Design and Evaluation of Benchmarks for Financial Applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand*

Hari Subramoni, Gregory Marsh, Sundeep Narravula, Ping Lai, and Dhabaleswar K. Panda

Department of Computer Science and Engineering, The Ohio State University

{subramon, marshgr, narravul, laipi, panda}@cse.ohio-state.edu

Abstract

Message Oriented Middleware (MOM) is a key technology in financial market data delivery. In this context we study the Advanced Message Queuing Protocol (AMQP), an emerging open standard for MOM communication. We design a basic suite of benchmarks for AMQP's Direct, Fanout, and Topic Exchange types. We then evaluate these benchmarks with Apache Qpid, an open source implementation of AMQP. In order to observe how AMQP performs in a real-life scenario, we also perform evaluations with a simulated stock exchange application adapted to Apache Qpid. All our evaluations are performed over 1 Gigabit Ethernet as well as InfiniBand networks. Our results indicate that in order to achieve the high scalability requirements demanded by high performance computational finance applications, we need to use modern communication protocols, like RDMA, which place less processing load on the host. We also find that the centralized architecture of AMQP presents a considerable bottleneck as far as scalability is concerned.

Keywords: *AMQP, Computational Finance, InfiniBand*

1 Introduction

Message Oriented Middleware (MOM) plays a key role in financial data generation and delivery. The strength of MOM is that it allows for efficient communication between applications situated on heterogeneous operating systems and networks. MOM allows developers to by-pass the costly process of enabling explicit connections to these varied systems and networks. Instead applications need only communicate with the MOM. Typical MOM implementations feature asynchronous message delivery between unconnected applications via a message queue framework. However, there are prominent MOM implementations that operate without a queue framework [22].

Advanced Message Queue Protocol (AMQP) originated in the financial services industry in 2006 [1] [11] [23]. AMQP is an open standard for MOM communication. AMQP grew out of the need for interaction between MOM systems both within, and between, corporate enterprises. Due to the proliferation of proprietary, closed-standard, messaging systems such integration is considered challenging. As such, the primary goal of AMQP is to enable better interoperability between MOM implementations. Since AMQP's inception, several, open-source, messaging software distributions have emerged.

In this paper we evaluate one such AMQP distribution, Apache Qpid [2]. Our evaluation draws on the Ohio State University Network-Based Computing Laboratory's experience with the Message Passing Interface (MPI) standard [9]. MPI is used extensively in the scientific, High Performance Computing (HPC) arena. Our laboratory's main software product, MVAPICH [10], is an open source adaptation of MPI to HPC networks such as InfiniBand. As such, we are particularly interested in messaging performance with high speed interconnects. This paper's main contributions are:

- The design of a set of benchmarks for AMQP.

- Implementation of these AMQP benchmarks with the Apache Qpid C++ API, and their evaluation on 1 Gigabit Ethernet and InfiniBand networks.

- Evaluation of a Stock Market Simulation, adapted to Apache Qpid on these networks.

We designed our benchmarks to evaluate the various communication models offered by AMQP. Three variables inherent in any AMQP communication model are the number of Publishers (senders), the number of Consumers (receivers), and the Exchange type (message routing engine). Each of our benchmarks exercises one or more of these variables. In particular our

---

benchmarks focus on AMQP's Direct, Fanout, and Topic Exchange types. Furthermore our benchmarks measure performance for data capacity, message rate, and speed.

Our experiments achieved a 350 MB/sec throughput using a basic AMQP Direct Exchange using IPoIB (TCP/IP implementation over InfiniBand). With an increased number of Consumers receiving messages, we found that the achievable message rate decreases for all Exchange types. Further investigation showed that an increased CPU utilization creates a performance bottleneck on the AMQP Broker. In our experiments with SDP, we found that due to the low Kernel stack overhead of SDP (Sockets Direct Protocol over InfiniBand), we were able to obtain higher throughput. Using the TCP Nagle algorithm moderately increased IPoIB performance with the stock exchange application at high message generation rates. However, the algorithm worsens performance at lower message generation rates. Overall our results indicate the need for better AMQP Broker designs, including a more distributed Broker scheme to alleviate AMQP's centralized Broker bottleneck. Our results also indicate the need to use modern communication protocols, like RDMA, which impose less of a processing load on host resources.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of AMQP and InfiniBand technologies. In Section 3, we describe the design of our AMQP Benchmarks. Section 4 presents the experimental results of our Benchmark tests. Section 5 describes our tests with a Stock Market Simulation application. Section 6 overviews related work. Finally we summarize our conclusions and possible future work in Section 7.

2 Background

In this section we provide a brief overview of Advanced Message Queuing Protocol and InfiniBand.

2.1 Advanced Message Queuing Protocol

Figure 1 shows the general architecture of an AMQP compliant messaging system. An AMQP messaging system consists of 3 main components: Publisher(s), Consumer(s) and Broker/Server(s). Each component can be multiple in number and be situated on independent hosts. Publishers and Consumers communicate with each other through message queues bound to exchanges within the Brokers. AMQP provides reliable, guaranteed, in-order message delivery. We briefly explain the functionality of each component below.

Broker/Server: A server or daemon program that contains one or more Virtual Hosts, Exchanges, Message Queues, and Bindings.

Virtual Host: A name space that groups and identifies a set of Exchanges, Message Queues, and Bindings.

Consumer: An application that declares one or more Message Queues on the Broker and attaches the queue(s) to one or more Exchanges with Bindings.

Message Queue: A data structure that stores messages in memory or on disk, and delivers these in sequence to the Consumer that declared the queue. Each Message Queue is entirely independent.

Binding: A rule which helps the Exchange decide to which Message Queues (and therefore to which Consumers) it needs to copy a message. A Binding typically is identified with a text string known as a Binding Key.

Publisher/Producer: An application that constructs messages with Routing Keys and sends the messages to an Exchange on a Broker.

Exchange: A matching and routing engine which accepts messages from Publishers. An Exchange then copies the messages into zero or more Queues by matching the messages' Routing Keys against the Queues' Binding Keys. Exchanges are classified into types based on the kind of key matching they perform. Of the Exchange types that AMQP supports, we evaluate the following: Direct, Fanout, and Topic. Details of each Exchange type's operation follow in Section 3.

