Red Hat Enterprise MRG

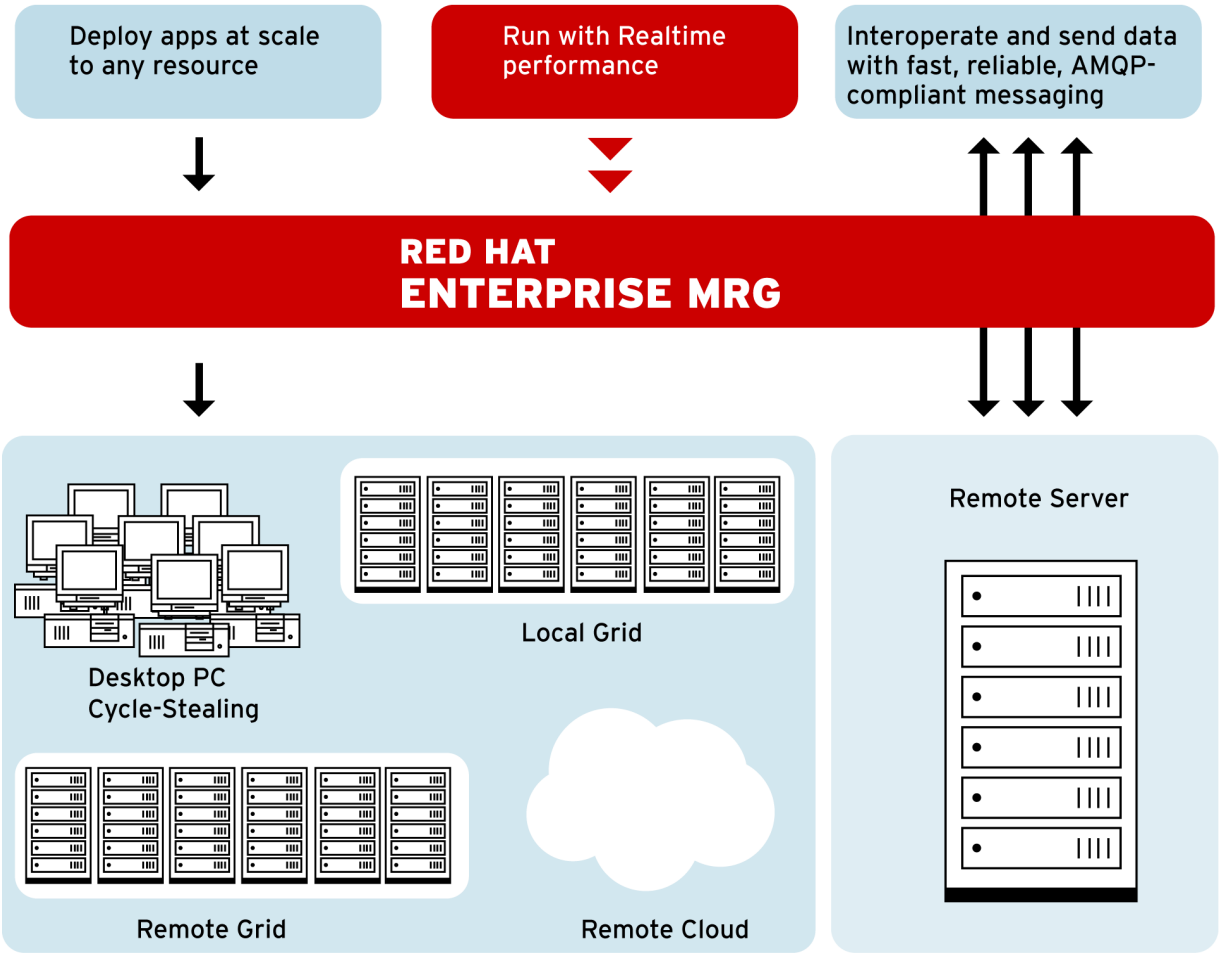
Messaging, Realtime, Grid

Bryan Che
bche@redhat.com

Last Updated 2/2010
MRG v1.2

About Red Hat Enterprise MRG

- Integrated platform for high performance distributed computing
  - High speed, interoperable, open standard **Messaging**
  - Deterministic, low-latency **Realtime** kernel
  - High performance & throughput computing **Grid** scheduler for distributed workloads and Cloud computing



MRG Messaging

- Enterprise Messaging System that
  - Implements AMQP (Advanced Message Queuing Protocol), the first open messaging standard
    - Participation from Red Hat, JPMC, Goldman, Credit Suisse, Deutsche Borse, Barclays, Bank of America, Microsoft, Cisco, etc
  - Spans all use cases in one implementation to consolidate architectural silos (fast messaging, reliable messaging, large file transfer, publish/subscribe, eventing, etc)
  - Uses Linux-specific optimizations to achieve breakthrough performance on Red Hat Enterprise Linux and MRG Realtime
  - Runs on non-Linux platforms without the full performance and quality of service benefits that Red Hat Enterprise Linux provides
- Provides open, high performance system for everything from financial exchanges to infrastructure management

MRG Messaging Feature Highlights

- Core Messaging
  - P2P, fanout, pub-sub, async
  - Reliable messaging
  - Transactions -local to dtx
  - Multiple clients (C++, .NET/WCF, Java, JMS, Python, etc)
- High Performance
  - C++ broker, optimized for RHEL
  - O-direct AIO for high-speed durable messaging over 500k messages/second per LUN
  - Infiniband RDMA support for ultra low latency messaging
- Management tools
  - Command line tools to Web-based GUI
  - AMQP-based framework & APIs
- Advanced Features
  - Queue Semantics: Ring Queue, Last Value Queue, TTL, Initial Value Exchange, etc
  - Routing patterns, including XML XQuery
  - Federation with dynamic routes
- High Availability
  - Active-Standby/Active-Active Broker Clustering
  - Federated disaster recovery
- Security
  - SASL auth
  - SSL encryption
  - role-based access control

What is AMQP?

An Open Standard for Middleware:

- Middleware: software that connects other software together. Middleware connects islands of automation, both within an enterprise and out to external systems.

Why it is different:

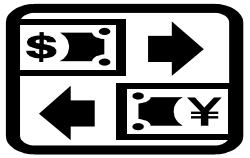
- A straight-forward and complete solution for business messaging
- Cost effective for pervasive deployment
- Totally open (developed in partnerships)
- Created by users and technologists(Messaging, OS, and Network) working together
- Made to satisfy real needs (needs to also provide things like IVQ, LVQ, Replay, ...)

AMQP = Internet Protocol for Business Messaging

The AMQP Model

The AMQP Architecture specifies modular components and rules as the building blocks

Exchanges



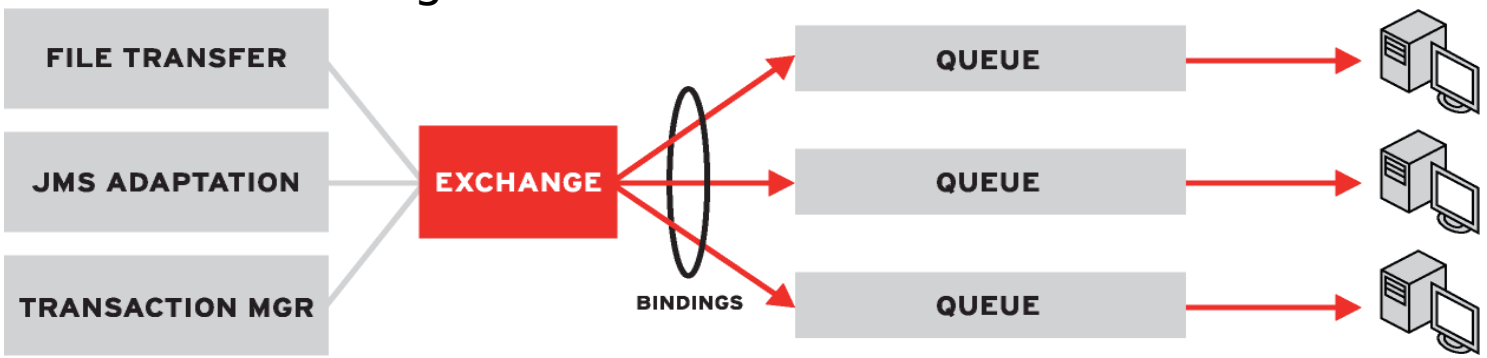
- The “Exchange” receives messages from publisher applications and routes these to queues, based on arbitrary criteria—typically topic & message headers

Queues

- The “Queue” stores messages until they can be safely processed by a consumer application (or multiple applications)

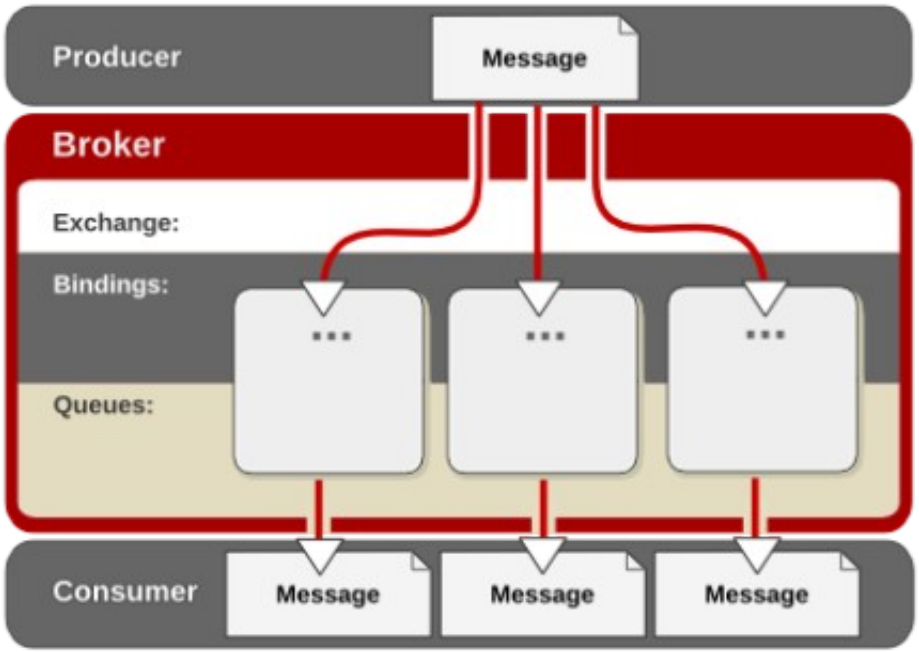
Bindings

- The “Binding” defines the relationship between a queue and an exchange and provides the message routing criteria

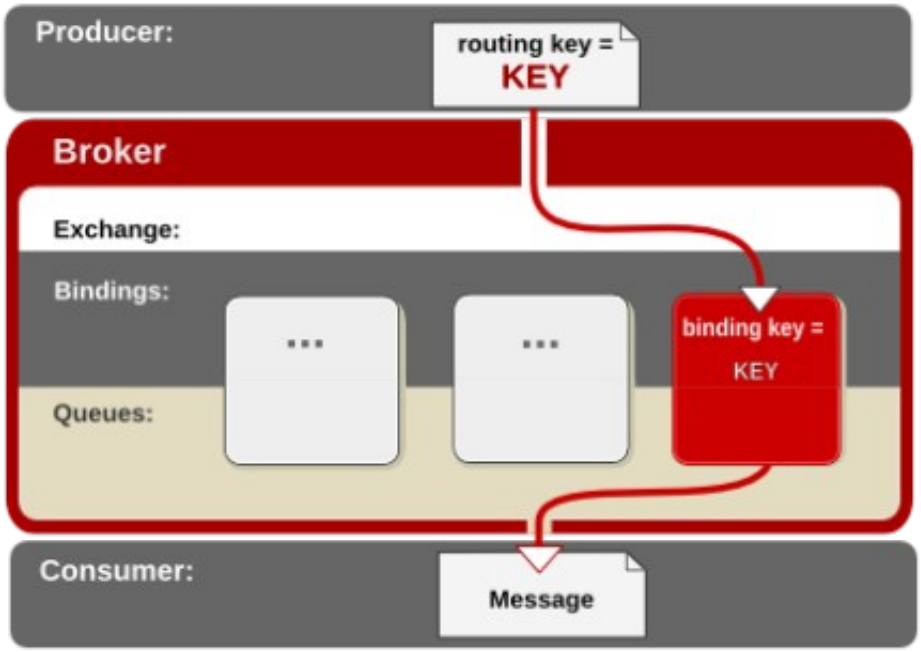


Sample AMQP Exchanges

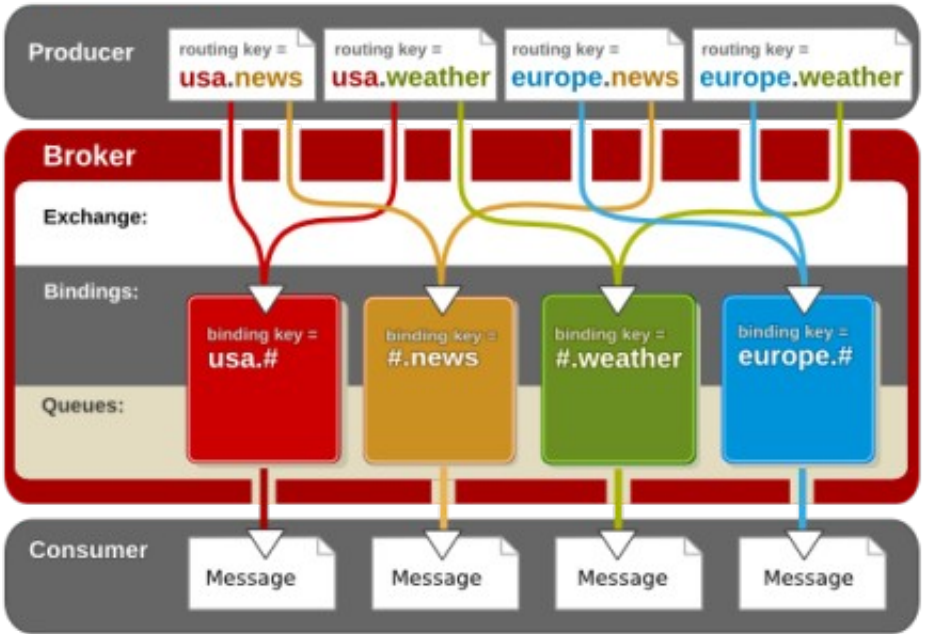
Fanout Exchange



Direct Exchange



Topic Exchange



A Simple Example

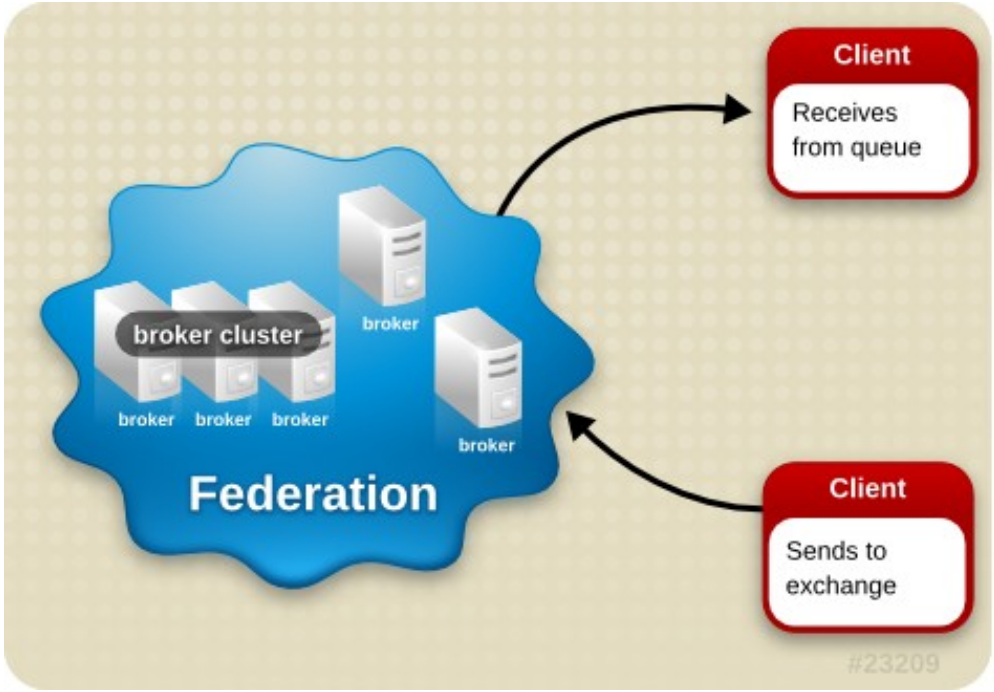
```
#!/usr/bin/env python
from qpid.connection import Connection
from qpid.util import connect
from qpid.datatypes import uuid4, Message
# connect to the server and start a session
conn = Connection(connect("127.0.0.1", 5672))
conn.start()
ssn = conn.session(str(uuid4()))
# create a queue
 ssn.queue_declare("test-queue")
# publish a message
dp = ssn.delivery_properties(routing_key="test-queue")
mp = ssn.message_properties(content_type="text/plain")
msg = Message(dp, "Hello World!")
ssn.message_transfer(message=msg)
# subscribe to a queue
ssn.message_subscribe(queue="test-queue",
destination="messages")
incoming = ssn.incoming("messages")
# start incoming message flow
incoming.start()
# grab a message from the queue
print incoming.get(timeout=10)
# cancel the subscription and close the session and
# connection
ssn.message_cancel(destination="messages")
ssn.close()
conn.close()
```

MRG Queue Semantics

- MRG Messaging provides queue semantics so that many capabilities people previously had to build on top of messaging software are now included directly in MRG
- **Ring Queue**: Queue with a configurable depth and ring buffer that will override the oldest messages as the buffer fills
- **Last Value Queue**: Queue that will update in-place messages that have not been consumed and have stale data with newer messages that have updated data
  - For example, useful for stock ticker symbols, when you just want the latest value
- **Initial Value**: Cache the last message in an exchange so that a late-binding queue or client can get an initial value even if the queue is empty
- **Time To Live (TTL)**: Set a configurable Time To Live so that late-joining clients can get a replay window
- **Global Sequencing**: The broker sequences messages to provide a global ID

