



## Red Hat Reference Architecture Series

# Red Hat Enterprise Linux 6.1 High Performance Network with MRG

## MRG Messaging: Throughput & Latency

### QDR Infiniband™ 10-GigE, 1-GigE

**Red Hat Enterprise MRG –  
MRG Messaging 1.3**

**Red Hat® Enterprise Linux® 6.1**

**HP Proliant DL380 G7  
12 CPU 48 GB**

**Mellanox QDR Infiniband  
Mellanox 10 GigE  
Broadcom 1 GigE**

**Version 1.0**

**April 2011**





1801 Varsity Drive  
Raleigh NC 27606-2072 USA  
Phone: +1 919 754 3700  
Phone: 888 733 4281  
Fax: +1 919 754 3701  
PO Box 13588  
Research Triangle Park NC 27709 USA

Linux is a registered trademark of Linus Torvalds. Red Hat, Red Hat Enterprise Linux and the Red Hat "Shadowman" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

© 2011 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The information contained herein is subject to change without notice. Red Hat, Inc. shall not be liable for technical or editorial errors or omissions contained herein.

Distribution of modified versions of this document is prohibited without the explicit permission of Red Hat Inc.

Distribution of this work or derivative of this work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from Red Hat Inc.

The GPG fingerprint of the [security@redhat.com](mailto:security@redhat.com) key is:  
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E



# Table of Contents

<u>1 Goals &amp; Executive Summary.....</u>	<u>5</u>
1.1 Goals.....	6
1.2 Executive Summary.....	6
<u>2 Red Hat MRG Messaging – Introduction &amp; Architecture.....</u>	<u>7</u>
2.1 The Basis of MRG Messaging.....	7
2.1.1 Advanced Message Queuing Protocol.....	7
2.1.2 Apache Qpid.....	7
2.1.3 Red Hat Enterprise MRG Messaging.....	7
2.2 How MRG Messaging Operates.....	7
2.3 Red Hat MRG Messaging (AMQP) Architecture .....	9
2.3.1 Main Features.....	10
2.3.2 Messages.....	11
2.3.3 Queues.....	11
2.3.4 Exchanges.....	11
2.3.5 Bindings.....	14
2.4 AMQP Communication Model.....	15
2.5 AMQP Object Model.....	16
<u>3 Performance Testing Methodology.....</u>	<u>17</u>
3.1 Test Harnesses .....	18
3.1.1 Throughput (Perftest).....	18
3.1.2 Latency (qpid-Latency-test).....	18
3.2 Tuning & Parameter Settings.....	18
3.2.1 Processes.....	19
3.2.2 SysCtl.....	20
3.2.3 ethtool.....	20
3.2.4 CPU Affinity.....	21
3.2.5 AMQP parameters.....	22
<u>4 Hardware/Software Versions.....</u>	<u>24</u>
4.1 Hardware.....	24
4.2 Network.....	24
4.3 Software .....	24
<u>5 Performance Results.....</u>	<u>25</u>
5.1 Latency.....	25
5.1.1 1-GigE.....	25
5.1.2 10-GigE.....	26
5.1.3 Internet Protocol over InfiniBand (IpIB).....	27
5.1.4 Remote Direct Memory Access (RDMA) with Mellanox 10Gig E.....	28
5.1.5 Remote Direct Memory Access (RDMA) with Mellanox Infiniband.....	29



<a href="#">5.1.6 Comparisons.....</a>	<a href="#">30</a>
<a href="#">5.2 Throughput.....</a>	<a href="#">34</a>
<a href="#">5.2.1 1-GigE.....</a>	<a href="#">34</a>
<a href="#">5.2.2 10-GigE.....</a>	<a href="#">35</a>
<a href="#">5.2.3 IpoIB.....</a>	<a href="#">36</a>
<a href="#">5.2.4 10-Gig-E RDMA.....</a>	<a href="#">37</a>
<a href="#">5.2.5 IB RDMA.....</a>	<a href="#">38</a>
<a href="#">5.2.6 Comparisons.....</a>	<a href="#">39</a>
<a href="#">5.3 System Metrics.....</a>	<a href="#">43</a>
<a href="#">6 Conclusions.....</a>	<a href="#">47</a>
<a href="#">Appendix A: Memlock configuration.....</a>	<a href="#">47</a>



# 1 Goals & Executive Summary

[High Performance Network Add-On](#) Red Hat Enterprise Linux 6.1 supports Mellanox MT 26428 ConnectX for Infiniband QDR (40Gbps) and MT26448 ConnectX EN (10GbE) for both TCP/IP and Remote Direct Memory Access (RDMA) using Red Hat's High Performance Network Add-On. This Add-on delivers RDMA over high performance Infiniband fabrics as well as RDMA over Converged Ethernet (RoCE) for those times when low network latency and high capacity are important. Because RDMA bypasses system and kernel calls to place data directly into remote system memory with less CPU overhead, the High Performance Networking Add-On is ideal for high-speed data processing applications that require low latency, for speeding up cluster locking, or for scaling up applications on distributed systems without investing in specialized networking technologies.

MRG is a next-generation IT infrastructure that makes enterprise computing 100-fold faster, defines new levels of interoperability and gives customers competitive advantage by running applications and transactions with greater performance and reliability. Red Hat Enterprise MRG integrates Messaging, Realtime, and Grid technologies. Messaging is the backbone of enterprise and high-performance computing, Service-Oriented Architecture (SOA) deployments, and platform services. MRG provides enterprise messaging technology that delivers:

- Unprecedented interoperability through the implementation of Advanced Message Queuing Protocol (AMQP), the industry's first open messaging standard. The solution is cross-language, cross-platform, multi-vendor, spans hardware and software, and extends down to the wire level. Red Hat is a founding member of the AMQP working group, which develops the AMQP standard.
- Linux-specific optimizations to achieve optimal performance on Red Hat Enterprise Linux and MRG Realtime. It can also be deployed on non-Linux platforms such as Windows and Solaris without the full performance and quality of service benefits that Red Hat Enterprise Linux provides.
- Support for most major development languages.

This paper generates performance numbers for MRG Messaging under the Red Hat Enterprise Linux 6.1 operating system using various interconnects and supported protocols include RoCE and RDMA.

- Performance numbers are useful to raise interest in AMQP and MRG Messaging technology when deployed on Red Hat Enterprise Linux.
- This report compares hardware network options to help in deployment and technology selection.
- This report makes no attempt to plot the maximum throughput of MRG, but rather a controlled comparison in a minimal network configuration.



## 1.1 Goals

- Generate MRG Messaging performance numbers, latency and throughput, with various interconnects and supported protocols.
- Evaluate the High Performance Network Add-On in Red Hat Enterprise Linux 6.1 with RoCE.
- Provide tuning and optimization recommendations for MRG Messaging.

## 1.2 Executive Summary

This study measured the throughput and 2-hop fully reliable latency of MRG Messaging V1.3 (using AMQP 0-10 protocol) using 8 to 32768-byte packets running on Red Hat Enterprise Linux 6.1. The results provided were achieved with the testbed described in **Section 5**. It is possible higher throughput could be achieved with multiple drivers and multiple instances of adapters.

Three physical interconnects were used. A single protocol, TCP, was used with the 1-GigE. Two protocols were used with the 10-GigE interconnects and InfiniBand (IB) interconnect. Internet Protocol over InfiniBand (IPoIB) allows users to take partial advantage of IB's latency and throughput using pervasive Internet Protocols (IP). Using the RDMA protocol allows users to take full advantage of the latency and throughput of the IB interconnect however, the cost is loss of the IP application and programming interfaces.

Latency results show that faster interconnects will yield lower latencies. Also shown is Mellanox 10 IB and Mellanox 10Gig-E provided the shortest latency. The Mellanox (QDR connectx2) IPoIB with RDMA had best with averaged readings at 49 microseconds. Mellanox's 10 GigE with RoCE had latencies of 69 microseconds. With IPoIB and RDMA, latencies increased more with transfer size but had results just over 100 microseconds. The 1-GigE best average was 130 microseconds.

For throughput, the faster interconnects prove their worth with larger transfer sizes. The results for smaller transfers had much smaller variances. While IB and 10 GigE hardware was consistently the best performer, the IB protocol to achieve the best throughput depended on the transfer size. For 64-byte transfers, IB with RDMA performed best with 93 MB/s recorded, followed by 10 Gig-E with RDMA (91 MB/s), 1GigE (81 MB/Sec), IB (71.97 MB/s), and 10-GigE (71 MB/s). Using 256-byte transfers, IB performed best with 339 MB/s, trailed closely by 10-GigE (333 MB/s), IB RDMA (276 MB/s), 10-GigE RDMA (261 MB/s), and 1-GigE (157 Mb/s). With a 1024-byte transfer size, 10-GigE ranks first at 1129 MB/s, IB follows with 921 MB/s, then 10GigE RDMA (822 MB/s), IB RDMA (812 MB/s), and 1-GigE (195 MB/s) complete the sequence.



## 2 Red Hat MRG Messaging – Introduction & Architecture

### 2.1 *The Basis of MRG Messaging*

#### 2.1.1 Advanced Message Queuing Protocol

AMQP is an open-source messaging protocol. It offers increased flexibility and interoperability across languages, operating systems, and platforms. AMQP is the first open standard for high performance enterprise messaging. More information about AMQP is available at <http://www.amqp.org>. The full protocol specification is available at <http://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf?version=1> .

#### 2.1.2 Apache Qpid

*Qpid* is an Apache project that implements the AMQP protocol. It is a multi-platform messaging implementation that delivers transaction management, queuing, distribution, security and management. Development on MRG Messaging is also contributed back upstream to the Qpid project. More information can be found on the <http://qpid.apache.org>.

#### 2.1.3 Red Hat Enterprise MRG Messaging

MRG Messaging is an open source messaging distribution that uses the AMQP protocol. MRG Messaging is based on Qpid but includes persistence options, additional components, Linux kernel optimizations, and operating system services not found in the Qpid implementation. For more information refer to <http://www.redhat.com/mrg>.

### 2.2 *How MRG Messaging Operates*

MRG Messaging was designed to provide a way to build distributed applications in which programs exchange data by sending and receiving messages. A message can contain any kind of data. Middleware messaging systems allow a single application to be distributed over a network and throughout an organization without being restrained by differing operating systems, languages, or network protocols. Sending and receiving messages is simple, and MRG Messaging provides guaranteed delivery and extremely good performance.

In MRG Messaging a *message producer* is any program that sends messages. The program that receives the message is referred to as a *message consumer*. If a program both sends and receives messages it is both a message producer and a message consumer.

The *message broker* is the hub for message distribution. It receives messages from message producers and uses information stored in the message's headers to decide where to send it. The broker will normally attempt to send a message until it gets notification from a consumer that the message has been received.

Within the broker are *exchanges* and *queues*. Message producers send messages to exchanges, message consumers subscribe to queues and receive messages from them.

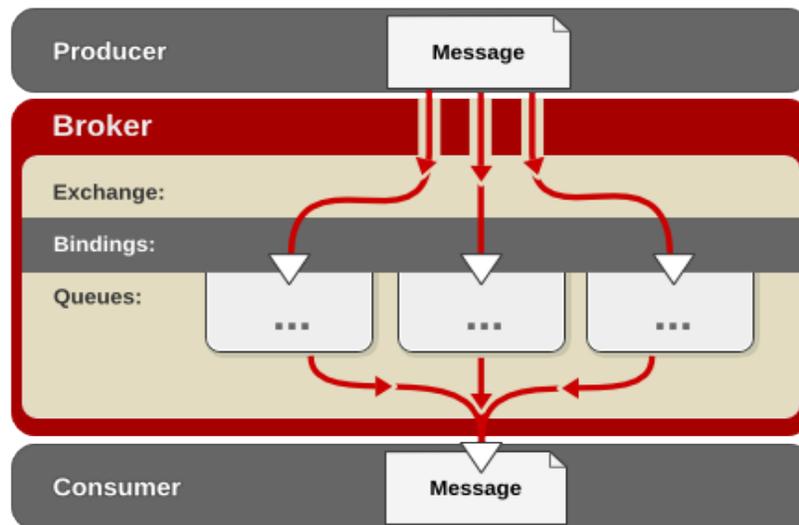


The message headers contain *routing* information. The *routing key* is a string of text that the exchange uses to determine which queues to deliver the message to. *Message properties* can also be defined for settings such as message durability.

A *binding* defines the relationship between an exchange and a message queue. A queue must be bound to an exchange in order to receive messages from it. When an exchange receives a message from a message producer, it examines its active bindings, and routes the message to the corresponding queue. Consumers read messages from the queues to which they are subscribed. Once a message is read, it is removed from the queue and discarded.

As shown in **Figure 1**, a producer sends messages to an exchange. The exchange reads the active bindings and places the message in the appropriate queue. Consumers then retrieve messages from the queues.

**Producer Consumer**



**Figure 1**



## 2.3 Red Hat MRG Messaging (AMQP) Architecture

AMQP was born out of the frustrations in developing front- and back-office processing systems at investment banks. No existing products appeared to meet all the requirements of these investment banks.

AMQP is a binary wire protocol and well-defined set of behaviors for transmitting application messages between systems using a combination of store-and-forward, publish-and-subscribe, and other techniques. AMQP addresses the scenario where there is likely to be some economic impact if a message is lost, does not arrive in a timely manner, or is improperly processed.

The protocol is designed to be usable from different programming environments, operating systems, and hardware devices, as well as making high-performance implementations possible on various network transports including TCP, Stream Control Transmission Protocol (SCTP), and InfiniBand.

From the beginning, AMQP's design objective was to define enough MOM semantics (**Figure 2**) to meet the needs of most commercial computing systems and to do so in an efficient manner that could ultimately be embedded into the network infrastructure. It's not just for banks. AMQP encompasses the domains of store-and-forward messaging, publish-and-subscribe messaging, and point to point. It incorporates common patterns to ease the traversal of firewalls while retaining security, and to permit network QoS. To ease adoption and migration, AMQP is also designed to encompass Java Message Service (JMS) semantics.

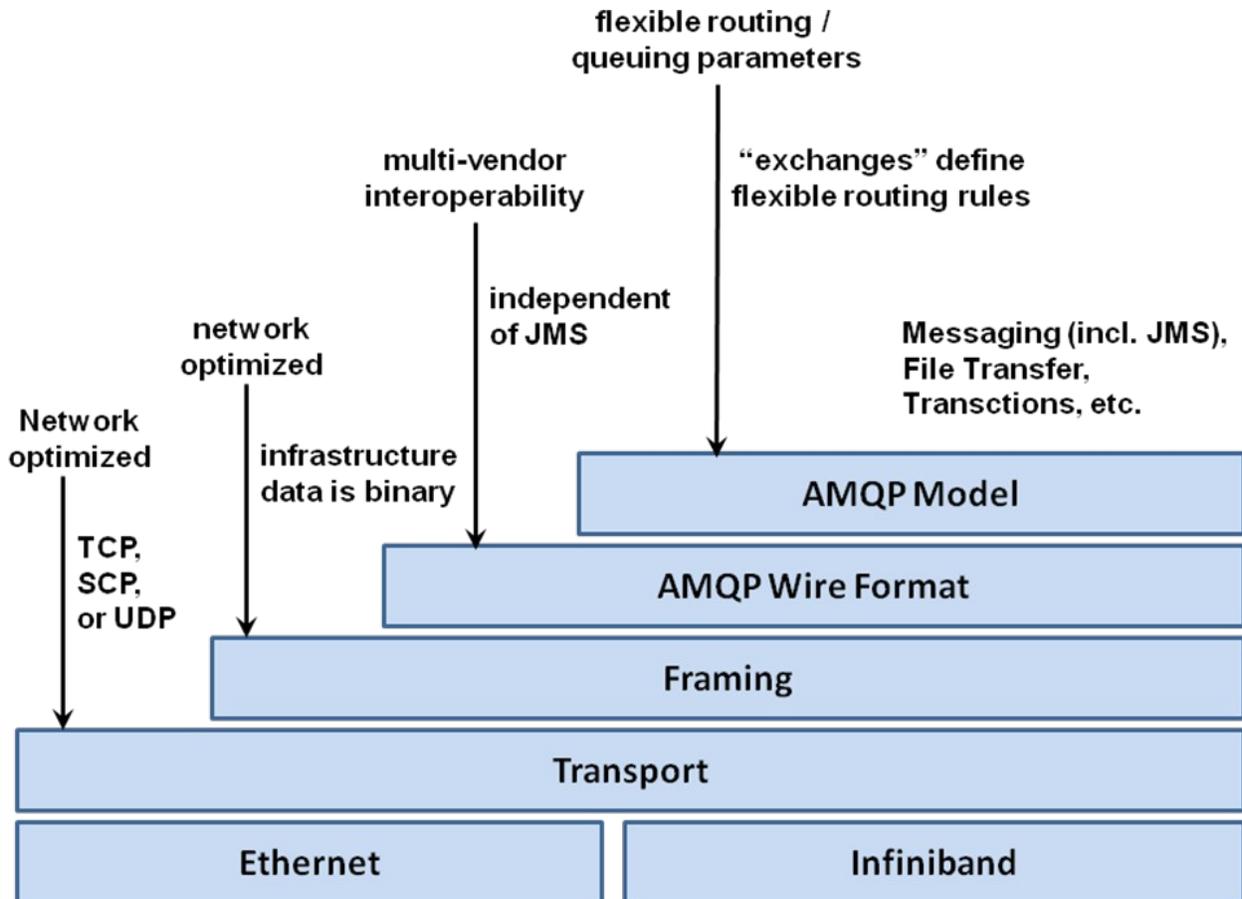


Figure 2

AMQP goes further, however, and includes additional semantics not found in JMS that members of the working group have found useful in delivering large, robust systems over the decades. Interestingly, AMQP does not itself specify the API a developer uses, though it is likely that will happen in the future.