2.2 InfiniBand Architecture

InfiniBand Architecture (IB) [4] is an industry standard for low latency, high bandwidth, System Area Networks (SAN). An increasing number of InfiniBand network clusters are being deployed in high performance computing (HPC) systems as well as in E-Commerce-oriented data centers. IB supports two types of communication models: Channel Semantics and Memory Semantics. Channel Semantics involve discrete send and receive commands. Memory Semantics involve Remote Direct Memory Access (RDMA) [18] operations. RDMA allows processes to read or write the memory of processes on a remote computer without interrupting that computer's CPU. Within these two communication semantics, various transport services are available that combine reliable/unreliable, connected/unconnected, and/or datagram mechanisms.

The popular TCP/IP network protocol stack can be adapted for use with InfiniBand by either the IP over IB (IPoIB) driver or the Sockets Direct Protocol (SDP) [7]. IPoIB is a Linux kernel module than enables InfiniBand hardware devices to encapsulate IP packets into IB datagram or connected transport services. When IPoIB is applied, an InfiniBand device is assigned an IP address and accessed just like any regular TCP/IP hardware device. SDP contains a kernel module and a software library that allow applications written with TCP sockets to transparently use IB without re-writing existing code.
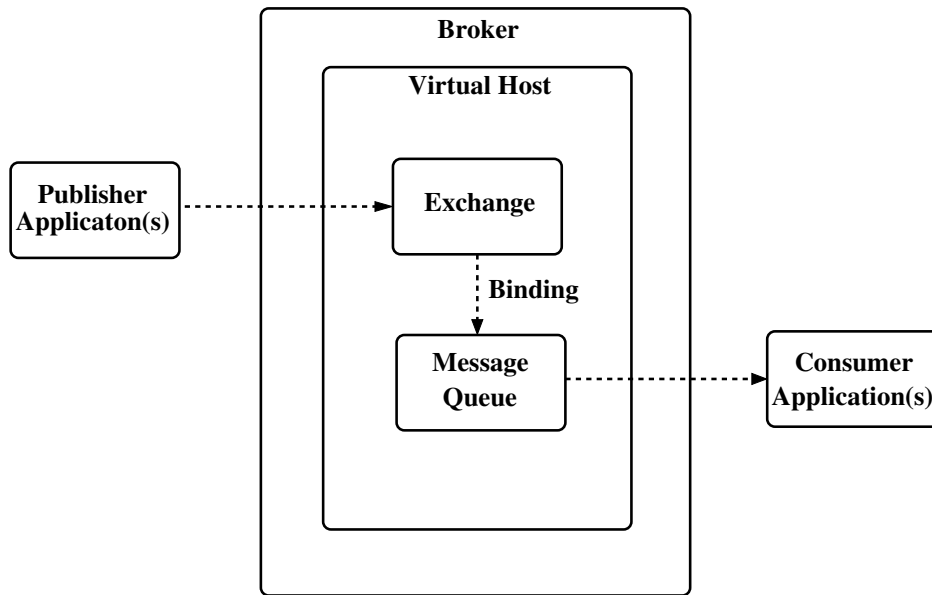
Figure 1. The AMQP Architecture

During SDP-enabled execution, an application's TCP sockets are automatically replaced with SDP sockets. Although IPoIB and SDP are similar in purpose, they provide different modes of execution and in some cases SDP can provide lower latency and better throughput than IPoIB.

3 Design of AMQP Benchmarks

In this section we describe the design of our AMQP benchmarks which are modeled after the OSU Micro-benchmarks for MPI [14]. One thing to note here is that, unlike the OSU benchmarks, our benchmarks do not assume a direct, one link, point-to-point network connection. Within AMQP, a message must always traverse the Broker host in route to the destination Consumer. This incorporates at least two network links into the travel path of any message.

Three variables inherent in any AMQP operation are the number of Publishers, the number of Consumers, and the Exchange type. Each of our benchmarks exercises one or more of these variables. Furthermore each of our benchmarks measures performance for data capacity, message rate, and speed. Data capacity is the amount of raw data in MegaBytes (MB) that may be transmitted per second, irrespective of the number of messages. This is also known as *Bandwidth*. Message rate is similar to data capacity, but measures the number of discrete messages transmitted per second. Message rate is also known as *Throughput*. Speed is the average time one message takes to travel from the publisher to the consumer. This speed measure is commonly referred to as *Latency*.

3.1 Direct Exchange - Single Publisher Single Consumer (DE-SPSC) Benchmark

This benchmark tests basic Publisher-Consumer message transmission across AMQP's direct exchange type. This test is analogous to a network point-to-point test. The direct exchange type provides routing of messages to zero or more queues based on an exact match between the Routing Key of the message, and the Binding Key used to bind the Queue to the Exchange. So the Exchange performs a text string equality computation for each message.

The Publisher sends out a pre-defined number of messages, with sizes varying from 1 Byte to 1 MegaByte (MB), to a Direct exchange running on a single Broker host. Once the Consumer has received the pre-defined number of messages of a particular size, it sends back a 0 byte reply to the Publisher. Upon receiving this reply, the Publisher computes the performance metrics for that message size, and then starts transmitting the next set of messages to the Consumer. (Note: to receive reply messages from the Consumer, the Publisher is also a Consumer with its own receive queue. Unlike the Consumer, the publisher only processes a small number of these reply messages.)

3.2 Direct Exchange - Multiple Publishers Multiple Consumers (DE-MPMC) Benchmark

This benchmark tests the scalability of the AMQP architecture with the Direct Exchange type. Here we use multiple, independent, Publisher-Consumer pairs. These pairs simultaneously communicate with each other through the same Direct Exchange residing on a single Broker host. This increases text string equality computations, as well as network connection overhead on the Broker host. The operations of this benchmark are the same as DE-SPSC, just with a higher number of Publishers and Consumers.

3.3 Direct Exchange - Ping Pong (DE-PP) Benchmark

This benchmark tests round trip transmission between a Publisher-Consumer pair. This test is similar to the sending of market trade orders and the receipt of order confirmations.

The Publisher sends out a pre-defined number of messages, with sizes ranging from 1 Byte to 1 MegaByte (MB), to a Direct exchange running on a single Broker host. However after sending a single message, the publisher waits for a reply from the Consumer. Upon receiving one message from the Publisher, the Consumer sends a same sized reply back to the Publisher. When a reply for each sent message is received, the Publisher computes the performance metrics for that message size, and then starts transmitting the next set of messages to the Consumer.