Clustering and Federation

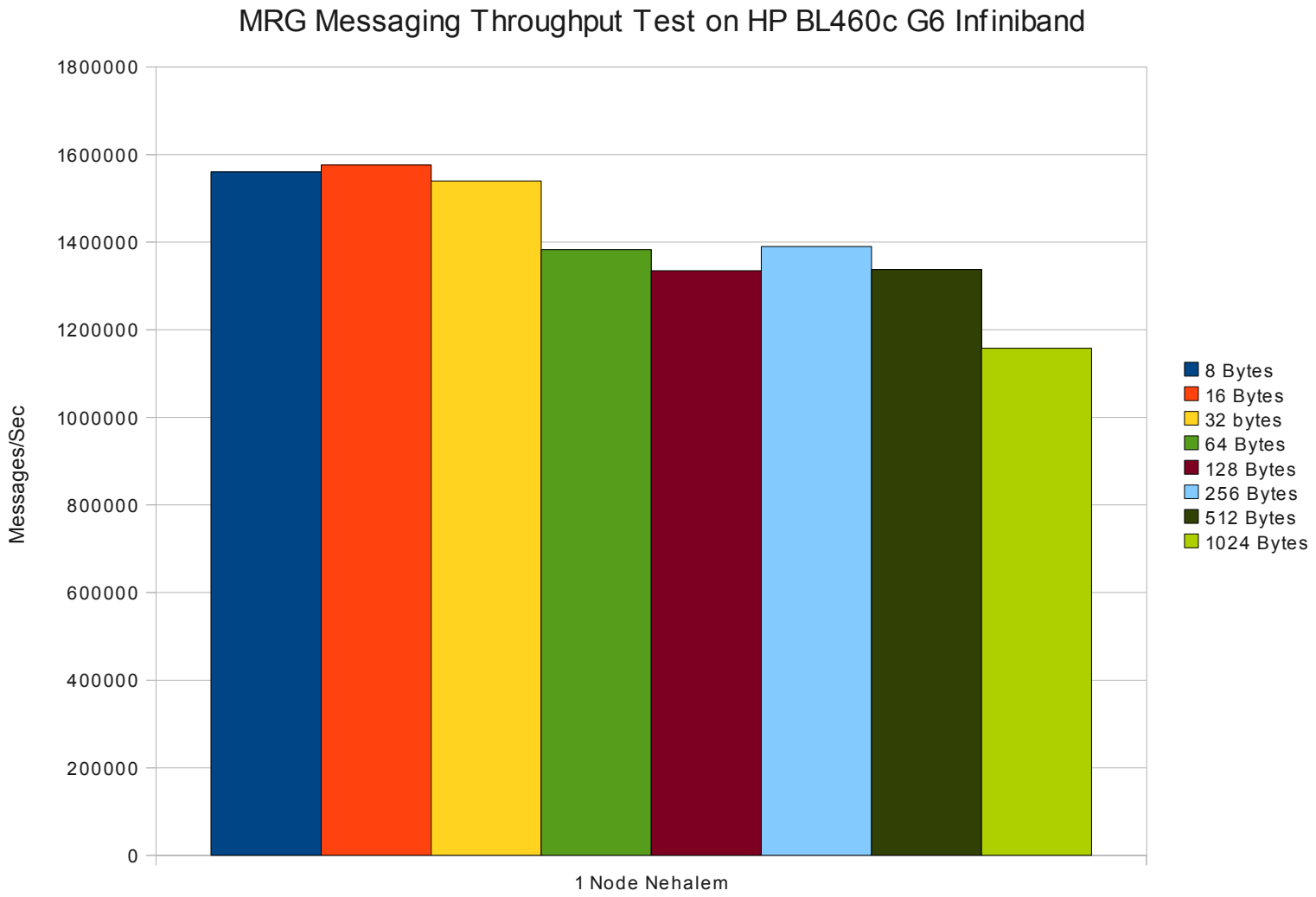
- Active/Active Clustering provides scalability and enhanced load-balancing
  - Producers and consumers can be connected to any broker in the cluster
  - based on RHEL5 OpenAIS technology
- Federation provides geographical distribution of brokers and Disaster Recovery
  - configured via *links* and *routes*
  - link: connection between two brokers that allows messages to be passed between them
  - route: path that messages take from one broker to another; can run along one or more links to the final destination. Routes can be dynamic or static



MRG Messaging Security

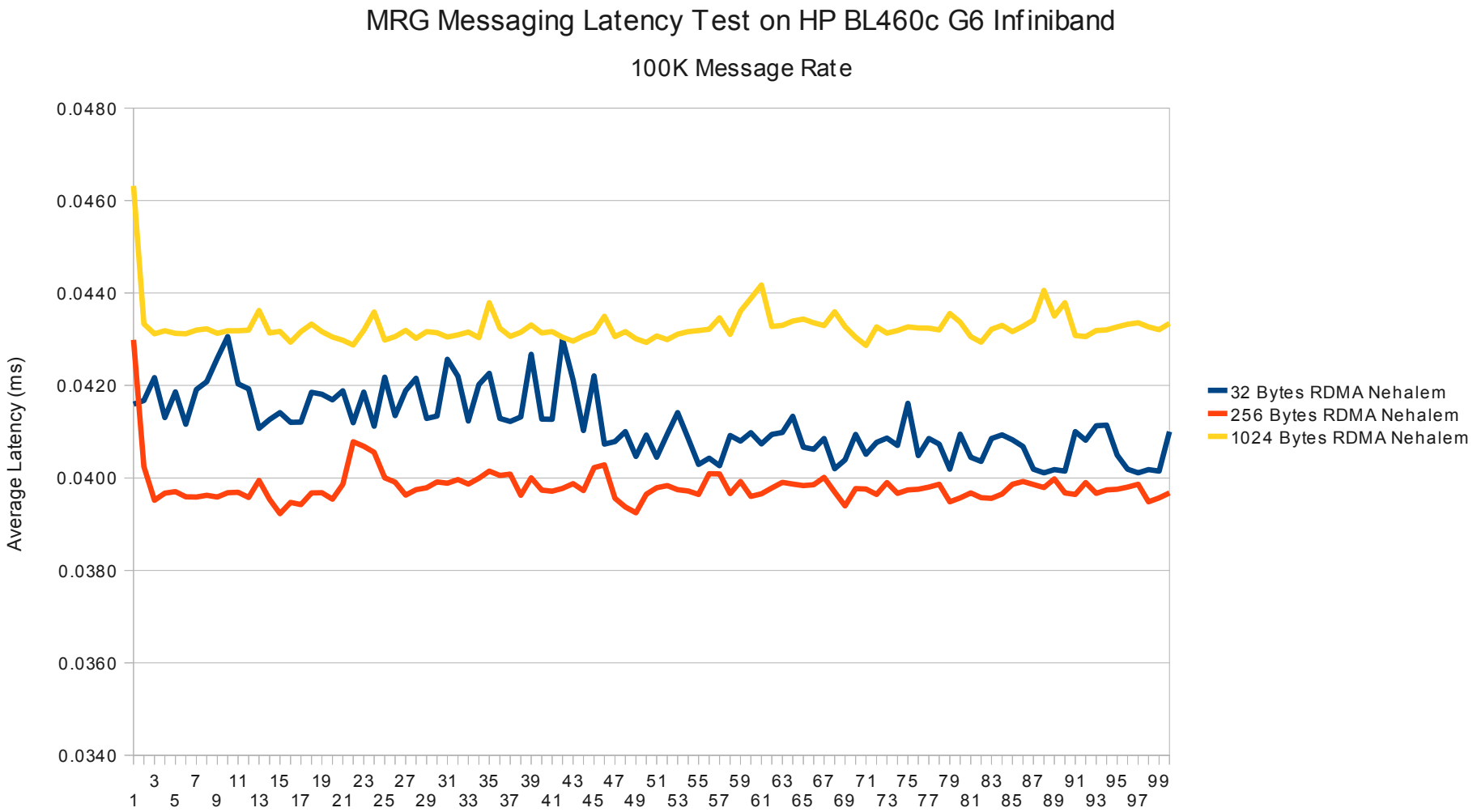
- Simple Authentication and Security Layer (SASL) security layer
  - identifies and authorizes connections to the broker
  - multiple authentication methods
  - full SASL support in broker; SASL PLAIN for clients
- Multiple users support
- Role-based access control
  - based on plain text files
- SSL encryption

MRG Messaging Infiniband Throughput: >1.5 Million Reliable Messages/Second per System

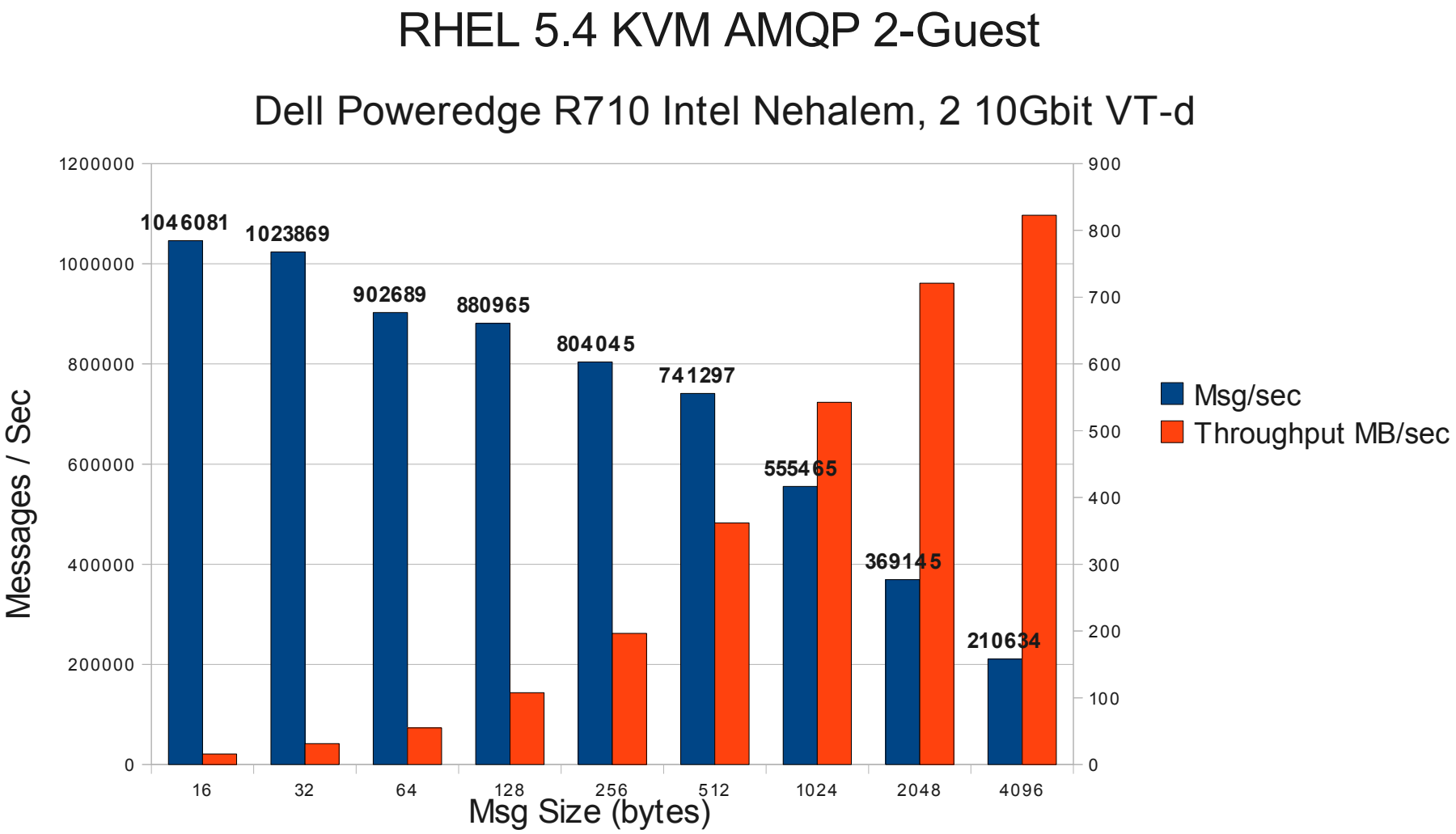


- HP BL460c G6 Intel(R) Xeon(R) CPU X5570
- 2.93 GHz, 8MB L3 cache, 95W,
- Memory Type DDR3-1333, HT, Turbo 2/2/3/3)
- Memory 24GB(6x4GB)
- Infiniband 4X QDR IB Dual-port Mezzanine
- Infiniband Switch BLc 4X QDR IB Switch

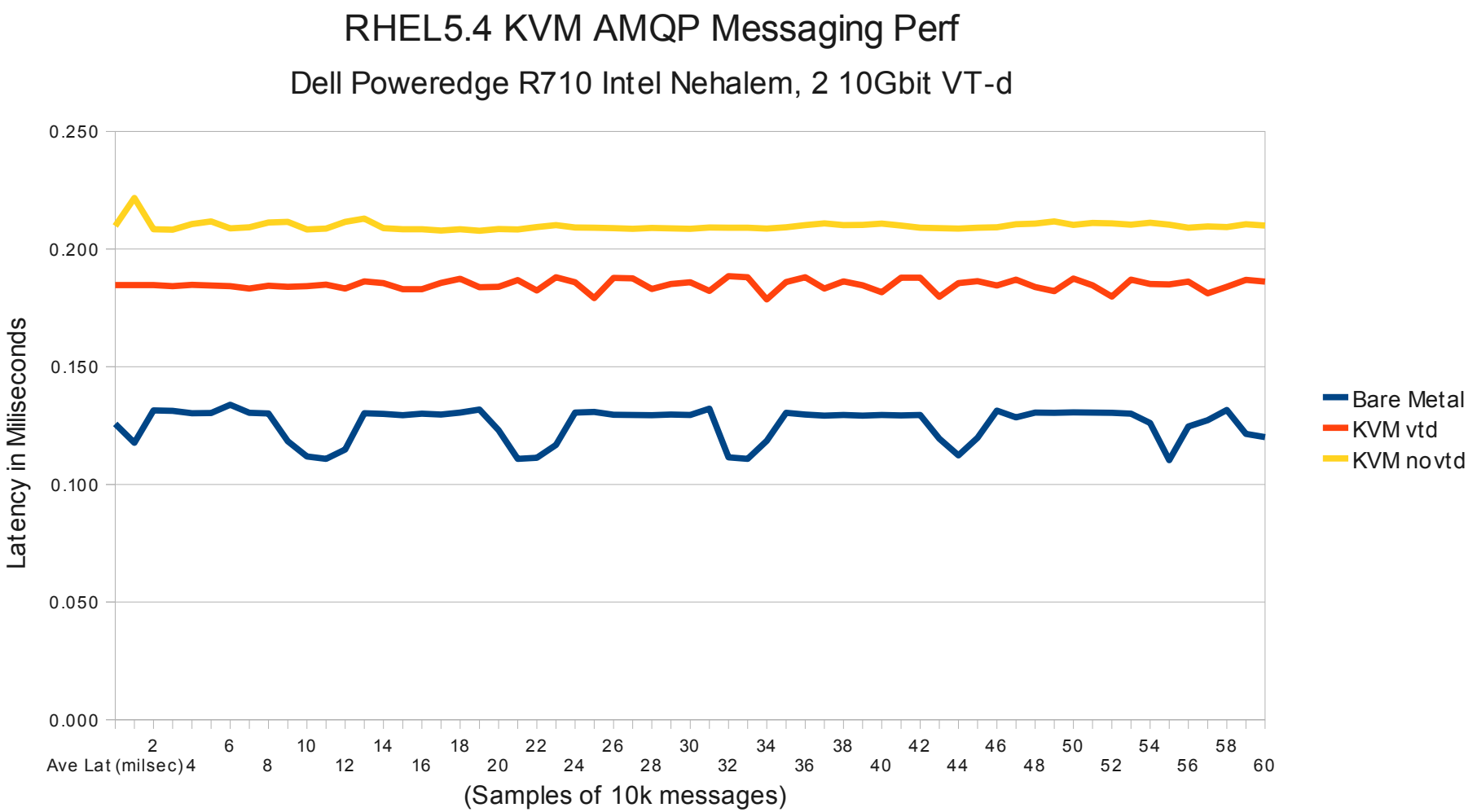
MRG Messaging Infiniband RDMA Latency: Under 40 Microseconds Reliably Acknowledged



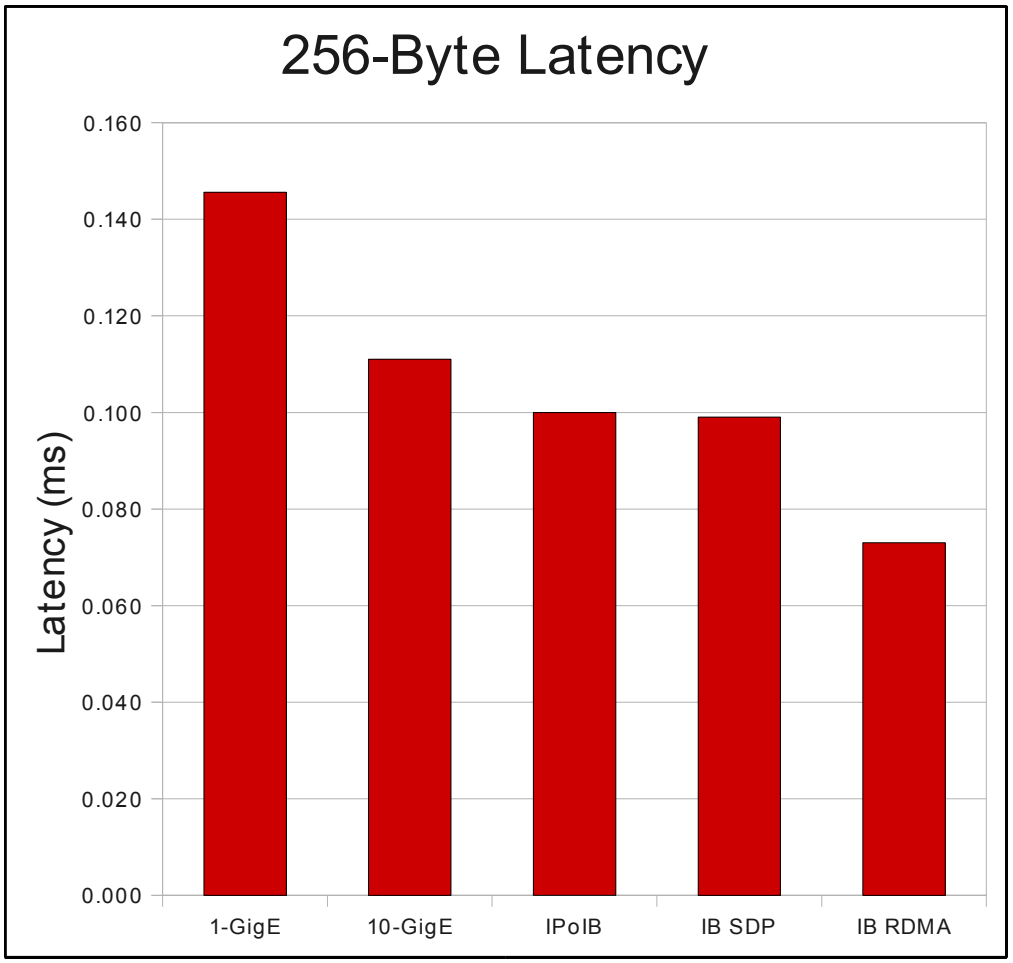
MRG Messaging on KVM Virtualized Performance: Over 1 million Messages/Second Throughput