### 2.3.1 Main Features

AMQP is split into two main areas: transport model and queuing model. AMQP is unusual in that it thoroughly specifies the semantics of the services it provides within the queuing model; since applications have a very intimate relationship with their middleware, this needs to be well defined or interoperability cannot be achieved. In this respect, AMQP semantics are more tightly defined than JMS semantics.

As stated, AMQP's transport is a binary protocol using network byte ordering. AMQP strives to be high performance and flexible, and to be hardware friendly rather than human friendly. The protocol *specification* itself, however, is written in XML so implementers can code-generate large portions of their implementations, making it easier for vendors to support the technology.

The transport model itself can use different underlying transports. MRG supports TCP/IP, InfiniBand RDMA, 10GigE RDMA, and TSL.



## 2.3.2 Messages

Messages in AMQP are self-contained and long-lived, and their content is immutable and opaque. The content of messages is essentially unlimited in size; 4GB messages are supported just as easily as 4KB messages. Messages have headers that AMQP can read and use to help in routing.

This can be likened to a postal service: a message is the envelope, the headers are information written on the envelope and visible to the mail carrier, and the carrier may add various postmarks to the envelope to help deliver the message. The valuable content is within the envelope, hidden from and not modified by the carrier. The analogy holds quite well except that it is possible for AMQP to make unlimited copies of the messages to deliver if required.

## 2.3.3 Queues

Queues are the core concept in AMQP. Every message *always* ends up in a queue, even if it is an in-memory private queue feeding a client directly. To extend the postal analogy, queues are mailboxes at the final destination or intermediate holding areas in the sorting office.

Queues can store messages in memory or on disk. They can search and reorder messages, and they may participate in transactions. The administrator can configure the service levels they expect from the queues with regard to latency, durability, availability, etc. These are all aspects of implementation and not defined by AMQP. This is one way commercial implementations can differentiate themselves while remaining AMQP-compliant and interoperable.

## 2.3.4 Exchanges

Exchanges are the delivery service for messages. In the postal analogy, exchanges provide sorting and delivery services. In the AMQP model, selecting a different carrier is how different ways of delivering the message are selected. The exchange used by a publish operation determines if the delivery will be direct or publish-and-subscribe, for example. The exchange concept is how AMQP brings together and abstracts different middleware delivery models. It is also the main extension point in the protocol.

A client chooses the exchange used to deliver each message as it is published. The exchange looks at the information in the headers of a message and selects where they should be transferred to. This is how AMQP brings the various messaging idioms together - clients can select which exchange should route their messages.

Several exchanges must be supported by a compliant AMQP implementation:

- The fanout exchange will distribute messages to every queue. Any routing information provided by the producer is ignored. See **Figure 3**.



## Fanout Exchange

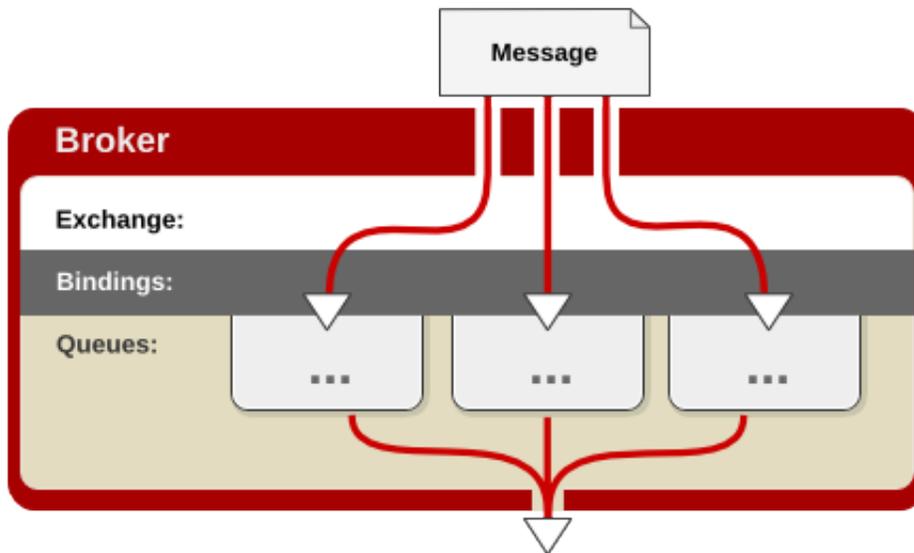


Figure 3

- The direct exchange will queue a message directly at a single queue, choosing the queue on the basis of the "routing key" header in the message and matching it by name. This is how a letter carrier delivers a message to a postal address. See **Figure 4**.

## Direct Exchange

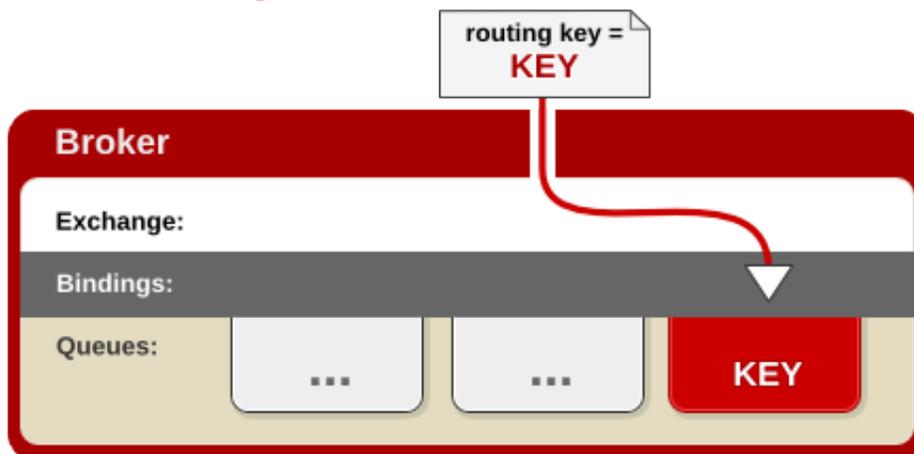
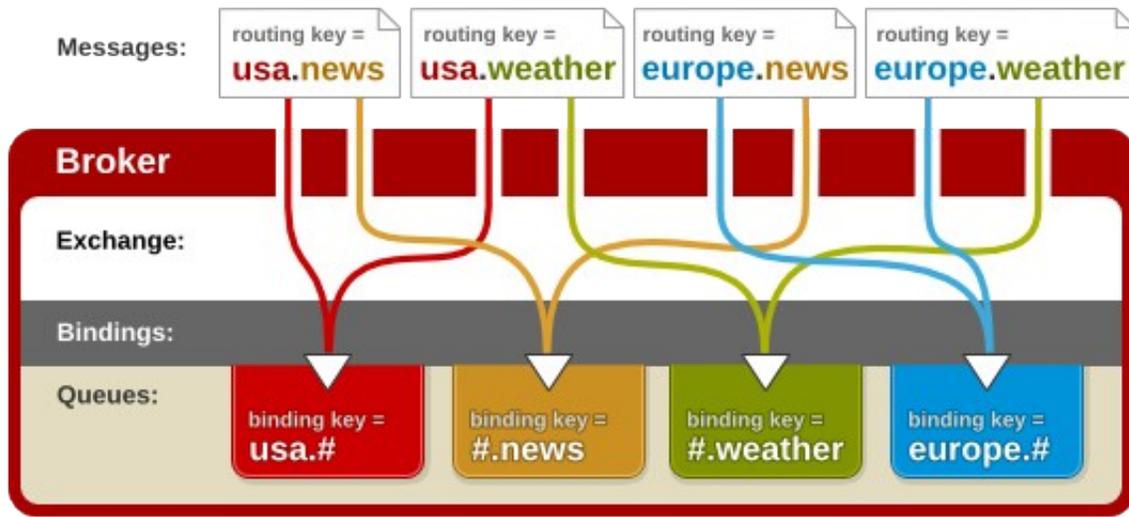


Figure 4



- The topic exchange will copy and queue the message to all clients that have expressed an interest based on a rapid pattern match with the routing key header. You can think of the routing key as an address, but it is a more abstract concept useful to several types of routing. See **Figure 5**.

## Topic Exchange



**Figure 5**

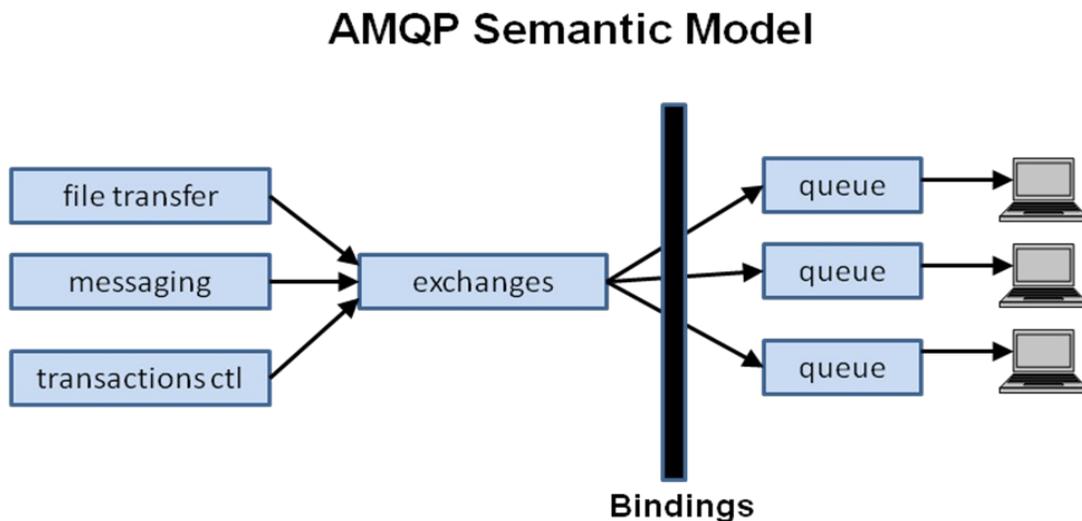
- The headers exchange will examine all the headers in a message, evaluating them against query predicates provided by interested clients using those predicates to select the final queues, copying the message as necessary.
- In addition the implementation provides an additional exchange for XML routing. This exchange allows for the routing of XML based messages using XQuery. The XML exchange has not been benchmarked in this report.

Throughout this process, exchanges never store messages but they do retain binding parameters supplied to them by the clients using them. These bindings are the arguments to the exchange routing functions that enable the selection of one or more queues.



## 2.3.5 Bindings

The arguments supplied to exchanges to enable the routing of messages are known as bindings (see **Figure 6**). Bindings vary depending on the nature of the exchange; the direct exchange requires less binding information than the headers exchange. Notably, it is not always clear which entity should provide the binding information for a particular messaging interaction. In the direct exchange, the sender is providing the association between a routing key and the desired destination queue. This is the origin of the "destination" addressing idiom so common to JMS and other queuing products.



**Figure 6**

In the topic exchange, it is the receiving client that provides the binding information, specifying that when the topic exchange sees a message that matches any given client(s) binding(s), the message should be delivered to all of them.

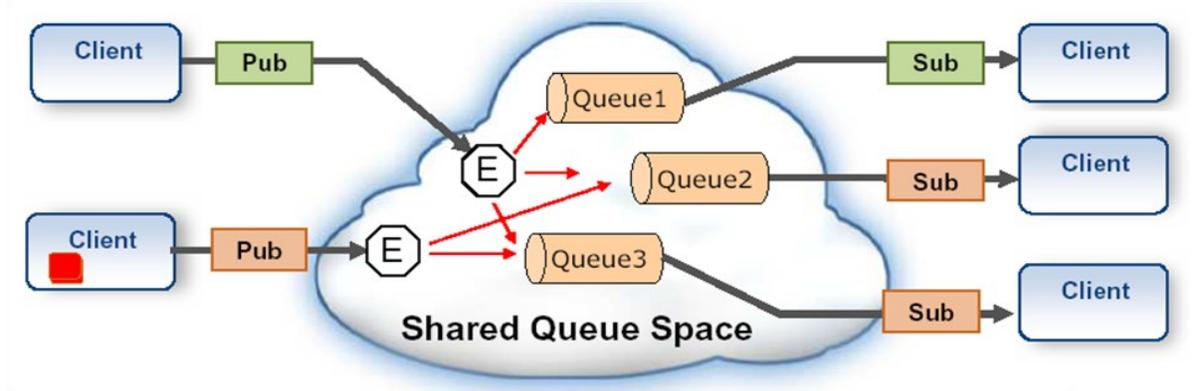
AMQP has no concept of a "destination" since it does not make sense for consumer-driven messaging. It would limit its abstract routing capabilities. The concept of bindings and the convention of using a routing key as the default addressing information overcome the artificial divisions that have existed in many messaging products.



## 2.4 AMQP Communication Model

Provides a “**Shared Queue Space**” that is accessible to all interested applications:

- **Messages** are sent to an **Exchange**
- Each message has an associated **Routing Key**
- **Brokers** forward messages to one or more **Queues** based on the **Routing Key**
- Subscribers get messages from **named Queues**
- Only **one subscriber** can get a given message from each **Queue**



*Figure 7*



## 2.5 AMQP Object Model

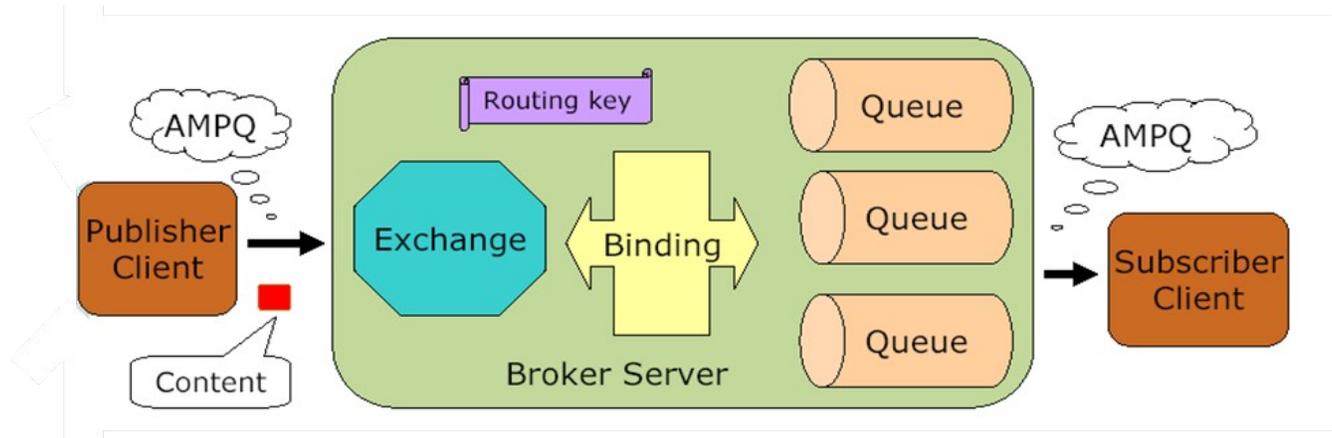
**Exchange** – Receives messages and routes to a set of message queues

**Queue** – Stores messages until they can be processed by the application(s)

**Binding** – Routes messages between Exchange and Queue. Configured externally to the application – Default binding maps routing-key to Queue name

**Routing Key** – label used by the exchange to route Content to the queues

**Content** – Encapsulates application data and provides the methods to send receive, acknowledge, etc



**Figure 8**



### 3 Performance Testing Methodology

The performance tests simulate a 3-tier configuration where producers publish to the broker and consumers subscribe to the broker.

