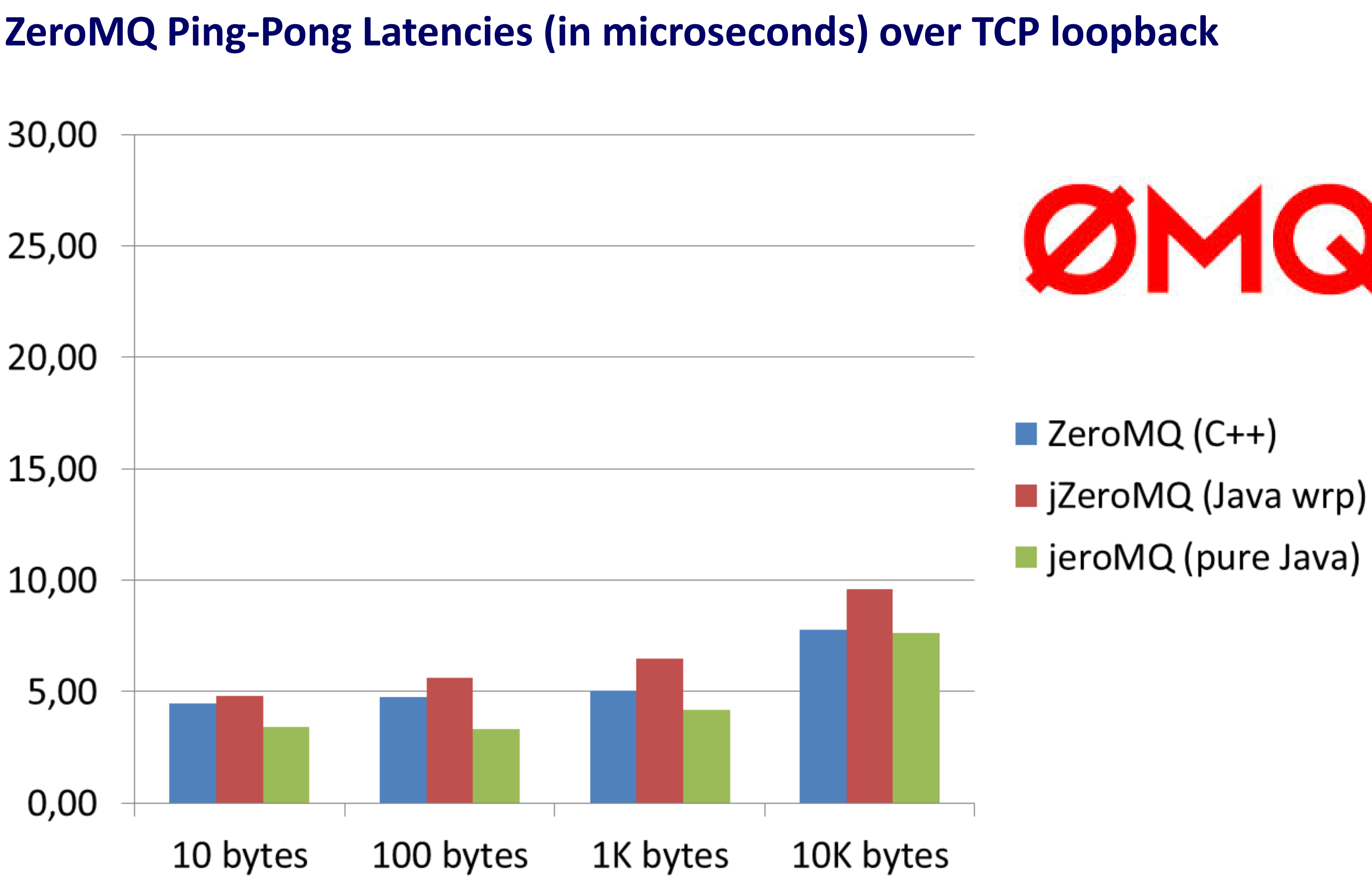


Universal Fast Sockets (UFS)

- UFS skips the TCP/IP processing overhead for shared memory and high-speed networks
- UFS is just plug&play, user and application transparent, without source code changes
- Further information and demo downloads at http://www.torusware.com

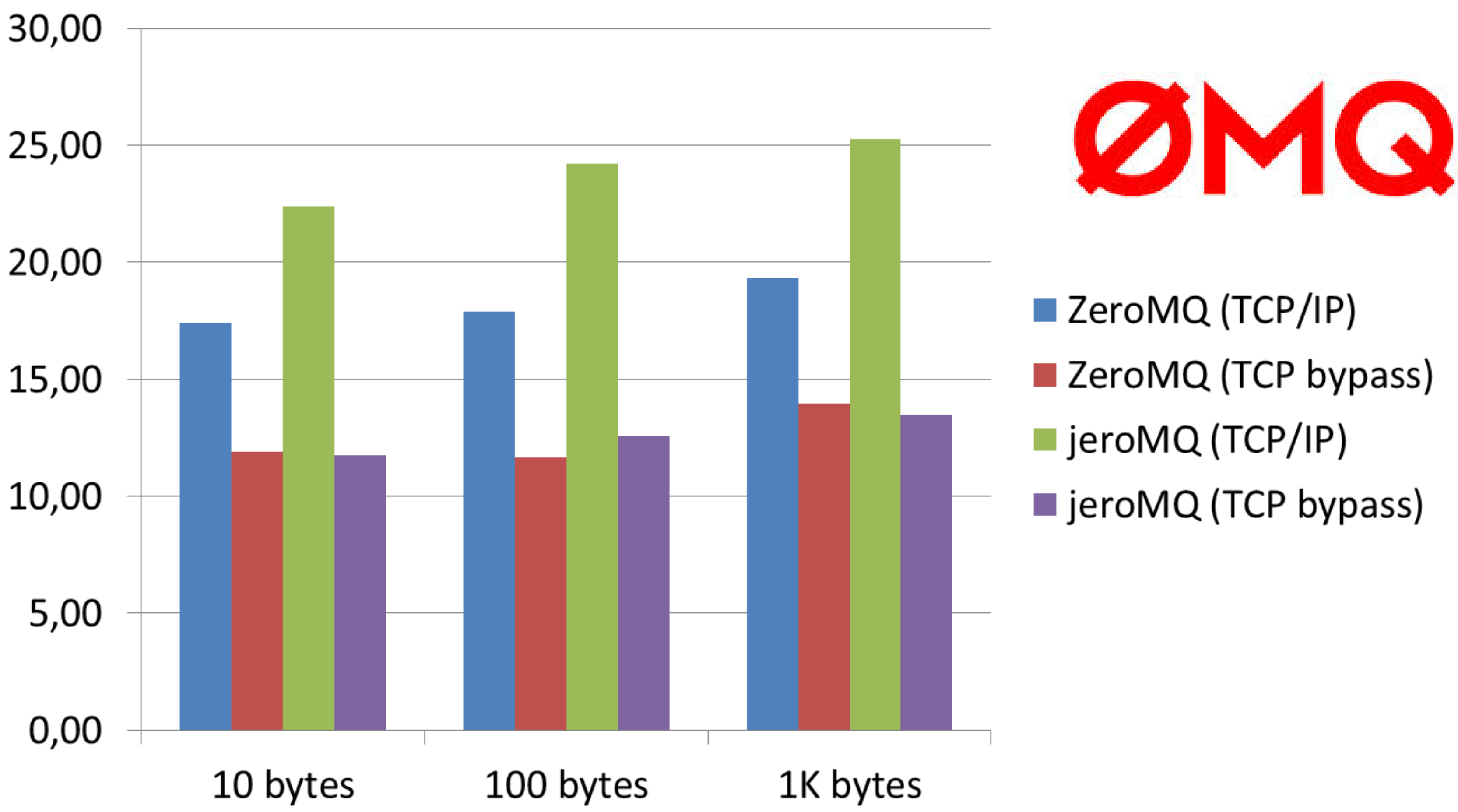


Accelerating C++ and JVM sockets (bypassing TCP/IP)

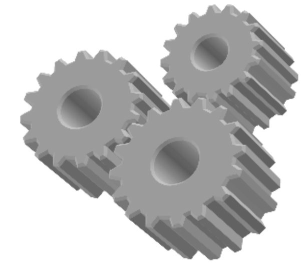


Now on a Low-latency Network

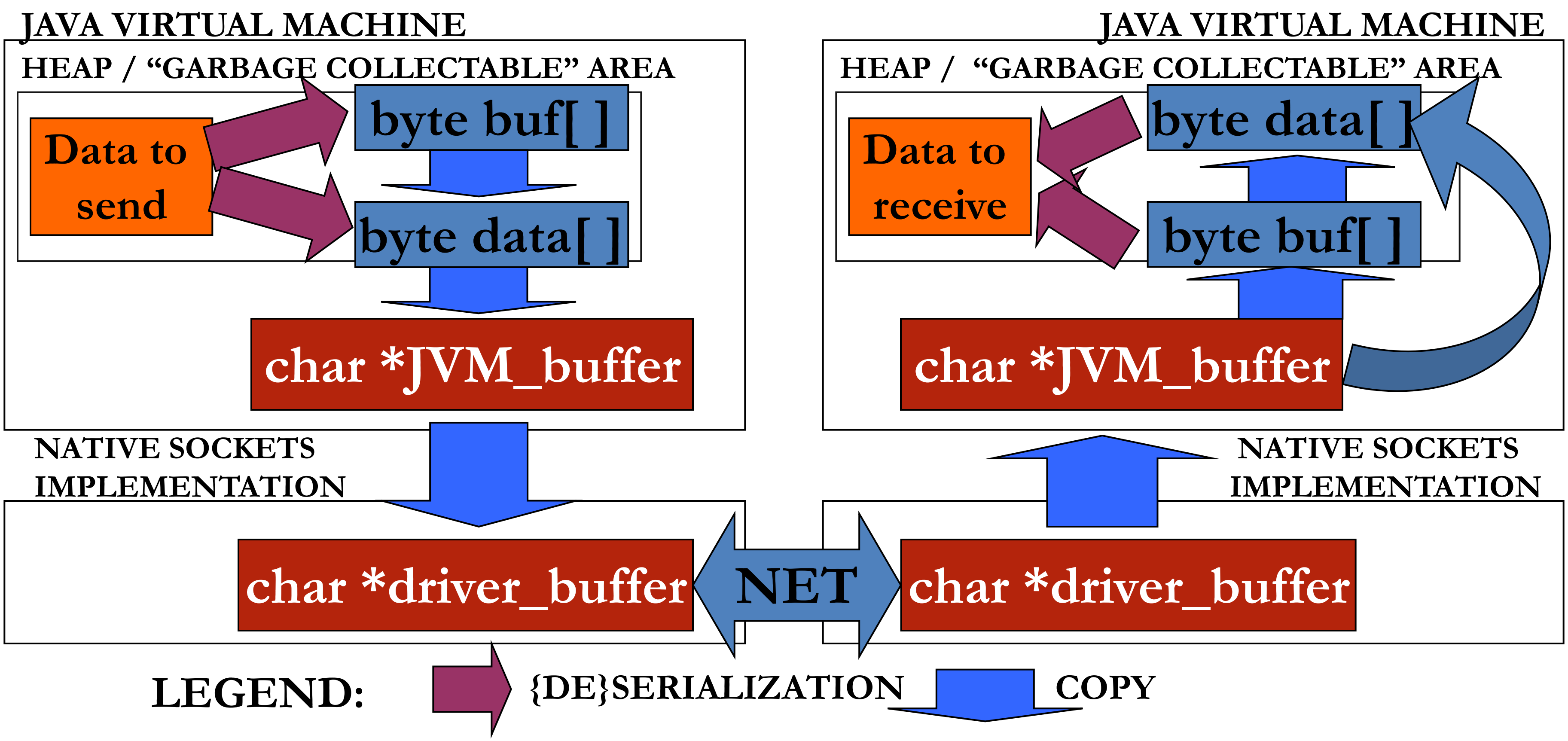
ZeroMQ Ping-Pong Latencies (in microseconds) over Mellanox cards