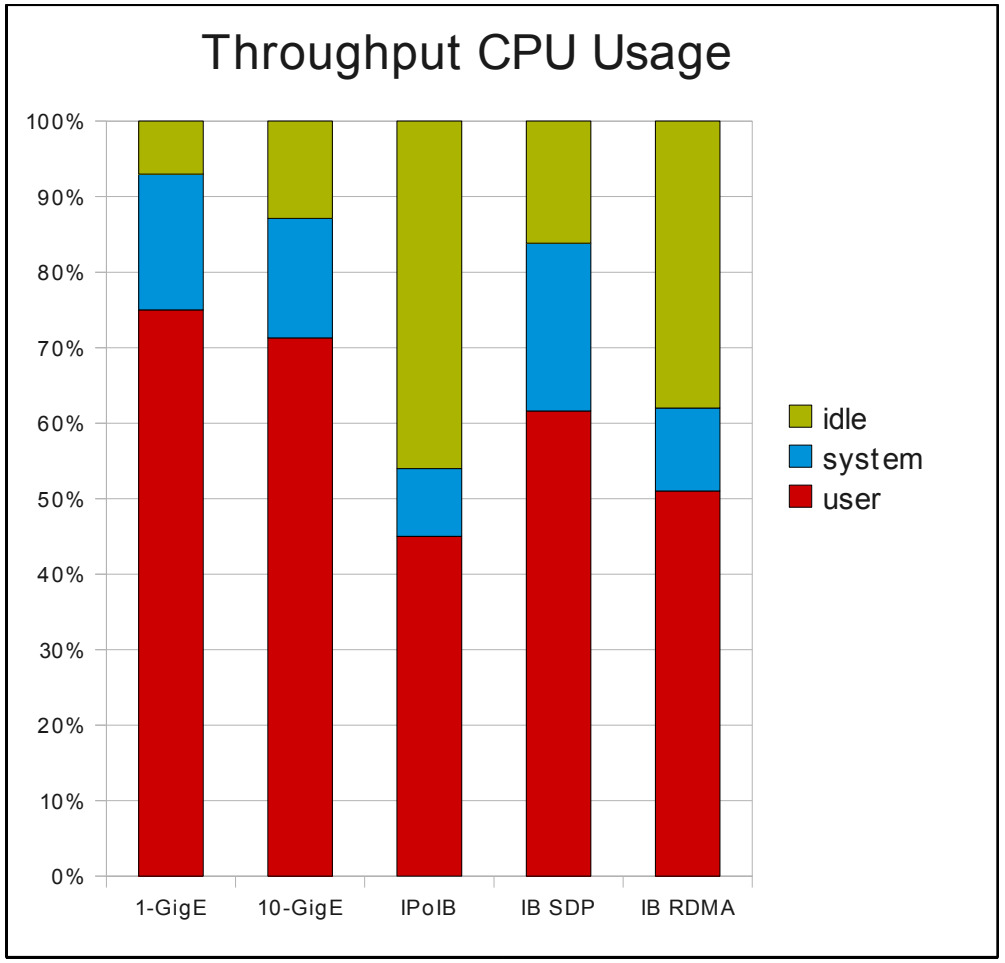


MRG Messaging on KVM Virtualized Performance: <200 Microsecond Latency, Reliably Acknowledged



Comparing Latency/CPU per technology

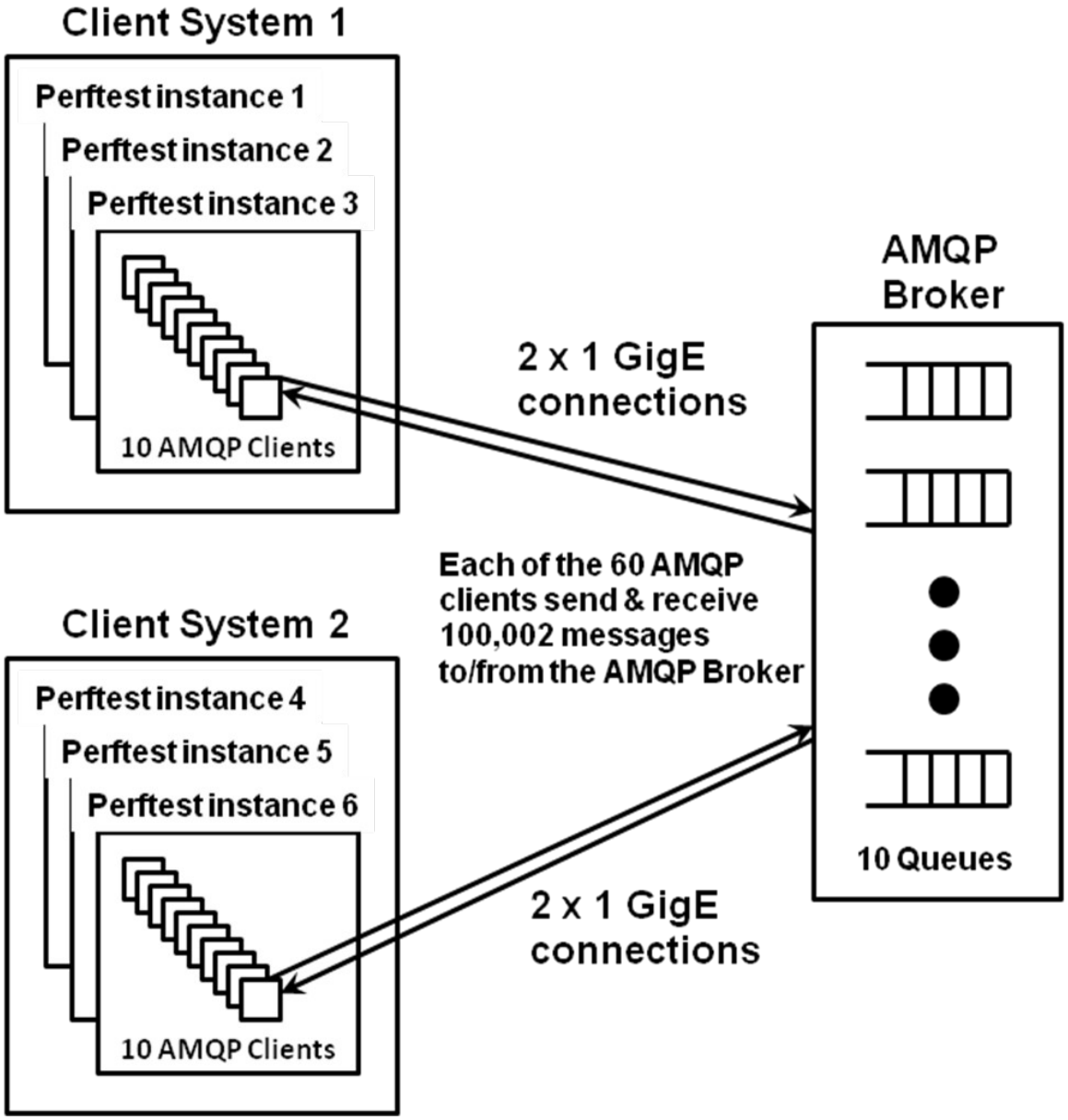




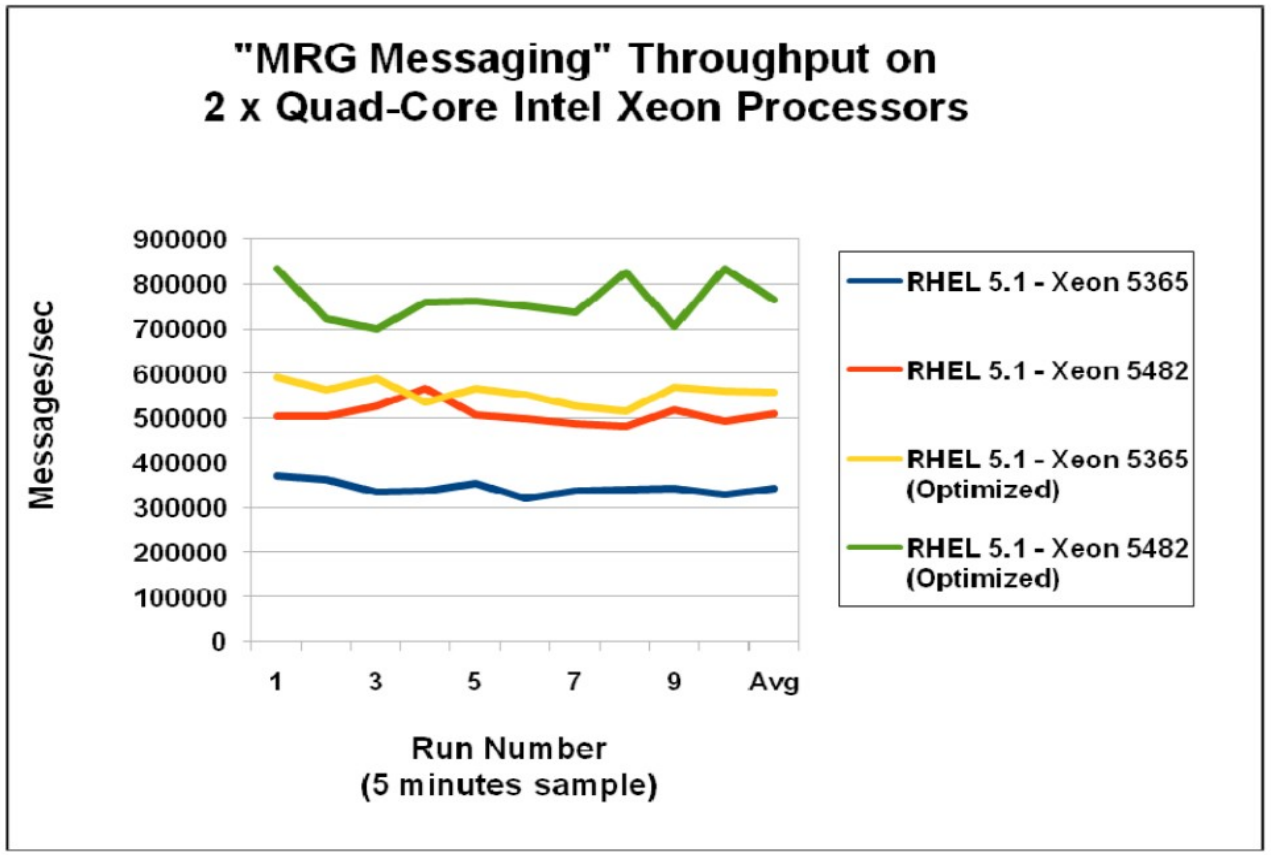
All measurements are between 3 peers (brokered) and fully reliable

Legacy Test: MRG/Intel Throughput Test Setup

- Intel 2x Quad-Core Xeon systems with 1GB ethernet
  - Intel Xeon 5300 series SUT
    - 2x X5365 (Quad cores), 3.00 Ghz,
    - 8 GB RAM
    - FSB 1333
  - Intel Xeon 5400 series SUT
    - 2x X5482 (Quad cores), 3.20 Ghz,
    - 8 GB RAM
    - FSB 1600
- 256-byte messages
- Fanout to 60 clients on two client systems from one broker
- 10 shared queues

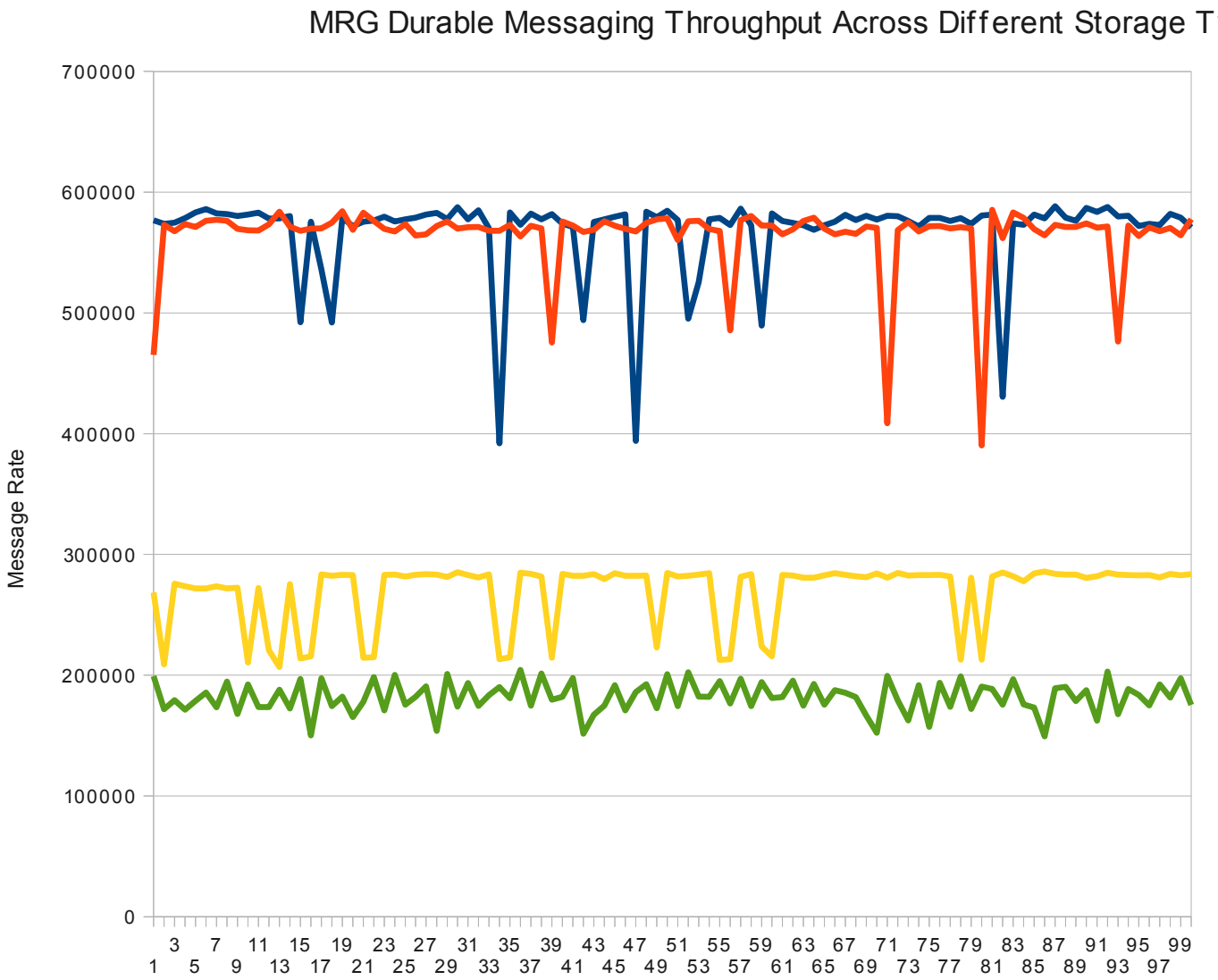


Throughput Results



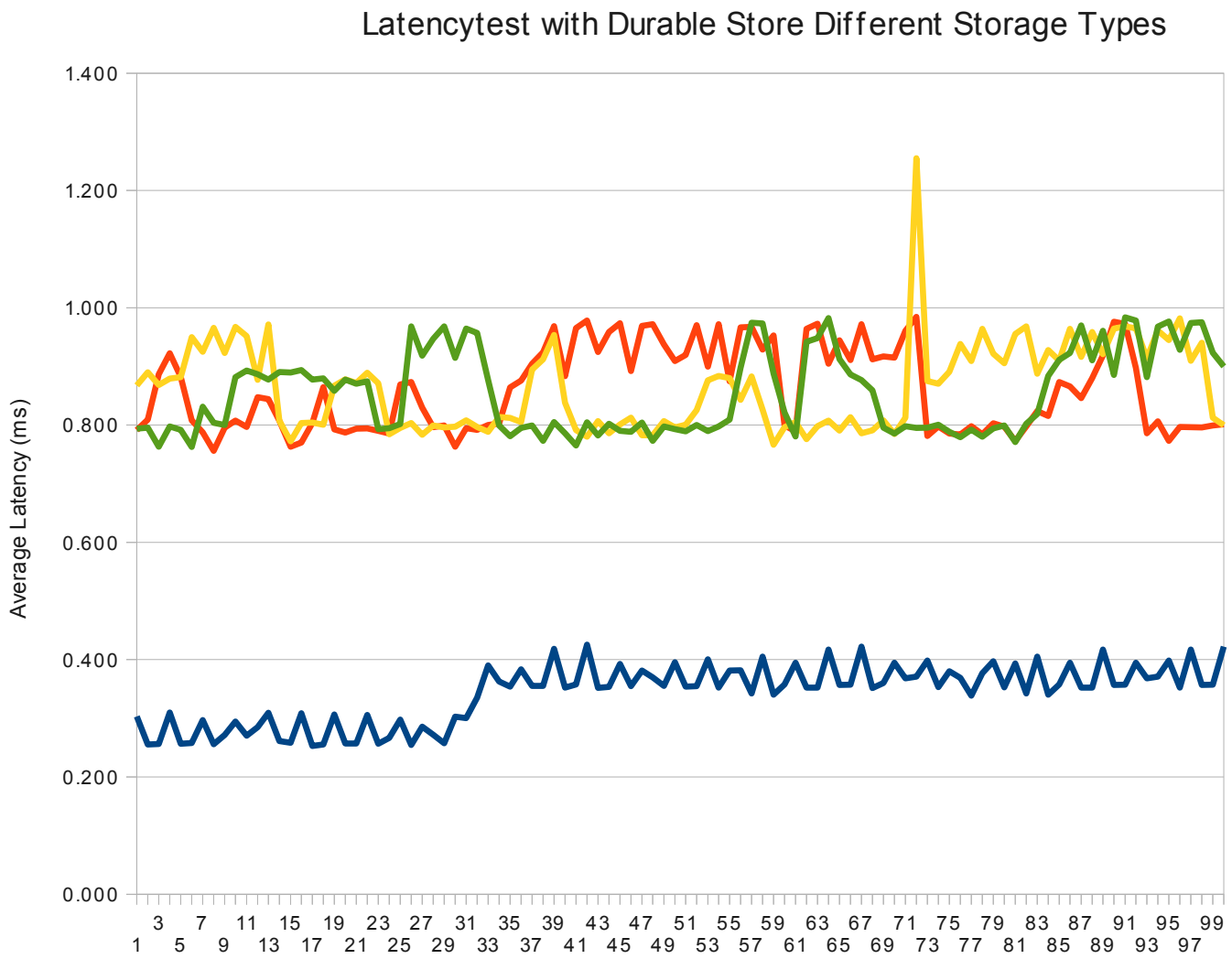
- 760,000+ ingress messages/sec on Xeon 5482
- Equivalent to
  - 6,080,000 ingress OPRA messages/ second
  - 2,432,000 fully reliable OPRA messages per second