#### Logical Configuration for Performance Testing

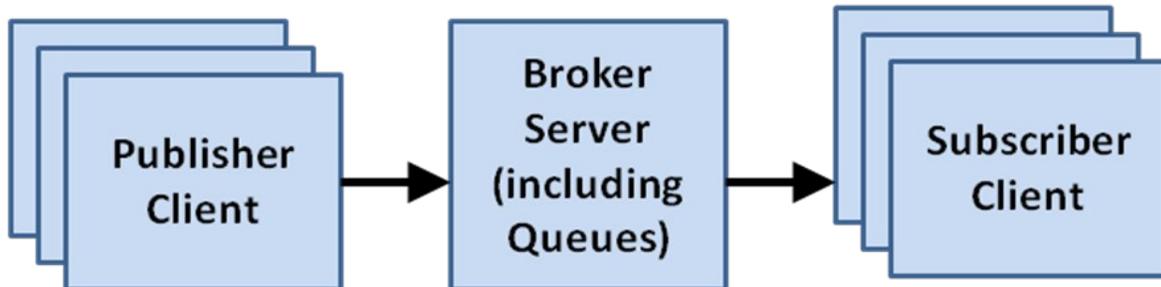


Figure 9

In the actual test setup, the Client System can run multiple instances of Publishers / Subscribers. Further, each Publisher doubles as both a Publisher and a Subscriber. Thus, a Publisher (in the Client System) generates messages which are sent to a Queue in the Broker Server and then returned to the originating Publisher (in the originating Client System) which also doubles as the Subscriber for that message.

#### Physical Configuration for Performance Testing

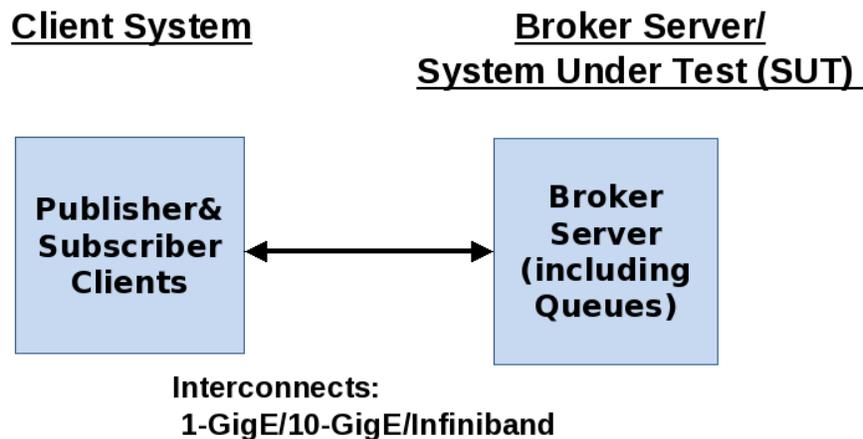


Figure 10



## 3.1 Test Harnesses

Red Hat Enterprise Linux MRG supplies AMQP test harnesses for throughput and latency testing.

### 3.1.1 Throughput (Perftest)

For throughput, `qpido-perftest` is used to drive the broker for this benchmark. This harness is able to start up multiple producers and consumers in balanced (n:n) or unbalanced(x:y) configurations.

What the test does:

- creates a control queue
- starts x:y producers and consumers
- waits for all processors to signal they are ready
- controller records a timestamp
- producers reliably en-queues messages onto the broker as fast as they can
- consumers reliably de-queue messages from the broker as fast as they can
- once the last message is received, the controller is signaled
- controller waits for all complete signals, records timestamp and calculates rate

The throughput is calculated as the total number of messages reliably transferred divided by the time to transfer those messages.

### 3.1.2 Latency (qpido-Latency-test)

For latency, `qpido-latency-test` is used to drive the broker for this benchmark. This harness is able to produce messages at a specified rate or for a specified number of messages that are timestamped, sent to the broker, and looped back to client node. All the measurements are 2-hop, from the client to the broker and back. The client will report the minimum, maximum, and average time for a reporting interval when a rate is used, or for all the messages sent when a count is used.

## 3.2 Tuning & Parameter Settings

For the testing in this paper the systems were not used for any other purposes. Therefore, the configuration and tuning that is detailed should be reviewed when other applications along with MRG Messaging.

Fine tuning the throughput tests proved difficult since the results from one data collection to another was fairly variable. This variability is a result of the fact that the test can start up a number of processes and connections. Any tuning that did not produce a significant difference compared to the variability could not be detected.



### 3.2.1 Processes

For the testing performed, the following were disabled (unless specified otherwise):

- SELinux
- auditd
- avahi-daemon
- certmonger
- cgconfig
- cgroup
- cpuspeed
- crond
- cups
- haldaemon
- ip6tables
- iptables
- irqbalance
- kdump
- matahari-host
- matahari-net
- matahari-service
- mdmonitor
- netconsole
- nfs
- ntpd
- ntpdate
- oddjobd
- opensm
- psacct
- quota\_nld
- rdisc
- restorecond
- rhnsd
- rhsmcertd
- rpcgssd
- rpcidmapd
- rpcsvcgssd
- saslauthd
- smartd
- sssd
- tuned
- ypbind
- libvirt-guests
- libvirtd



## 3.2.2 SysCtl

The following kernel parameters were added to */etc/sysctl.conf*.

fs.aio-max-nr	262144	The maximum number of allowable concurrent requests.
net.ipv4.conf.default.arp_filter	1	The kernel only answers to an ARP request if it matches its own IP address.
net.ipv4.conf.all.arp_filter	1	Enforce sanity checking, also called ingress filtering or egress filtering. The point is to drop a packet if the source and destination IP addresses in the IP header do not make sense when considered in light of the physical interface on which it arrived.

**Table 1**

## 3.2.3 ethtool

Some of the options `ethtool` allows the operator to change relate to coalesce and offload settings. However, during experimentation only changing the ring settings had noticeable effect for throughput testing.

1Gb Network

```
# ethtool -g eth0
Ring parameters for eth0:
Pre-set maximums:
RX:          4096
RX Mini:     0
RX Jumbo:    0
TX:          4096
Current hardware settings:
RX:          256
RX Mini:     0
RX Jumbo:    0
TX:          256

# ethtool -G eth0 rx 4096 tx 4096
# ethtool -g eth0
Ring parameters for eth0:
Pre-set maximums:
RX:          4096
RX Mini:     0
RX Jumbo:    0
TX:          4096
Current hardware settings:
RX:          4096
RX Mini:     0
RX Jumbo:    0
```



```
TX:          4096
```

```
10GigE Network
```

```
# ethtool -g eth2
```

```
Ring parameters for eth2:
```

```
Pre-set maximums:
```

```
RX:          8192
```

```
RX Mini:     0
```

```
RX Jumbo:    0
```

```
TX:          8192
```

```
Current hardware settings:
```

```
RX:          1024
```

```
RX Mini:     0
```

```
RX Jumbo:    0
```

```
TX:          512
```

```
# ethtool -G eth2 rx 8192 tx 8192
```

```
# ethtool -g eth2
```

```
Ring parameters for eth2:
```

```
Pre-set maximums:
```

```
RX:          8192
```

```
RX Mini:     0
```

```
RX Jumbo:    0
```

```
TX:          8192
```

```
Current hardware settings:
```

```
RX:          8192
```

```
RX Mini:     0
```

```
RX Jumbo:    0
```

```
TX:          8192
```

### 3.2.4 CPU Affinity

For latency testing, all interrupts from the cores of one CPU socket were reassigned to other cores. The interrupts for the interconnect under test were assigned to cores of this vacated socket. The processes related to the interconnect (e.g. `ib_mad`, `ipoib`) were then scheduled to run on the vacated cores. The Qpid daemon was also scheduled to run on these or a subset of the vacated cores. How `qpid-latency-test` was scheduled was determined by the results of experiments limiting or not limiting the `qpid-latency-test` test process to specific cores. Experiments with `qpid-perftest` show that typically the best performance was achieved with the affinity settings after a boot without having been modified.



Interrupts can be re-assigned to specific cores or set of cores. `/proc/interrupts` can be queried to identify the interrupts for devices and the number of times each CPU/core has handled each interrupt. For each interrupt, a file named `/proc/irq/<IRQ #>/smp_affinity` contains a hexadecimal mask that controls which cores can respond to specific interrupt. The contents of these files can be queried or set.

Processes can be restricted to run on a specific set of CPUs/cores. `taskset` can be used to define the list of CPUs/cores on which processes can be scheduled to execute. Also, `numactl` can be used to define the list of Sockets on which processes can be scheduled to execute

The MRG – Realtime product includes the application `tuna`, which allows for easy setting of affinity of interrupts and processes, via a GUI or command line.

### 3.2.5 AMQP parameters

Qpid parameters can be specified on the command line, through environment variables or through the Qpid configuration file.

The tests were executed with the following `qpidd` options:

<code>--auth no</code>	disable connection authentication, makes setting the test environment easier
<code>--mgmt-enable no</code>	disable management data collection
<code>--tcp-nodelay</code>	disable packet batching
<code>--worker-threads &lt;#&gt;</code>	set the number of IO worker threads to <#> This was used only for the latency tests, where the range used was between 1 and the numbers of cores in a socket plus one. The default, which was used for throughput, is one more than the total number of active cores.

**Table 2**



**Table 3** details the options specified for `qpid-perftest`. For all testing in this paper a count of 100000 was used. Experimentation was used to detect if setting `tcp-nodelay` was beneficial. For each size reported, the `npubs` and `nsubs` were set equally from 1 to 8 by powers of 2 while `qt` was set from 1 to 8, also by powers of 2. The highest value for each size is reported.

<code>--nsubs &lt;#&gt;</code> <code>--npubs &lt;#&gt;</code>	number of publishers/ subscribers per client
<code>--count &lt;#&gt;</code>	number of messages send per pub per qt, so total messages = count * qt * (npub+nsub)
<code>--qt &lt;#&gt;</code>	number of queues being used
<code>--size &lt;#&gt;</code>	message size
<code>--tcp-nodelay</code>	disable the batching of packets
<code>--protocol &lt;tcp  rdma&gt;</code>	used to specify RDMA, default is TCP

**Table 3**

The parameters used for `qpid-latency-test` are listed in **Table 4**. A 10,000 message rate was chosen so all test interconnects would be able to maintain this rate. When specified, the `max-frame-size` was set to 120 greater than the size. When a `max-frame-size` was specified, `bound-multiplier` was set to 1.

<code>--rate &lt;#&gt;</code>	target message rate
<code>--size &lt;#&gt;</code>	message size
<code>--max-frame-size &lt;#&gt;</code>	the maximum frame size to request, only specified for ethernet interconnects
<code>--bounds-multiplier &lt;#&gt;</code>	bound size of write queue (as a multiple of the max frame size), only specified for ethernet interconnects
<code>--tcp-nodelay</code>	disable packet batching
<code>--protocol &lt;tcp  rdma&gt;</code>	used to specify RDMA, default is TCP

**Table 4**



## 4 Hardware/Software Versions

### 4.1 Hardware

Client System	HP DL380 G7 Dual Socket, Six Core (Total of 12 cores) Intel Xeon X5650 @ 2.67GHz 48 GB RAM
Broker Server	HP DL380 G7 Dual Socket, Six Core (Total of 12 cores) Intel Xeon X5650 @ 2.67 GHz 48 GB RAM

**Table 5**

### 4.2 Network

All load driver (client) systems have point-to-point connections to the system under test (SUT). Network cards used during the sessions were:

1 GigE Adapter	DL380 - Embedded Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet
10 GigE Adapters	Mellanox Technologies MT26448 [ConnectX EN 10GigE, PCIe 2.0 5GT/s]direct connect with SR Fibre
Infinband QDR	Mellanox Technologies MT26428 [ConnectX VPI PCIe 2.0 5GT/s - IB QDR / 10GigE]

**Table 6**

### 4.3 Software

Qpid	0.7.946106-13.el6.x86_64
Red Hat Enterprise Linux	2.6.32-117.el6.x86_64
bnx2	2.0.17
lgb	2.1.0-k2
mlx4_en	1.5.1.6
mth4_ib	1.0
ofed/openib	1.4.1.3

**Table 7**



## 5 Performance Results

### 5.1 Latency

qpid-Latency-test, with specified options, reports the minimum, maximum, and average latency for each 10,000 messages sent every second. The test runs performed collected 100 seconds of data for each size.

Each graph of latency results, plots 100 averages for each size tested. The legend on the right side identifies the transfer size with the average of the 100 values in parenthesis.

#### 5.1.1 1-GigE

As expected, the latency increases with on-board 1GigE with the transfer size.

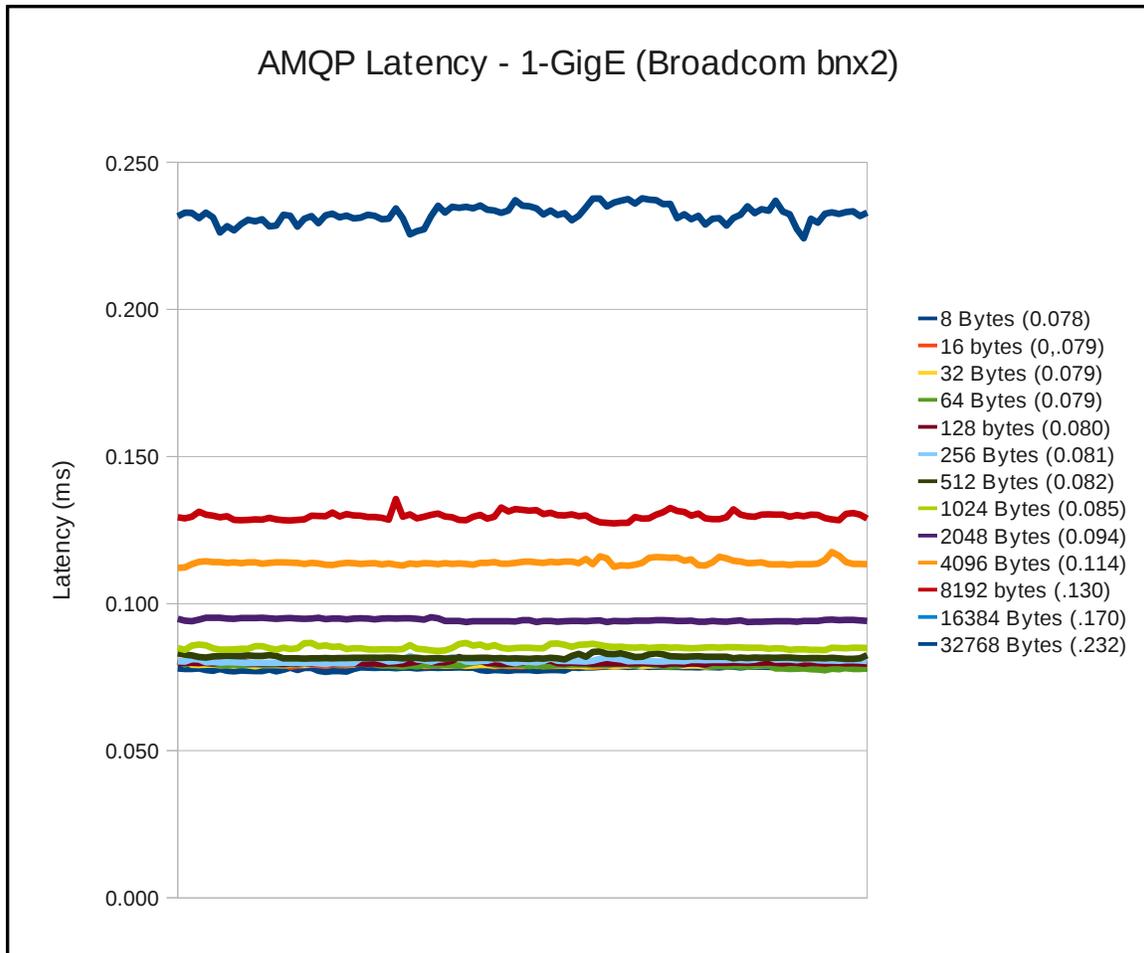


Figure 11



### 5.1.2 10-GigE

The latencies for Mellanox 10-GigE are lower than those of 1-GigE and IPoIB. This is the only interface where the latencies do not consistently increase as the transfer size gets larger using TCP/IP protocol.