JFS: The Secret Recipe

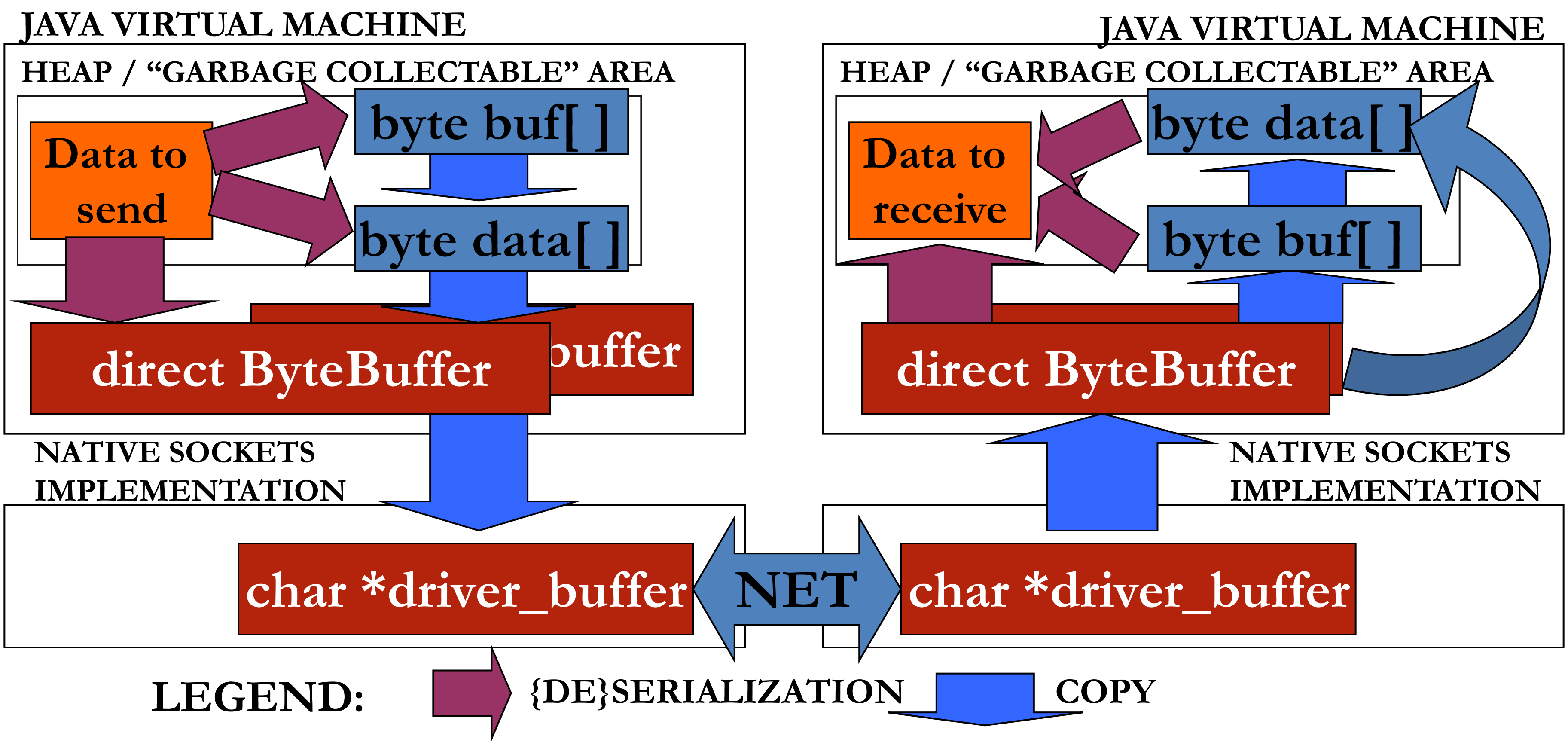


- Default scenario in JVM sockets communication



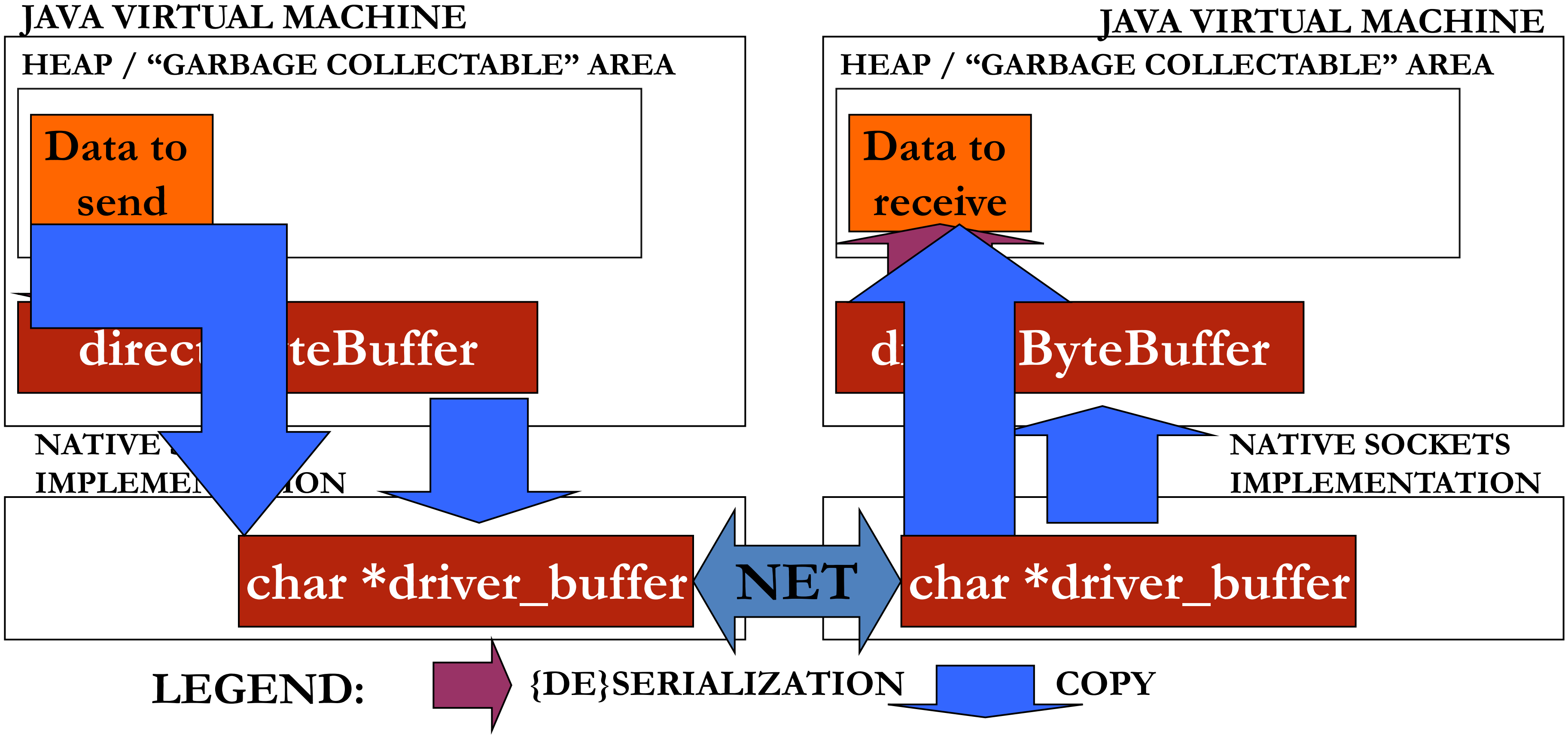
JFS: The Secret Recipe

- Attempt to improve the situation in Java NIO

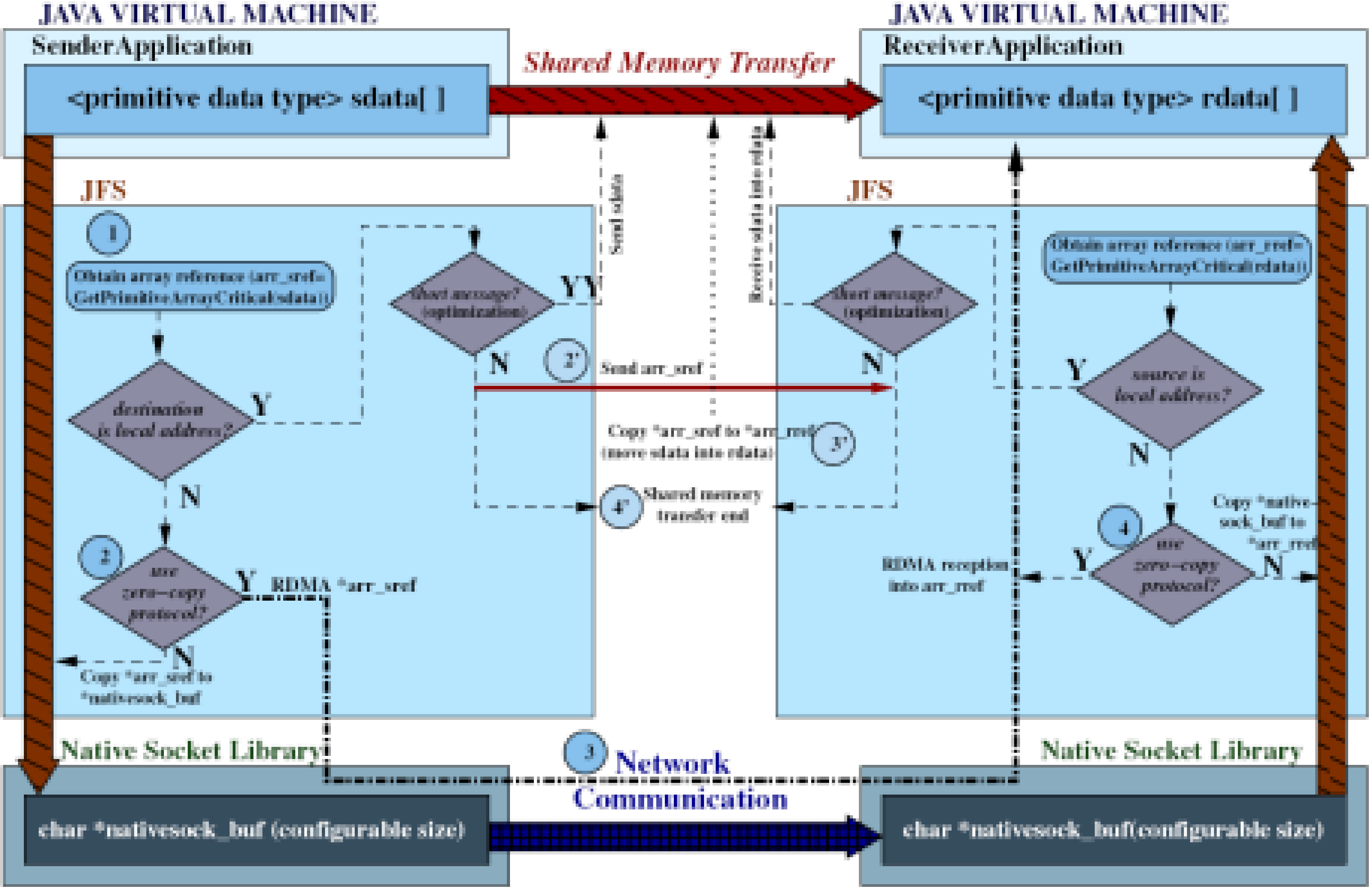


JFS: The Secret Recipe

- **JFS Zero-copy protocol**



Java Fast Sockets



Java Fast Sockets: Key points

- GetPrimitiveArrayCritical avoids buffering
- Combination of polling and waiting, depending on frequency of communication
- Optimization of NIO select (NIO calls epoll and writes a “slow” pipe for notifying waiting threads)
- Extended API for reducing serialization overhead:

```
write(byte array[ ])  /* This is the only write method supported in Java */
write(int array[])
write(long array[])
write(double array[])
write(float array[])
write(short array[])
write((direct) ByteBuffer bb, int position, int size)
write((array) Object oarray, int position, (direct) ByteBuffer, int init, int size)
```

MPI Java

Cisco Blog > High Performance Computing Networking

Resurrecting MPI and Java

January 28, 2012
at 8:21 am PST

Jeff Squyres

hadoop

Among the projects produced were several that tried to bring MPI to Java. That is, they added a set of Java bindings over existing C-based MPI implementations. However, many in the HPC crowd eschewed Java for compute- or communication-heavy applications because of performance overheads inherent to the Java language and runtime implementations.

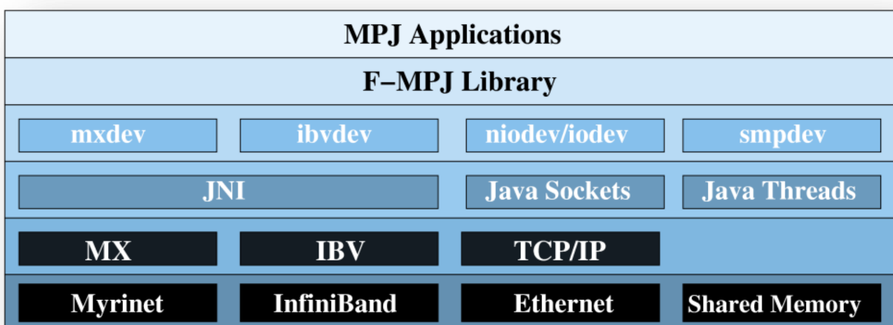
Hence, the Java+MPI=HPC efforts didn't get too much traction.