MRG Messaging Durable Messaging Throughput



- Intel 16 CPU Hapertown
- 12GB memory 667 Memory speed
- Intel 82571EB Gigabit Ethernet
- HP IO Fusion
- 32-byte messages

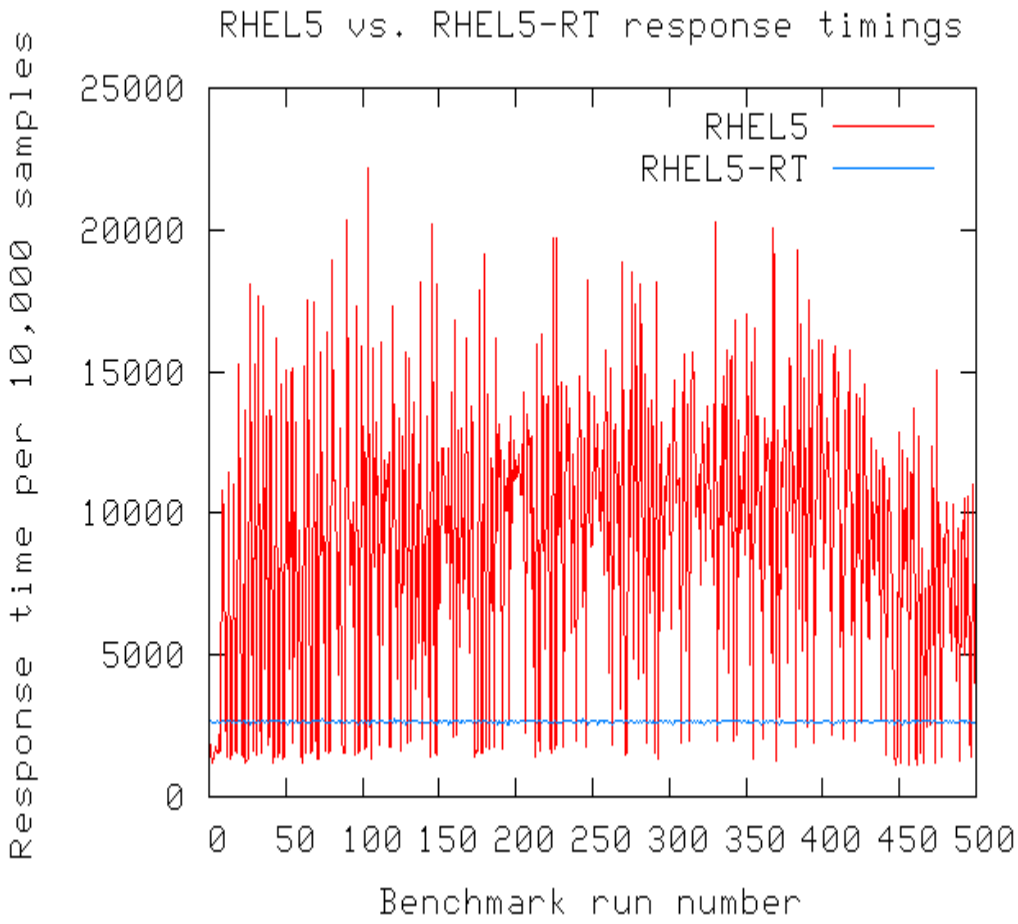
MRG Messaging Durable Messaging Latency



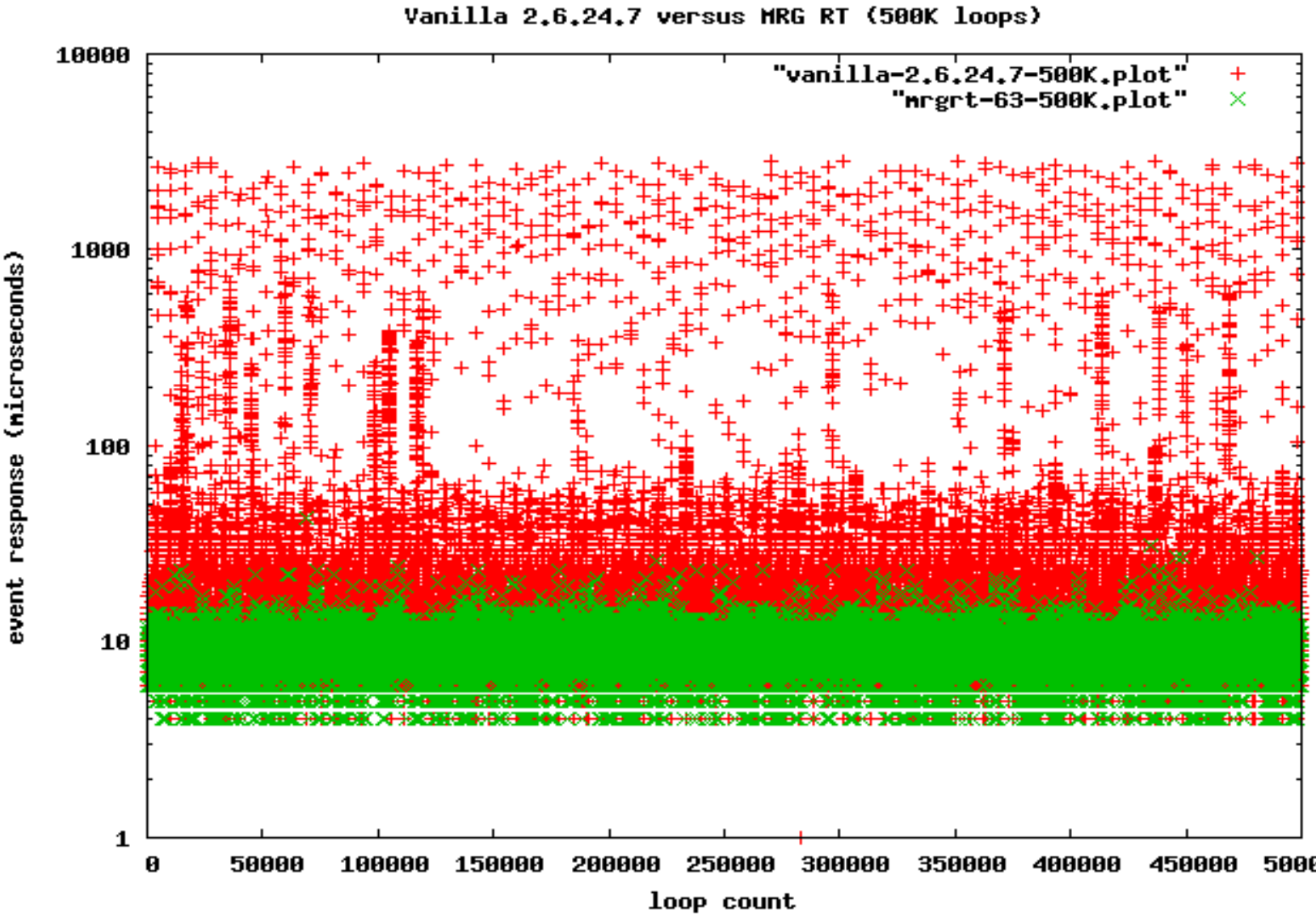
- Intel 16 CPU Hapertown
- 12GB memory 667 Memory speed
- Intel 82571EB Gigabit Ethernet
- HP IO Fusion
- 32-byte messages

MRG Realtime

- Enables applications and transactions to run predictably, with guaranteed response times
  - Provides microsecond accuracy
- Provides competitive advantage & meets SLA's
  - Travel web site: missed booking
  - Program trading: missed trades
  - Command & Control: life & death
- Upgrades RHEL 5 to realtime OS
  - Provides replacement kernel for RHEL 5; x86/x86_64
  - Preserves RHEL Application Compatibility
- Red Hat Leads Upstream Linux Realtime Development
  - maintainer, wrote 90% of code base



MRG Realtime Scheduling Latency



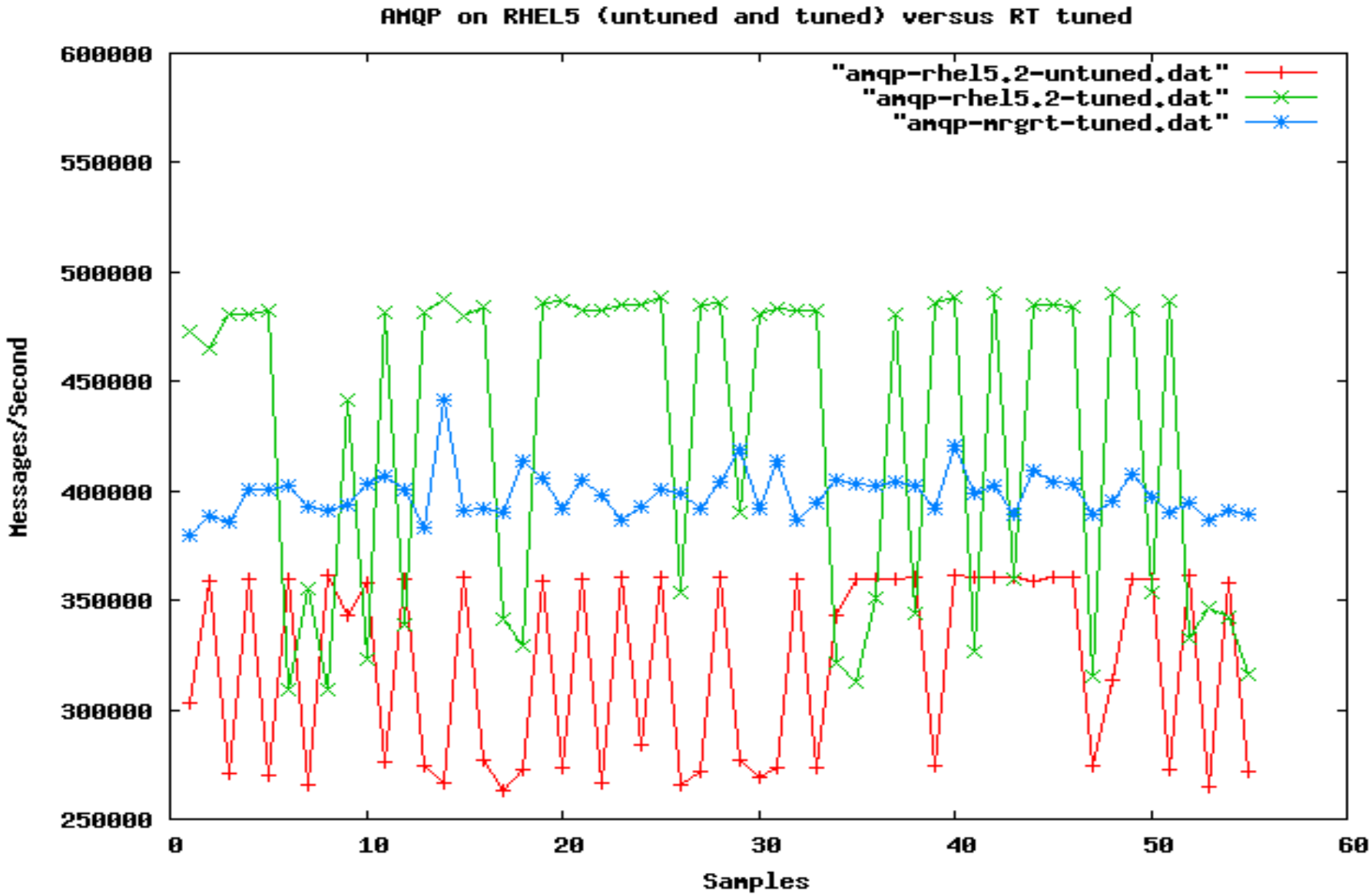
Vanilla

- Min: 1
- **Max: 2857**
- Mean: 11.47
- Mode: 9.00
- Median: 9.00
- **Std. Deviation: 54.94**

MRG RT

- Min: 4
- **Max: 43**
- Mean: 8.34
- Mode: 8.00
- Median: 8.00
- **Std. Deviation: 1.49**

MRG Realtime Throughput



Realtime exchanges ~10% throughput for better deterministic latency

MRG Realtime Kernel Features

- Preemption
  - Interrupts not turned off for almost all operations
  - threaded interrupt handlers
- Most locks converted to rt_mutex
  - priority inheritance *
  - Spinlocks can sleep
- high-resolution timers *
- Performance optimizations like Virtual Dynamic Shared Object (VDSO) gettimeofday
- Completely Fair Scheduler (CFS) *
- Read-Copy-Update (RCU) *
- Ftrace tracing infrastructure *

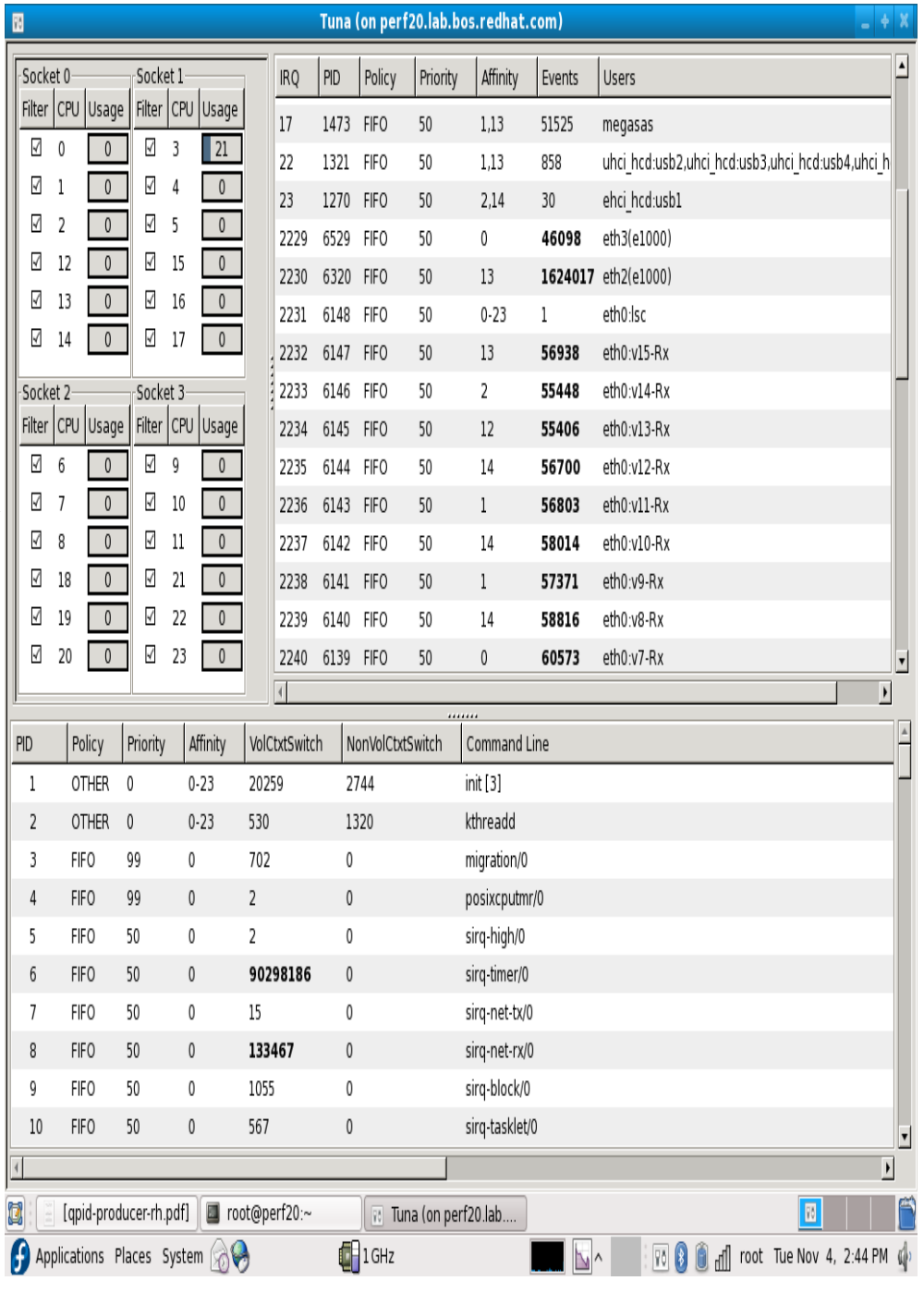
* *now in upstream kernel*

MRG Realtime Preserves RHEL 5 Application Compatibility

- MRG Realtime uses the existing RHEL5 C library with no changes
- APIs to access timers, get/set scheduler policies, lock memory and many others have been around for years
  - `gettimeofday()`
  - `clock_gettime()`
  - `sched_setsched()`
  - `mlockall()`
- What changed is the underlying kernel code
- Note that you don't have to have an RT kernel for these APIs to work
  - Need RT kernel for them to work *well*

