Experience with a Middleware Infrastructure for Service Oriented Financial Applications

José Pedro Oliveira   José Pereira

ParadigmaXis, S.A.   U. do Minho
{*jpo,jop*}@*di.uminho.pt*

Abstract

Financial institutions, acting as financial intermediaries, need to handle numerous information sources and feed them to multiple processing, storage, and display services. This requires filtering and routing, but these feeds are usually provided in custom formats and protocols that are not the best fit for further processing. Moreover, the sheer volume of information and stringent timeliness and reliability requirements make this a substantial task.

In this paper, *i)* we characterize one of these information feeds (the Exchange Data Publisher feed from the NYSE Euronext European Cash Markets) and *ii)* we present and evaluate a dissemination system for this particular feeder based on commodity hardware and open-source message-oriented middleware (Apache Qpid). This allows us to assess the feasibility of this approach and to point out the main challenges to be overcome.

1 Introduction

Market data feeds such as the NYSE Euronext XDP [5] provide to financial institutions a detailed account of orders, trades, and quotes in real time. This information is needed for trading activities within the institution as well as to serve external clients through Web- based trading and home banking platforms. This requires that these feeds are processed, filtered, and routed towards a number of different services that encapsulate processing, storage, and further dissemination activities.

This is however a challenging task. First, due to the sheer volume of information and to stringent timeliness and reliability requirements. But also because the protocol used to deliver the data across a wide area network from the market systems is custom tailored to a specific set of requirements and not fit for further processing and usage within the institution. As further detailed in Section 2, this protocol is tailored to providing reliable transmission with minimal recovery latency while minimizing feedback to the sender for greater scalability and resilience of market systems. As a consequence, the coarse granularity of subscriptions and upfront redundant data transmission cause a large bandwidth overhead.

Reconciling this with the typical requirements of a financial institution means publishing the market data feed to a more flexible event dissemination system. In this paper we describe an experiment to assess the feasibility of achieving this with off-the-shelf hardware and software and minimal additional configuration and performance tuning effort. This leads to two contributions:

- We characterize the workload imposed by a typical market data feed in terms of number and type of events, but also how frequently and far apart related events are found in the incoming data.
- We deploy a test system using Apache Qpid [3] message broker and several event consumers and measure the latency introduced while accounting for resources used.

The rest of the paper is structured as follows. The NYSE Euronext data feed is characterized in Section 2, followed by a description of our experimental setting in Section 3. Then, in Section 4 we present the experimental results. Finally, we summarize the lessons learned, discuss the results and lay plans for future work.

2 NYSE Euronext XDP

The NYSE Euronext European Cash Markets Exchange Data Publisher (XDP) feed is disseminated in real-time via dual multicast channels with different Market Data product sets having its own pair of dedicated multicast channels (Figure 1). In particular, the Exchange Data Publisher feed, has seven main different product sets, or services: Euronext Equities - Referential Data (101), Euronext Equities - Trades (102), Euronext Equities - Quotes (103), Euronext Equities - Orders (104), Euronext Warrants - Trades (105), Euronext Warrants - Quotes (106), and Euronext Indices - Composition and Values (107).

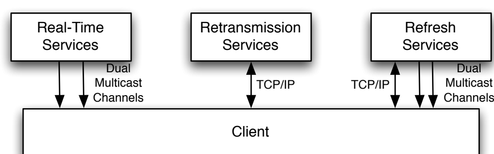

Figure 1: NYSE Euronext UTP-MD platform

The usability of these feeds, in terms of packet recovery, is assured by three components: *i)* the Market Data Server (MDS) that provides the real-time data via dual multicast channels (the data of each of the above services is received independently in two channels), *ii)* the Retransmission Server (RTS) that is able to fill, upon request and by TCP connections, the packet gaps that clients may experience, and *iii)* the Refresh Server (RFS) that is able to provide a snapshot of the current

market state using a second set of multicast channels or, upon request, via TCP connection.