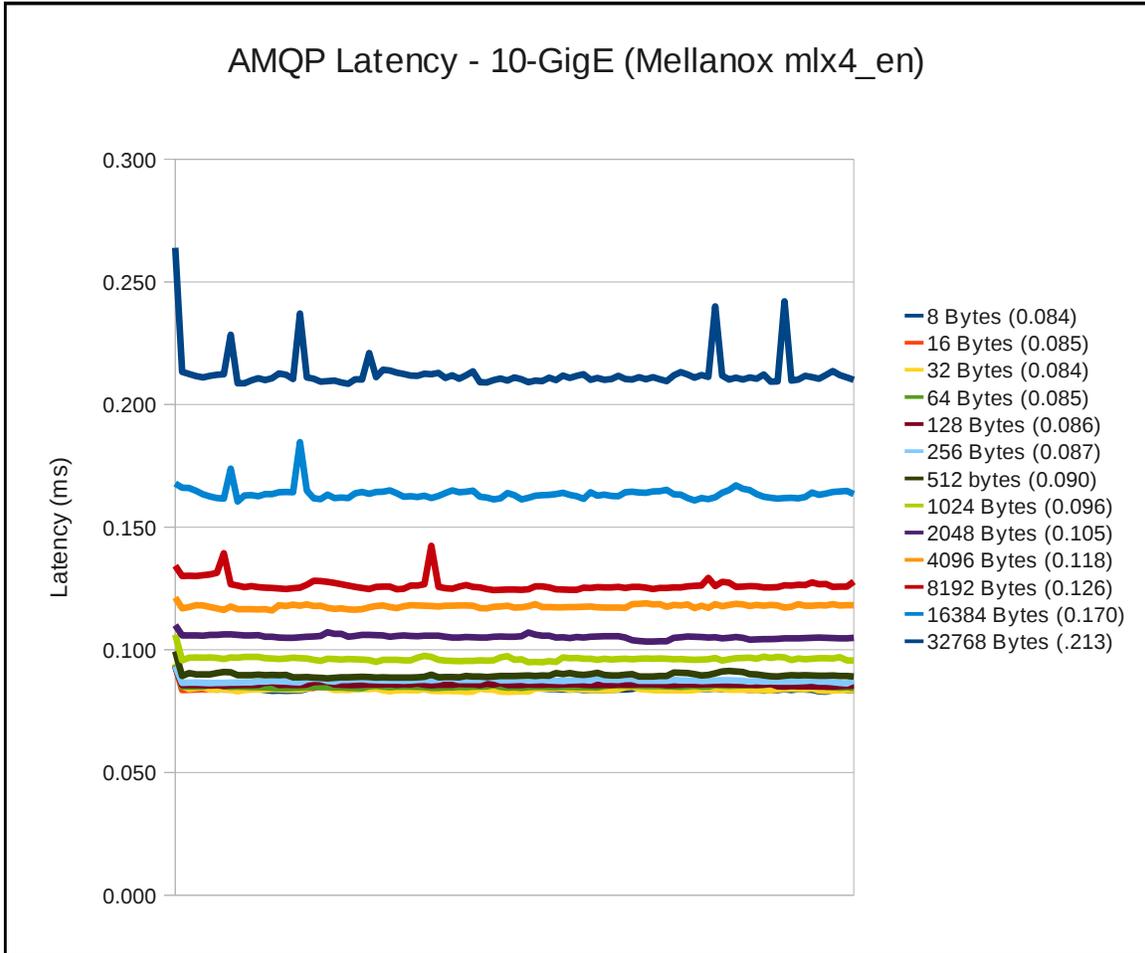


Figure 12



### 5.1.3 Internet Protocol over InfiniBand (IPoIB)

The IPoIB data is grouped more closely except for the larger transfer sizes (4K and higher).

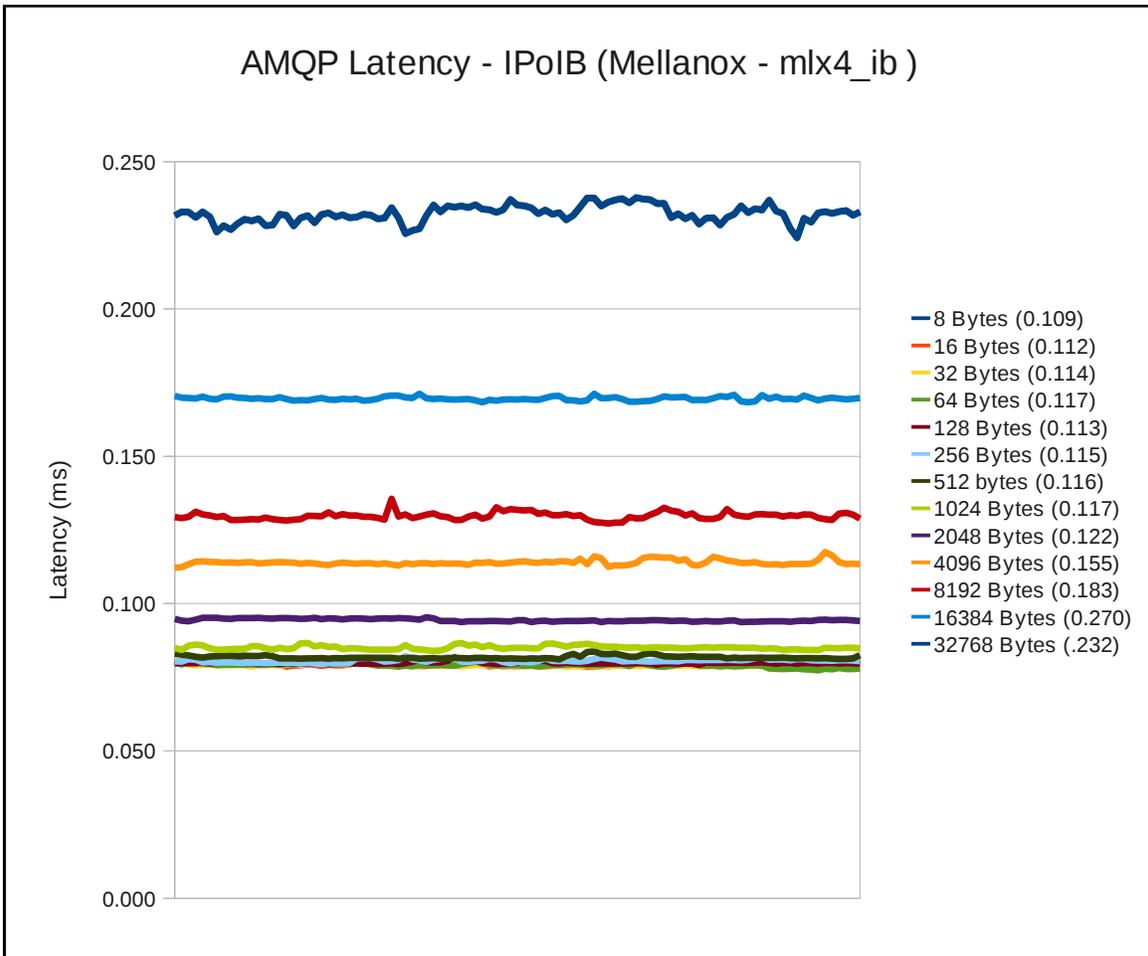


Figure 13



## 5.1.4 Remote Direct Memory Access (RDMA) with Mellanox 10Gig E

The Mellanox 10GigE with RoCE results are the second lowest with a 100 microsecond difference between the 8 Byte and 32K Byte results.

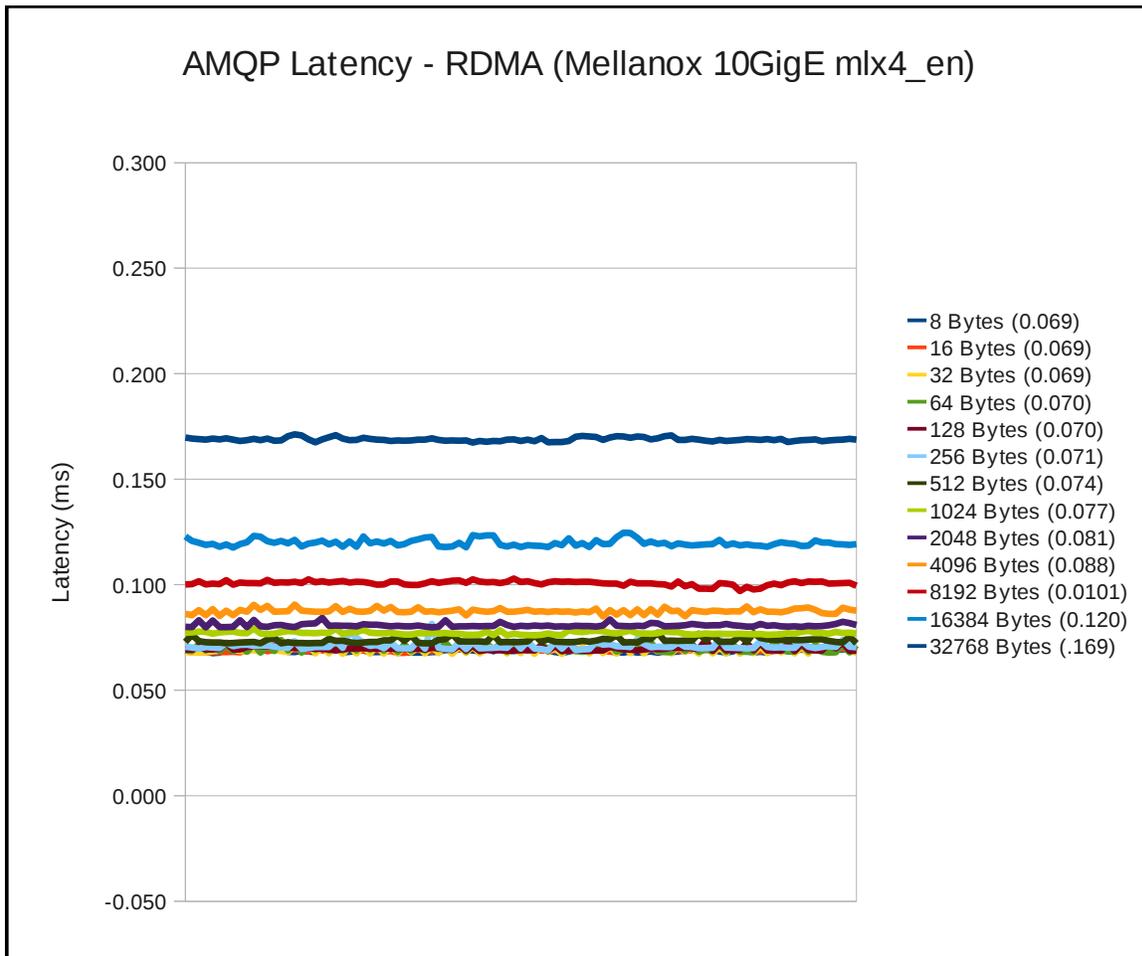


Figure 14



## 5.1.5 Remote Direct Memory Access (RDMA) with Mellanox Infiniband

The Mellanox Infiniband with RDMA results show the best latency performance with a 60 microsecond difference between 8 Byte and 32K Byte results.