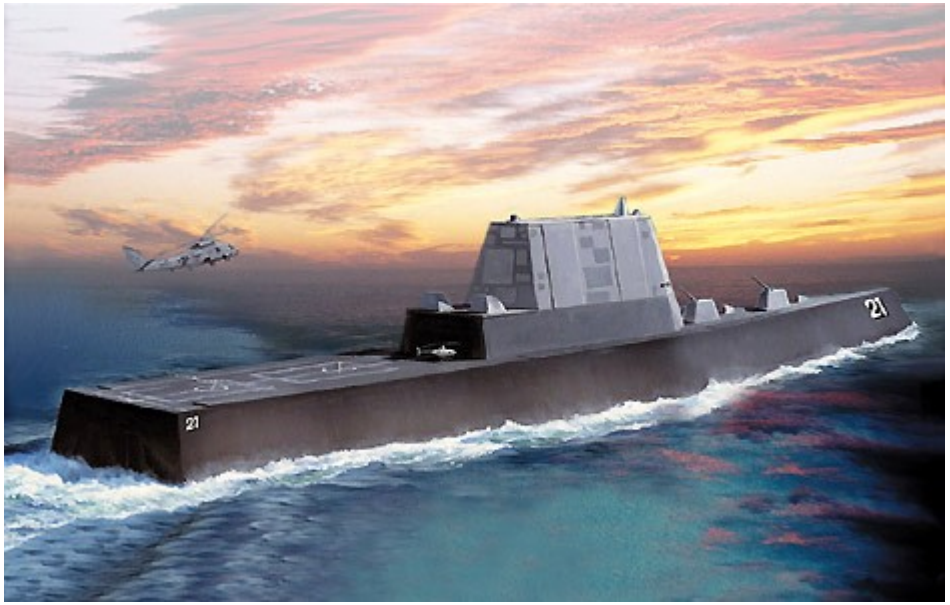
MRG Realtime Tools

- TUNA: System Tuning Tool
  - Dynamically control tuning parameters like process affinity, parent & threads, scheduling policy, device IRQ priorities, etc.
- FTrace: Latency Tracer
  - Runtime trace capture of longest latency codepaths – both kernel and application. Peak detector
  - Selectable triggers for threshold tracing
  - Detailed kernel profiles based on latency triggers
- RTEval: Hardware Latency Detector
  - Tool that finds hardware latencies in your system so that you can achieve low latency across your entire platform
  - Complements MRG Realtime hardware certification program
- Existing standard RHEL5 based performance monitoring tools remain relevant



Realtime Java With MRG Realtime

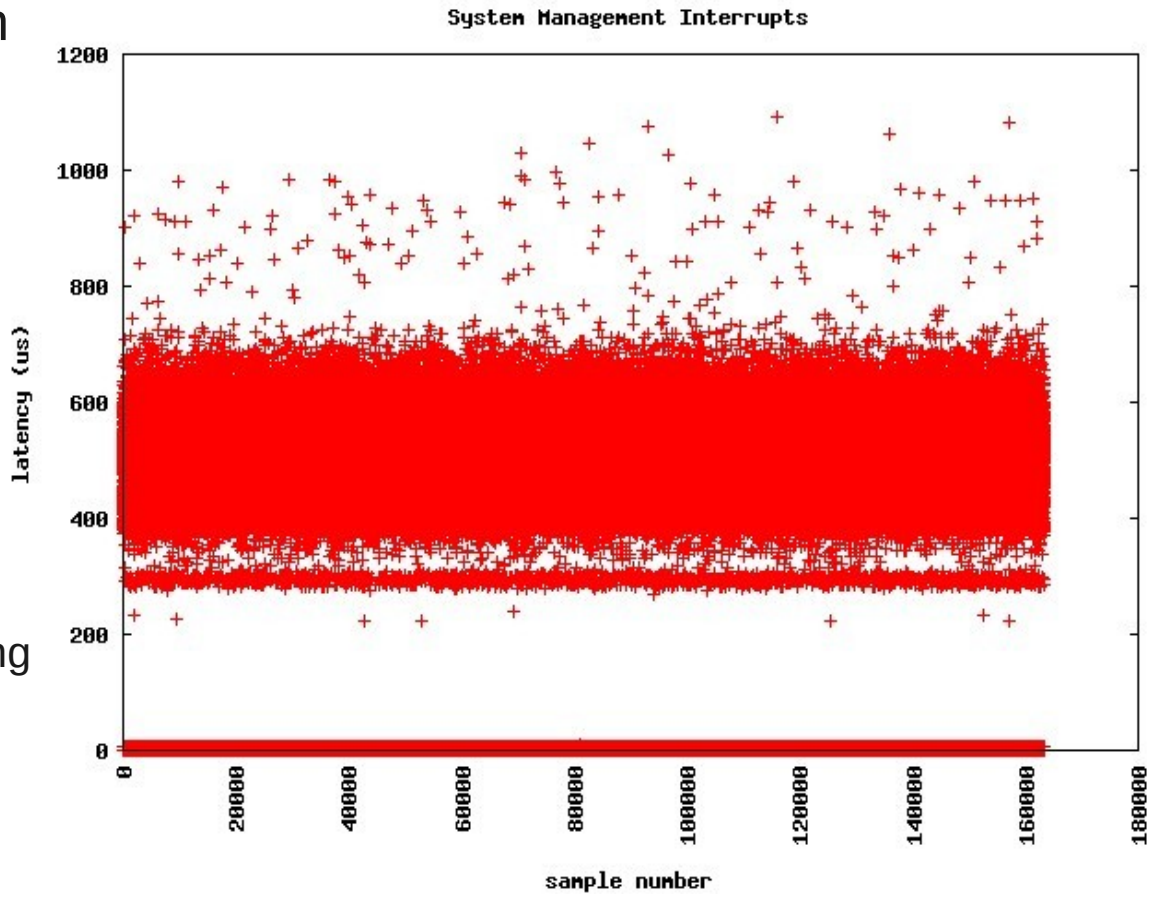
- Standard Java deployments typically have highly undeterministic performance—especially because of garbage collection
- JSR 1 provides a realtime specification for Java and realtime JVMs
  - Requires an underlying realtime operating system to provide *priority inheritance* and *preemption* —like realtime Linux!
  - Provides deterministic garbage collection, realtime threads, and deterministic performance
- Red Hat has partnered with IBM and Sun to certify their Realtime JVMs for MRG Realtime



Red Hat and IBM have partnered to deliver Realtime Java on Realtime Linux for the US Navy DDG 1000 Zumwalt Class Destroyer Program

Hardware Matters

- Hardware can have a big effect on realtime performance
- Hardware drivers may need to be updated to handle threaded interrupts
- Many system BIOS's include System Management Interrupts (SMIs)
  - Cause non-deterministic latency *beneath* the operation system by taking CPU cycles for things like power management, administration
  - SMI latencies *cannot* be resolved by realtime linux—they require the hardware OEM to remove SMIs or make them configurable
- Red Hat has worked with OEMs to certify systems for MRG Realtime



SMI latencies on untuned system

How to Develop for MRG Realtime

- Use POSIX threads
  - finer grained applications mean more parallelism, so can take advantage of multiple cores
- Use POSIX threads synchronization mechanisms
  - Mutexes
  - Barriers
  - Condition variables
- Set appropriate priorities for your threads
  - Any SCHED_FIFO thread is higher priority than any SCHED_OTHER thread
  - ensure that your high priority threads don't hog the processor

How to Deploy MRG Realtime

- Tune your system!
  - No two applications behave the same
  - use *tuna* to tweak priorities and affinities
  - use *oprofile* to find application hotspots
  - use *ftrace* to find long latency areas
- Dedicate processors to your application threads
  - Use *tuna* or *taskset* to bind threads to specific processors and move other threads off
  - 4-way and 8-way processors getting cheaper
- Use cpu affinity field in /proc/irqs/<n>/smp_affinity to bind interrupts to specific processors
  - *tuna* can do this easily

MRG Grid

- Provides leading High Performance & High Throughput Computing
  - Brings advantages of scale-out and flexible deployment to any application or workload
  - Delivers better asset utilization, allowing applications to take advantage of all available computing resources
  - Handles “Holiday Rushes”
- Enables building cloud infrastructure and aggregating multiple clouds
  - Integrated support for virtualization as well as public clouds
  - Seamlessly aggregates multiple cloud resources into one compute pool
- Provides seamless and flexible computing across:
  - Local grids
  - Remote grids
  - Private and hybrid clouds
  - Public clouds (Amazon EC2)
  - Cycle-harvesting from desktop PCs

MRG Grid is Based on Condor

- MRG Grid is based on the Condor Project created and hosted by the University of Wisconsin, Madison
- Condor has a >20-year history and runs many of the largest grids in the world
- Red Hat and the University of Wisconsin have signed a strategic partnership around Condor:
  - University of Wisconsin makes Condor source code available under OSI-approved open source license
  - Red Hat & University of Wisconsin jointly fund and staff Condor development on-campus at the University of Wisconsin
- Red Hat and the University of Wisconsin's partnership will:
  - Add enhanced enterprise features, management, and supportability to Condor and MRG Grid
  - Add High Throughput Computing capabilities to Linux

Red Hat Additions To Condor Include:

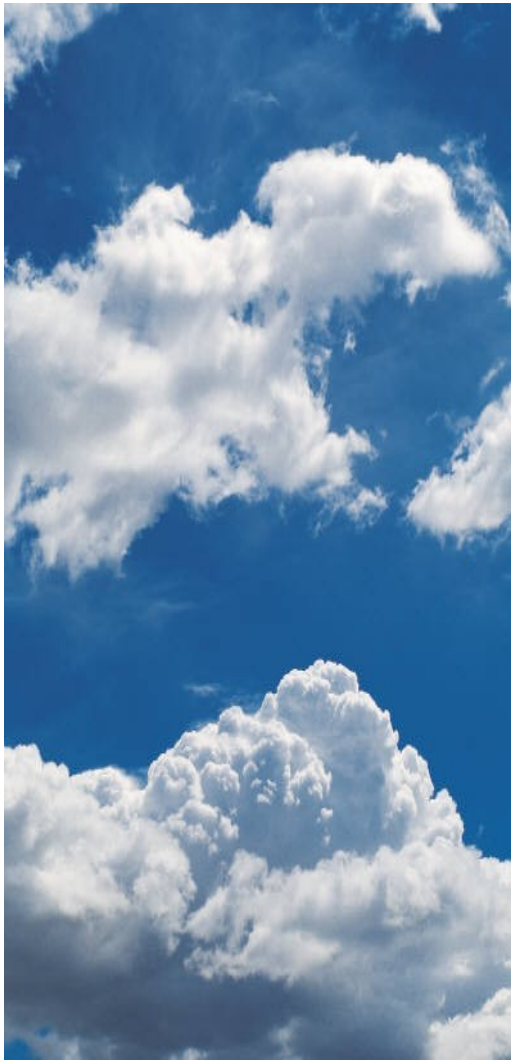
- Enterprise Supportability
  - Break out Condor from statically-linked blob to multiple well-maintained and individually patchable rpm's
- Web-Based Management Console
  - Unified management across all of MRG for job, system, license management, and workload management/monitoring
- Low Latency Scheduling
  - Enable job submission to Condor via AMQP Messaging clients
  - Enable sub-second, low-latency scheduling for sub-second jobs
  - Back MS Excel calculations with a grid via MRG C# client
- Virtualization Support via libvirt Integration
  - Support scheduling of virtual machines on Linux using libvirt API's
- Cloud Integration with Amazon Ec2
  - Enable automatic cloud provisioning, job submission, results storage, teardown via Condor scheduler
  - Runs as a job, so it can be a dependency for other jobs or executed based on rules (e.g. add capacity in in the cloud if local grid out of capacity)
- Concurrency Limits
  - Set limits on how much of a certain resource (e.g. software licenses, db connections) can be used at once
- Dynamic Slots
  - Mark slots as partitionable and sub-divide them dynamically so that more than one job can occupy a slot at once

Other MRG Grid Features Include:

- **Desktop Cycle-Harvesting** - Desktop cycle-harvesting allows you to leverage the unused capacity of desktops to add processing power to your grid.
- **ClassAds** - A flexible language for policy and meta-data description.
- **Policies** - Flexible, customizable policies specified by jobs and resources via ClassAds.
- **Federated Grids/Clusters** - A mechanism known as flocking allows independent pools to use each others' resources, controllable by customizable policies.
- **Multiple Standards-Based APIs** - Web Service interface provides job submission and management functionality; CLI provides a highly scriptable, with consistent output, interface to all functionality.
- **Workflow Management** - The ability to specify job dependencies, via DAGMan, allows for construction and execution of complex workflows.
- **Compute On-Demand (COD)** - The ability for a node or set of nodes to be claimed by a user in such a way that others may use the claimed nodes until the user needs them.
- **High Availability** - The Negotiator and Collector, via HAD, and the Schedd, via Schedd Fail-over, can have their state replicated to allow for graceful fail-over upon service disruption.
- **Dynamic Pool Creation** - Through a technology known as Glide-ins, nodes can be dynamically added to a pool to service user jobs.
- **Priority Based Scheduling** (including fair share)
- **Parallel Universe** – run parallel (including MPI) jobs. Co-allocation of compute nodes is done automatically.
- **Accounting** - User and group resource utilization is tracked and accessible to administrators.
- **And much more...**

Cloud Computing with MRG

- Cloud computing is a hot topic, but many people have important questions and challenges they need addressed before they can adopt cloud:
  - How do I build an internal cloud?
  - How do I avoid lock-in to a single cloud?
  - How do I mix, match, and blend different cloud resources—including internal and external clouds?
  - How do I manage a variety of applications and groups with different SLAs, priorities, and resource requirements across clouds?
  - How do I manage and track cloud resources?
- Red Hat Enterprise MRG provides solutions to all these issues



Building Clouds with MRG

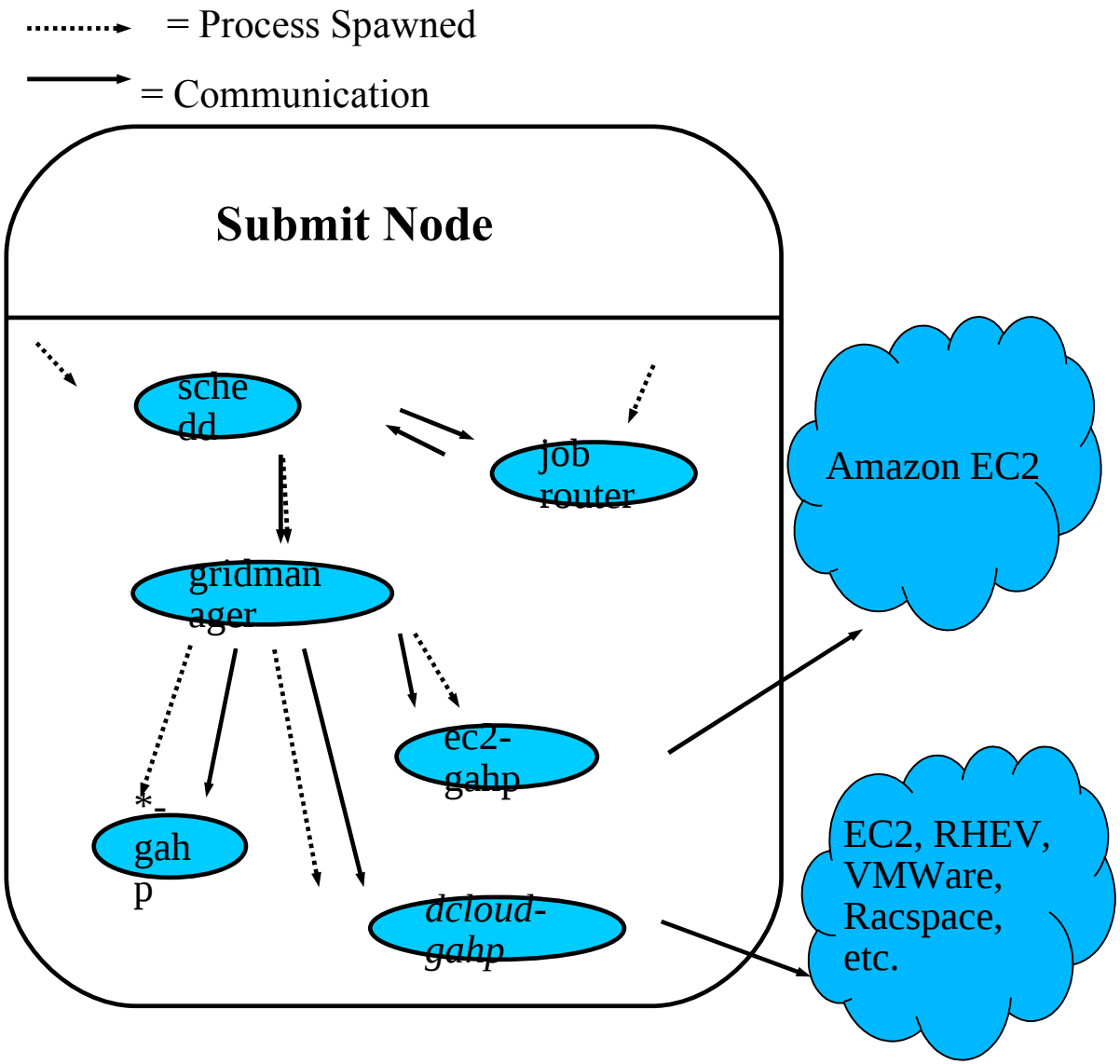
- Scalable Virtualization
  - Schedule VMs directly as jobs via libvirt
  - Provision VMs via Red Hat Enterprise Virtualization
  - Inject jobs into VMs
- Resource Accounting
  - Track resources via Condor's resource accounting
- SLA's
  - Apply priorities and policies
  - Apply security – Authentication (e.g. SSL, …), Integrity, Encryption
- Powerful Policies
  - VMs – run multiple concurrent instances, start on Black Friday or semi-monthly, re-run after fault
  - Machines – only run VMs from owner’s group between 9 and 5, everyone else has a low priority shot from 5 to 9
  - Global – control limiters (e.g. NFS mount users, licenses),
- Various Cloud Services
  - IaaS clouds: run all workloads as VMs
  - PaaS clouds: leverage job scheduling with VM scheduling