3.4 Fanout Exchange - Single Publisher Multiple Consumers (FE-SPMC) Benchmark

This benchmark tests Fanout Exchange delivery to a varying number of Consumers. The Fanout Exchange is similar to the traditional multicast model of network transmission. A Fanout Exchange routes messages to all bound queues regardless of the message's Routing Key. Therefore it does not have the text string matching overhead of the Direct Exchange type.

The Publisher sends out a pre-defined number of messages, with sizes varying from 1 Byte to 1 MegaByte (MB), to a Fanout exchange running on a single Broker host. Upon receiving a message, the exchange copies the message to all queues which have been bound by a pre-defined number of Consumers. Once each Consumer has received the pre-defined number of messages of a particular size, it sends back a reply to the Publisher. After receiving a reply from all Consumers, the Publisher computes the performance metrics for that message size, and then starts transmitting the next set of messages to the Consumers. To understand how the Fanout Exchange scales, this test may then be repeated with an increased number of Consumers.

3.5 Topic Exchange - Single Publisher Single Consumer (TE-SPSC) Benchmark

This benchmark tests Topic Exchange delivery with a varying amount of Binding Key topics. A Topic Exchange routes a message to bound queues if the message's Routing Key matches a pattern provided by the Binding Key. For example a Routing Key of "news.usa" would match a Binding Key of "news.*". So the exchange's Broker host incurs the computational overhead of text string pattern matching for each message.

The Publisher sends out a pre-defined number of messages, with sizes varying from 1 Byte to 1 MegaByte (MB), to a Topic exchange running on a single Broker host. Each message has the same Routing Key. A pre-defined number of Consumers have bound their queues to the Exchange using different Binding Key patterns. However, only one of these Binding Keys will pattern match the messages' Routing Key. Therefore only one Consumer will receive messages. Once this Consumer has received the pre-defined number of messages of a particular size, it sends back a reply to the Publisher. Upon receiving this reply, the Publisher computes the performance metrics for that message size, and then starts transmitting the next set of messages. To understand the pattern matching overhead incurred by the Exchange, the number of non-match, Binding Key patterns may be increased. Therefore the Exchange must work through an increased number of failed pattern matches to find the one true match.

4 Experimental Results

In this section, we present results from our evaluation of Apache Qpid over the 1 Gigabit Ethernet (1 GigE) and InfiniBand networks.

4.1 Experimental Setup

Figure 2 shows the basic setup which we used to conduct our tests. We use a cluster consisting of Intel Xeon Quad dual-core processor host nodes. Each node has 6GB RAM and is equipped with a 1 GigE Network Interface Controller (NIC), as well as with an InfiniBand Host Channel Adapter (HCA). The IB HCAs are DDR ConnectX using Open Fabrics Enterprise Distribution (OFED) 1.3 [12] drivers. The operating system for each node is Red Hat Enterprise Linux 4U4.

4.2 Basic Performance

To establish the maximum IPoIB performance that we might achieve in our experimental setup, we first performed a low-level network test using Socket Benchmarks [20]. This test is independent of AMQP and establishes baseline network performance. Since the Broker has to service both the Publisher(s) and Consumer(s) at the same time, the Socket Benchmarks have each host send and receive packets to/from the Broker host. This tests how multiple streams affect basic performance across the Broker host. Our Socket Benchmark tests established a average maximum bandwidth of *550* MBps using IPoIB and *650* MBps using SDP for our experimental setup.
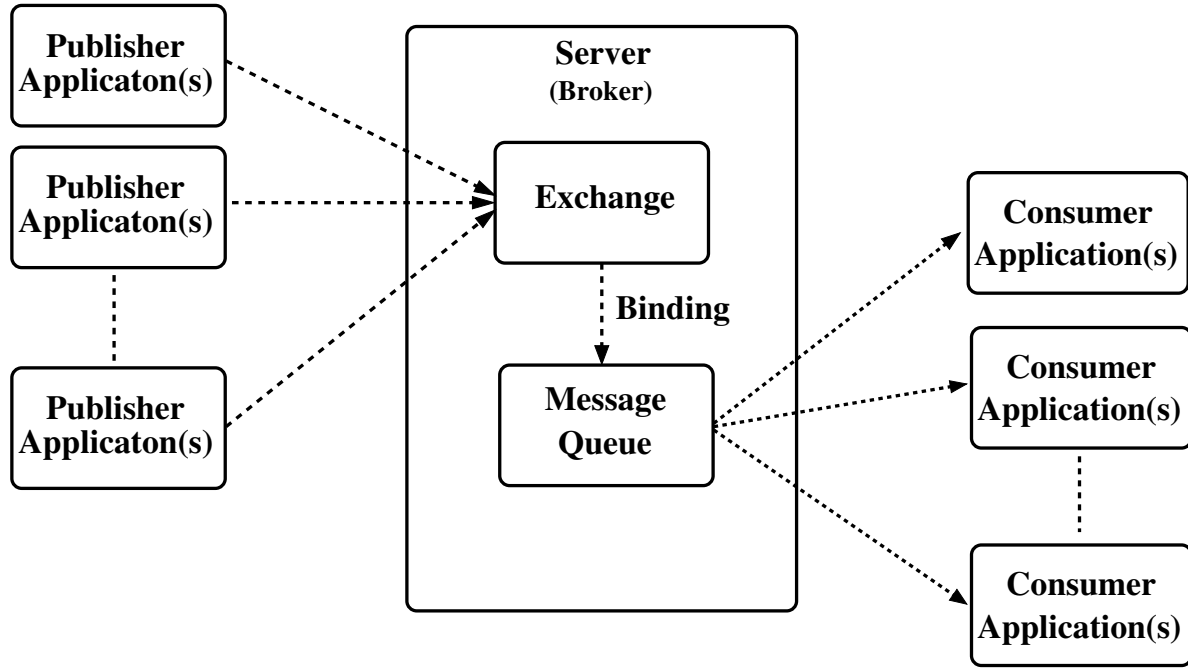
Figure 2. Experimental Setup