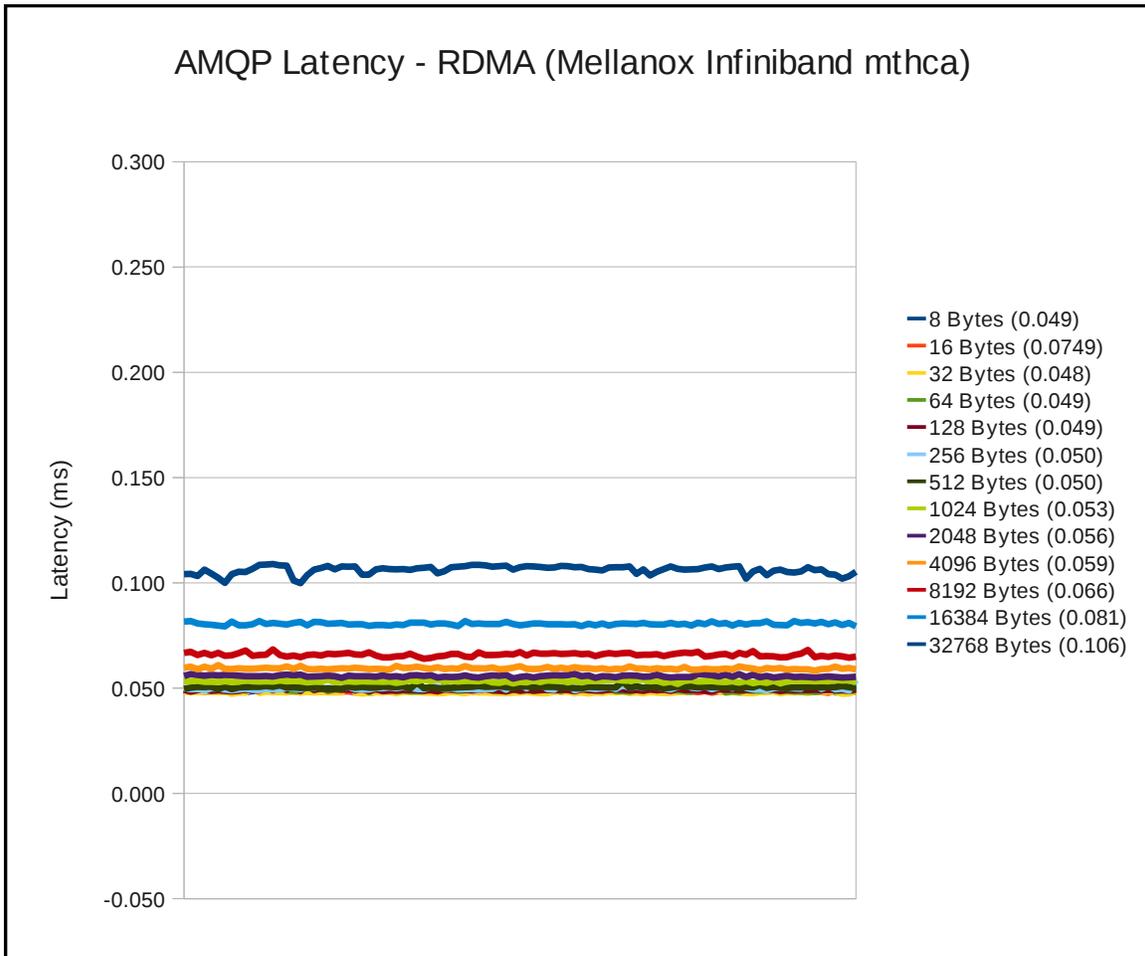
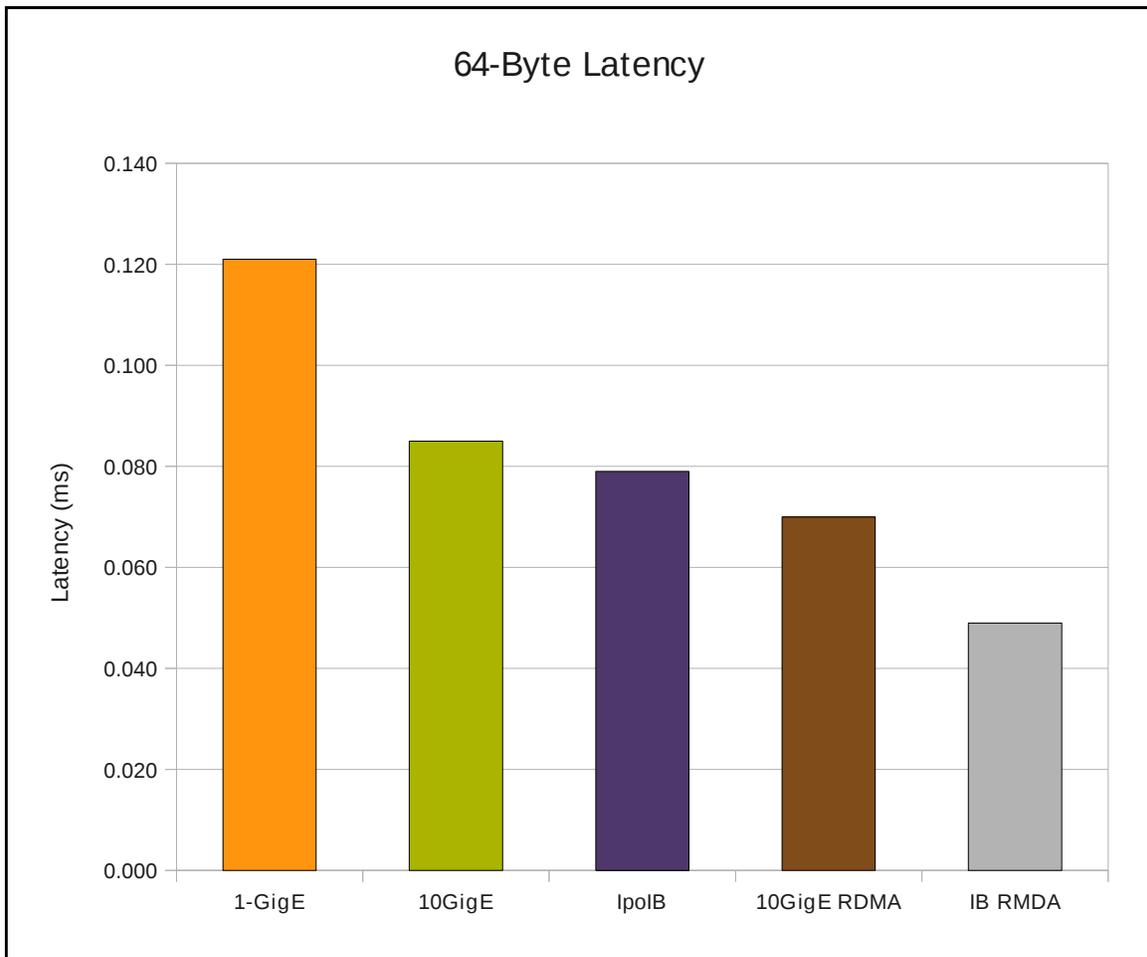


Figure 15

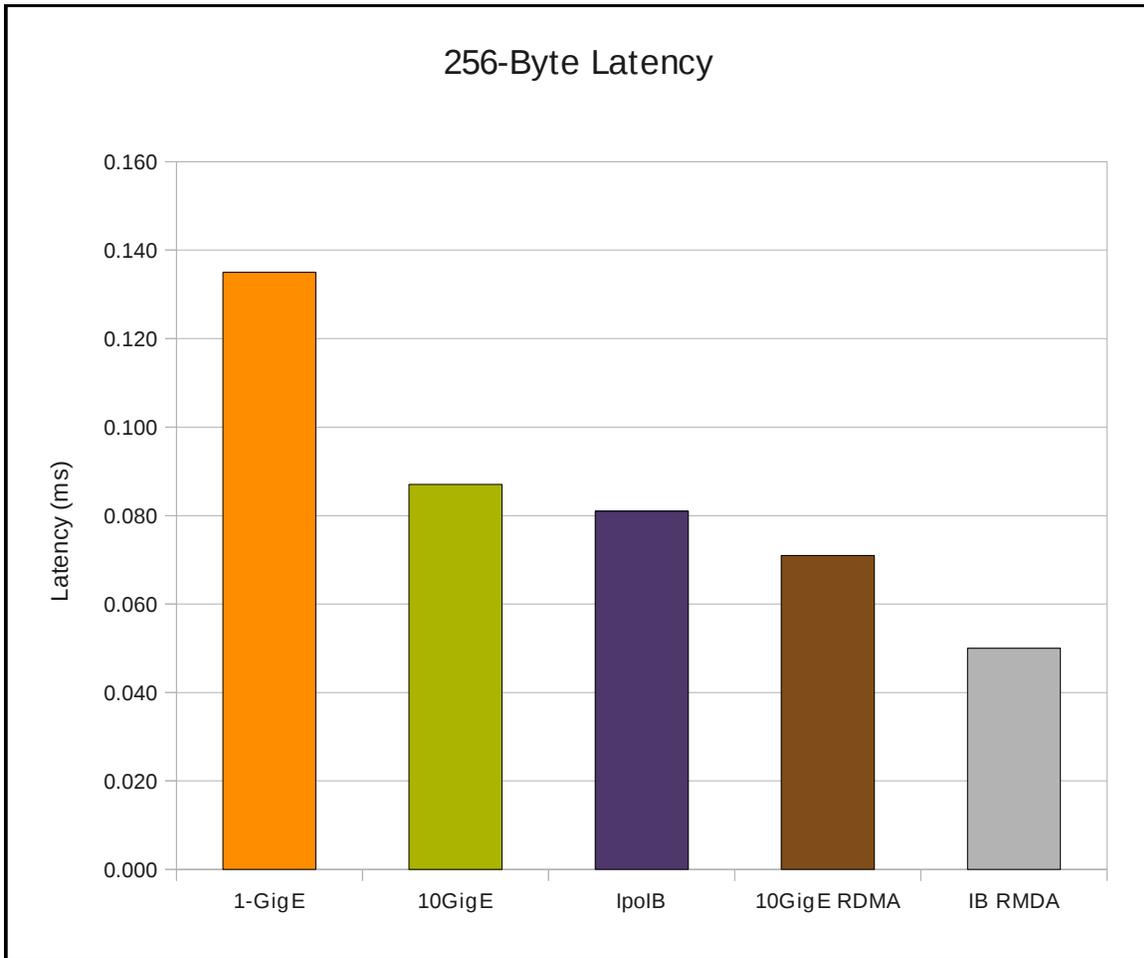


## 5.1.6 Comparisons

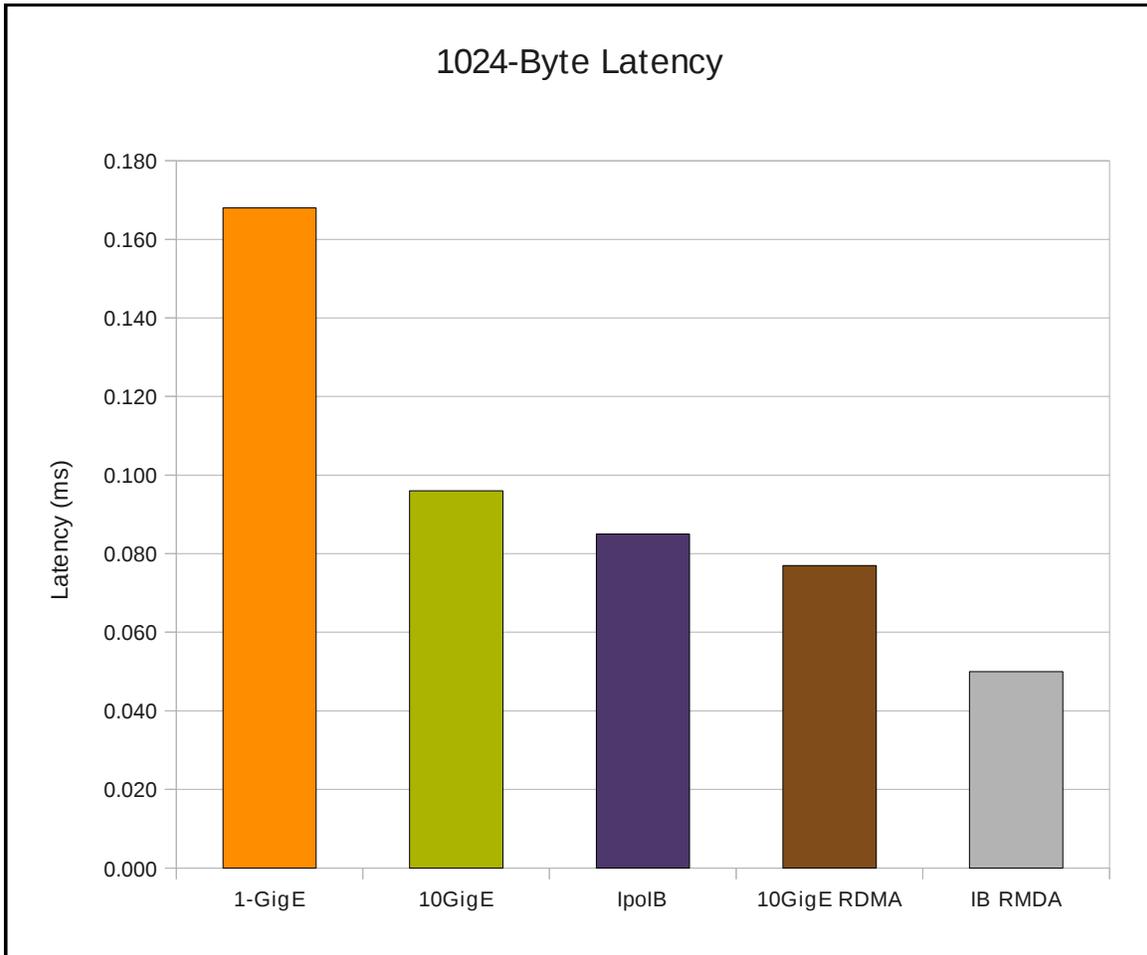
The averages for three commonly used message sizes are plotted for each of the interconnects/protocols. For each of the plots, 1-GigE has the highest latency. Considerably less, but consistently second, is 10 GigE followed by IPoIB. The 10GigE RDMA and IB RDMA both provide the lowest latency in all tests.



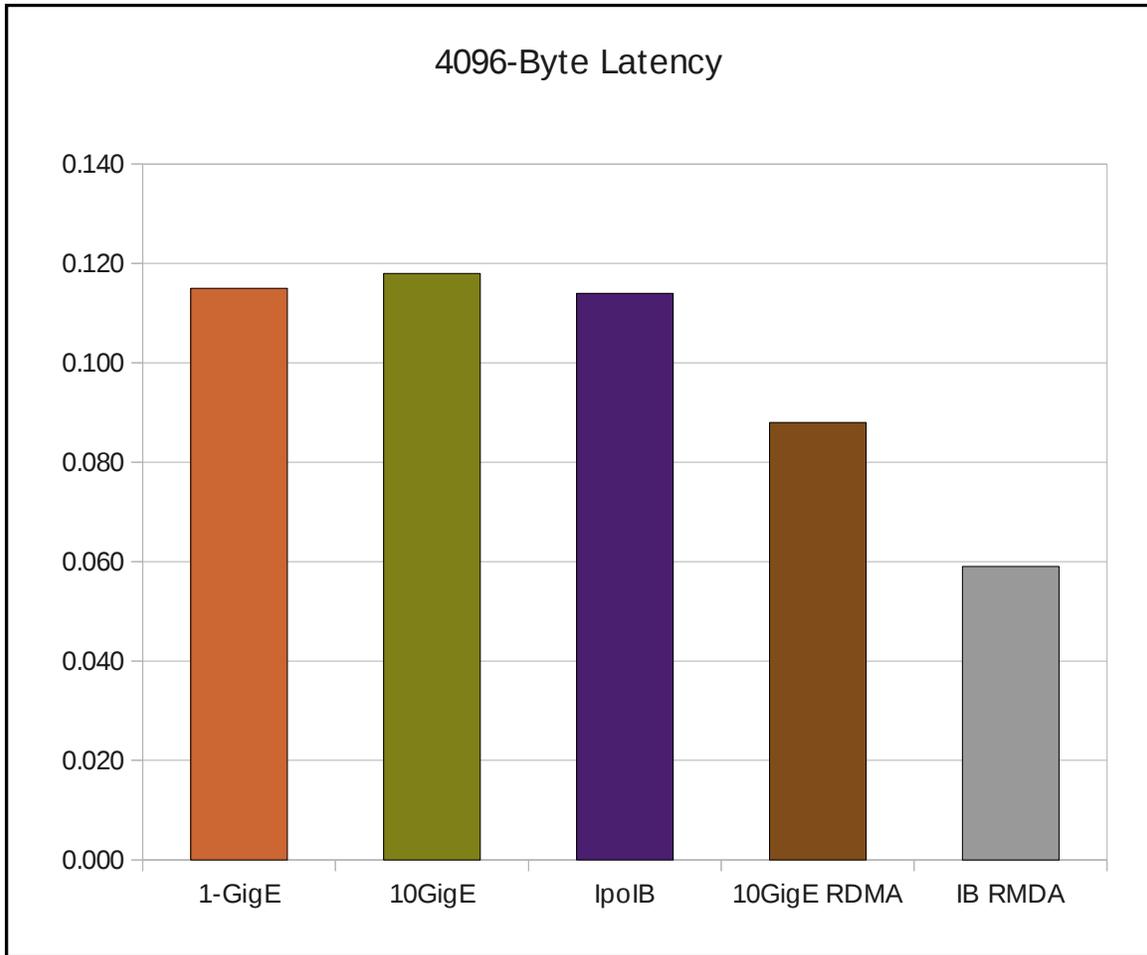
**Figure 16**



**Figure 17**



**Figure 18**



**Figure 19**



## 5.2 Throughput

The following bar graphs present the best throughput results for each transfer size for each of the interconnects/protocols. The blue bars represent the number of transfers per second and use the left side y-axis scale of 1000 transfers per second. The red bars represent the corresponding MB/s and use the right side y-axis scale. The scales are consistent for all interconnects/protocols.

In general, smaller transfer sizes were limited by the number of transfers the network stack could perform. However the larger results are limited by the line capacity of the interconnect.

Higher throughput may be achieved using multiple interfaces simultaneously.

### 5.2.1 1-GigE

The maximum transfers per second, 1,498,285.2, occur using the 8-byte transfer size. The maximum throughput was 211 MB/s.