2.1 Protocol

The real-time market data is delivered as payload of IPv4 UDP datagrams with fixed length fields. Each Euronext packet has, at least, 16 bytes in a packet header and never exceeds 1400 bytes. It can also have several market data messages in its payload (the number of messages is specified in one field of the packet header). And each packet will only contain complete messages.

The 16-byte packet header has these fields: PacketLength(2), PacketType(2), PacketSeqNum(4), SendTime(4), ServiceID(2), DeliveryFlag(1), and NumberMsgEntries(1). Each Market Data message also has a 4 byte message header with two fields: MsgSize(2) and MsgType(2).

2.2 Session traffic

Table 1 and Figures 2 and 3 represent the raw data, in terms of packets and bytes, received from the Euronext network in a typical session during a March 2011 session starting at 05:10 and finishing at 22:00. The resulting data is representative of most sessions, although it is not the worst case scenario that was observed.

This Euronext session traffic, arriving through a single 48 Mbps leased line, was captured [6] by a software based solution - dumpcap [1] - without any packets being dropped. The capture process only listened for multicast packets and no switch port-mirror facility was used.

	Packets		Data size	
Service	Number	%	Bytes	%
Realtime	**78790668**	**97.0**	**26872066702**	**95.9**
101	69943	0.1	17595944	0.1
102	4450056	5.5	561680548	2.0
103	28504078	35.1	12073599450	43.1
104	28762398	35.4	10014669786	35.7
105	649279	0.8	125315792	0.4
106	15731559	19.4	3851438560	13.7
107	623355	0.8	227766622	0.8
Refresh	**2426944**	**3.0**	**1153843507**	**4.1**
Total	**81217612**	**100.0**	**28025910209**	**100.0**

Table 1: Real-time services: traffic summary

For this particular session more than 81 million packets (ethernet frames), totaling 26.1 GiB of data, were captured. For Euronext traffic these numbers can be

[1] Dumpcap is part of the Wireshark [2] software suite.

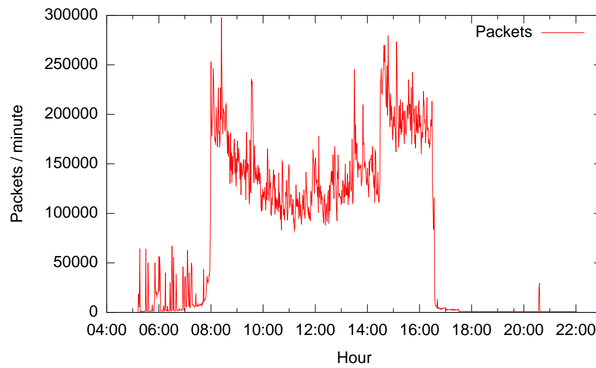

Figure 2: Packets per minute

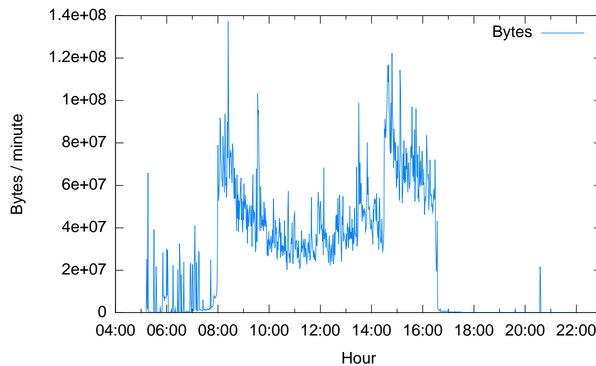

Figure 3: Bytes per minute

further divided in Realtime and Refresh traffic, where 97% of the packets belong to the first category and 3% belong to the second. The 81 millions of Euronext packets contained more than 410 millions of Euronext messages, the average Euronext packet size (the UDP datagram payload) was 287.53 bytes, the average Euronext message size was 53.74 bytes, and the average number of Euronext messages per packet was 5.05.

Figures 4 and 5 show the raw incoming data splitted by the seven real-time services. The ten bigger traffic peaks for this particular session, in terms of packet and bytes rates, are listed in Tables 2 and 3.