4.3 Benchmark Tests

4.3.1 Direct Exchange - Single Publisher Single Consumer (DE-SPSC) Benchmark
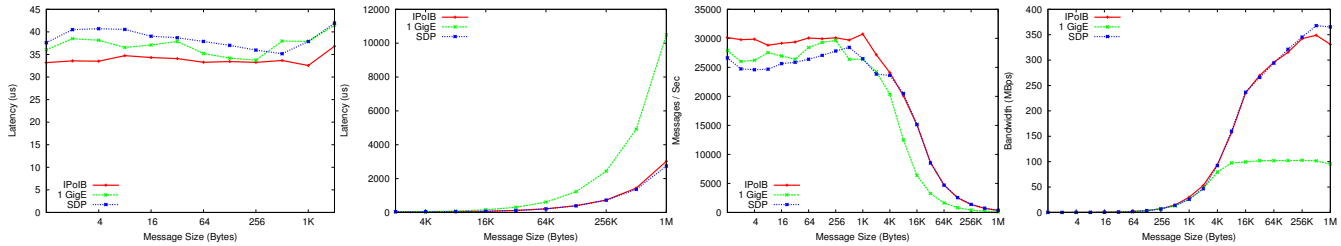

Figure 3. (a) DE-SPSC Small Message Latency, (b) DE-SPSC Large Message Latency, (c) DE-SPSC Message Rate, and (d) DE-SPSC Bandwidth

Figures 3 (a), (b) and (c) show the speed and message rates achieved for varying message sizes using the *DE-SPSC* Benchmark over IPoIB, 1 GigE and SDP, respectively. As we can see, for small messages, IPoIB achieves the best latency. Where as for larger messages, SDP achieves better latency. SDP requires a larger connection setup time as compared to IPoIB. As a result, the connection setup time dominates the total data transfer time for smaller messages resulting in higher latencies when we use SDP. This behavior of SDP has been well studied in [15].

Figures 3 (d) show the bandwidth achieved for varying message sizes using the *DE-SPSC* Benchmark over IPoIB, 1 GigE and SDP.

4.3.2 Direct Exchange - Multiple Publishers Multiple Consumers (DE-MPMC) Benchmark

Figure 4 (a) shows the bandwidth (data capacity) curve with multiple Publishers-Consumers pairs. The 1 GigE interface saturates the link early on and hence shows no variation. For IPoIB, we are able sustain performance up to four simultaneous Publisher-Consumer pairs. But, performance drops drastically as we increase the number of pairs up to eight.

To gain more insights into this drop in performance, we looked at the CPU utilization on the Publisher, Consumer and the Broker while running the *DE-MPMC* benchmark. Figure 4 (b) shows the normalized CPU utilization for a varying number of Publisher-Consumer pairs. It is to be noted that our experiments are performed on machines with multiple (8) processing cores. Hence, CPU utilization of more than 100% is not unexpected. As we can see, with an increasing number of pairs, the CPU utilization on the Broker shows a linear increase. With the kind of scalability that is expected of High Performance Finance applications these days, there is a strong motivation for us to explore better designs for the Broker, including distributing the functionality of the Broker among multiple cores/machines. As pointed out in [15], lower CPU utilization is the reason why SDP is able to maintain performance for larger number of Publisher-Consumer pairs.
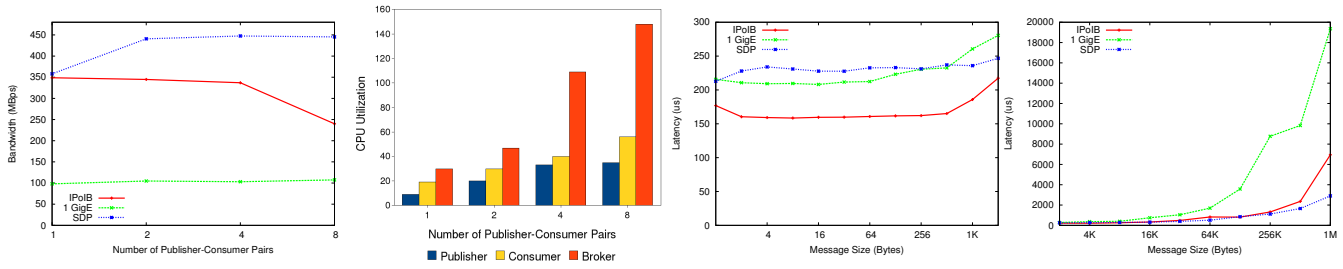
Figure 4. (a) DE-MPMC Bandwidth, (b) DE-MPMC CPU Utilization, (c) DE-PP Small Message Latency, and (d) DE-PP Large Message Latency

In this context modern high performance protocols such as RDMA, which incurs very low overhead on the host CPU will help. Figure 5 compares the MPI level message rate achieved over IPoIB as well as Native IB Verbs. As we can see, message rates achieved over RDMA using Native IB verbs are an order of magnitude higher than what we can achieve using IPoIB. This coupled with the fact that the communication using RDMA incurs very low overhead on the host CPU makes this an ideal choice for the underlying protocol.
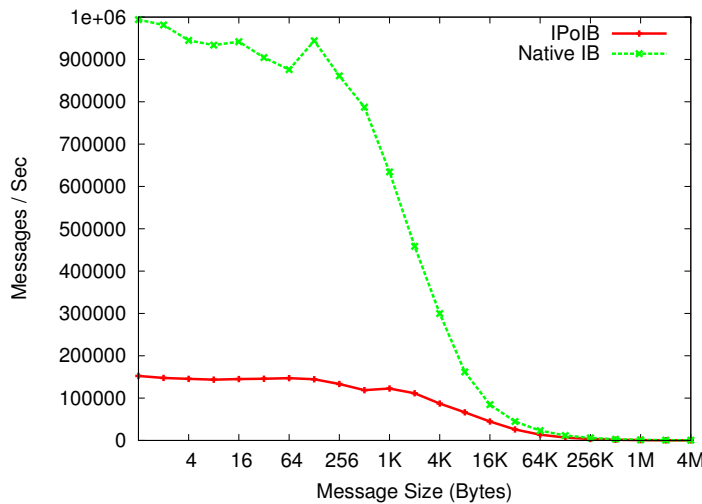

Figure 5. MPI Level Message Rate

4.3.3 Direct Exchange - Ping Pong (DE-PP) Benchmark

Figures 4 (c) and (d) show the Ping Pong latency measured between one Publisher-Consumer pair for small messages and for large messages, using IPoIB, 1 GigE and SDP. The graphs show the same trends as seen in the Section 4.3.1.