It is the Hadoop community who has presented the idea of re-introducing Java for MPI. Their idea is that the "reduce" applications are getting larger and more computationally expensive. Hence, they want to parallelize their computations by spreading across multiple processor cores.

With parallelization, they want to have efficient inter-process communication (IPC) between cores. MPI is a well-established IPC API, so why re-invent the wheel? Just add some reasonable Java MPI bindings to an underlying C MPI implementation (or even a Java class library with some nice Java-ish abstractions).

FastMPJ

- FastMPJ is the fastest Java message-passing library
- FastMPJ supports efficiently shared memory and high-speed networks (RDMA IB)
- Scales performance up to thousands of cores and outperforms Hadoop for Big Data
- FastMPJ is fully portable, as Java
- Further information and demo downloads at http://www.torusware.com



Testbed

Configuration:

- Dell PowerEdge™ R620x8 – Sandy Bridge E5-2643 4C (3.30GHz) 32 Gb DDR3-1600MHz
- Mellanox ConnectX-3 RoCE (40 Gbps) and InfiniBand (56 Gbps) JFS, on a PCIe Gen3
- Solarflare SFN6122F, on a PCIe Gen3
- Red Hat Linux 6.2, kernel 2.6.32-220, OpenJDK 1.6
- Sockets benchmarked with ping pong NetPIPE (both Java and natively compiled tests)
- FastMPJ benchmarked with pingpong of Java version of Intel MPI Benchmarks
- Testing methodology:

  100,000 iterations warm-up & 100,000 iterations per message size

  Shared memory communication within a single processor

  No stopped Linux services, normal operational conditions

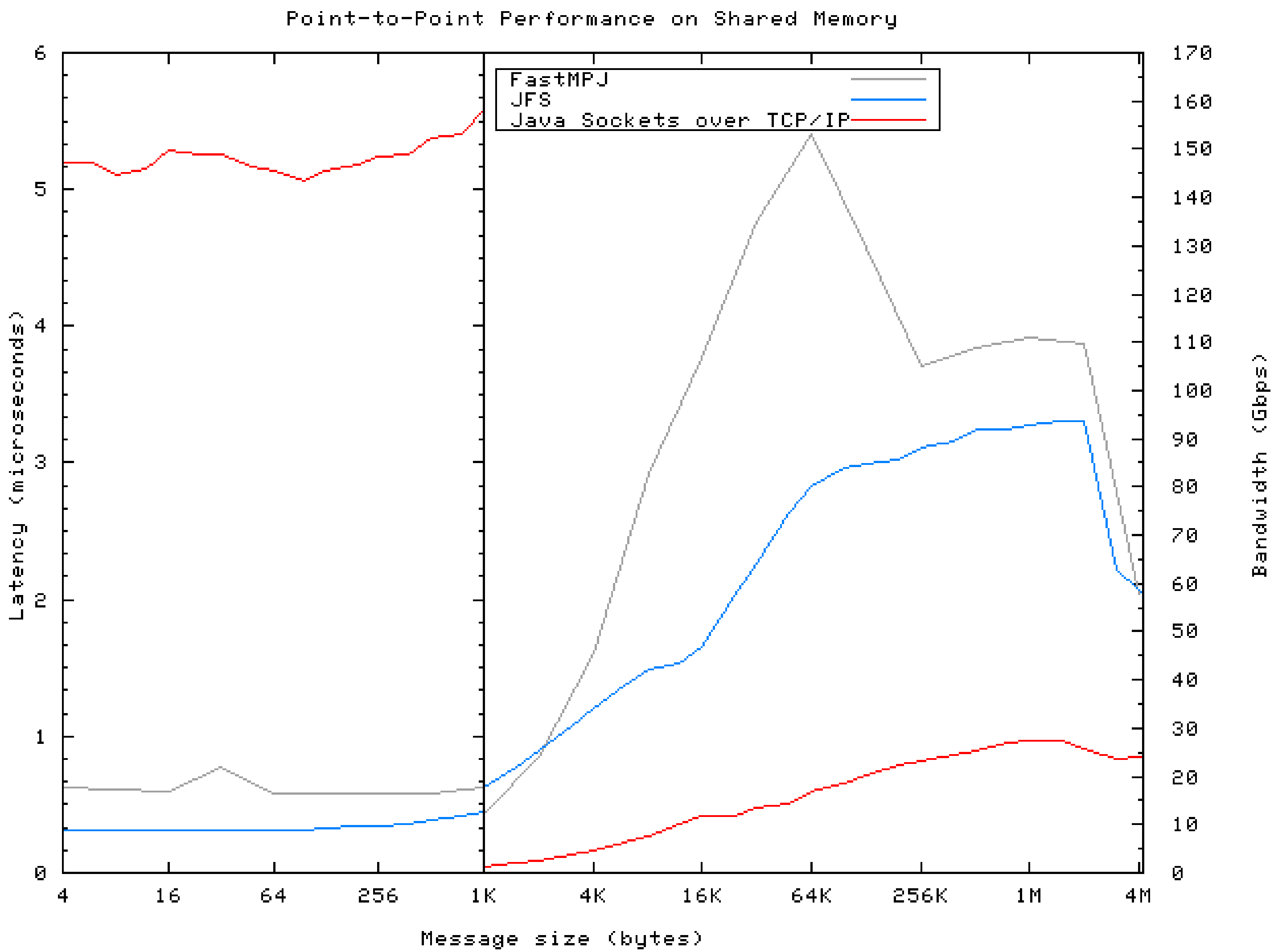
Performance Results

List of performance graphs:

1. JFS & FastMPJ performance on shared memory
2. JFS & FastMPJ vs VMA performance on InfiniBand
3. Comparison of JFS/FastMPJ vs ZeroMQ (shmem and IB)
4. Applications of JFS: optimizing JGroups
5. Applications of JFS: optimizing NIO - Netty
6. JFS & FastMPJ jitter analysis

Localhost Performance

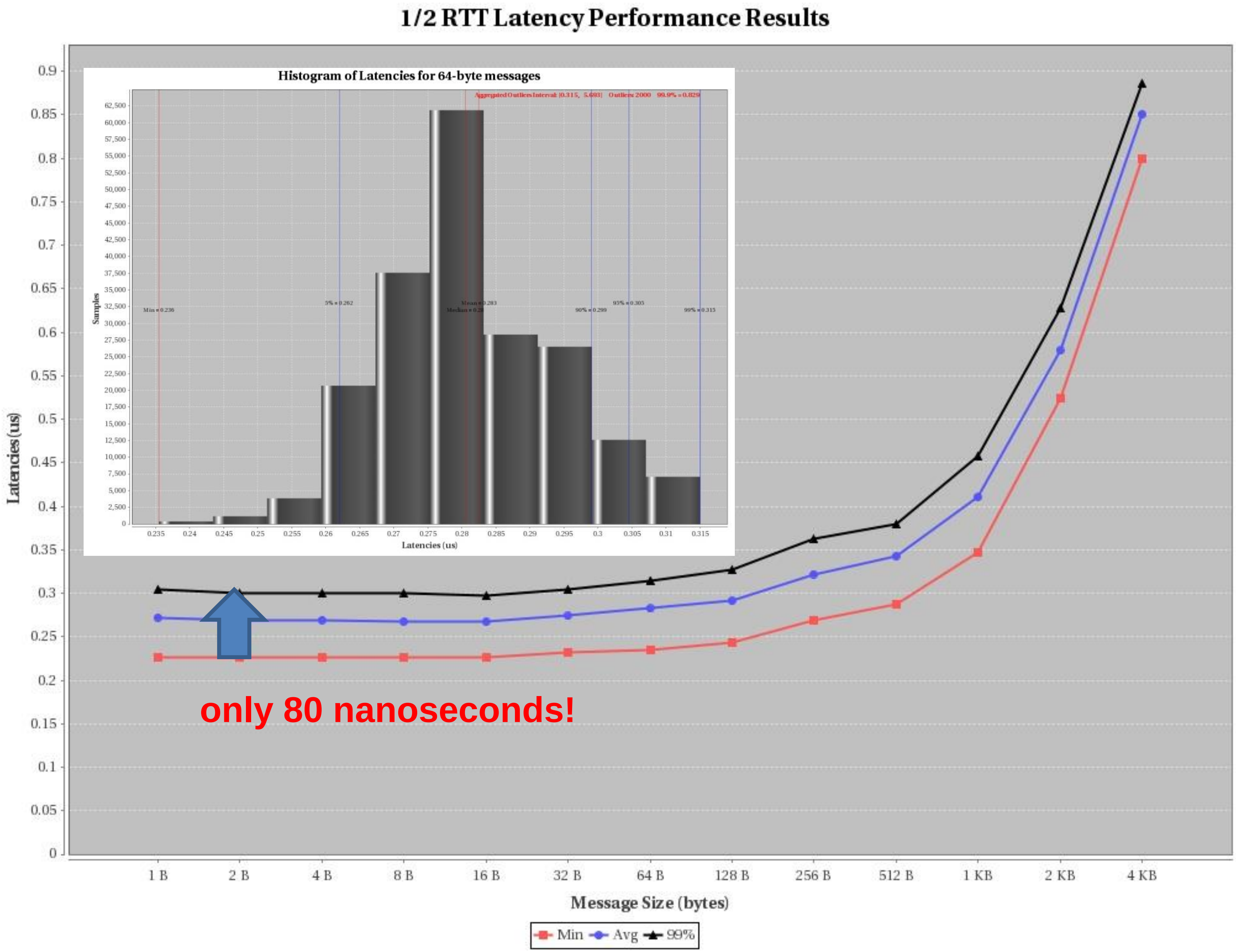
NOTE: In latency (left-hand side) the lower the better. In bandwidth (right-hand side) the higher the better



TORUS_PUBLIC_High_Performance_
Communications

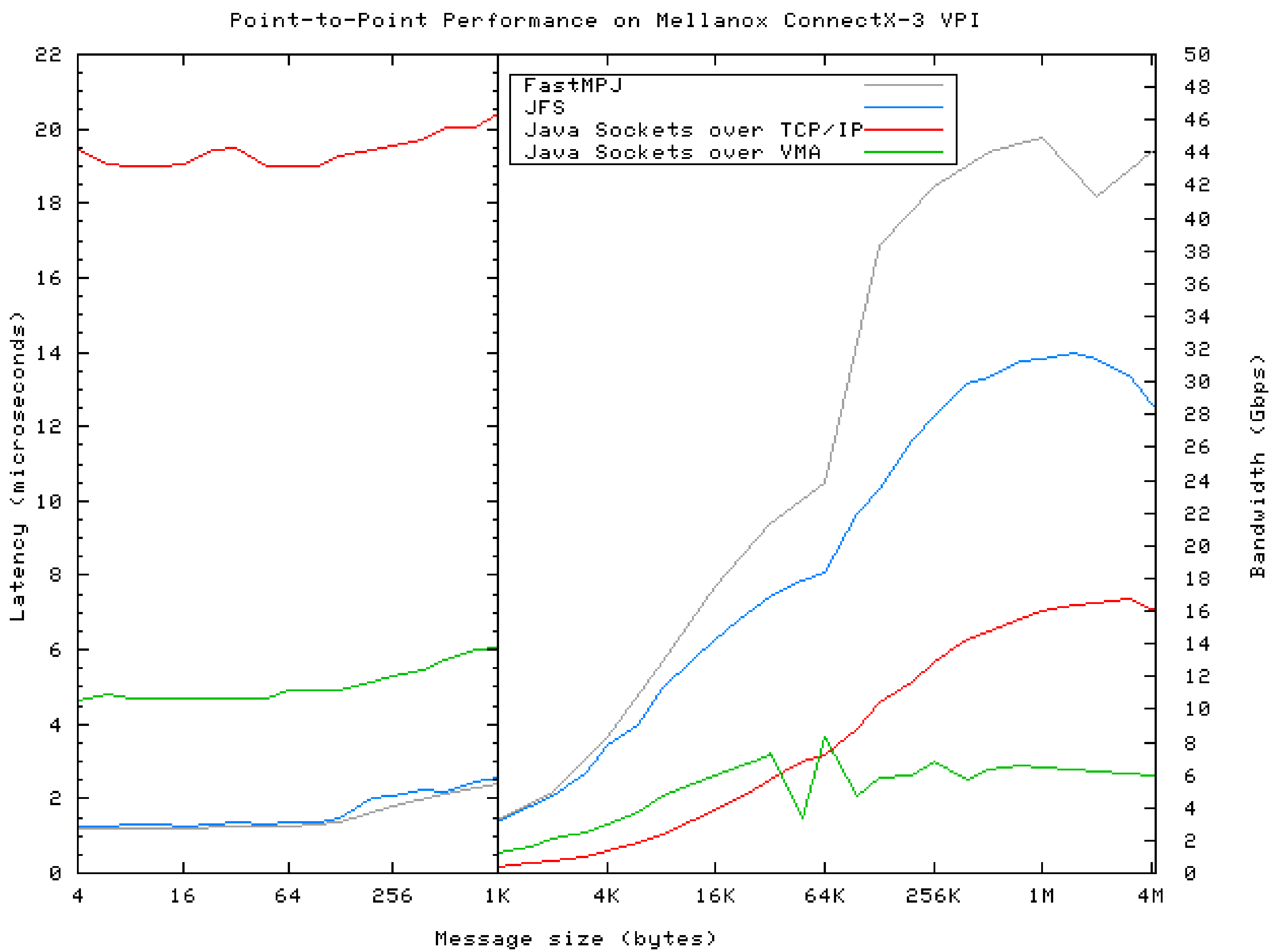
Source: Torus lab tests

Jitter (shared memory)