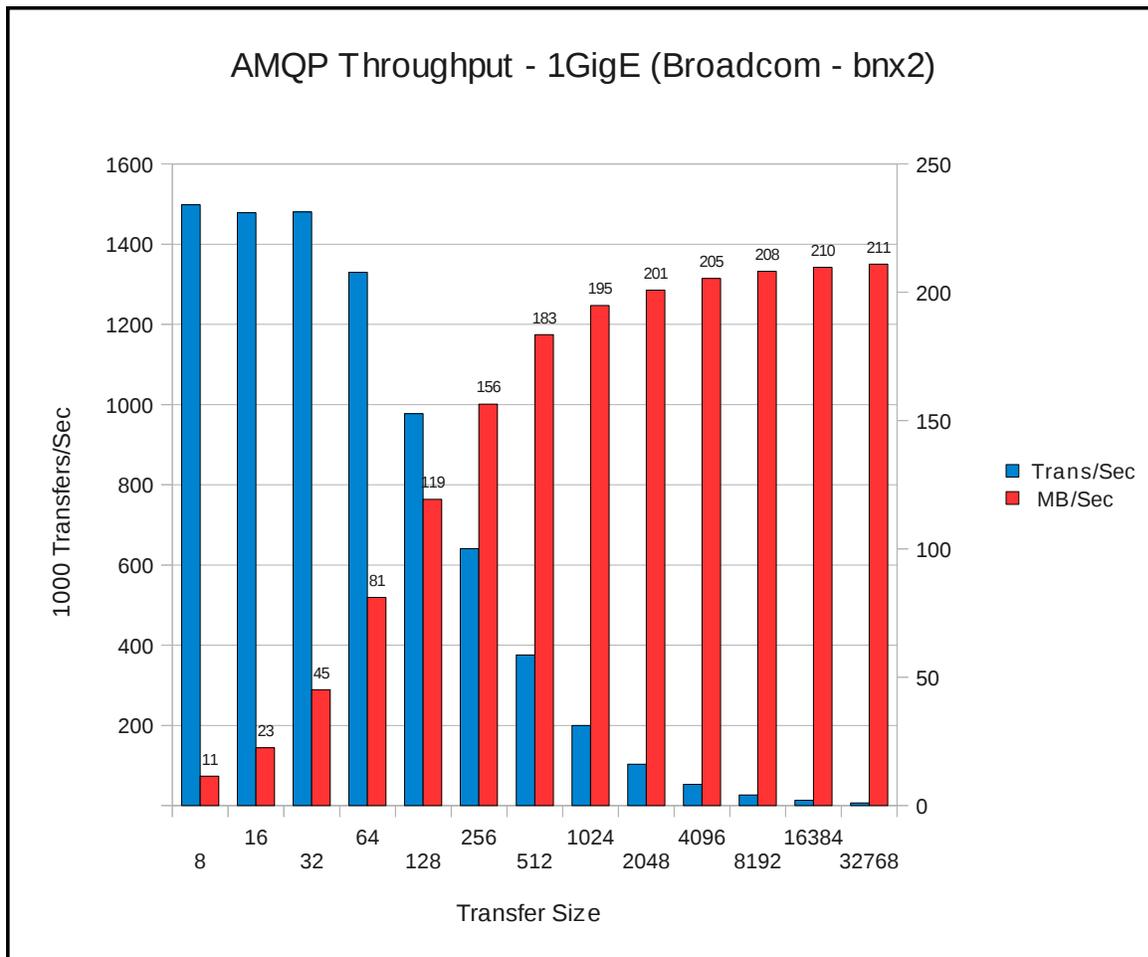


Figure 20



## 5.2.2 10-GigE

The higher line capacity of 10-GigE yields 1,505,432 8-byte transfers per second and peaks at 2036 MB/s.

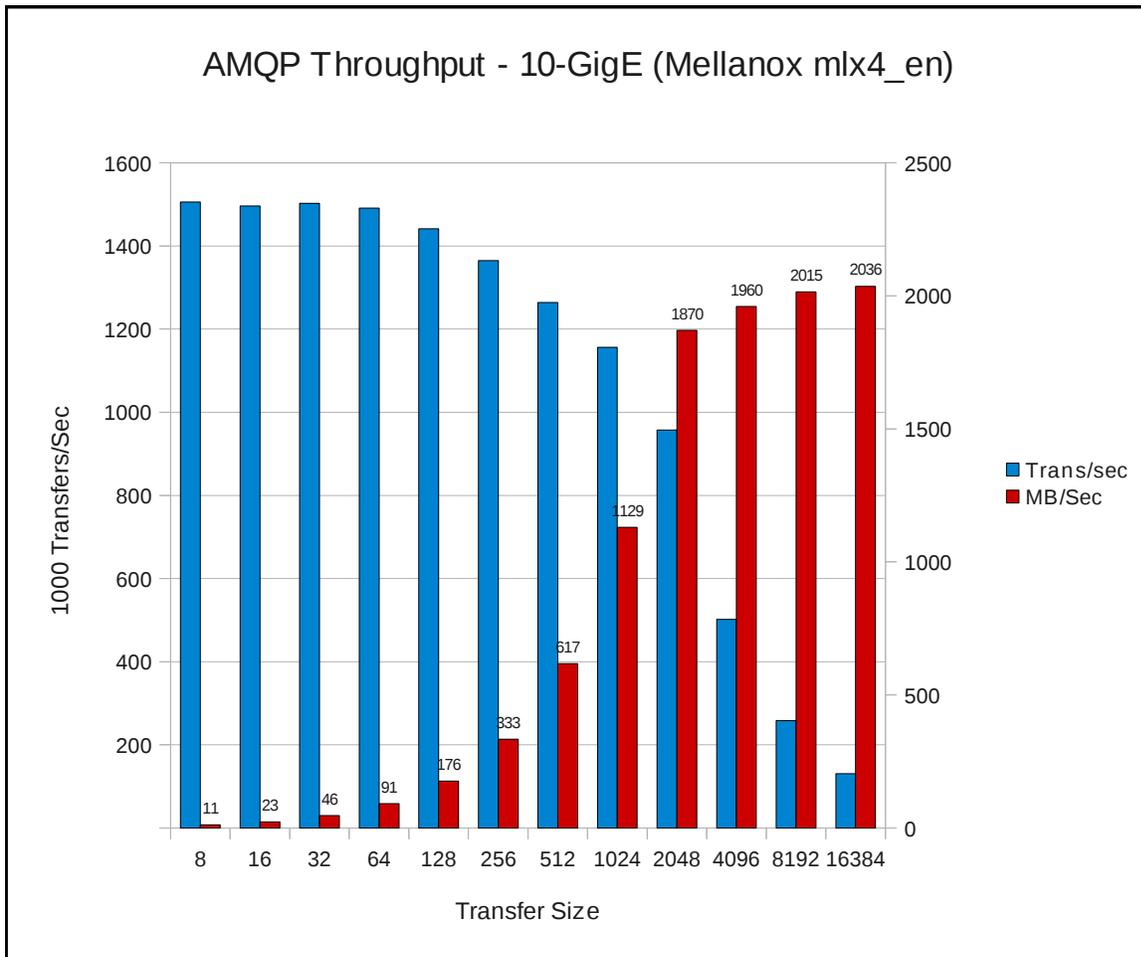


Figure 21



### 5.2.3 IpoIB

The 16-byte transfers proved to yield the highest transfer rate of 1,546,674. Throughput peaks at 1046 MB/s.

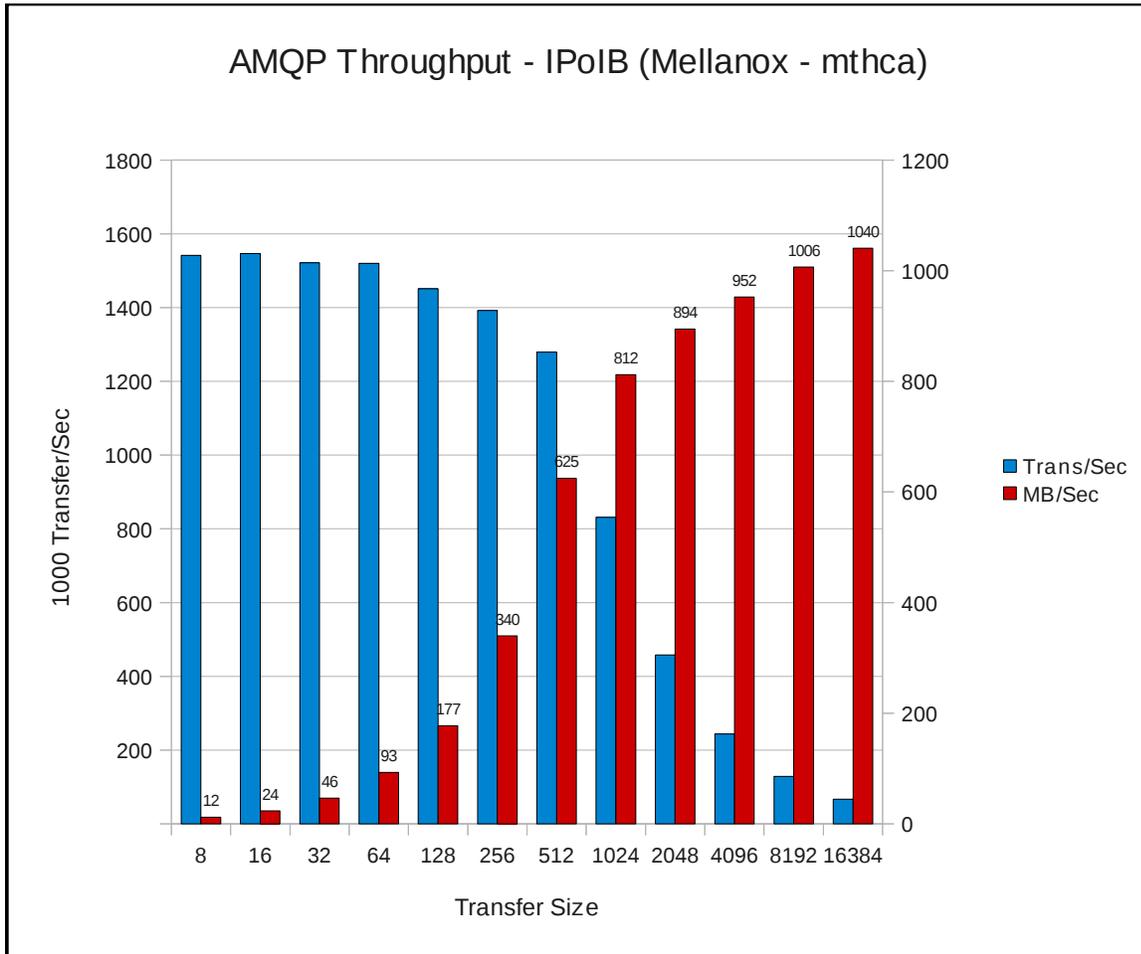
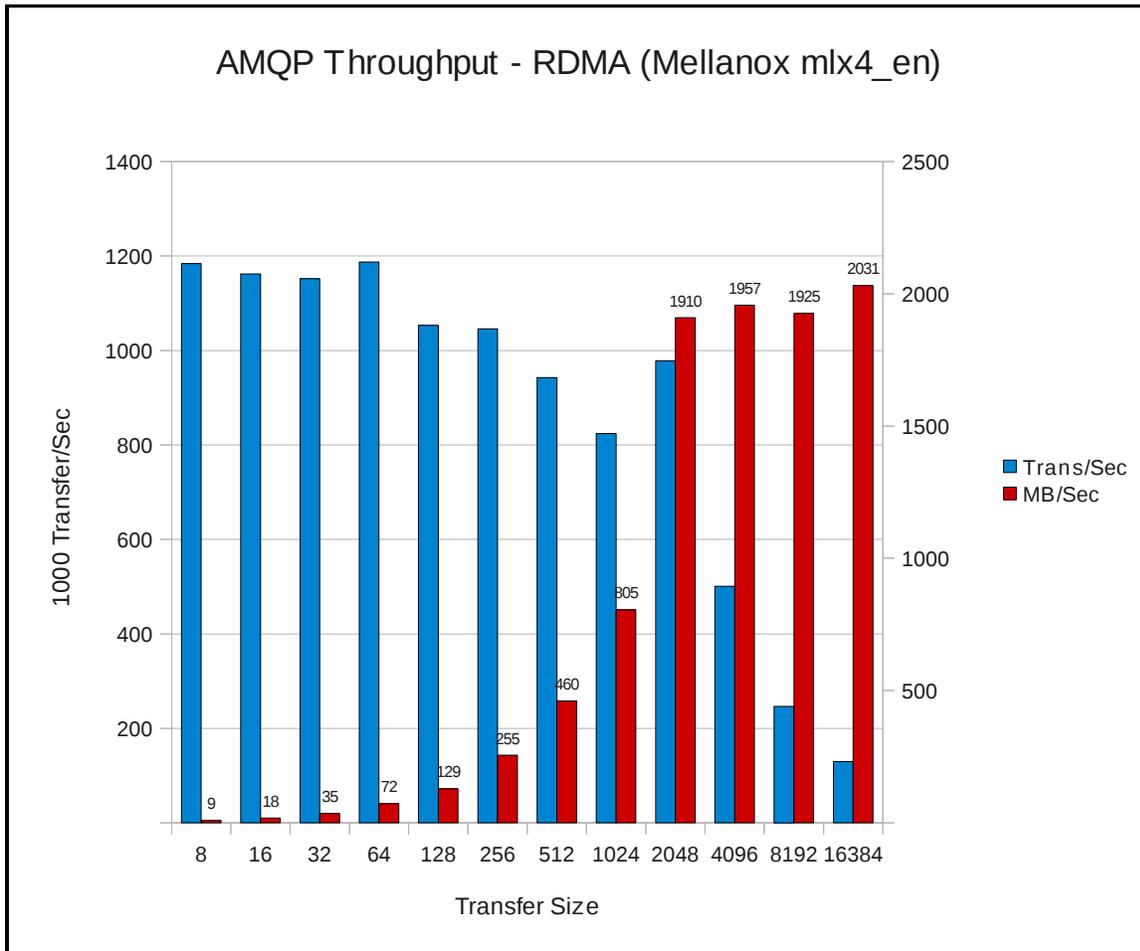


Figure 22



## 5.2.4 10-Gig-E RDMA

Using 10-GigE with RDMA improves the large transfer throughput rate producing the overall rate of 2031 MB/s. The most transfers were 1,191,374 for 64-bytes.

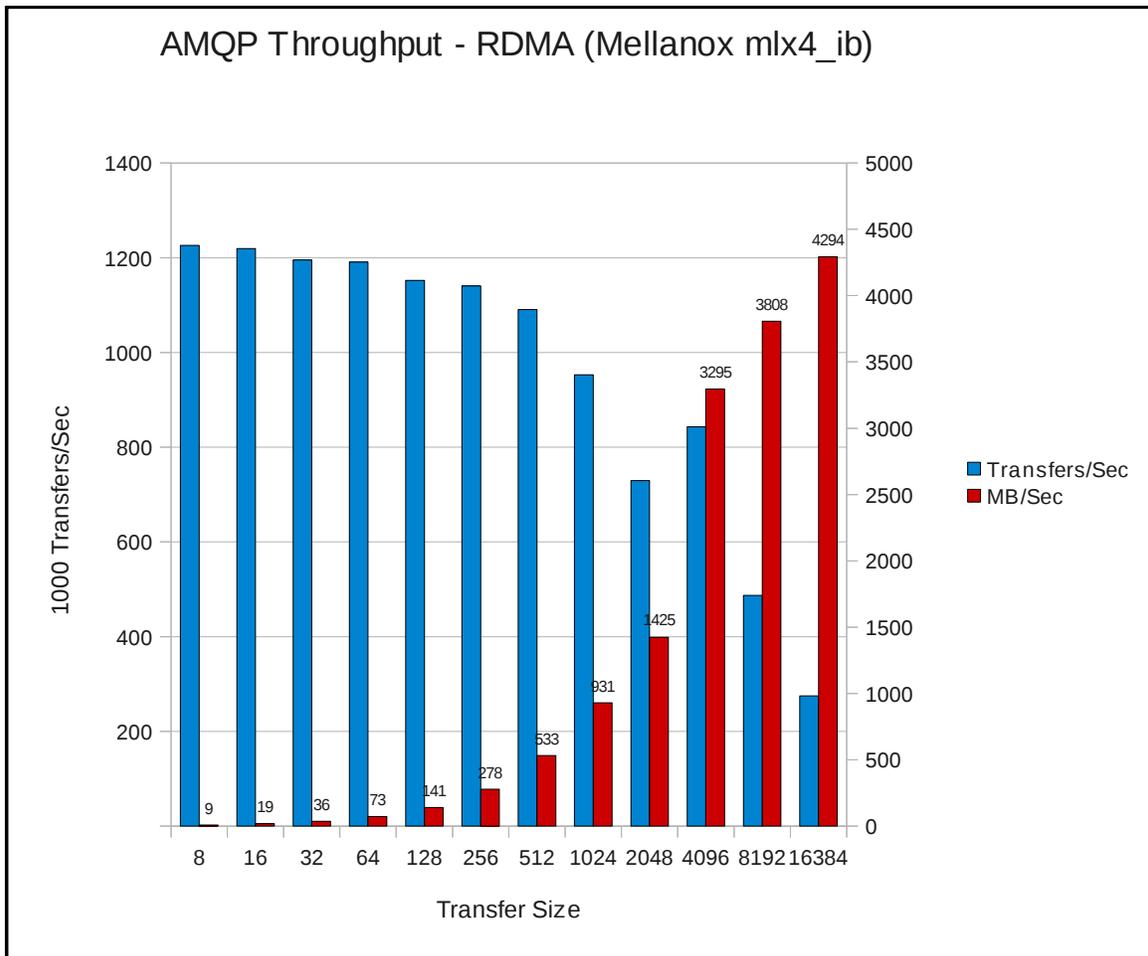


**Figure 23**



## 5.2.5 IB RDMA

For RDMA the 8-byte transfers/s slight beat out the 16-byte results at 1,226,407. The throughput tops out at 4294 MB/s.