4.3.4 Fanout Exchange - Single Publisher Multiple Consumers (FE-SPSC) Benchmark

Figure 6 (a) shows a comparison of the number of messages sent over IPoIB, SDP, and 1 GigE for a fanout factor of 32. Figures 6 (b), (c), and (d) show the number of messages sent per second using Fanout Exchange with a varying fanout factor over IPoIB, 1 GigE and SDP, respectively.
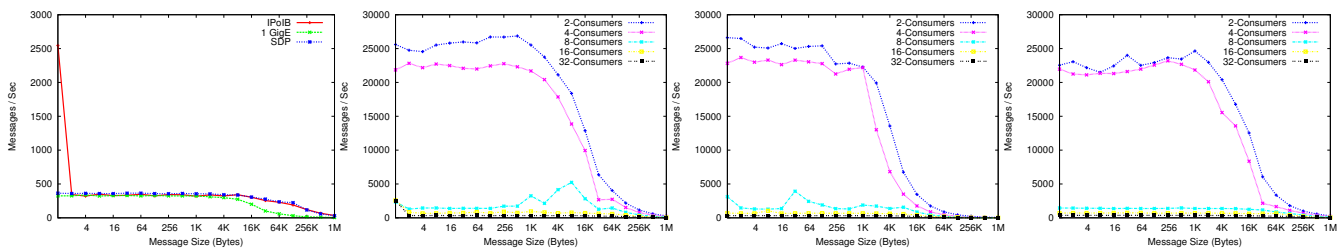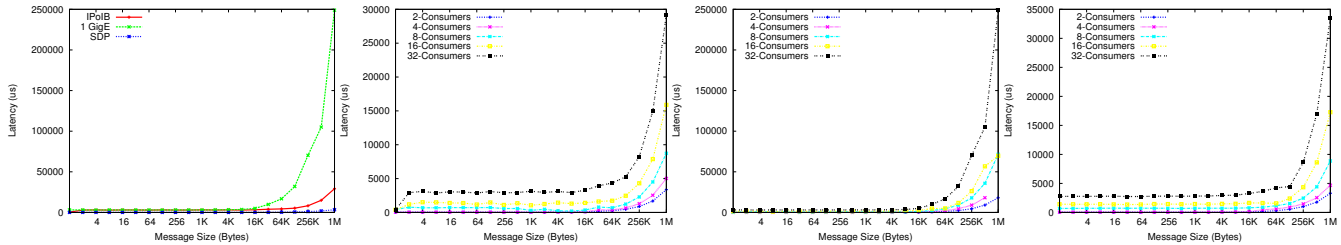

Figure 6. FE-SPSC Performance with Fanout up to 32: (a) Message Rate comparison with fanout of 32, (b) IPoIB Message Rate, (c) 1 GigE Message Rate, and (d) SDP Message Rate

Figure 7 (a) shows a comparison of the data capacity obtained over IPoIB, SDP, and 1 GigE for a fanout factor of 32. Figures 7 (b), (c), and (d) show the data capacity obtained using Fanout Exchange with a varying fanout factor over IPoIB, 1 GigE and SDP, respectively.
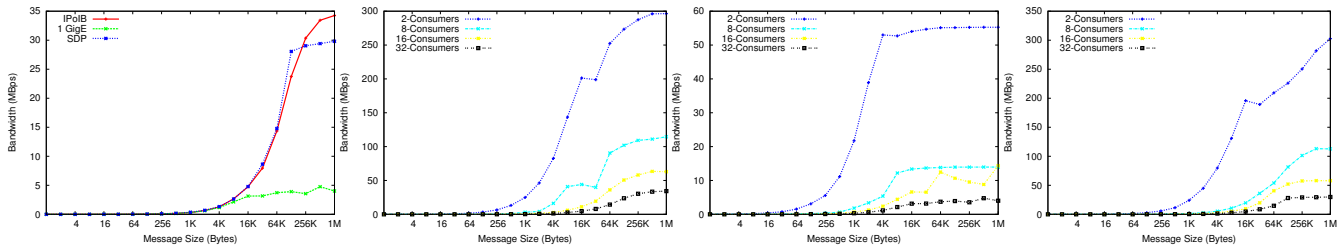
Figure 7. FE-SPSC Performance with Fanout up to 32: (a) Bandwidth comparison with fanout of 32, (b) IPoIB Bandwidth, (c) 1 GigE Bandwidth, and (d) SDP Bandwidth

Figure 8 (a) shows a comparison of the latency seen over IPoIB, SDP, and 1 GigE for a fanout factor of 32. Figures 8 (b), (c), and (d) show latency seen using Fanout Exchange with a varying fanout factor over IPoIB, 1 GigE and SDP, respectively. The CPU utilization follows the same trends as seen in the DE-MPMC benchmark and hence are not included here.

Figure 8. FE-SPSC Performance with Fanout up to 32: (a) Latency comparison with fanout of 32, (b) IPoIB Latency, (c) 1 GigE Latency, and (d) SDP Latency

4.3.5   Topic Exchange - Single Publisher Single Consumer (TE-SPSC) Benchmark

Figure 9 (a) shows the comparison of the number of messages sent per second with 1024 Binding Key patterns in the Exchange between IPoIB, 1 GigE and SDP. Figures 9 (b), (c), and (d) show the number of messages sent per second sent with varying number of Binding Key patterns over IPoIB, 1 GigE and SDP, respectively. We can see that messages per second drops as the number of binding rules on the Exchange increase.
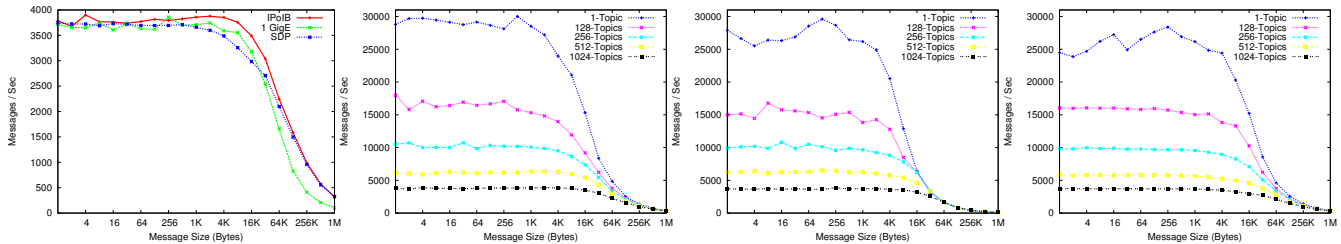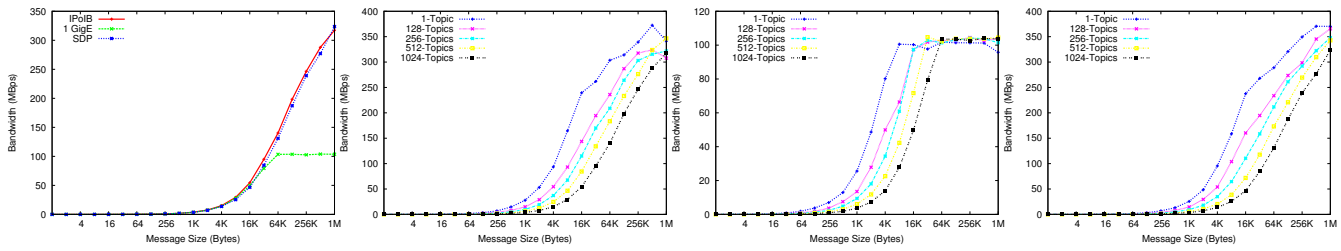