Network Performance

NOTE: In latency (left-hand side) the lower the better. In bandwidth (right-hand side) the higher the better

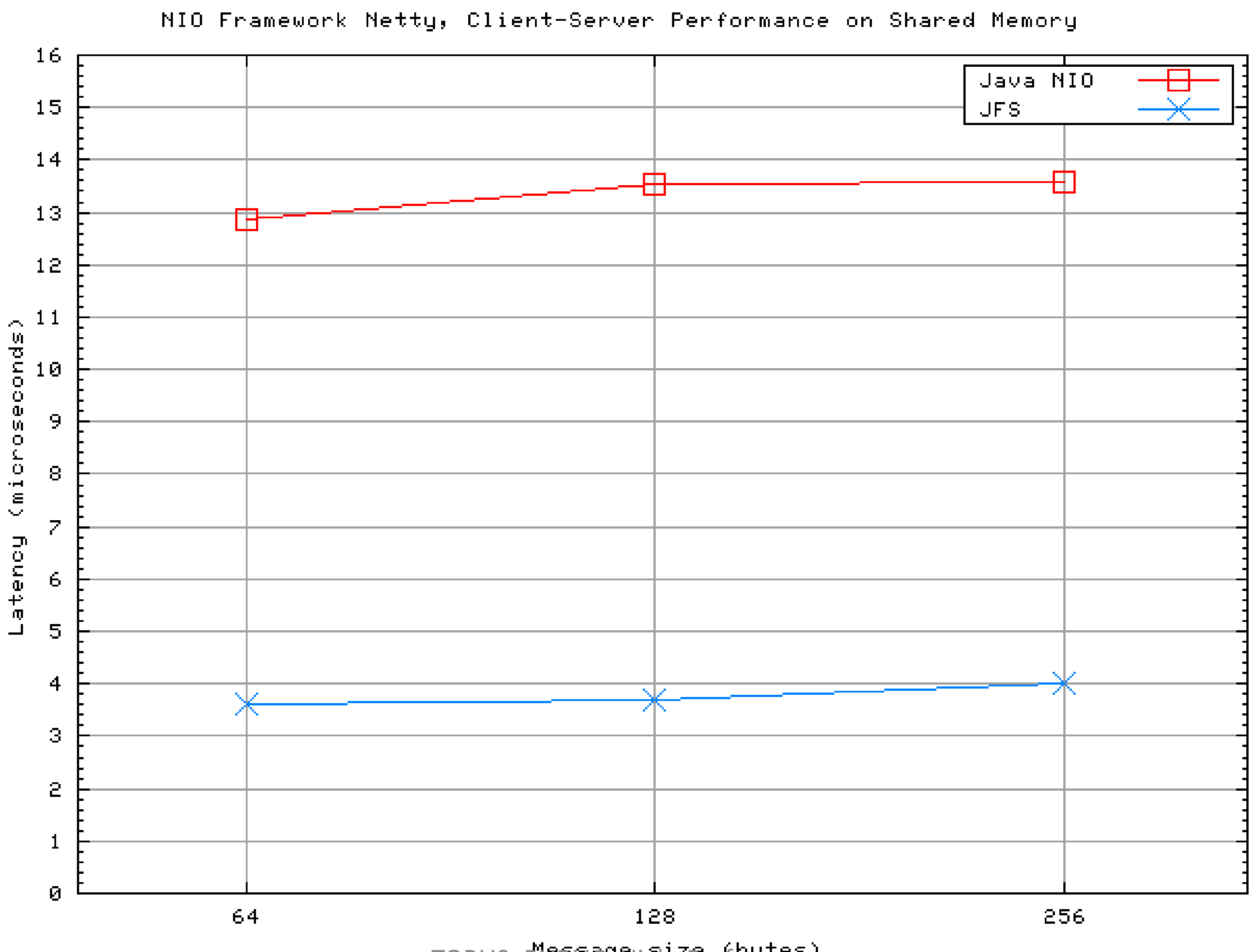


TORUS_PUBLIC_High_Performance_ Communications

Source: Torus lab tests

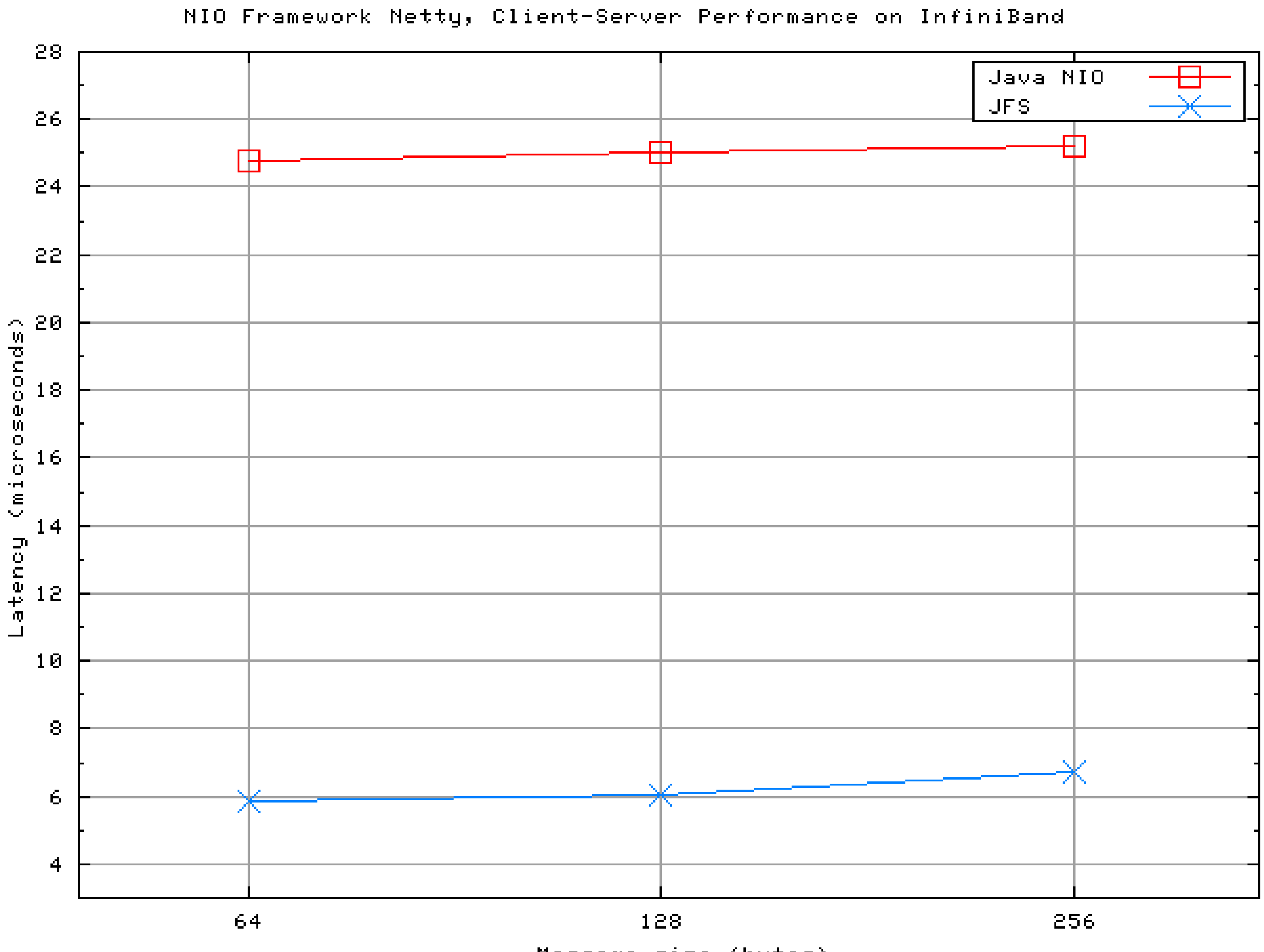
optimizing NIO - Netty (shmem)

NOTE: In latency (left-hand side) the lower the better.

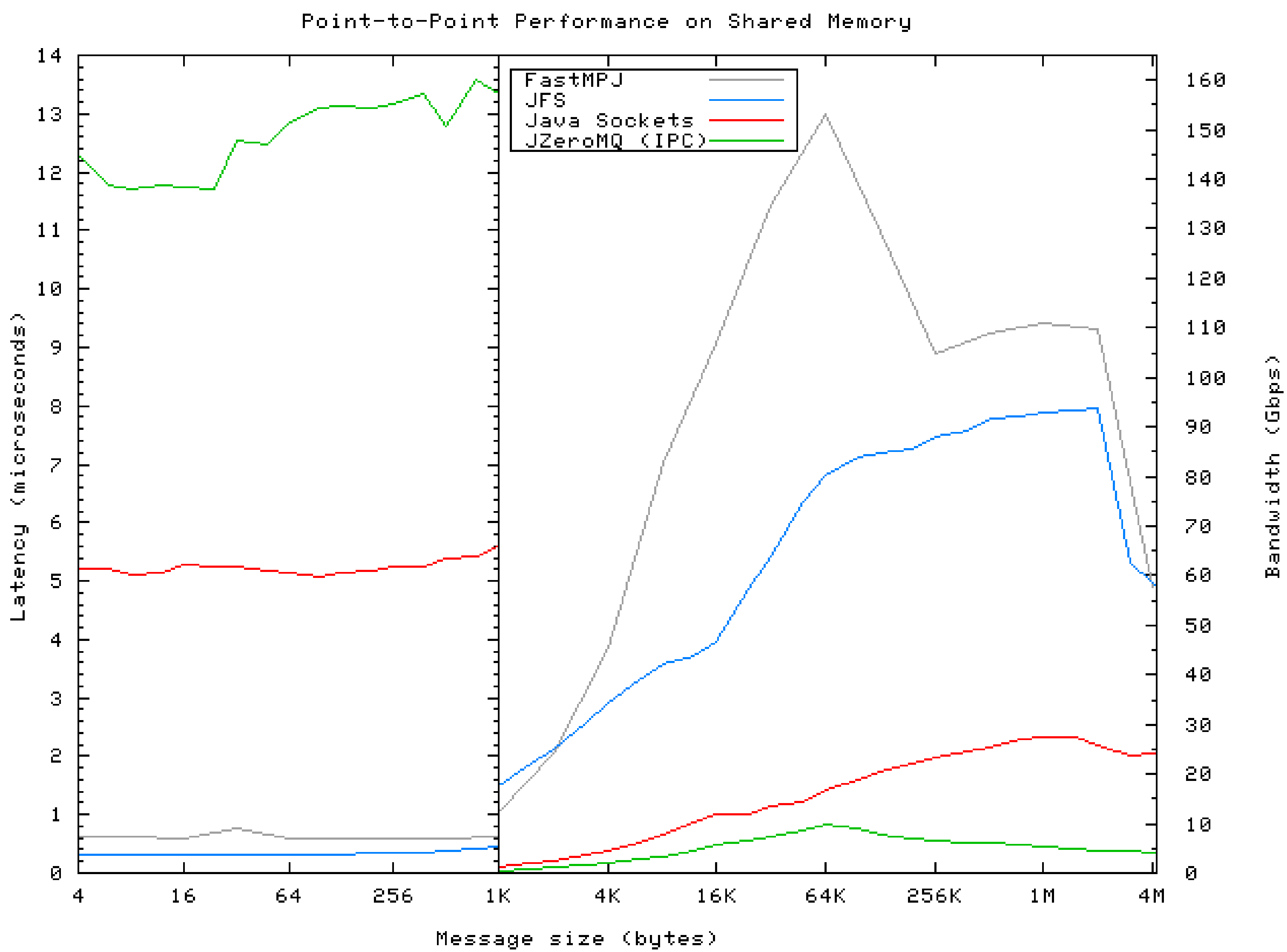


optimizing NIO - Netty (IB)