Aggregating and Bridging Clouds with MRG

- MRG includes the ability to schedule jobs and applications to multiple clouds, based on policy
  - MRG has the ability to send VMs to other resource managers
  - MRG becomes the unified interface to many types of resources – internal VM resources and multiple external clouds
  - MRG's life-cycle management, accounting and policy benefits still available
- Use cases include
  - Manage overflow/spillover
  - Access to specialized resource managers
  - Transformation between VM types/systems
  - Allow a single app/stack to bridge multiple clouds

MRG Cloud Aggregation Architecture

- Schedd accepts jobs over SOAP, AMQP, CLI
- GAHP: Grid ASCII Helper Protocol
  - An adapter to an external resource manager
  - Exist for many batch systems
  - Exists for EC2-like resource managers
  - Extensible to new resource managers
- Job Router transforms types, e.g. stack to VM to EC2 AMI

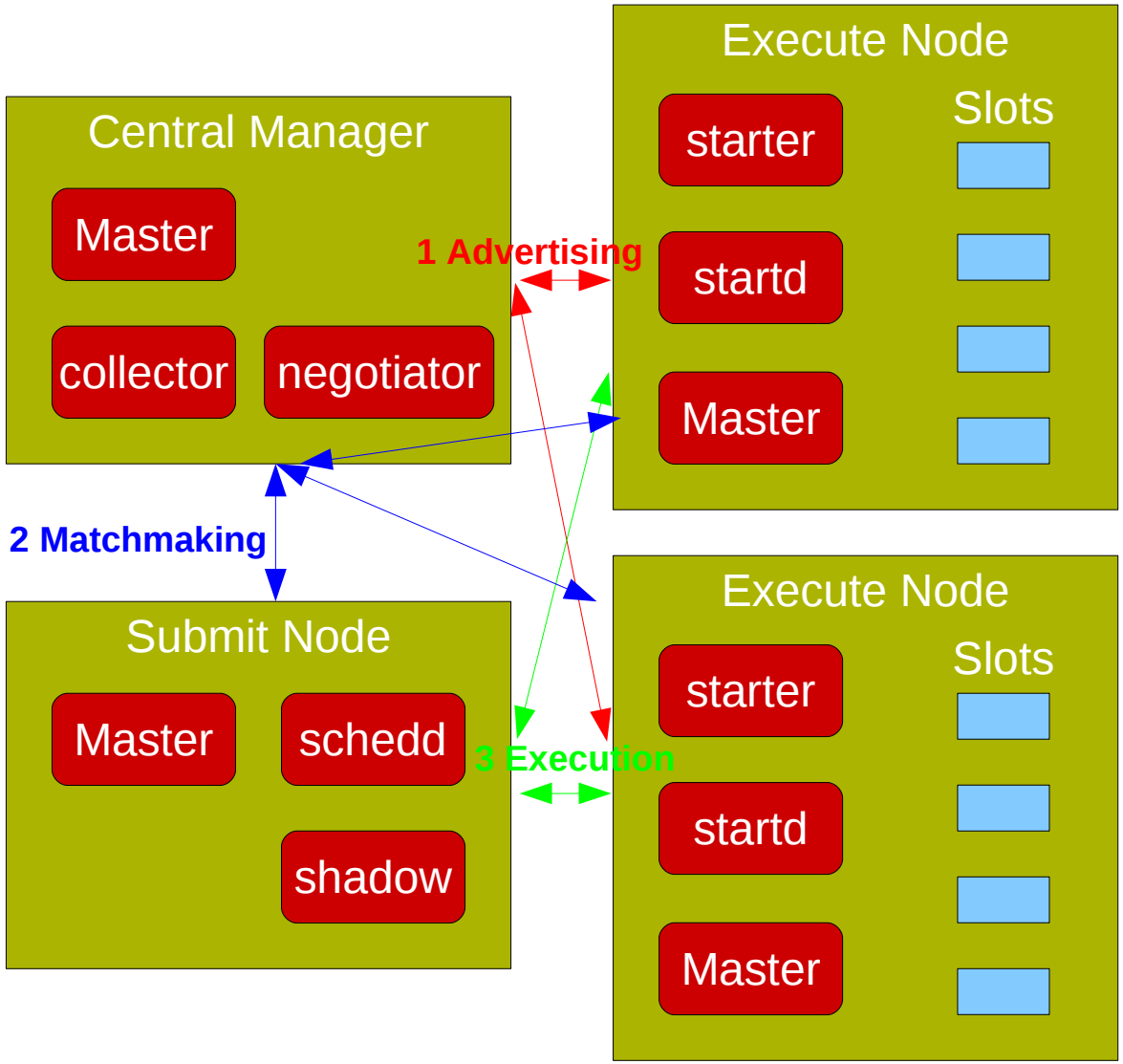


Cloud Computing with MRG
Demo Video

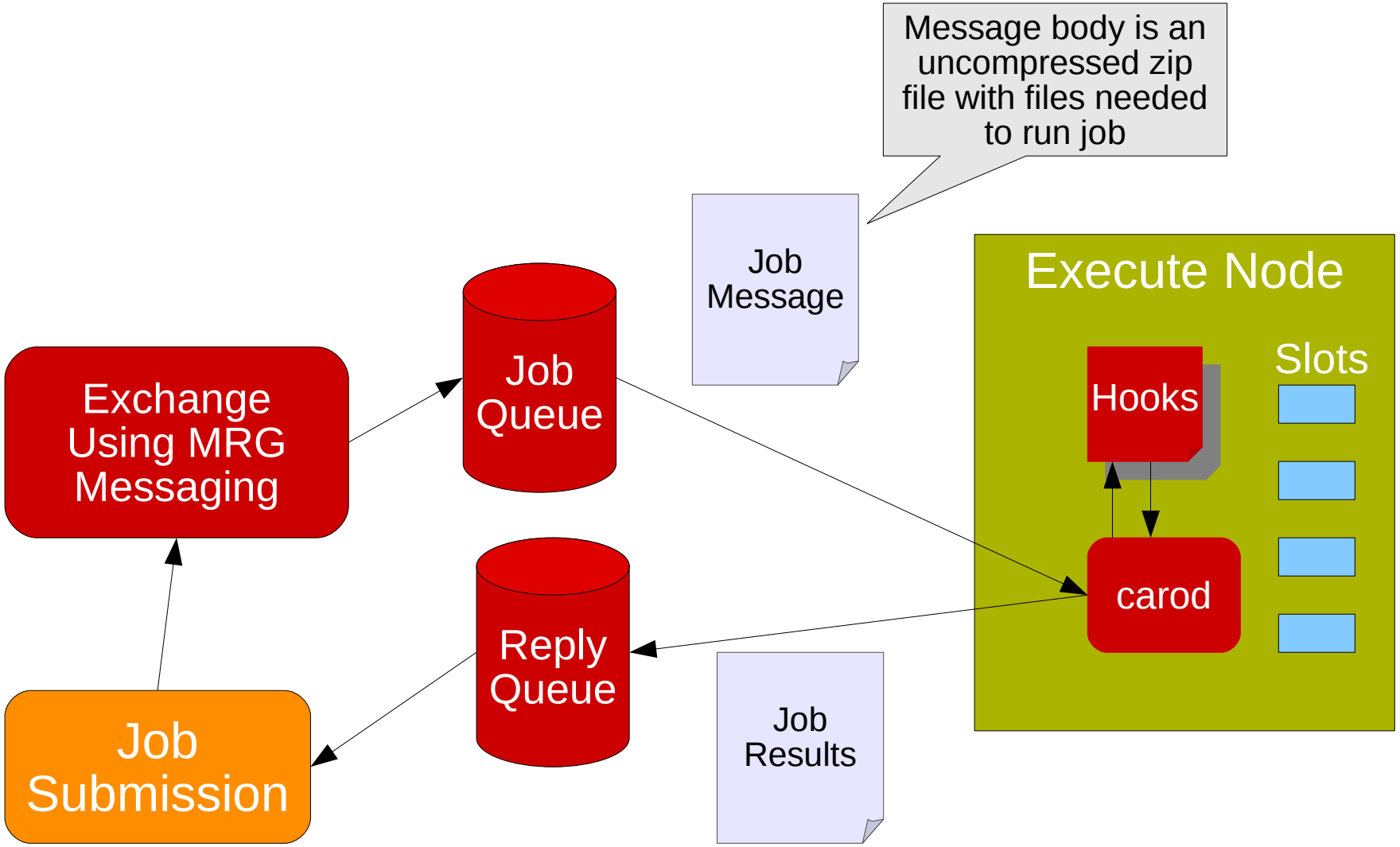
http://www.youtube.com/watch?v=oSm7Ff8kKjk

MRG Grid Architecture Components

- Central Manager: Schedules Jobs
  - collector: collects info about pool status
  - negotiator: responsible for match-making. Informs submit nodes about execute nodes & vice-versa
- Submit Node: Submit Jobs
  - schedd: schedules jobs and stores in job queue
  - shadow: spawned to manage jobs
- Execute Node: Executes Jobs
  - startd: enforces policies, spawns job to starter
  - starter: process that spawns remote job and sends statistics to submitter
- *Master Daemon manages other daemons*



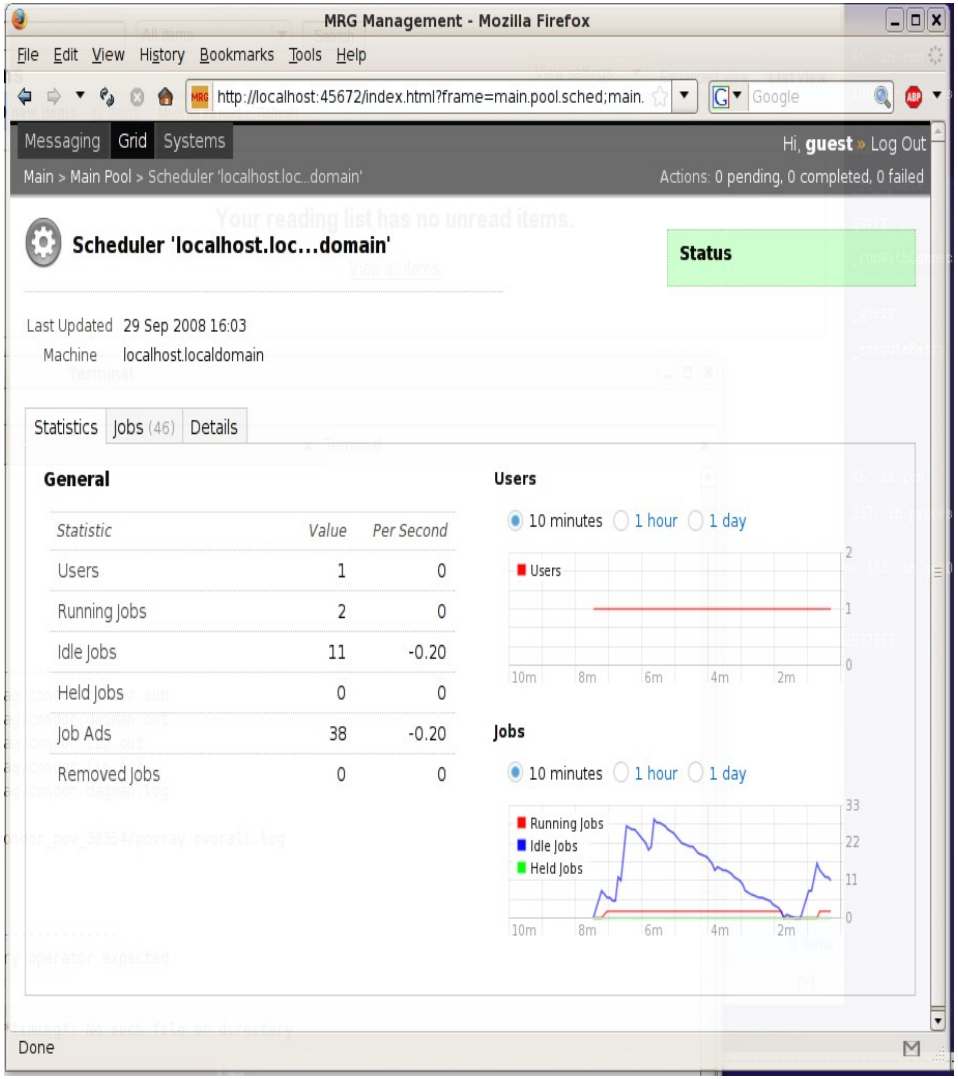
Low Latency Scheduling



**MRG Components in Red*

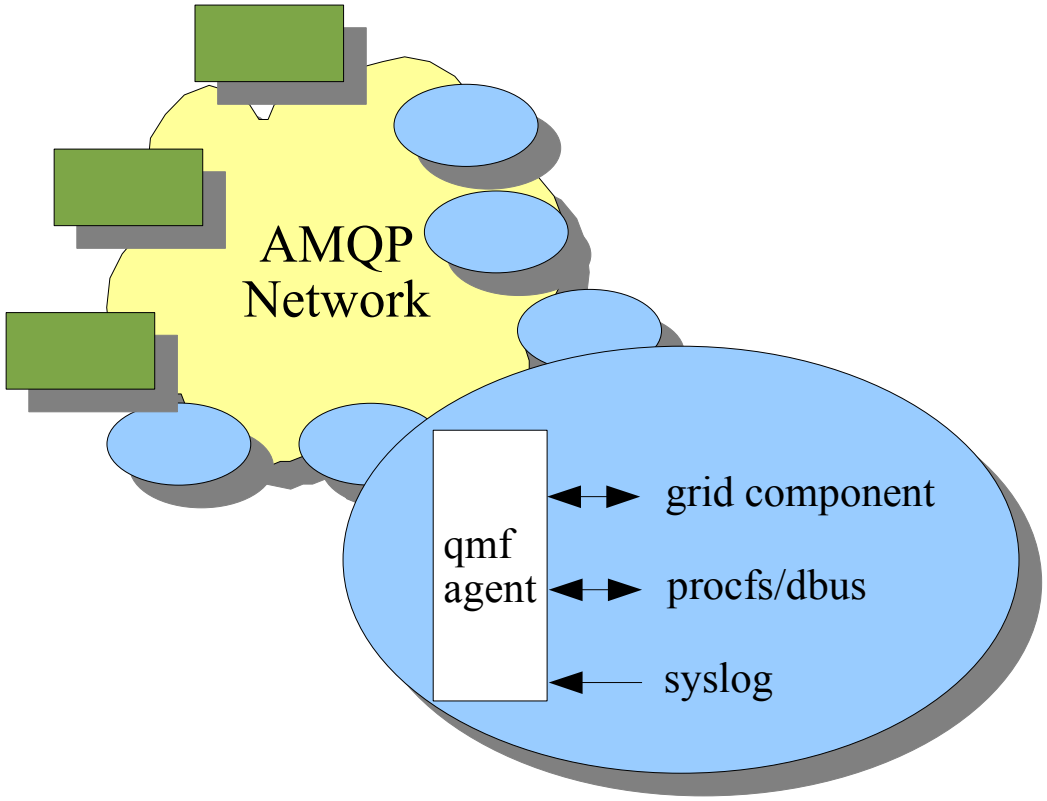
MRG Platform's Integrated Benefits

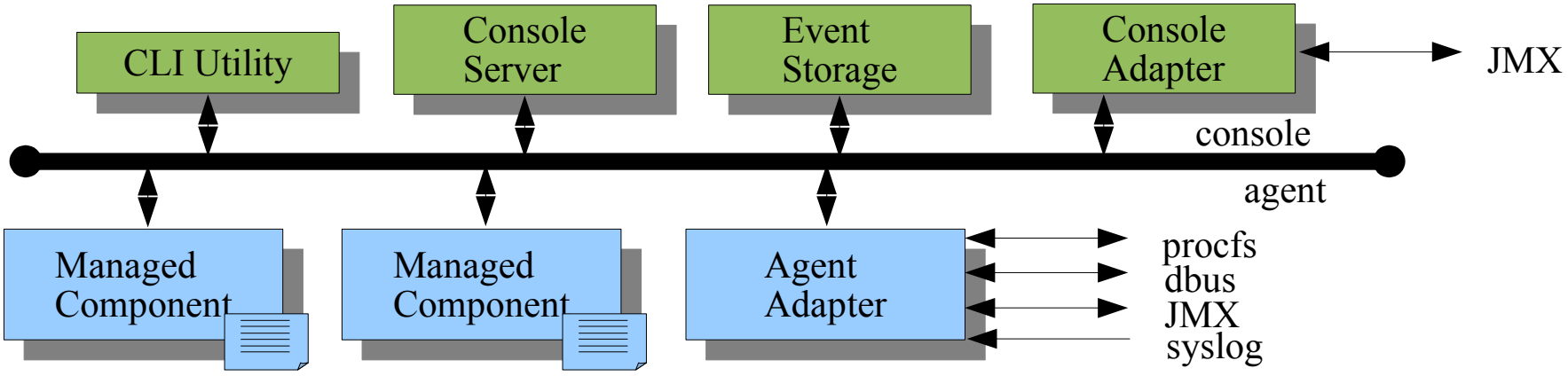
- Deterministic, low-latency messaging
  - Messaging and Realtime together provide deterministic, low-latency messaging
- Workload scheduling from reliable sub-second to long-running batch jobs, from small to large-scale
  - Grid scheduling via Realtime Messaging enables low-latency scheduling with high scalability
- Interoperability
  - AMQP provides full interoperability with an entire ecosystem (software and hardware) from stand-alone messaging to the grid
- Simplified software stack and flexible architecture
  - MRG provides one integrated platform to replace layers of specialized, incompatible point products
- Integrated management
  - MRG provides a unified management system built on MRG Messaging