Figure 9. TE-SPSC Performance with up to 1024 Topics: (a) Message Rate comparison with 1024 Topics , (b) IPoIB Message Rate, (c) 1 GigE Message Rate, and (d) SDP Message Rate

Figure 10 (a) shows the comparison of the data capacity with 1024 Binding Key patterns in the Exchange between IPoIB, 1 GigE and SDP. Figures 10 (b), (c), and (d) show the data capacity achieved with varying number of Binding Key patterns over IPoIB, 1 GigE and SDP, respectively.
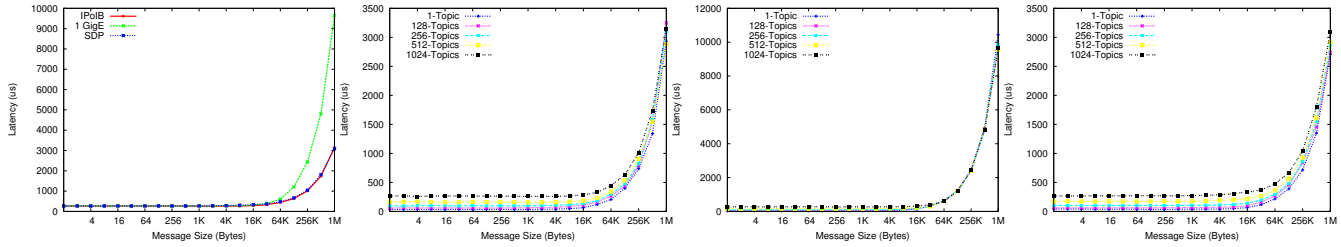
Figure 11 (a) shows the comparison of the latency seen with 1024 Binding Key patterns in the Exchange between IPoIB, 1 GigE and SDP. Figures 11 (b), (c), and (d) show the latency seen with varying number of Binding Key patterns over IPoIB, 1 GigE and SDP, respectively.

5   Stock Exchange Simulation Application and Evaluation

In this experiment we modified the stock exchange example application provided with the open source, ZeroMQ MOM distribution [25]. Our modifications were confined to having the application sending messages in Qpid's AMQP framework instead of ZeroMQ's framework.

Figure 10. TE-SPSC Performance with up to 1024 Topics: (a) Bandwidth comparison with 1024 Topics, (b) IPoIB Bandwidth, (c) 1 GigE Bandwidth, and (d) SDP Bandwidth

Figure 11. TE-SPSC Performance with up to 1024 Topics: (a) Latency comparison with 1024 Topics, (b) IPoIB Latency, (c) 1 GigE Latency, and (d) SDP Latency

5.1 Description

As Figure 12 shows, this application consists of three components: a gateway, a matching engine, and a statistics reporter. The gateway produces random, simulated buy and sell orders. It then sends the orders to the matching engine's order queue on the Qpid broker. The matching engine reads orders from the order queue and simulates a functional stock exchange by matching the prices in buy orders to the prices in sell orders. An exact match in prices results in trades. The matching engine produces three types of messages which are sent to the gateway's trade queue: trade confirmations, order confirmations, and price quotes. Order confirmations are produced when there is no matching price to make a trade. Price quotes occur when the stock's maximum bid or ask price changes due to trading activity.
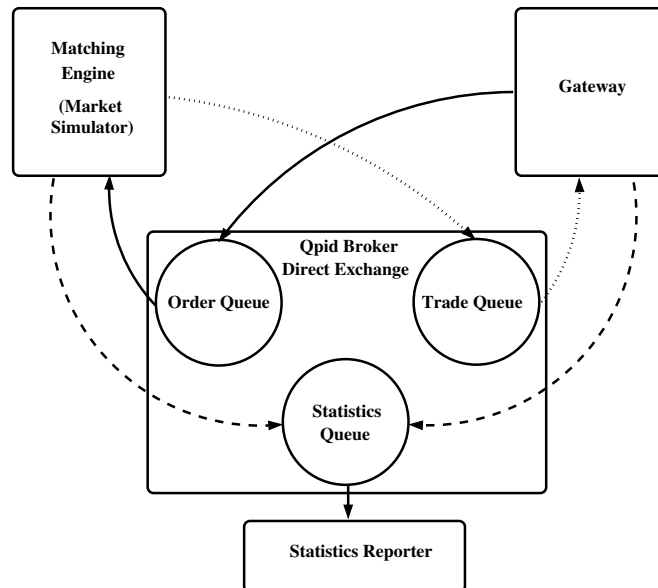
Figure 12. Stock Market Simulation Application

During the course of their operation, both the gateway and the matching engine produce performance measurement messages and send them via Qpid to the statistics reporter's queue. These measurements are taken with a sampling interval, which we set for every 10,000 orders. The statistic reporter writes these measurement messages to a file. The gateway produces the following measurements:

1. Orders per second sent from the gateway to the matching engine's order queue.

2. Sum of trade confirmations, order confirmations, and quotes received per second from matching engine via the gateway's trade queue.

3. Order Completion Time: Time in microseconds between when an order is sent to the order queue and when either a trade confirmation or order confirmation is received from the trade queue.

The matching engine produces the the following measurements:

1. Orders per second received from the gateway via the matching engine's order queue.

2. Sum of trade confirmations, order confirmations, and quotes sent per second from the matching engine to the gateway's trade queue.