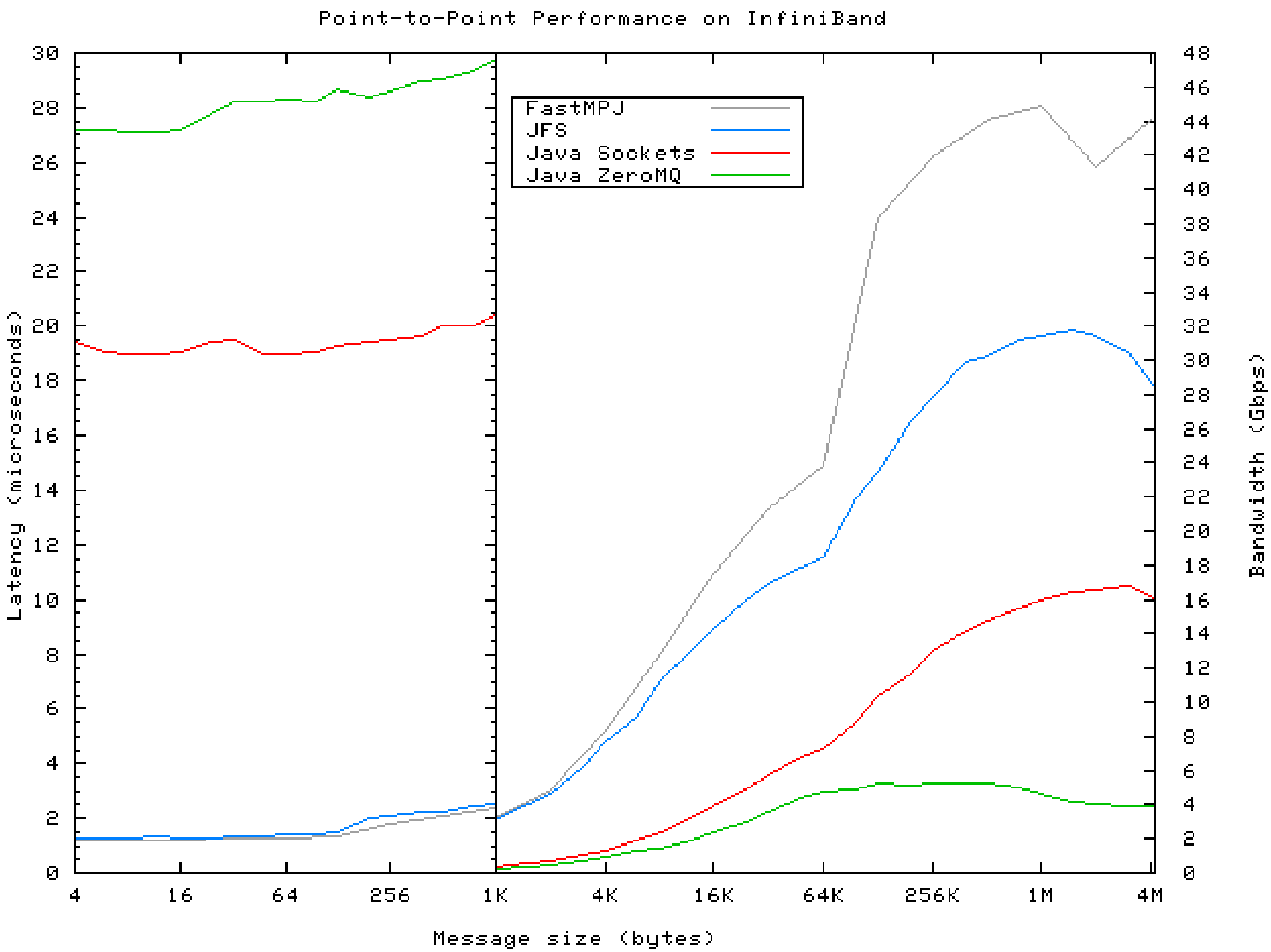
NOTE: In latency (left-hand side) the lower the better.



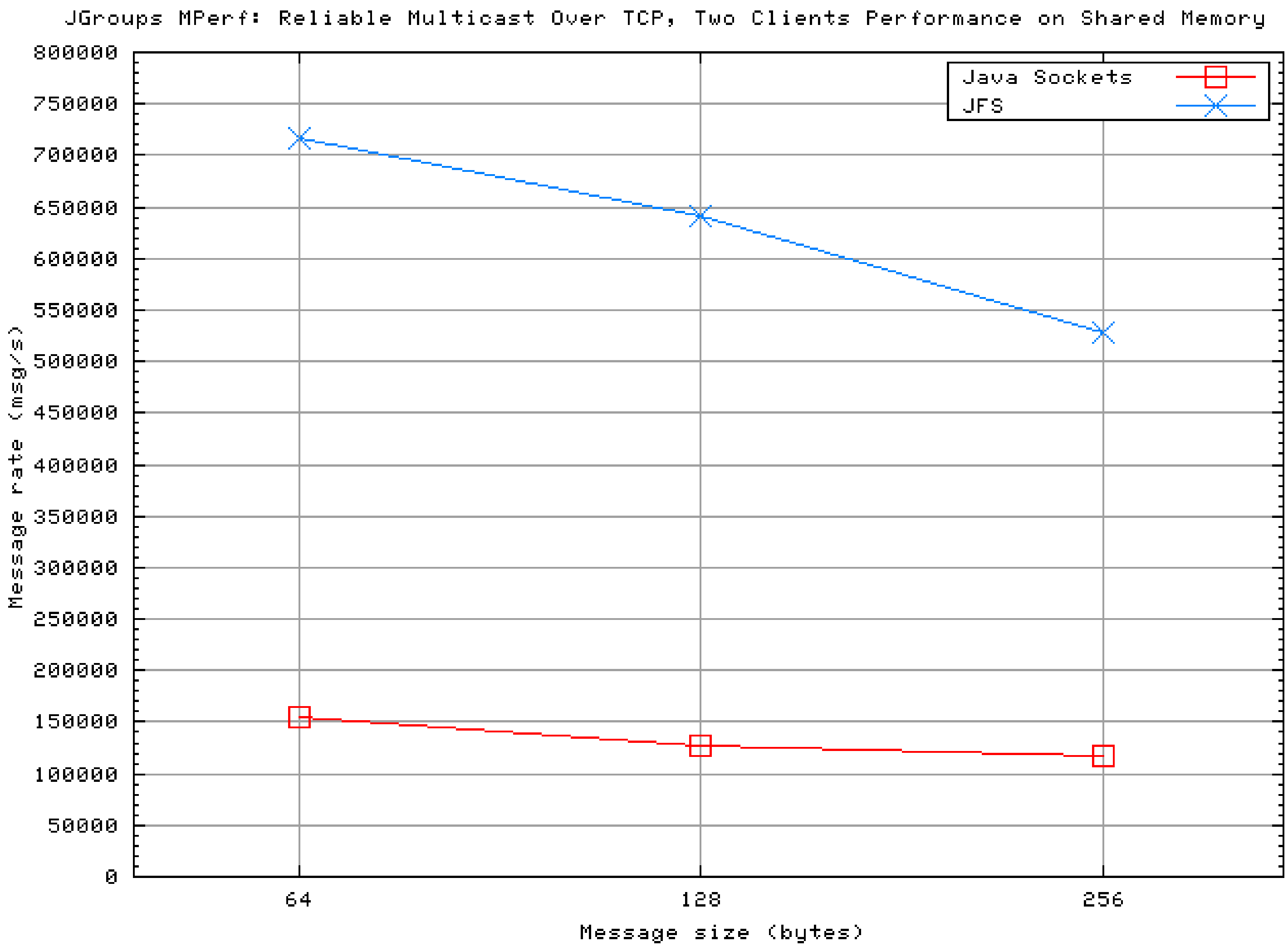
vs JZeroMQ (SHM)



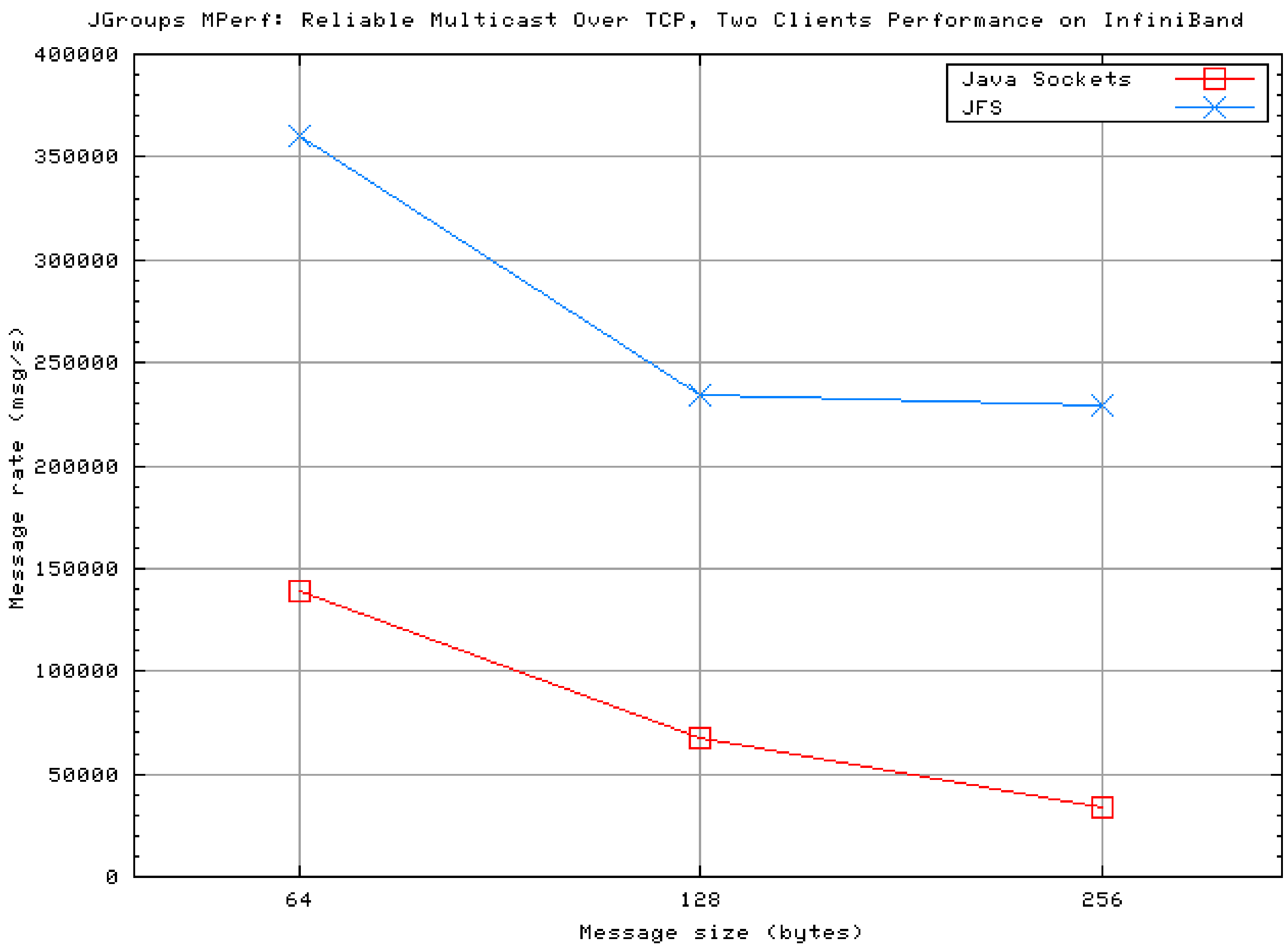
vs JZeroMQ (IB)