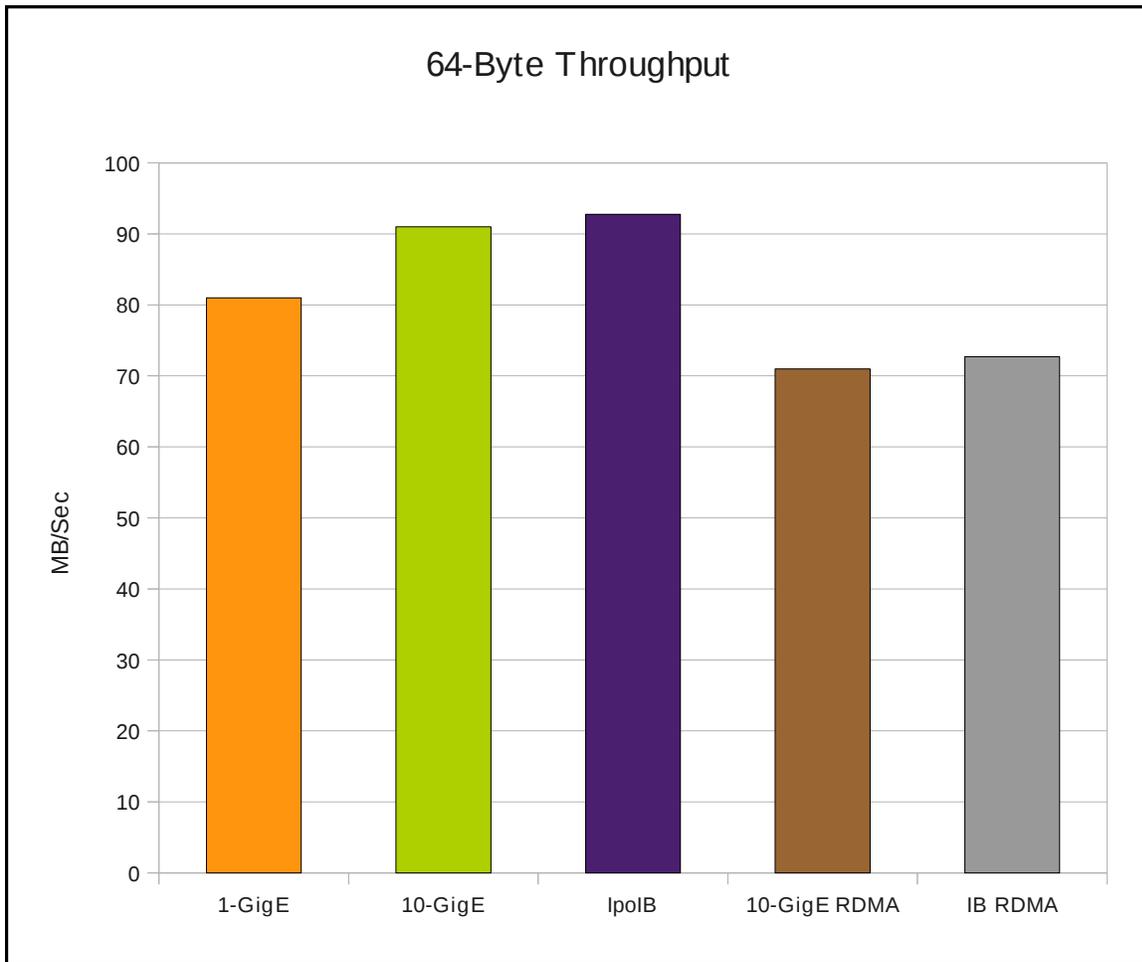


**Figure 24**



## 5.2.6 Comparisons

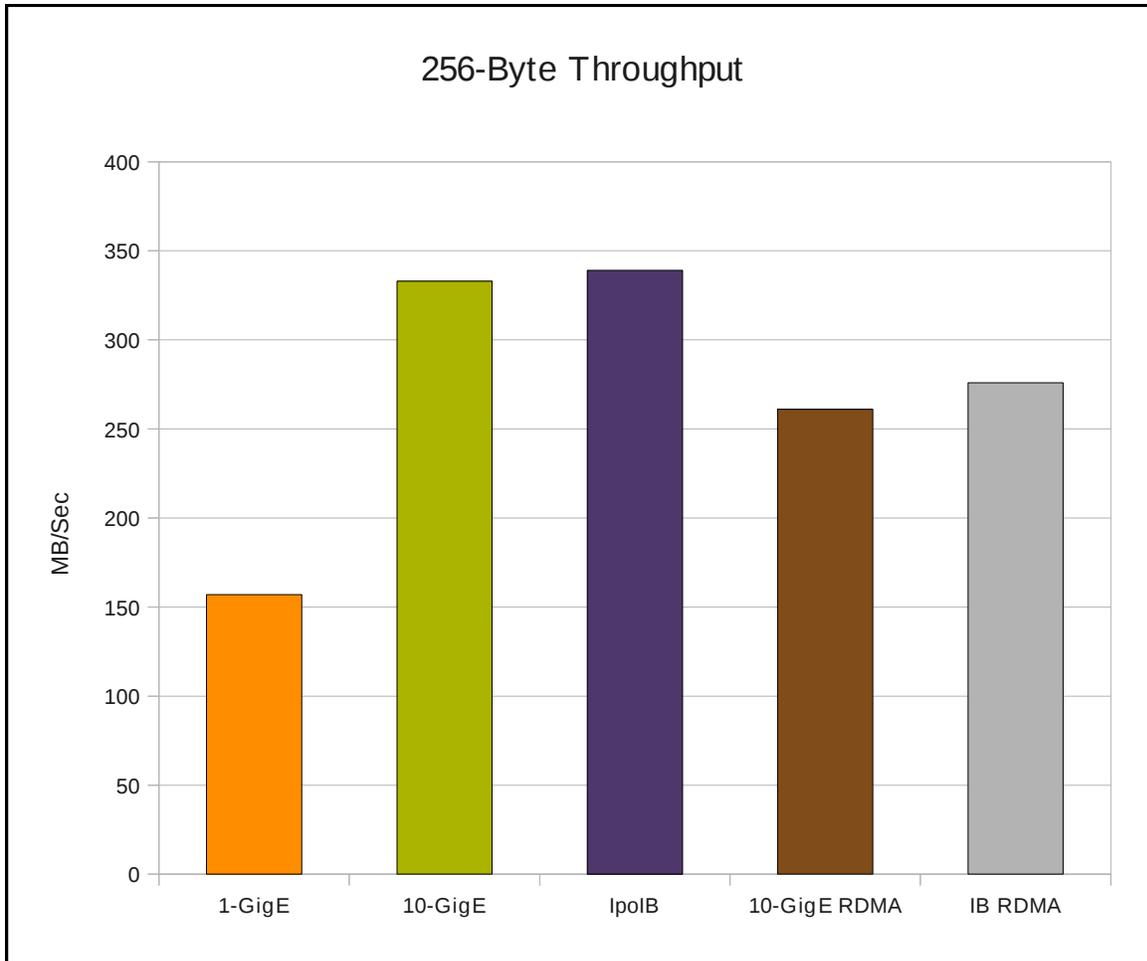
Charting the throughput of the same common transfer sizes for the various interconnects/protocols show the results are consistent with the latency results. For the 64-byte transfer, IpoIB performs best, followed by 10-GigE 1-GigE, IB RDMA, and 10-GigE RDMA.



**Figure 25**



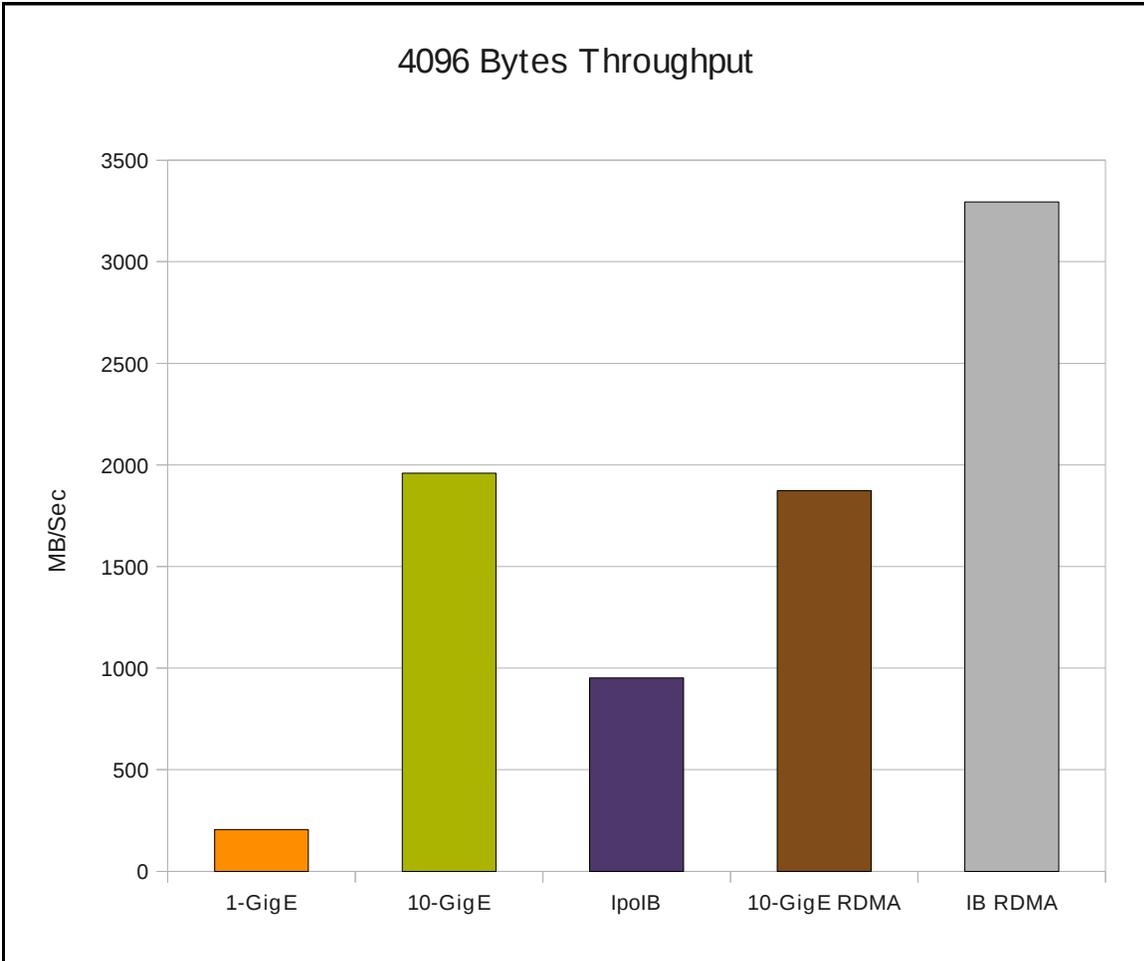
The order changes for 256-byte transfers. IPoIB edges out 10-GigE by less than 6 MB/s. IB-RDMA, 10-GigE RDMA, and 1-GigE finish out the order.



**Figure 26**



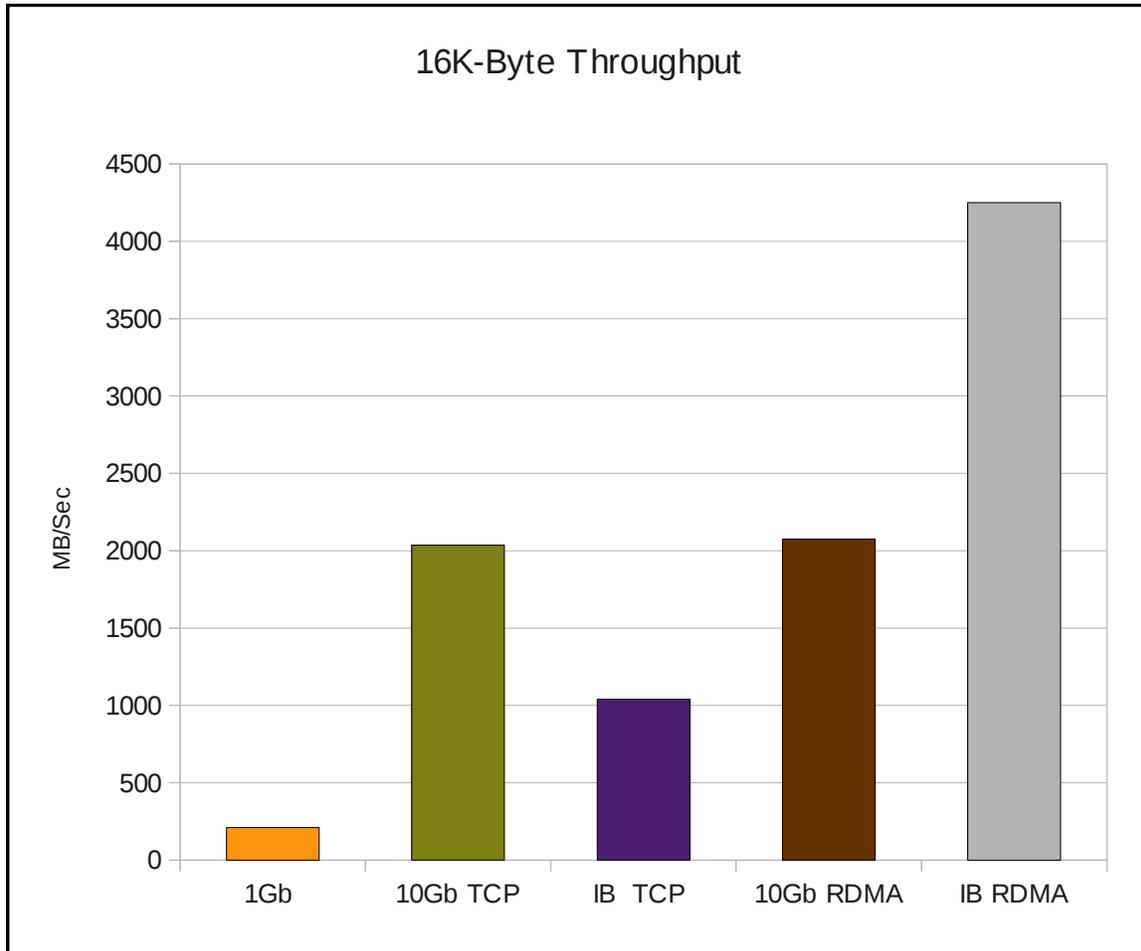
Using a 4K-byte transfer, the benefits of RDMA become evident. IB RDMA scores the highest followed by 10GigE, 10-GigE RDMA, IpoIB, and 1-GigE.