5.2 Experimental Setup

For the purposes of our tests each component runs on an independent host with a Qpid Broker running on yet another independent host. Each component has its own Queue on the Broker's Direct Exchange and each Queue is bound with a unique Routing Key. The application creates order, order confirmation, trade, quote, throughput rate, and completion timestamp message types. Each message type has a different sized payload. We counted the number of each message type created during a typical run and calculated a weighted average message size of around 9 bytes. The application's performance may be tested by varying the number of random orders per second created on the gateway. In our tests we used creation rates of 1000, 2000, 4000, 6000, and 8000 orders/second and created 1,000,000 orders at each rate. We recorded the resulting measurements at each rate. Beyond 8000 orders/second the application began failing and would not record a full set of measurements.

5.3 Experimental Results

Figures 13 and 14 show our experimental results. Figure 13 (a) summarizes measurements (1.) and (2.) from both the gateway and matching engine. The lower line shows that measurement (1.) for both components is the same. The matching engine was able to read orders from its order queue at the same rate as the gateway was able to generate them. The upper line shows measurement (2.) for both components is the same. The gateway was able to read confirmations and quotes from its trade queue at the same rate as the matching engine was able to generate them.
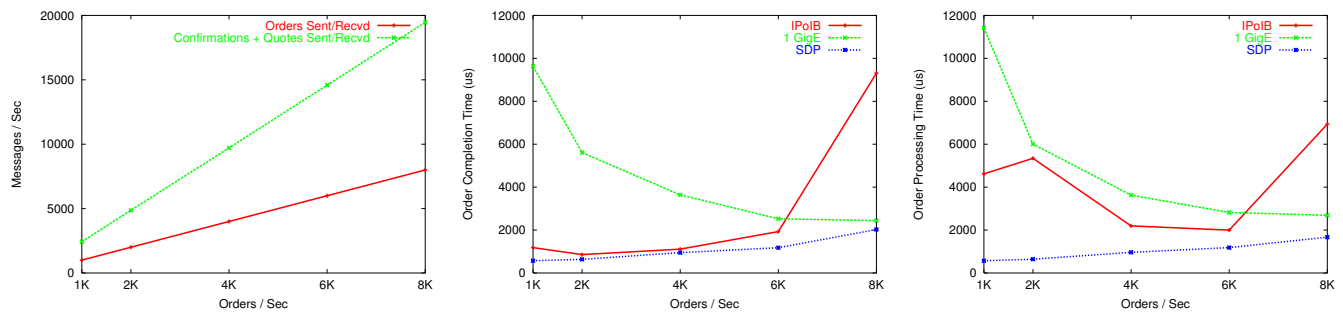
Figure 13. Performance of Stock Exchange Simulation: (a) Message Rate, (b) Order Completion Time without Nagle Algorithm, and (c) Order Completion Time with Nagle Algorithm

Figure 13 (b) summarizes gateway measurement (3.), order completion time. InfiniBand SDP achieved the best average order completion times across all order generation rates. However, Figure 14 (a) shows that SDP utilized a considerably higher amount of CPU on the Broker than the other network schemes. The other InfiniBand-TCP adaptation, IPoIB achieved better performance than regular TCP on 1 GigE for all generation rates except 8000 orders/sec. As shown in Figure 13 (b), IPoIB's average order processing time increases drastically at this generation rate. Our first attempt to explain this was to look for CPU utilization bottlenecks. However, as Figure 14 shows, IPoIB's CPU utilization never exceeds 35% on any component.

In an attempt to improve IPoIB performance, we re-ran our experiments without the "TCP No Delay" option on the Broker. While this did not not alter the number of messages at the Qpid application level, it did alter the number of packets at the the network level. The "TCP No Delay" option allows a sender to submit packets as fast as possible to the network. Omitting this option enables TCP's Nagle algorithm for all network transmissions. The Nagle algorithm prevents a sender from sending many small packets. If possible, it forces TCP to wait until there is enough data to send a full packet. This reduces TCP processing overhead by reducing the number of packets generated.
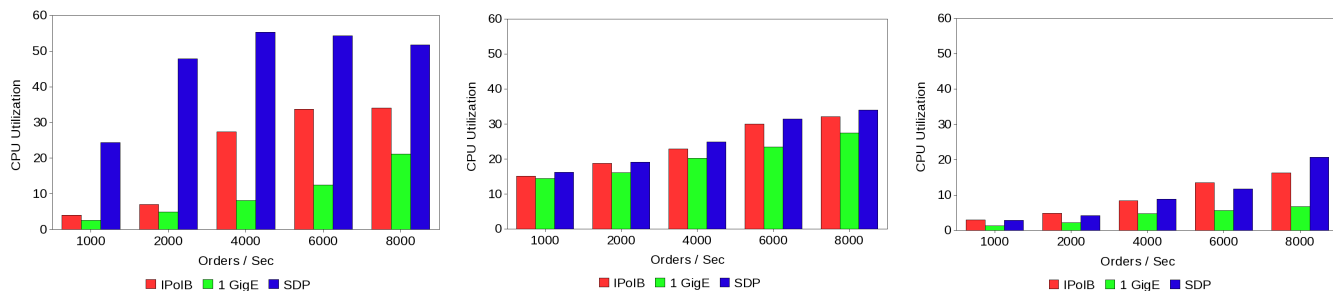
Figure 14. Performance of Stock Exchange Simulation: (a) Qpid Broker CPU Utilization, (b) Gateway CPU Utilization, and (c) Matching Engine CPU Utilization

Figure 13 (c) shows that the Nagle algorithm reduced IPoIB's average order completion from 9300 to 7000 microseconds at 8000 orders/second. However, the figure also shows that the Nagle algorithm drastically increased completion time at 1000 and 2000 orders/second. Seeking explanatory evidence, we re-ran the 1000 and 8000 orders/second experiments, capturing network traffic to and from the Qpid Broker with the *tcpdump* utility. Table 1 shows the results of this experiment. Without the Nagle algorithm (i.e. running with "TCP No Delay") each order generation rate produces a high number of small sized packets. With the Nagle algorithm enabled, the number of packets decreased and average packet size increased. However, the average time between packets, which also indicates the time required to build a full packet, also increased. This did not cause too much delay at 8000 orders/second as the increased time is still quite small. However at 1000 orders/second, the time between packets is higher. We believe that this high delay is exacerbated by the inherently slow 1000 orders/second generation rate.