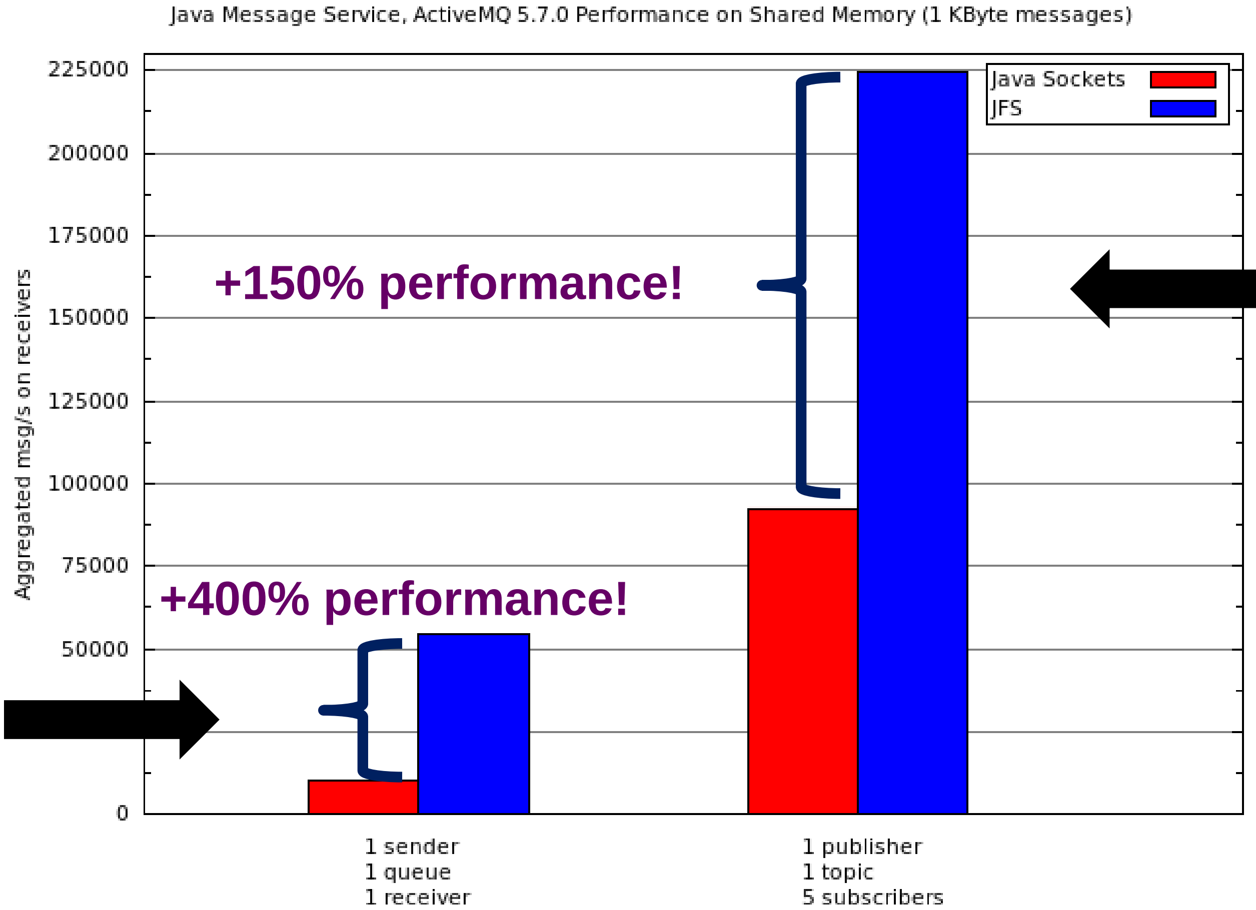
JGroups (Message rates SHM)



JGroups (Message rates IB)



optimizing JMS (ActiveMQ) on Shared Memory

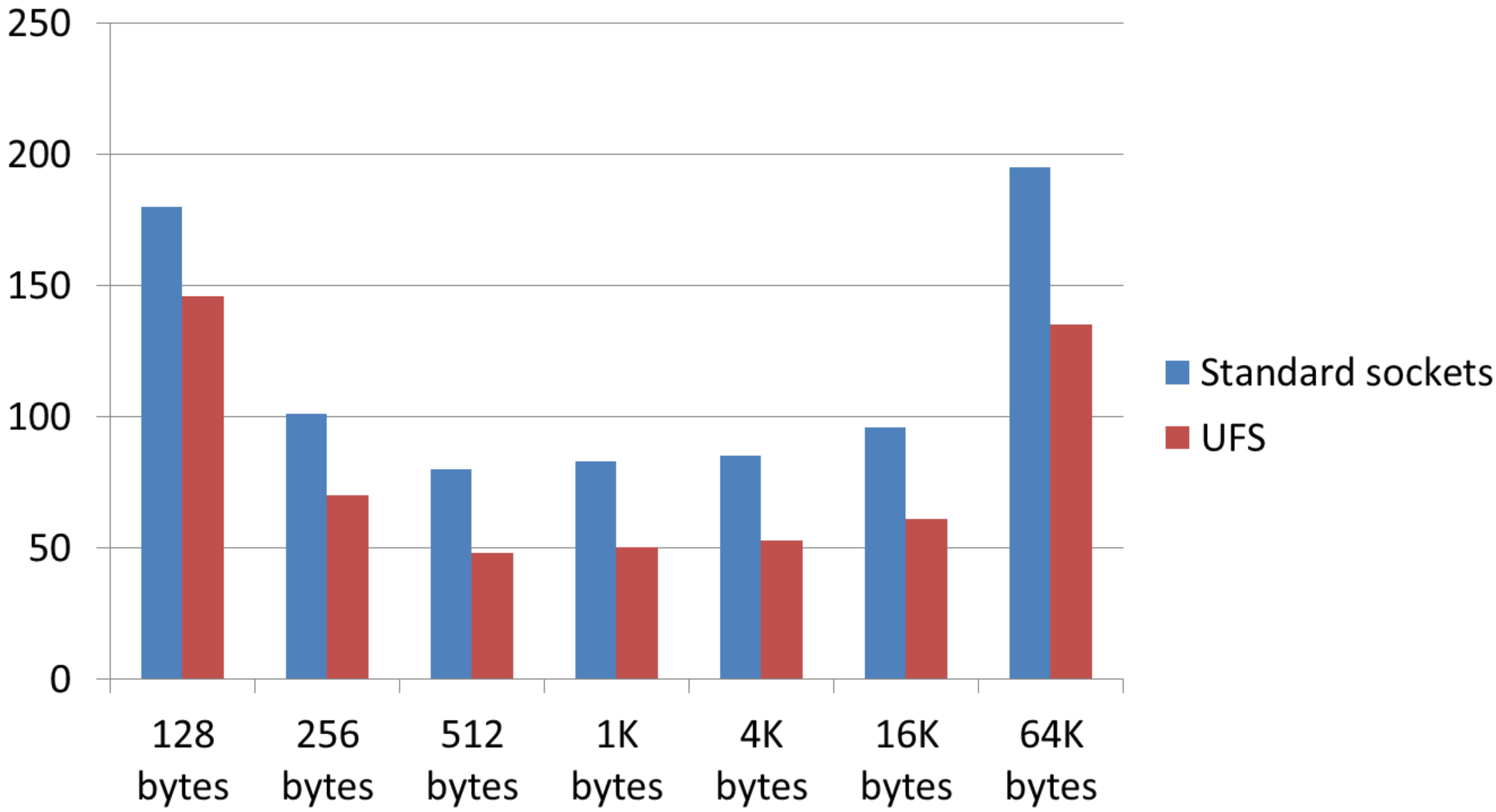


Pub/Sub

Send/Receive

optimizing Java<->Qpid C++ in capital markets

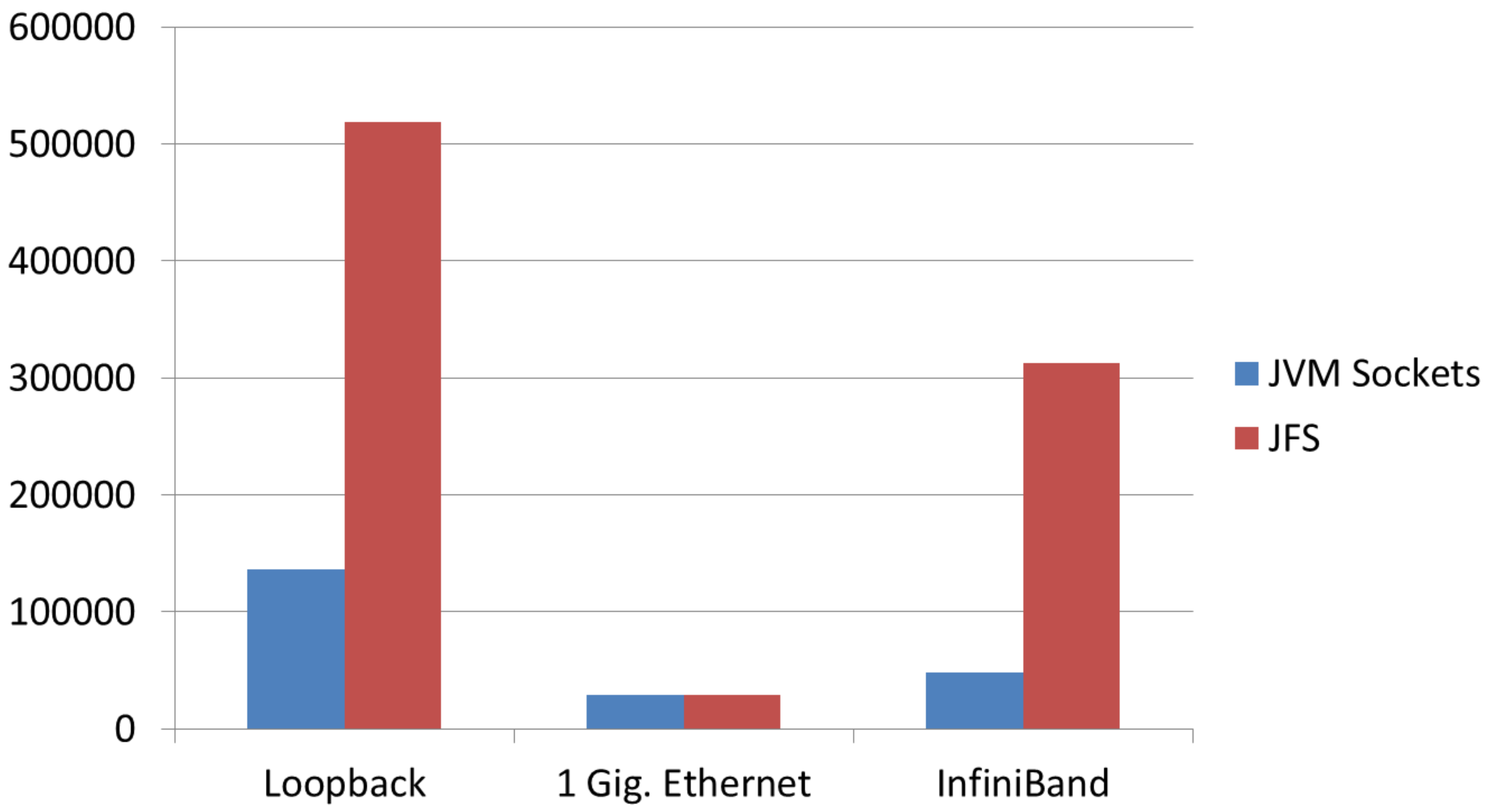
- **Latency (microseconds) in shared memory**



JAVA FAST SOCKETS

optimizing Oracle Coherence

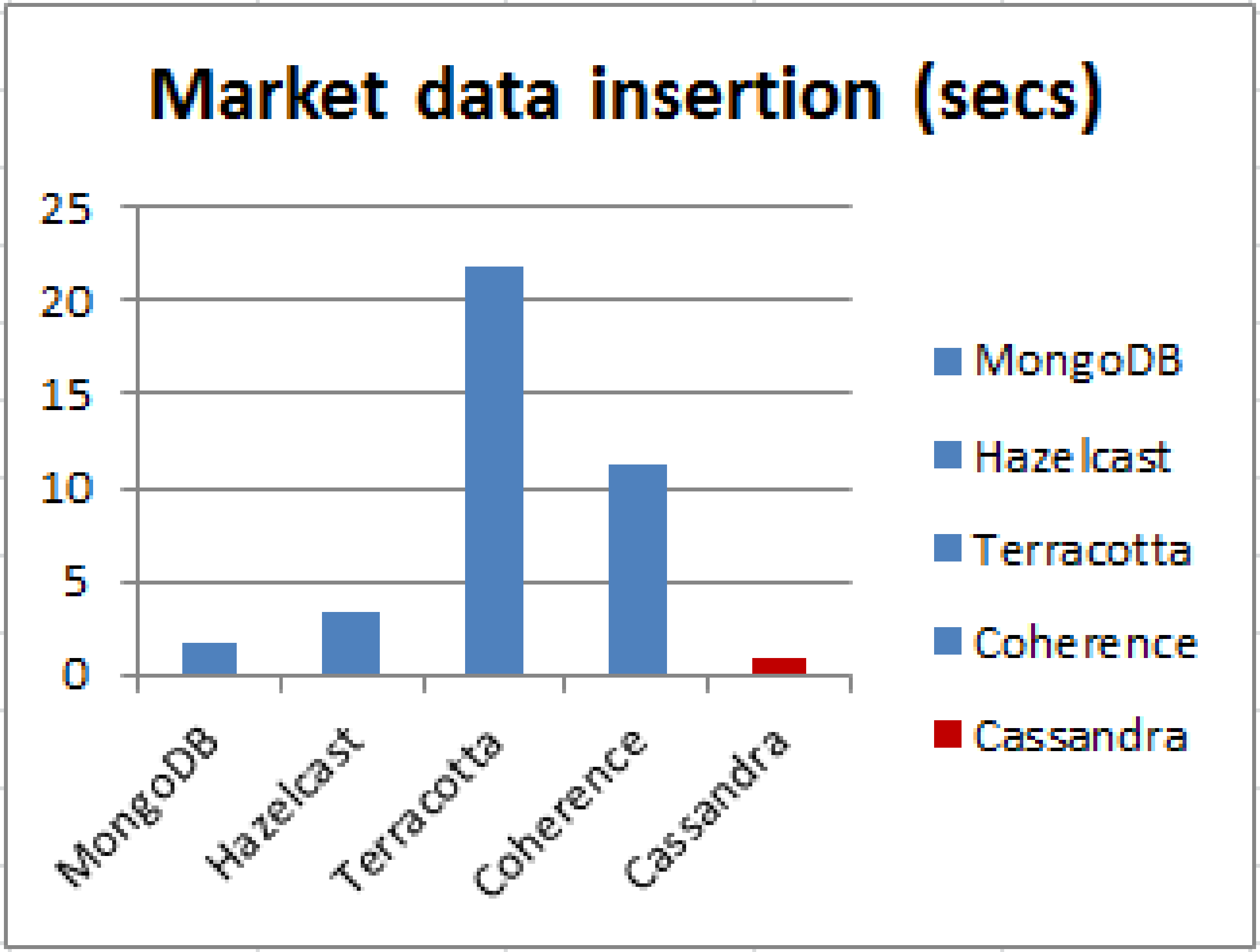
Oracle Coherence Exabus TCP SocketBus (Exalogic) boost (MessageBusTest bench)