QMF: AMQP Messaging-Based Management

- Red Hat Enterprise MRG's entire management/monitoring system is AMQP messaging-based
  - Asymmetric, Efficient, Scalable, and Secure
  - Any messaging client can manage
- QMF: AMQP Messaging-Based Management Framework
  - Agent-defined management model (self-describing)
  - Objects (properties, statistics, and methods/controls), Events
  - Ease of development and extensibility





Red Hat Enterprise MRG Availability

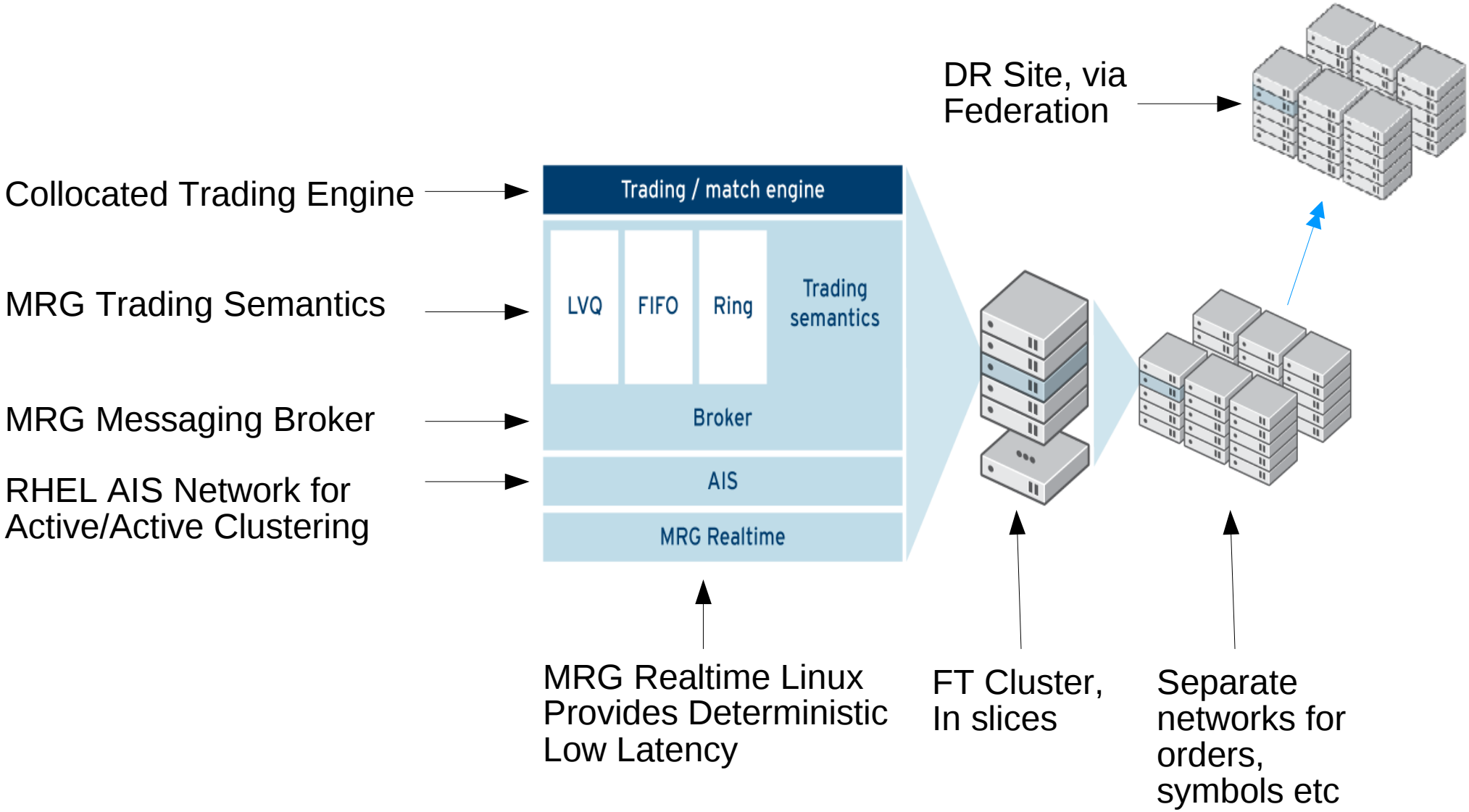
- 3 ways to buy MRG:
  - Red Hat Enterprise MRG Realtime component (priced per server)
  - Red Hat Enterprise MRG Messaging component (priced per CPU)
  - Red Hat Enterprise MRG MRG platform (priced per CPU)
    - includes messaging, realtime, and grid capabilities
- Red Hat Enterprise MRG is available on a limited basis, depending on geography
  - MRG Realtime component is available broadly worldwide
  - MRG Messaging component is available in NA and Europe
  - MRG Grid/Platform is available in NA

Additional Information

http://www.redhat.com/mrg

Addendum:
MRG Customer Trading System Example

MRG Customer Trading System Example



Illustrating trading semantics

-- setting up --

```
connection.open(host, port);
Session session =  connection.newSession();

// Create a queue named "message_queue", and route all messages whose
// routing key is "routing_key" to this FIFO queue.

session.queueDeclare(arg::queue="TICKER.NYSE", arg::exclusive=false);
session.exchangeBind(arg::exchange="amq.topic", arg::queue="TICKER.NYSE",
            arg::bindingKey="TICKER.NYSE.#");

session.queueDeclare(arg::queue="TICKER.NASDAQ", arg::exclusive=false);
session.exchangeBind(arg::exchange="amq.topic", arg::queue="TICKER.NASDAQ",
             arg::bindingKey="TICKER.NASDAQ.#");


// At this point we have two FIFO Queues for NYSE & NASDAQ
```

/* Fully worked example of this located in examples/tradedemo */

Illustrating trading semantics

--receive latest symbols --

```
void Listener::subscribeLVQQueue(std::string queue) {
 // Declare and subscribe to the queue using the subscription manager.
  QueueOptions qo;
  qo.setOrdering(LVQ);
  std::string binding = queue + ".#";
  queue += session.getId().getName();

  session.queueDeclare(arg::queue=queue, arg::exclusive=true, arg::arguments=qo);
  session.exchangeBind(arg::exchange="amq.topic", arg::queue=queue, arg::bindingKey=binding);
  subscriptions.subscribe(*this, queue, SubscriptionSettings(FlowControl::unlimited(), ACCEPT_MODE_NONE));
}

//  Then to subscribe....

        Listener listener(session);

        // Subscribe to messages on the queues we are interested in
           listener.subscribeTTLQueue("TICKER.NASDAQ");
           listener.subscribeTTLQueue("TICKER.NYSE");
           listener.subscribeLVQQueue("MRKT.NASDAQ");
           listener.subscribeLVQQueue("MRKT.NYSE");

        // Give up control and receive messages
        listener.listen();
```

Illustrating trading semantics

-- publish symbol data --

```
Message message;

std::string routing_key = "TICKER." + symbol;
std::cout << "Setting routing key:" << routing_key << std::endl;
message.getDeliveryProperties().setRoutingKey(routing_key);

curr_price = // { update the price ...  }

message.setData(curr_price);

// Set TTL value so that message will timeout after a period and be purged from queues
// This also creates a REPLAY window for late joining subscribers

message.getDeliveryProperties().setTtl(ttl_time);

// Asynchronous transfer sends messages as quickly as possible without waiting for confirmation.
async(session).messageTransfer(arg::content=message, arg::destination="amq.topic");
```

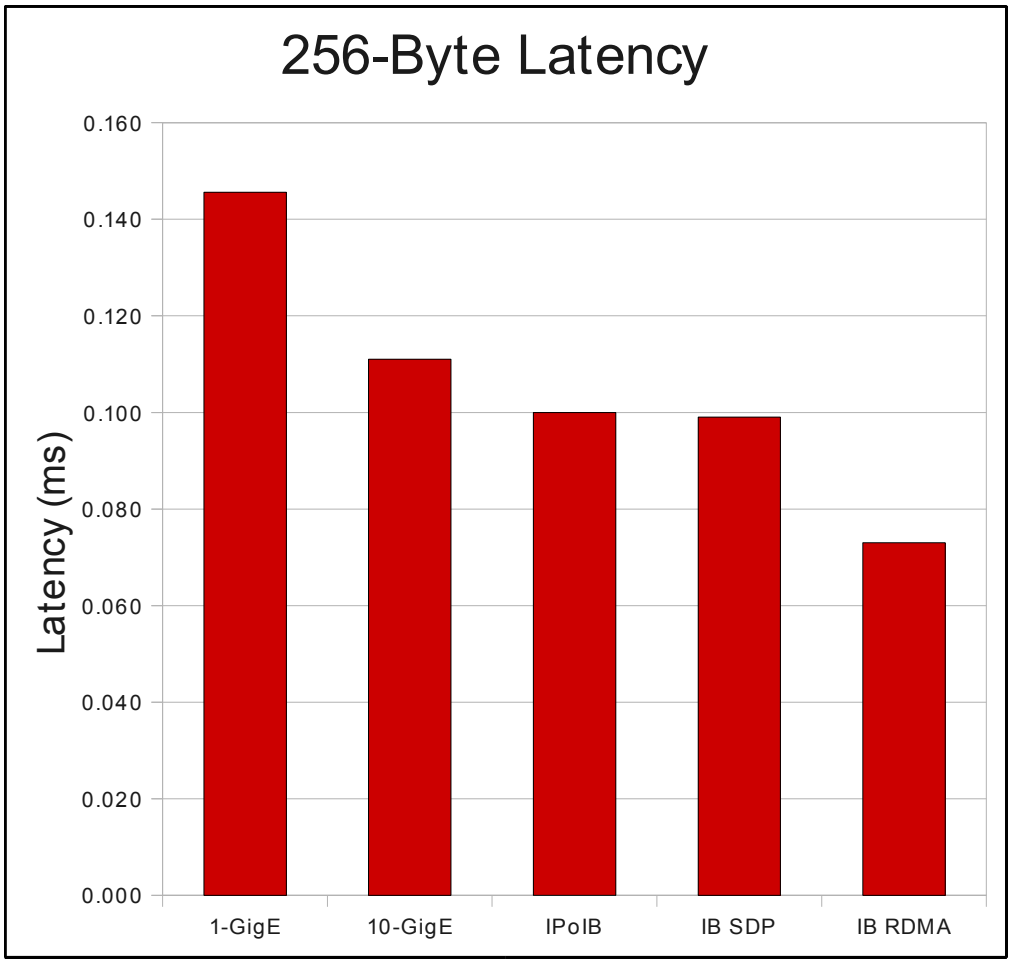
Illustrating trading semantics

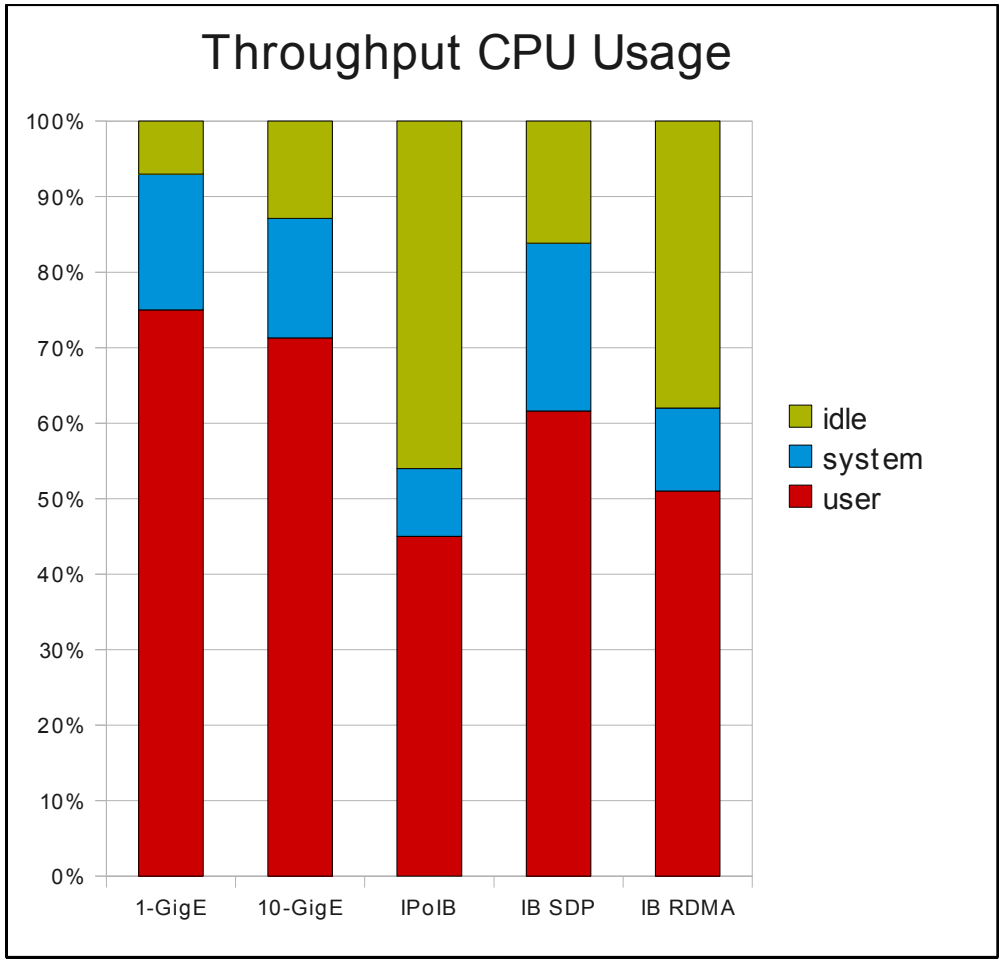
-- example consumer --

```
[MARKET] Symbol:NASDAQ.GOOG    Volume: 39350   Hi:125      Lo:113      MktCap:35796M      SEQ[485]
[TICKER] Symbol:NYSE.RHT       Price[20]   [0] [--]
[MARKET] Symbol:NYSE.RHT       Volume: 43165   Hi:24       Lo:8        MktCap:3800M       SEQ[486]
[TICKER] Symbol:NYSE.IBM       Price[37]   [1] [UP]
[MARKET] Symbol:NYSE.IBM       Volume: 36640   Hi:53       Lo:36       MktCap:49580M      SEQ[487]
[TICKER] Symbol:NASDAQ.MSFT    Price[25]   [1] [UP]
[MARKET] Symbol:NASDAQ.MSFT    Volume: 38089   Hi:26       Lo:8        MktCap:222250M     SEQ[488]
[TICKER] Symbol:NASDAQ.CSCO    Price[35]   [1] [UP]
[MARKET] Symbol:NASDAQ.CSCO    Volume: 39998   Hi:50       Lo:34       MktCap:205100M     SEQ[489]
[TICKER] Symbol:NASDAQ.YHOO    Price[8]    [0] [--]
[MARKET] Symbol:NASDAQ.YHOO    Volume: 38346   Hi:15       Lo:2        MktCap:11120M      SEQ[490]
[TICKER] Symbol:NASDAQ.GOOG    Price[114]  [0] [--]
[MARKET] Symbol:NASDAQ.GOOG    Volume: 40284   Hi:125      Lo:113      MktCap:35796M      SEQ[491]
[MARKET] Symbol:NYSE.RHT       Volume: 43989   Hi:24       Lo:8        MktCap:4180M       SEQ[492]
[TICKER] Symbol:NYSE.RHT       Price[22]   [2] [UP]
[MARKET] Symbol:NASDAQ.MSFT    Volume: 46230   Hi:26       Lo:8        MktCap:151130M     SEQ[596]
[MARKET] Symbol:NYSE.IBM       Volume: 43605   Hi:53       Lo:32       MktCap:42880M      SEQ[595]
[TICKER] Symbol:NASDAQ.MSFT    Price[23]   [2] [DOWN]
[TICKER] Symbol:NYSE.IBM       Price[37]   [0] [--]
[MARKET] Symbol:NASDAQ.CSCO    Volume: 47550   Hi:50       Lo:27       MktCap:158220M     SEQ[597]
[MARKET] Symbol:NYSE.RHT       Volume: 52990   Hi:28       Lo:8        MktCap:5320M       SEQ[594]
[TICKER] Symbol:NASDAQ.CSCO    Price[34]   [1] [DOWN]
[TICKER] Symbol:NYSE.RHT       Price[22]   [0] [--]
[MARKET] Symbol:NASDAQ.YHOO    Volume: 45910   Hi:15       Lo:2        MktCap:8340M       SEQ[598]
[TICKER] Symbol:NASDAQ.YHOO    Price[9]    [1] [UP]
[TICKER] Symbol:NYSE.IBM       Price[37]   [0] [--]
[MARKET] Symbol:NASDAQ.GOOG    Volume: 46082   Hi:125      Lo:111      MktCap:36110M      SEQ[599]
[TICKER] Symbol:NASDAQ.GOOG    Price[112]  [2] [DOWN]
```

Selecting the network fabric:

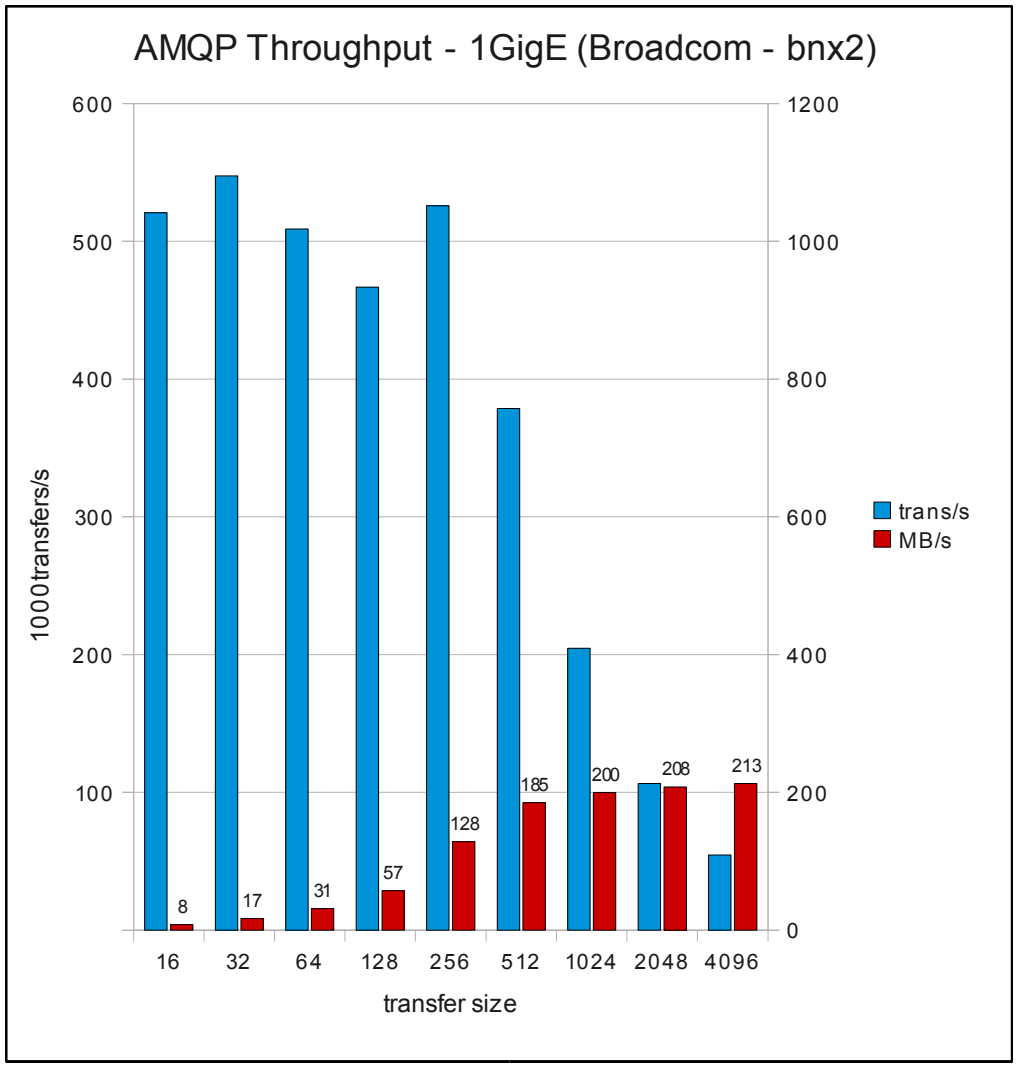
Comparing Latency per technology, per CPU cost at full load.