Table 1. Performance of Stock Exchange Simulation: Impact of Nagle Algorithm Delay on IPoIB and TCP

Delay Type	Orders / Sec	Total Packets	Avg Packet Size (bytes)	Avg Time Btw Packets (us)
Without Nagle	1000	5,097,597	159	196
	8000	4,775,966	170	26
With Nagle	1000	2,269,111	359	440
	8000	2,277,870	358	55

6 Related Work

A number of vendors offer proprietary products to meet the demand for MOM applications. Examples include IBM's Websphere MQ [3], Microsoft Message Queuing Server [8], TIBCO's Rendezvous and Enterprise Messaging Servers [22], and Progress Software Corporation's Sonic MQ [16]. Furthermore Sun Microsystems has incorporated the Java Message Service (JMS) API into the Java Platform Enterprise Edition [5]. The Standard Performance Evaluation Corporation (SPEC) has also built a standard MOM benchmark oriented to JMS called SPECjms2007 [21]. Within the AMQP standard, other open source implementations have emerged in addition to Apache Qpid. These implementations include OpenAMQ [13] and RabbitMQ [17]. Furthermore, RedHat Enterprise Messaging uses Qpid as it's underlying base [19].

The stock exchange simulation application was created by ZeroMQ, which is another, open source, messaging implementation [24]. ZeroMQ's evaluation of their application resulted in higher performance numbers than ours. However, there are some differences between our experimental setups. ZeroMQ's architecture is different than Qpid's AMQP architecture. ZeroMQ is more minimal, does not have central Broker queues, and is optimized for high throughput of small sized messages. AMQP is a more general purpose messaging architecture. Next, the hardware used in ZeroMQ's evaluation was purposefully configured for the stock exchange simulation. For example the gateway and matching engine hosts each had two NICs, one for sending messages and the other for receiving messages. Furthermore the gateway and matching engine's NICs were connected to each other via direct cables without an intermediary switch [25]. Our experimental cluster on the other hand is more general purpose.

Pang and Maheshwari [6] have also published benchmarks for MOM. Their benchmarks are similar to ours in that they test the interaction between varying numbers of publishers and consumers. However, they compare two MOM software products. Our focus is on the evaluation of one product (Apache Qpid) using advanced networking technologies like InfiniBand, on modern multi-core platforms. We also developed a more extended set of benchmarks applicable to Direct, Fanout and Topic Exchange types.

7 Conclusions and Future Work

In this paper we devised a set of AMQP benchmarks inspired from those developed for MPI, a messaging standard from the high performance computing field. We then evaluated Apache Qpid, an AMQP compliant messaging software, against these benchmarks using the 1 GigE and InfiniBand networks. The DE-SPSC and DE-PP benchmark tests both found that IPoIB performed better for small message sizes, while SDP performed better for large message sizes. This was due the larger connection establishment overhead for SDP. Our DE-MPMC benchmark test showed that performance decreases with the addition of multiple Publishers and Consumers. This was caused by an increasing load on the Broker's CPU. The FE-SPSC and TE-SPSC benchmarks both found that IPoIB performance is comparable to that of 1 GigE.

In our stock exchange application experiment, we found that IPoIB is highly sensitive to the message generation rate. Applying the Nagle algorithm did not yield performance improvements at all message generation rates. However, the algorithm appears to be useful when the message generation rate is high (e.g. 8000 orders/second) and full sized packets may be built quickly. Furthermore the algorithm does little for this application's performance with SDP and slightly worsens 1 GigE performance.

Our analysis leads us to believe that the IPoIB limitations seen in our tests are due to the high IPoIB stack overhead. We believe that applying modern communication protocols, like RDMA, would improve this performance. This is because such protocols impose less processing load on host resources such as CPU. As part of our future work we plan to implement a native InfiniBand verbs adaptation for Apache Qpid. Furthermore our results indicate the need to combine these modern protocols with better designs for the Broker, including a distributed Broker scheme to alleviate AMQP's centralized Broker bottleneck.

References

[1] Advanced Message Queuing Protocol Website. http://www.amqp.org/.

[2] Apache Qpid. http://cwiki.apache.org/qpid/.

[3] IBM Websphere MQ. http://www.ibm.com/software/integration/wmq/.

[4] Infiniband Trade Association. http://www.infinibandta.org.

[5] Java Message Service. http://java.sun.com/products/jms/.

[6] P. Maheshwari and M. Pang. Benchmarking message-oriented middleware: TIB/RV versus SonicMQ: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(12):1507–1526, 2005.

[7] Mellanox OFED Stack for Linux Users Manual. http://www.mellanox.com/pdf/products/software/Mellanox_OFED_Linux_User_Manual_1_20.pdf.

[8] Microsoft Message Queuing Server. http://www.microsoft.com/windowsserver2003/technologies/msmq/.

[9] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing*, 1993.

[10] MVAPICH2: High Performance MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.

[11] J. O'Hara. Toward a commodity enterprise middleware. *Queue*, 5(4):48–55, 2007.

[12] Open Fabrics Enterprise Distribution. http://www.openfabrics.org/.

[13] OpenAMQ. http://www.openamq.org/.

[14] OSU Micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[15] P Balaji and S Narravula and K Vaidyanathan and S Krishnamoorthy and J. Wu and D. K. Panda. OSU-CISRC-10/03-TR54 Sockets Direct Procotol over InfiniBand in Clusters: Is it Beneficial?, 2004.

[16] Progress Software Corporation. http://www.sonicsoftware.com/.

[17] RabbitMQ. http://www.rabbitmq.com/.

[18] RDMA Consortium. http://www.rdmaconsortium.org/home/draft-recio-iwarp-rdmap-v1.0.pdf.

[19] RedHat Enterprise Messaging. http://www.redhat.com/mrg/.

[20] Socket Benchmarks. ftp://ftp.netbsd.org/pub/pkgsrc/current/pkgsrc/benchmarks/nttcp/README.html.

[21] SPECjms2007 Benchmark. http://www.spec.org/jms2007/.

[22] TIBCO. http://www.tibco.com/.

[23] S. Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, Nov.-Dec. 2006.

[24] ZeroMQ. http://www.zeromq.org/.

[25] ZeroMQ Stock Exchange Example. http://www.zeromq.org/code:examples-exchange.