optimizing Hazelcast in capital markets

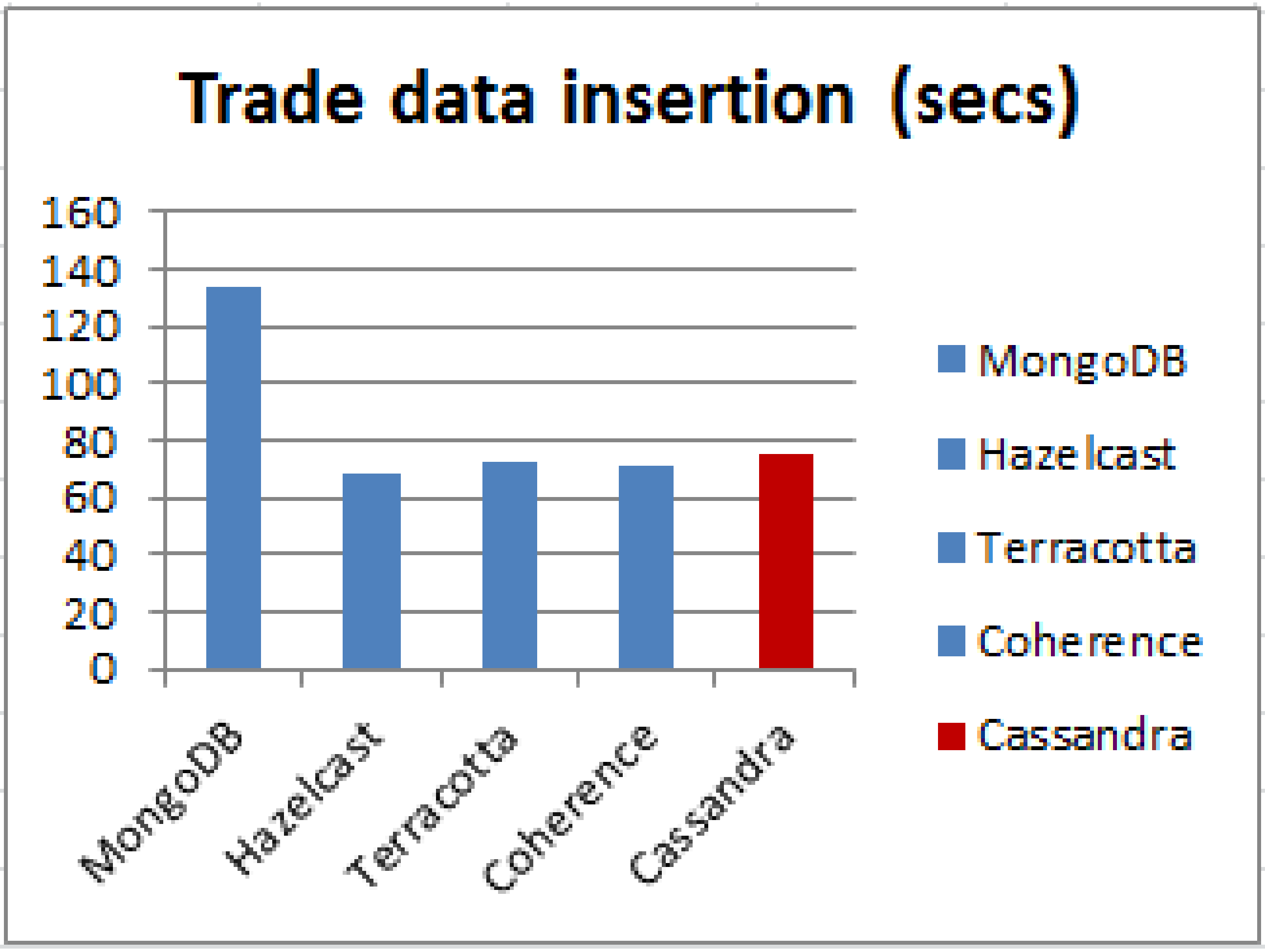
Results from "Raj Subramani (Quant School) "[Comparing NoSQL Data Stores]" plus our execution of the benchmark with Hazelcast+JFS. NB: Better HW+JFS