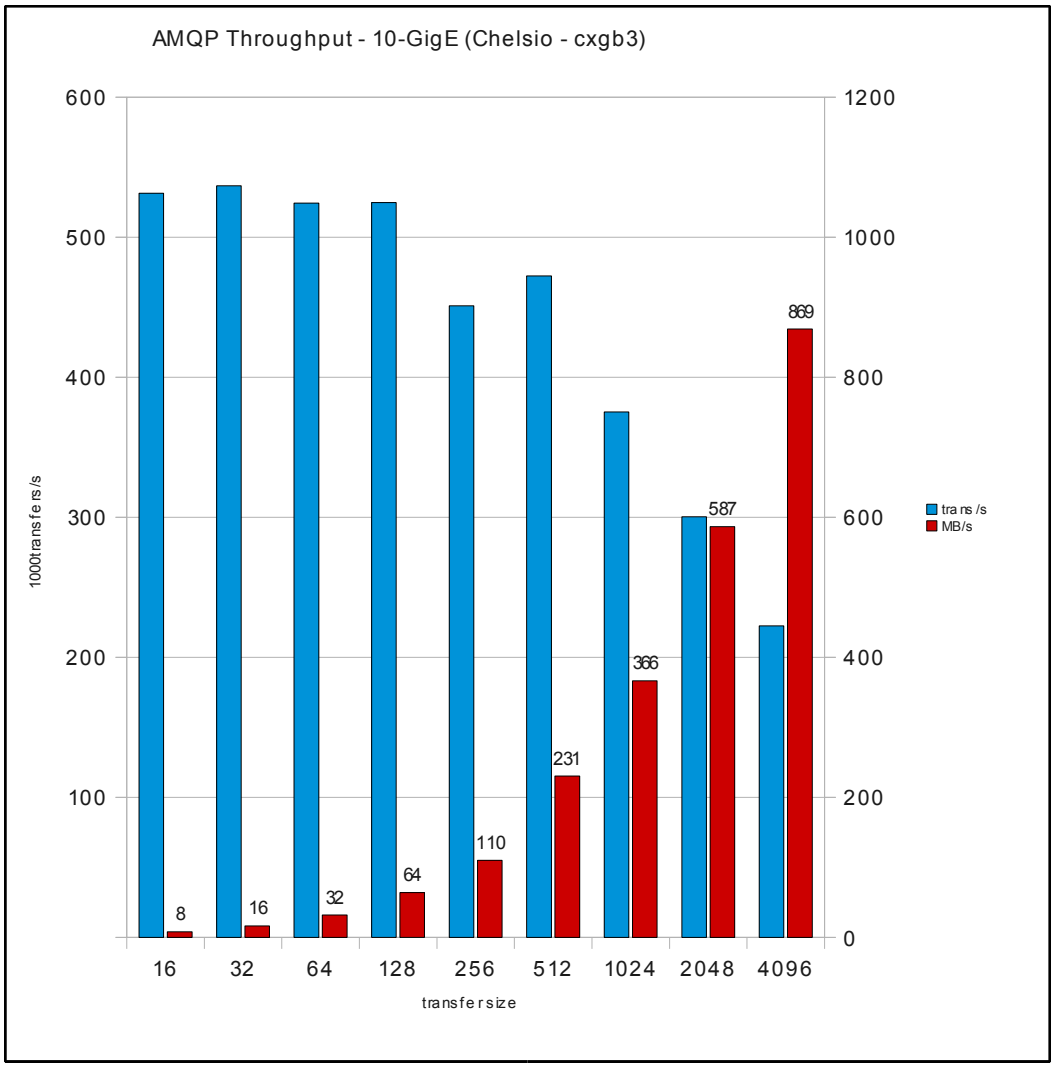




All measurements are AMQP between 3 peers (brokered) and fully reliable

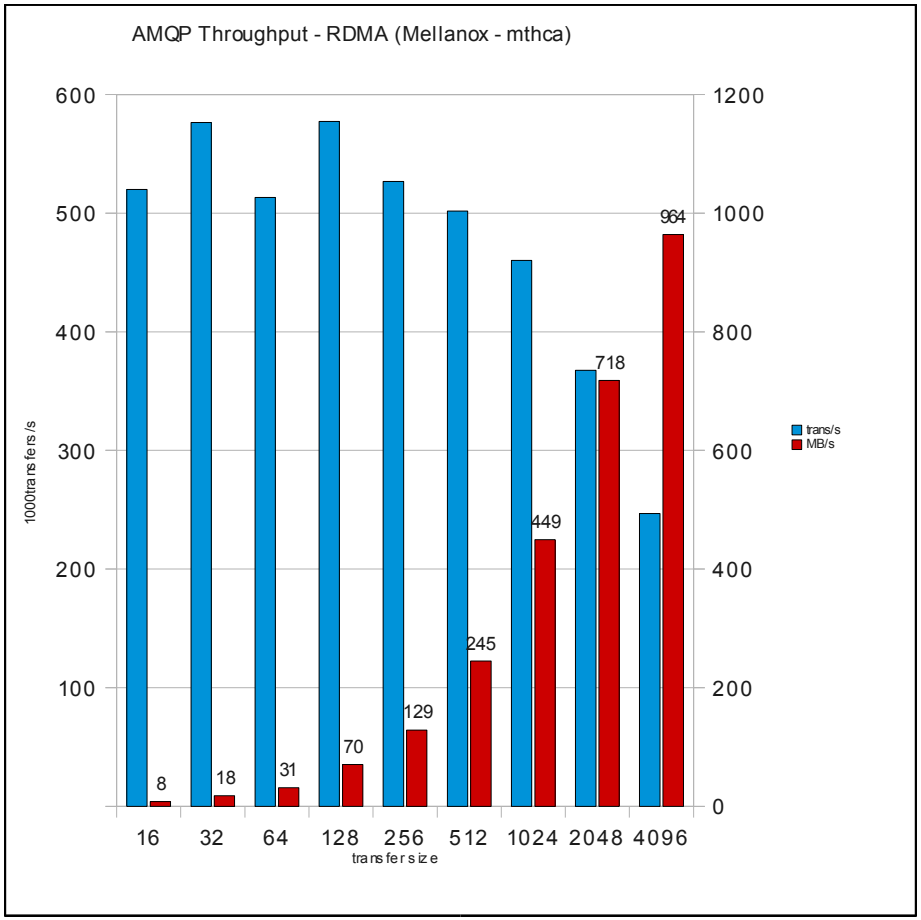
1 Gig versus 10 Gig, non-RDMA

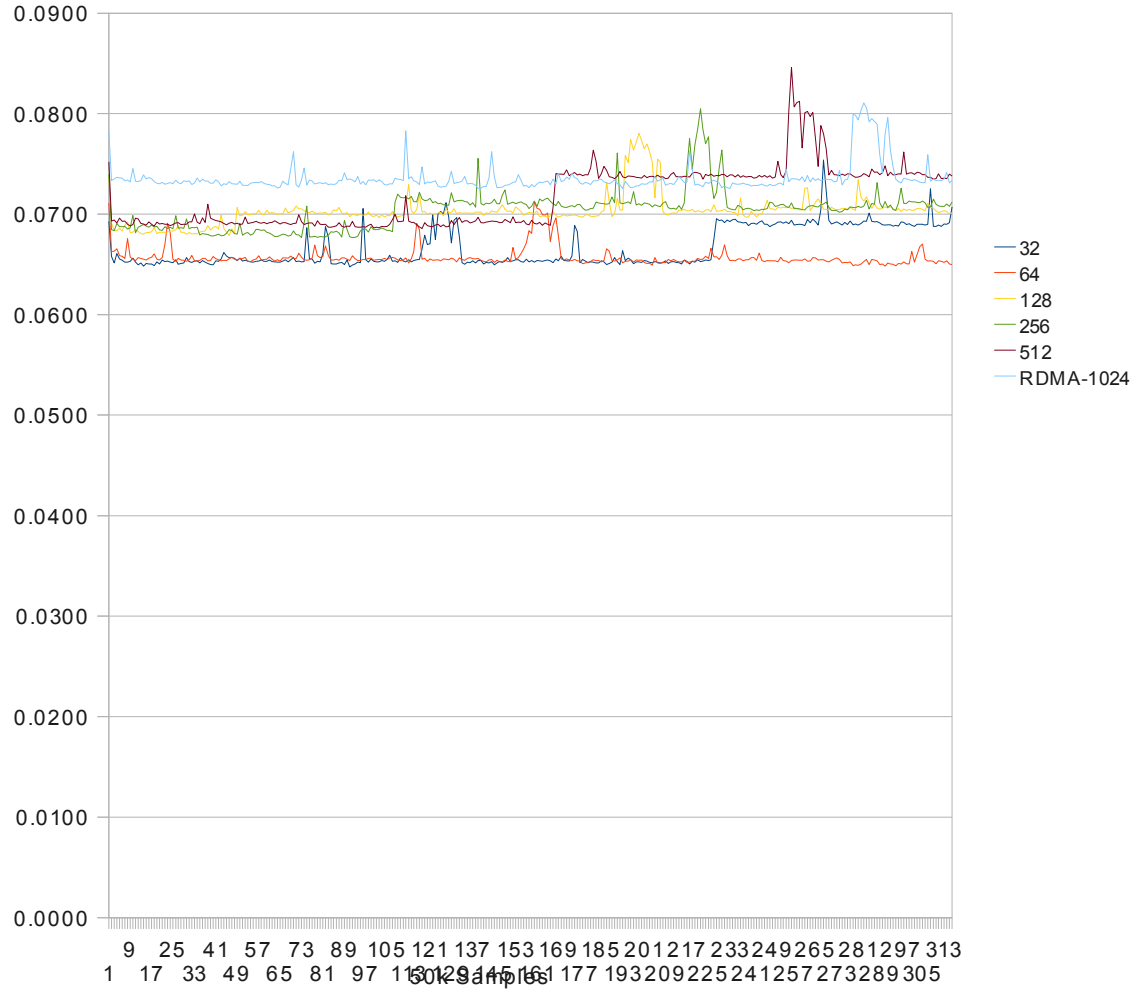




Rates and Throughput for 1 & 10G -- same load for direct comaparison

Messaging with native RDMA transport





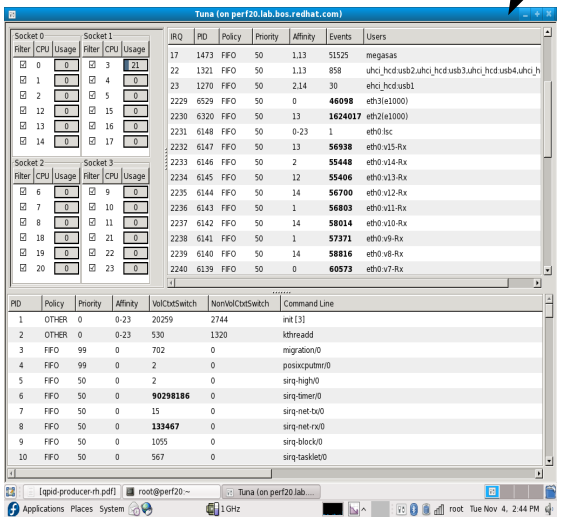
Rates, Throughput & Latency plot

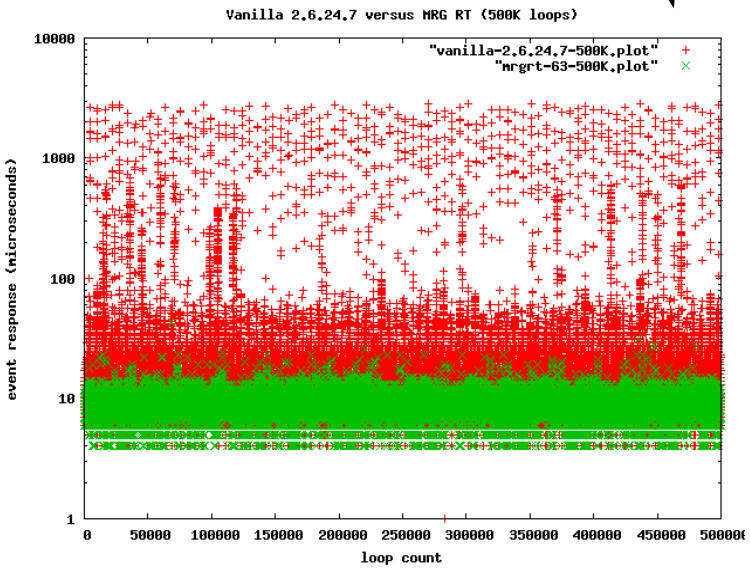
Dealing with other latency factors:

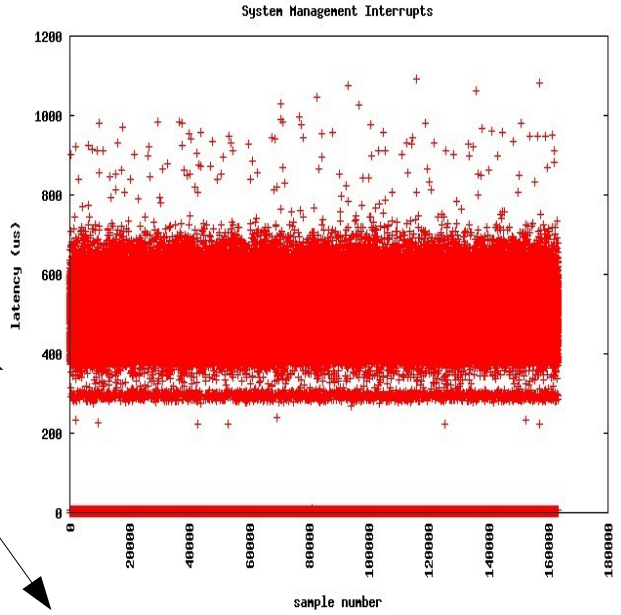
Impact of Realtime, SMIs, NUMA, Tuning, etc

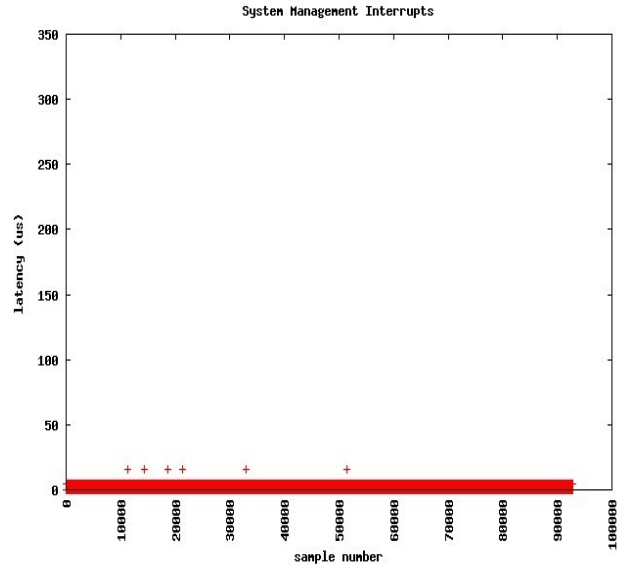
Market Data needs good latency & required determinism, which means each components needs to be able to deliver. (A hardware effect will 'spot' through all the layers for example)

- Two graphs on right show dealing with SMI's on hardware (same box, with and without SMIs)
- Graph center below, contrasts kernel schedule latency from RHEL to MRG-Realtime
- Image left below, MRG-tuna for setting up affinity, memory effects etc









Swapping your transport

-- no code changes --

$./qpidd –help

...

-- transport (tcp) The transport for which to return the port

-- load-module (file) Specifies additional module(s) to be loaded

...

... two of these options allow for the loading of modules and setting a transport, more than one can ran at a time

TIP: *./qpidd –load-module some_module.so –help* will show the help options for the loaded module

Now we start the broker with RDMA module loaded and specified as default.

$./qpidd –load-module rdma.so –transport rdma

Note: that SSL, clustering, federation, ACL, store, XQuery routing etc can all be loaded in the same way.