**Figure 27**



Highlighting the benefits of IB RDMA using a size of 16K, IB-RDMA is superior in bandwidth followed by 10-GigE RDMA, 10 Gig E, IBolP, and 1-Gig E.



**Figure 28**



### 5.3 System Metrics

This section presents various system metrics that were collected during 64-byte throughput tests for the various interconnects/protocols. The first is Memory Usage. IB RDMA and 10-GigE RDMA use a significant amount of memory compared to the other options. 10 GigE and IPoIB use the least.

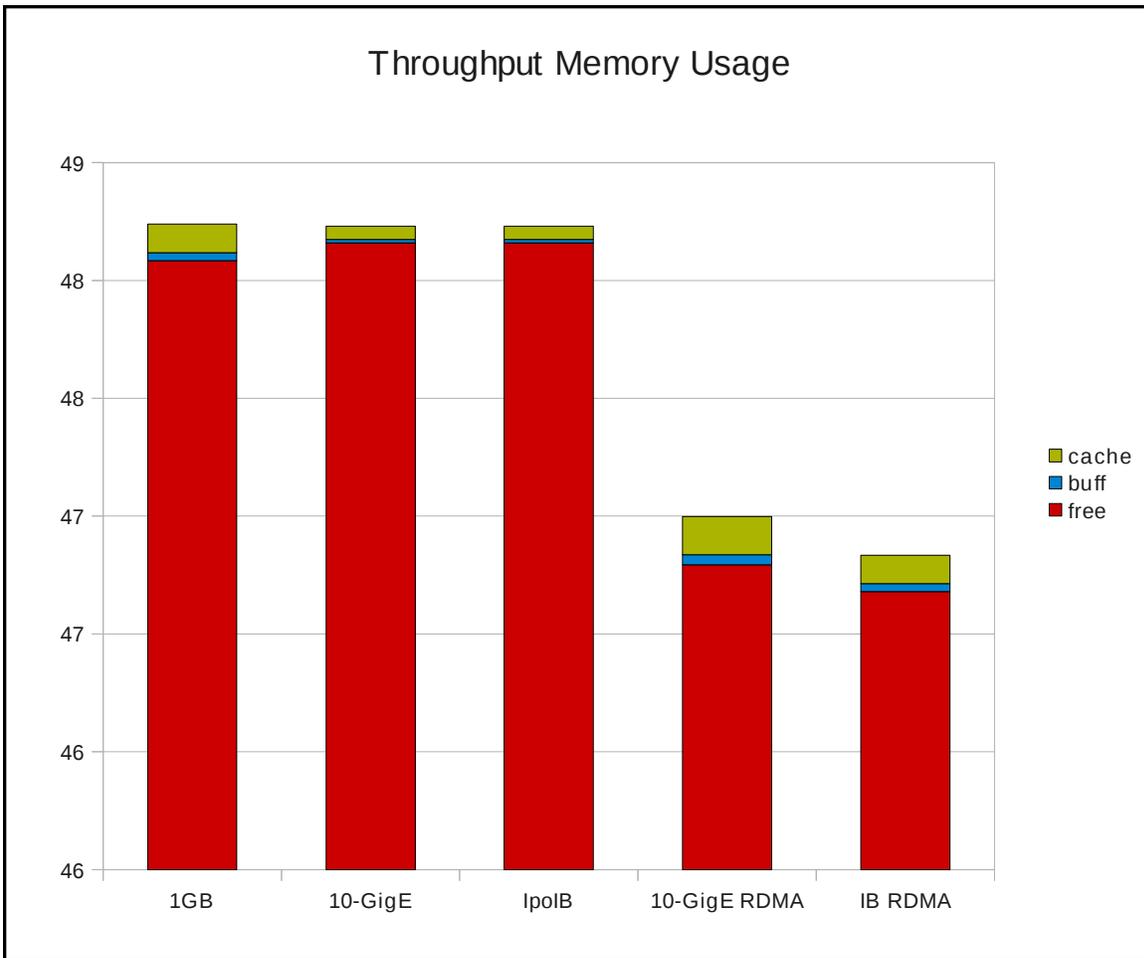
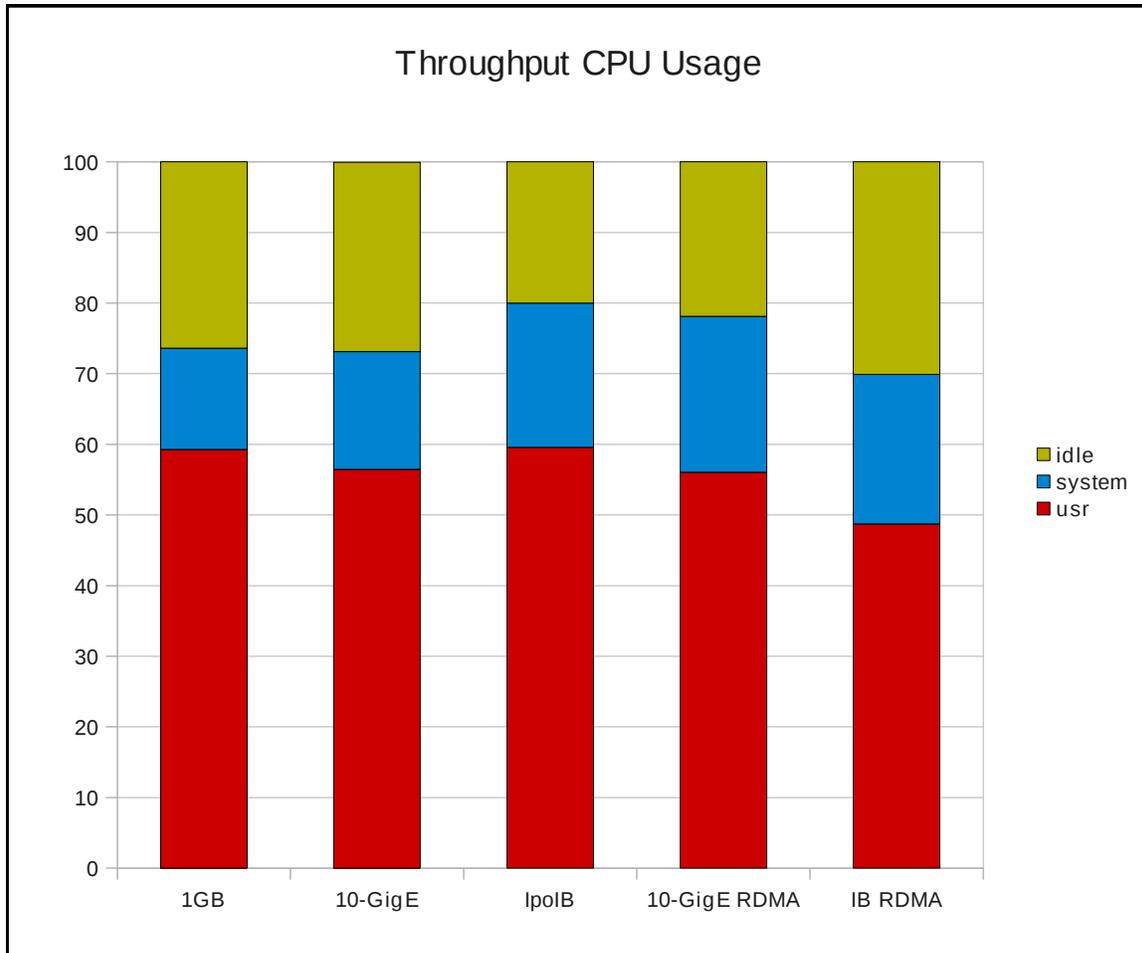


Figure 29



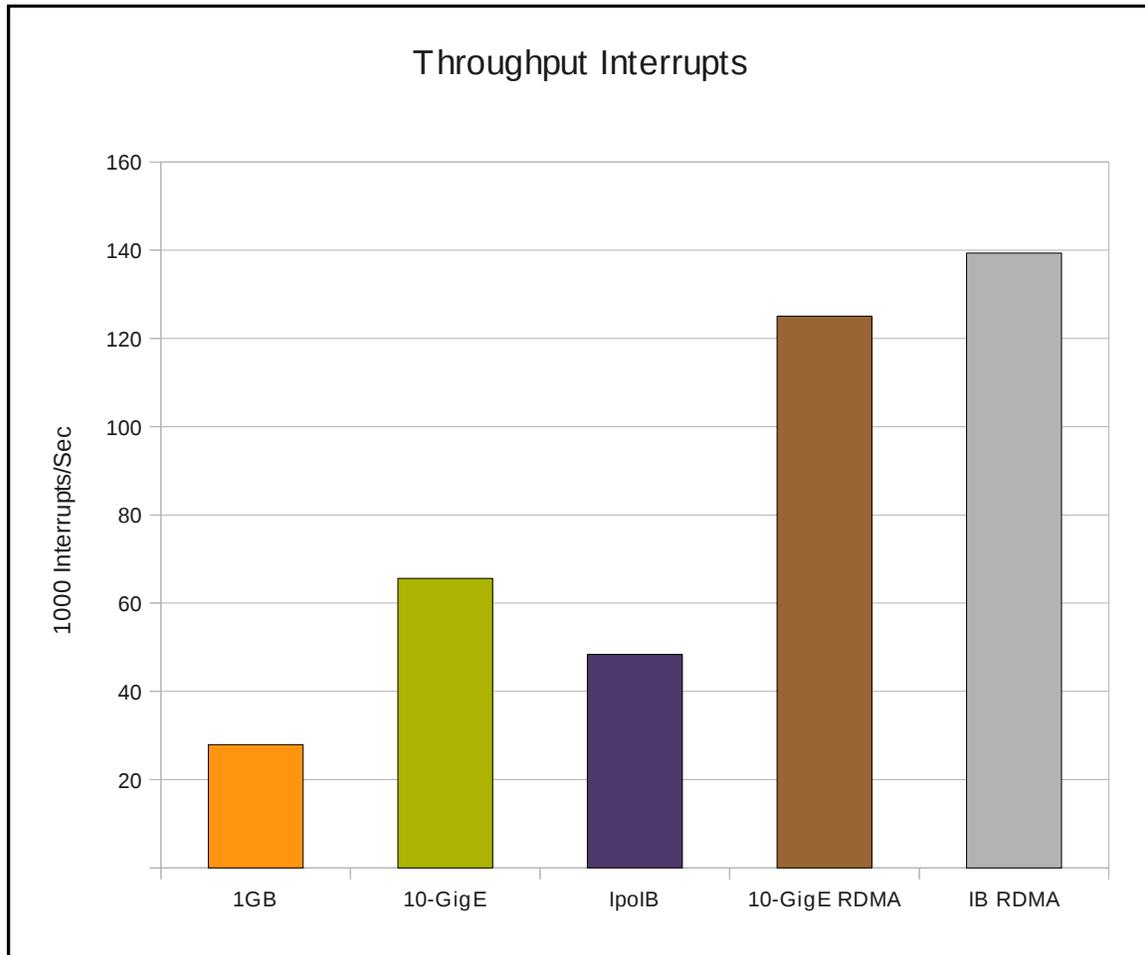
IB RDMA uses the least percentage of the available CPUs while 10-GigE uses the most followed by 10-GigE RDMA, IpoIB, and 1GB.



**Figure 30**



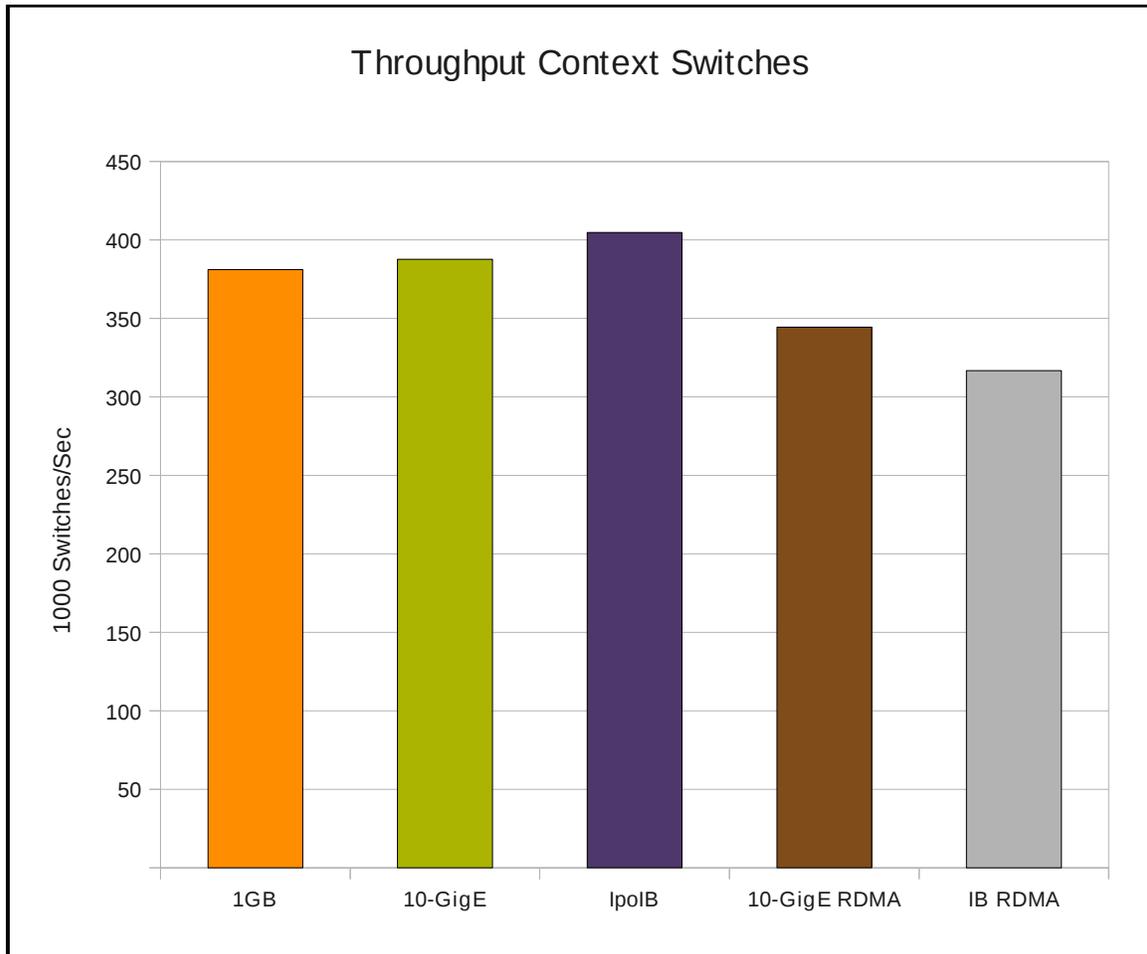
10Gig-E RDMA and IB RDMA processed a significantly larger amount of interrupts than the other protocols while 1GigE had the least.



**Figure 31**



The plotted rate of context switches illustrates that IB RDMA was significantly less than the others.



**Figure 32**



## 6 Conclusions

This study demonstrates how Red Hat Enterprise Linux 6.1, using Mellanox QDR IB with RDMA and using Red Hat's High Performance Network Add-on for Mellanox 10Gbit-E, running MRG had significant improvement in latency using RoCE compared to 1Gig-E network cards. Messaging provides a high throughput, low latency messaging infrastructure that can meet the needs of high-performance computing, SOA deployments, and platform services customers.

Furthermore, the results can be used to assist in determining configurations to meet user needs. The results show that IB and 10GigE provide overall lower-latency and higher throughput than the other interconnects studied. 1-GigE is a commodity interconnect and could be used for a respectable deployment especially for smaller transfer sizes or with multiple adapters for larger transfers.

## Appendix A: Memlock configuration

The default values for memlock limits were too small and generated error messages related to memory. One method of increasing the limit is by adding entries to `/etc/security/limits.conf` similar to the following.

```
* hard memlock unlimited
* soft memlock 262144
```