- **Hazelcast + JFS: 0.417 secs**

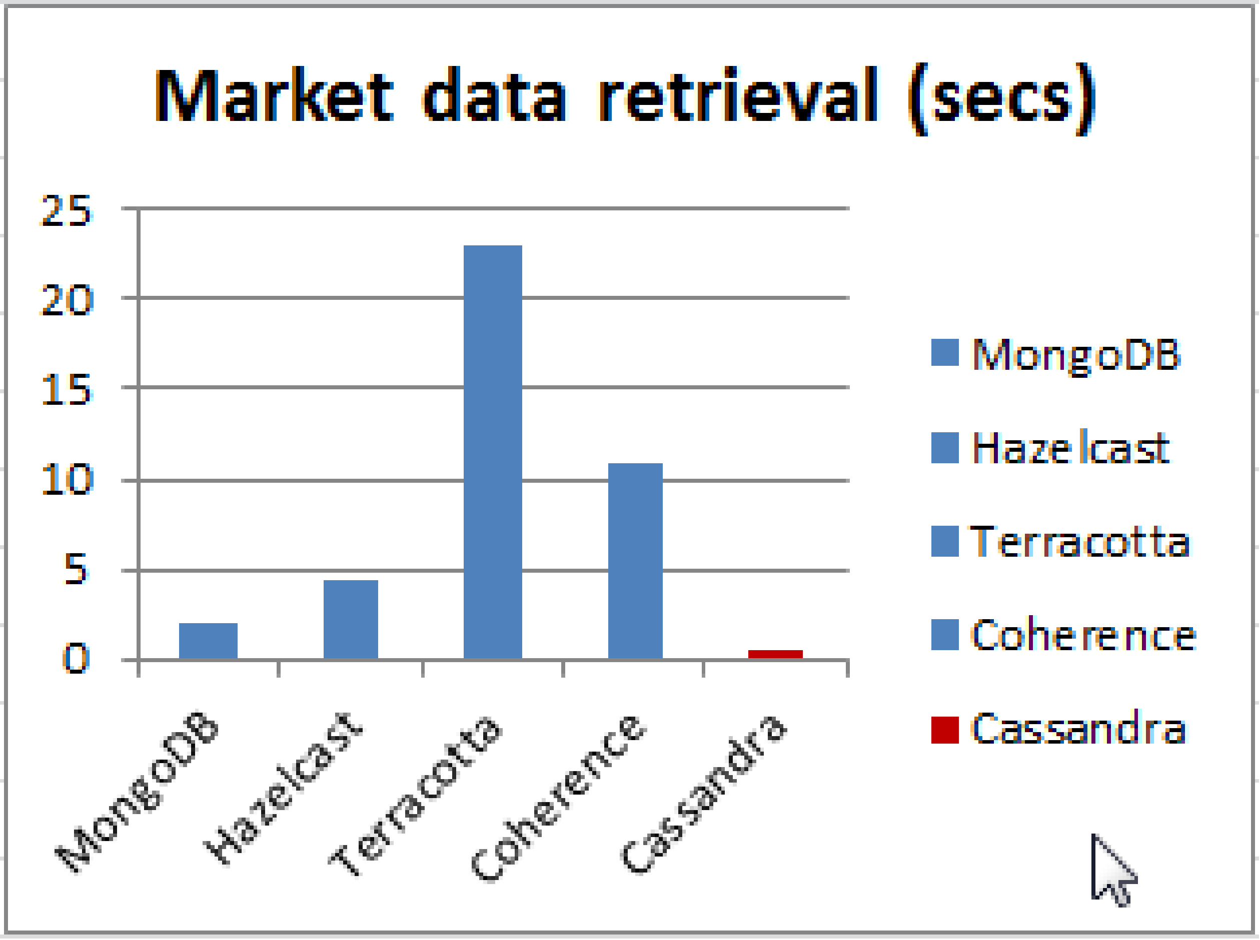


optimizing Hazelcast in capital markets

- **Hazelcast + JFS: 8.058 secs**

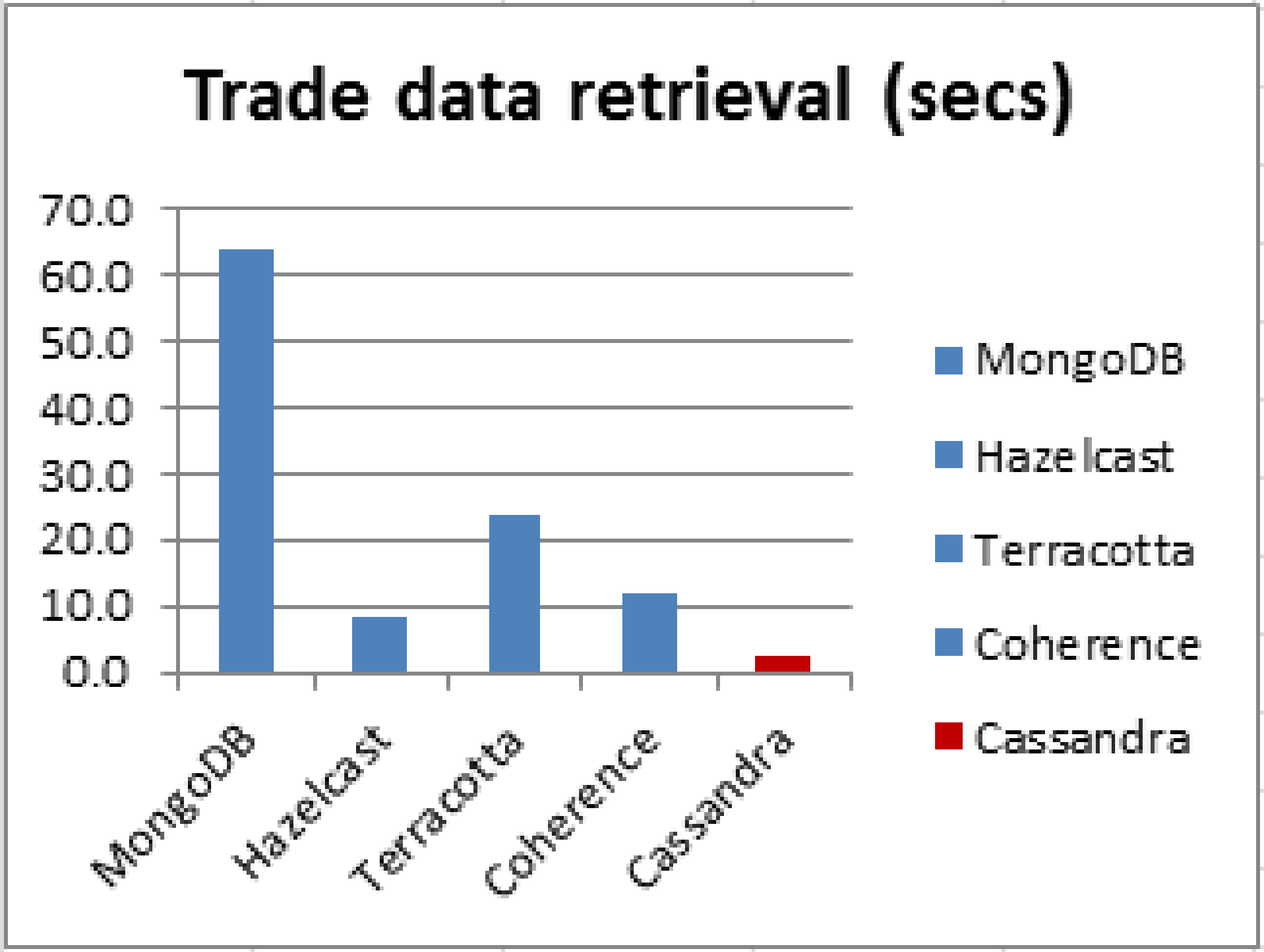


- **Hazelcast + JFS: 0.346 secs**



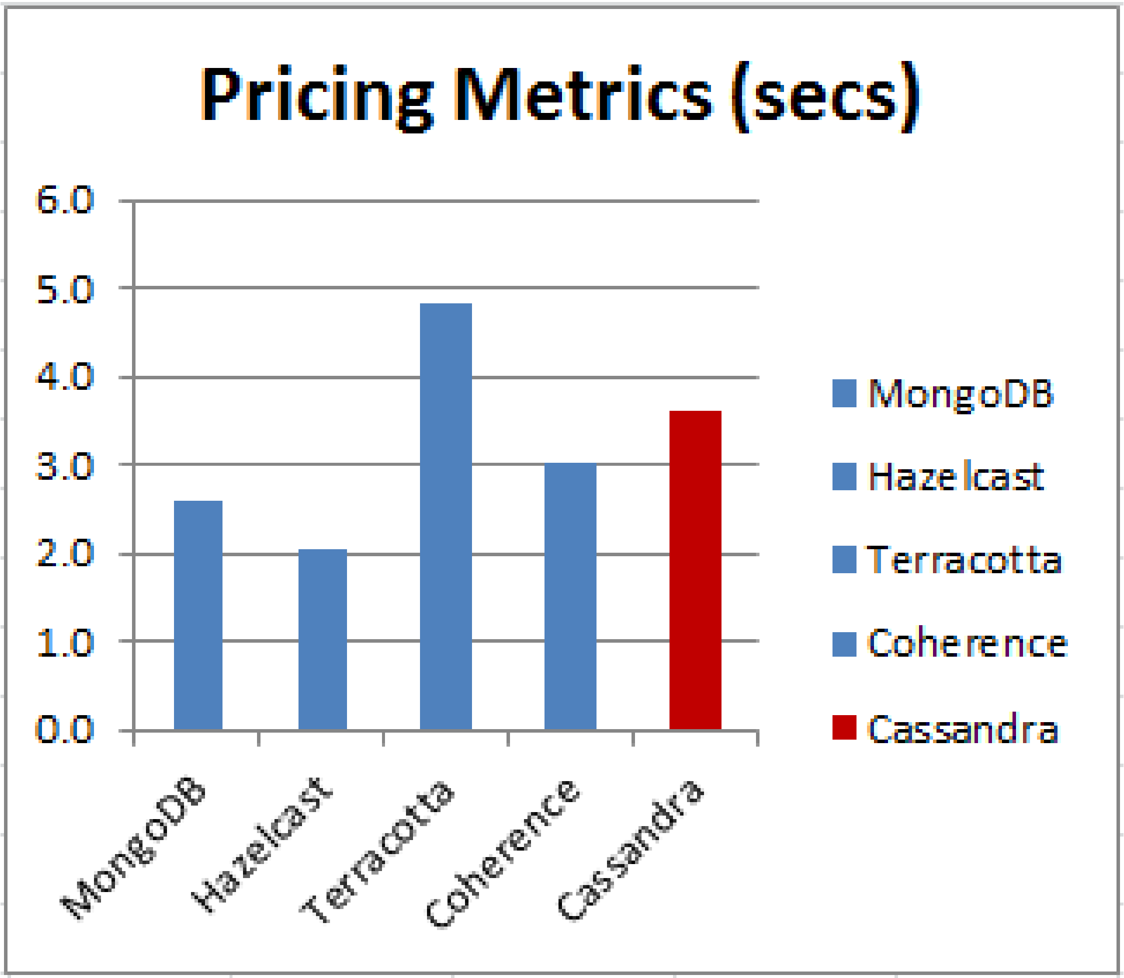
optimizing Hazelcast in capital markets

- **Hazelcast + JFS: 2.139 secs**

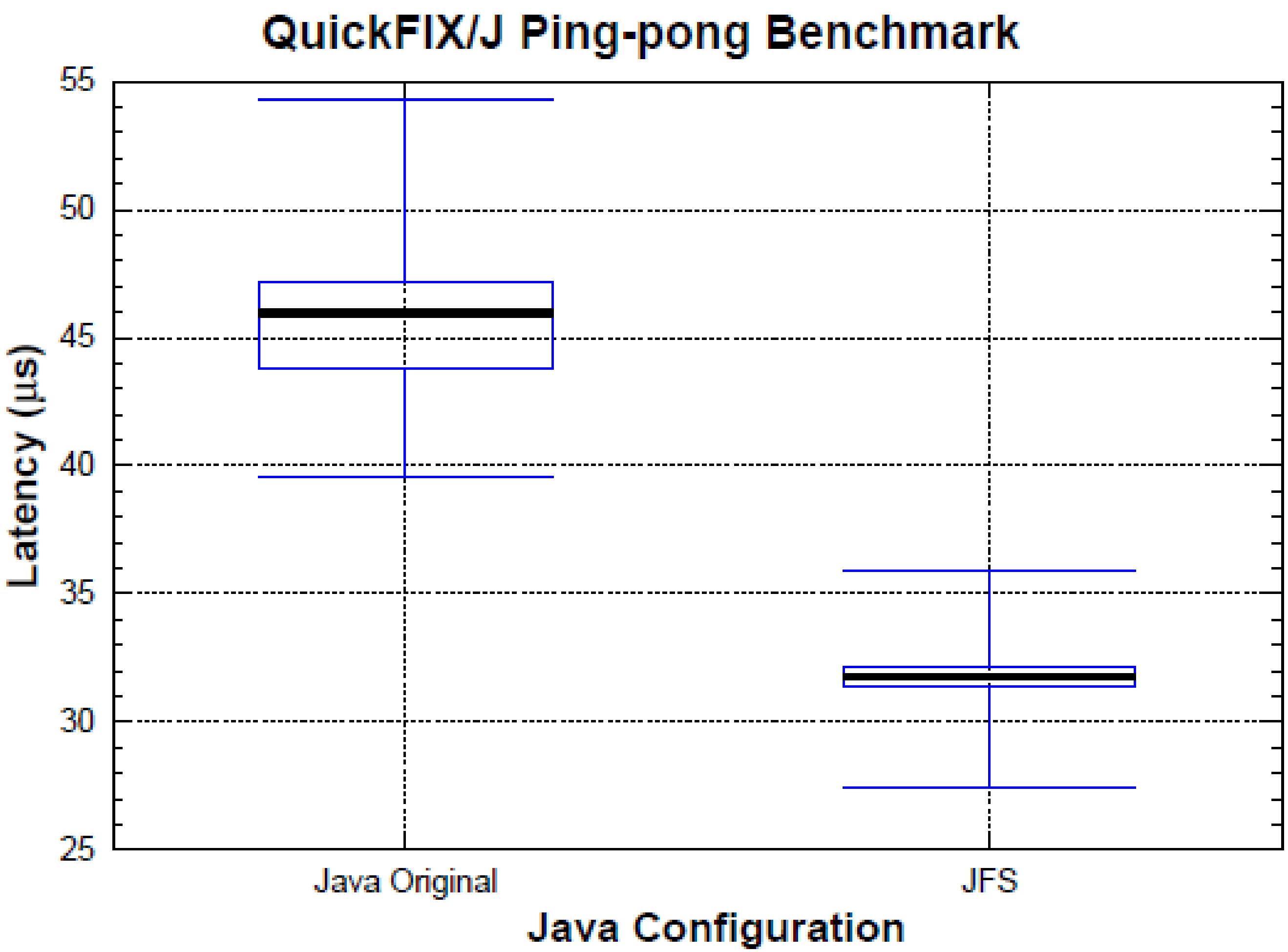


optimizing Hazelcast in capital markets

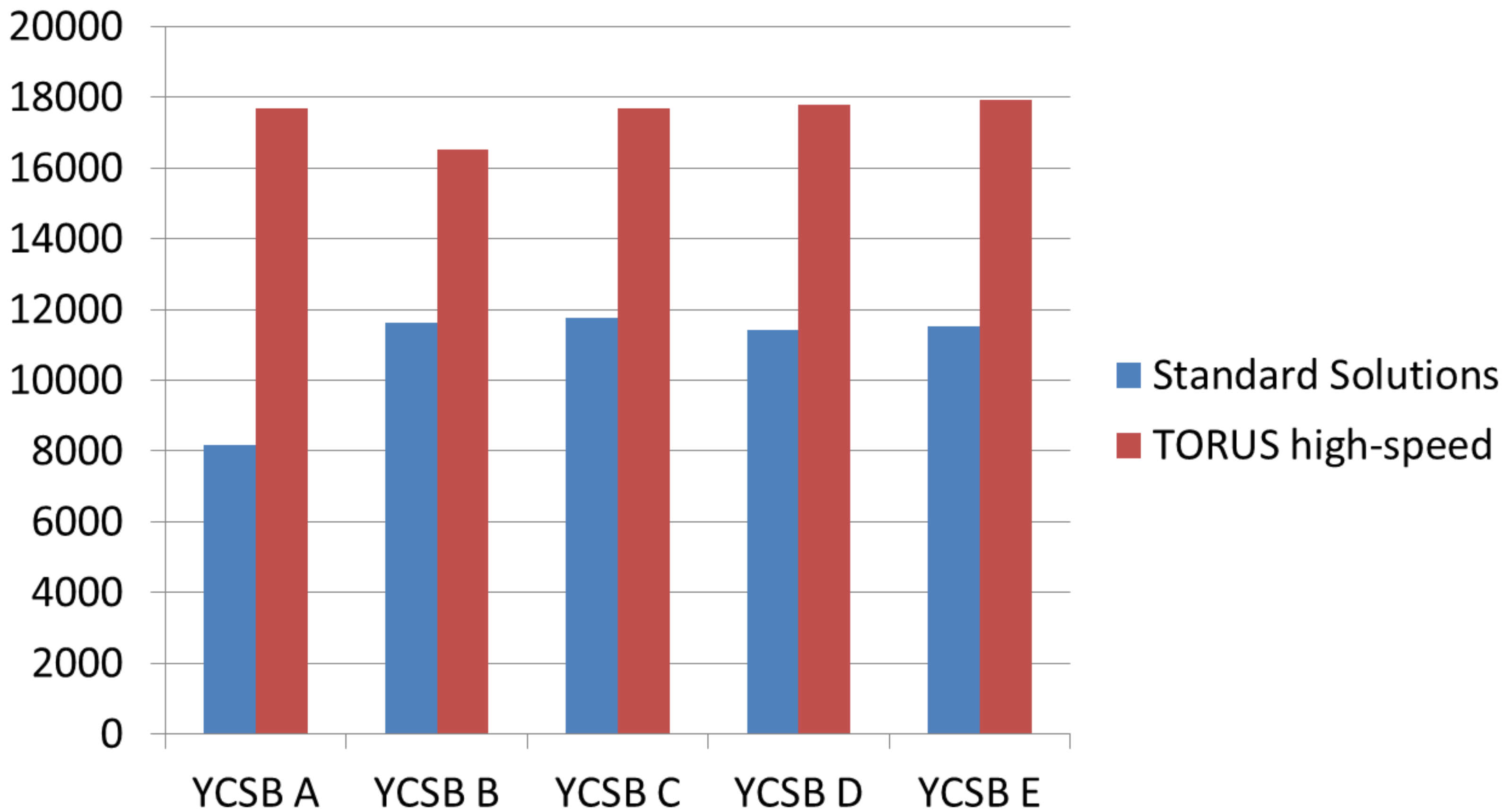
- **Hazelcast + JFS: 1.211 secs**



optimizing QuickFIX/J over Mina (Shmem)



optimizing Hbase (preliminary results)



Workload	Operations	Record selection	Application example
A—Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B—Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C—Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D—Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E—Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform*	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

optimizing Cassandra (Work-in-progress)

- The main bottleneck looks like the Thrift-based driver
- YCSB (A) performance results:
- Throughput Cassandra: 5846 ops Cassandra+JFS: 8097 ops
- Read Latency Cassandra: 166 us Cassandra+JFS: 120 us
- Write Latency Cassandra: 158 us Cassandra+JFS: 108 us
- Working on a pure Java client (promising first results)

torus® optimizing MongoDB

- **YCSB (A) performance results:**
- **Throughput Mongo: 5558 ops Mongo+TORUS: 12222 ops**
- **Read Latency Mongo: 122 us Mongo+TORUS: 42 us**
- **Write Latency Mongo: 176 us Mongo+TORUS: 78 us**
- **Update Latency Mongo: 146 us Mongo+TORUS: 59 us**

For more information on our solutions, please contact us:

✉guillermo.lopez@torusware.com
WWW: http://www.torusware.com

torus®