Finally, we characterize how often related events are found in the stream by computing the interarrival time of events referring to the same SymbolIndex. As can be seen in Figure 6, containing the corresponding empirical cumulative distribution, when a symbol is repeatead in the same stream, approximately 70% of times it will be within the same millisecond and 80% of times before 10 milliseconds. This means that multiple references to the same symbol appear in bursts. Note that the interarrival time was calculated using only packets from the first channel of each service. This means that there is potential for caching of rules within the broker when filtering and routing messages based on the symbol.

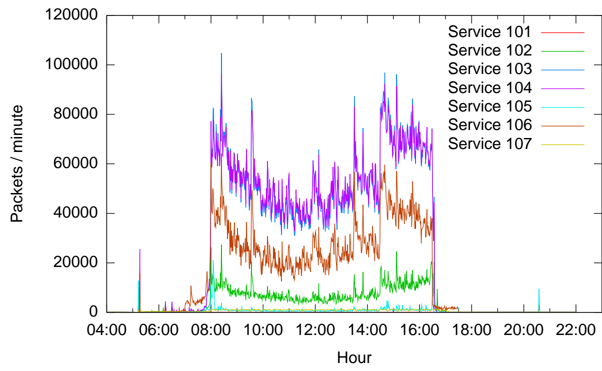

Figure 4: Real-time services: packets per minute

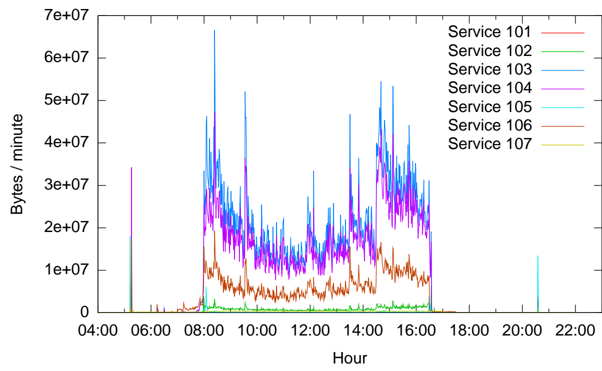

Figure 5: Real-time services: bytes per minute

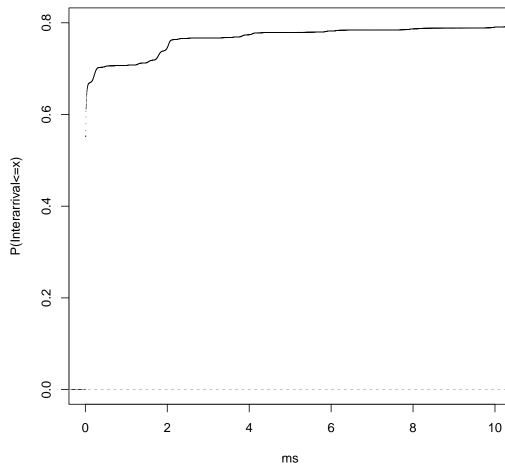

Figure 6: Same symbol interarrival

The challenge is thus to determine to what extent an off-the-shelf middleware package can be used on typical hardware components to perform the information

Time	Packets	Bytes
09:33:42	10297	5480480
13:29:56	9349	4802572
08:24:47	8871	5488584
13:50:39	8634	4603524
08:05:00	8545	3845246
09:33:43	8453	4870922
08:04:58	8386	3250192
08:05:11	8184	2088188
09:33:47	8162	4724620
09:33:46	8153	5414590

Table 2: Packet peaks

Time	Packets	Bytes
16:34:55	4790	5975840
05:16:29	4660	5912564
07:59:58	5989	5899770
05:16:24	5237	5897868
05:16:23	5496	5793488
05:16:25	4319	5670150
05:16:30	4459	5668254
05:16:26	4611	5628434
08:24:47	8871	5488584
09:33:42	10297	5480480

Table 3: Byte peaks

dissemination and filtering activities required to support a service oriented infrastructure. In particular, we are concerned with the ability to meet desired latency targets in spite of traffic bursts when performing dissemination and filtering at different granularities, namely, by service or by symbol.

3 Experimental Setup

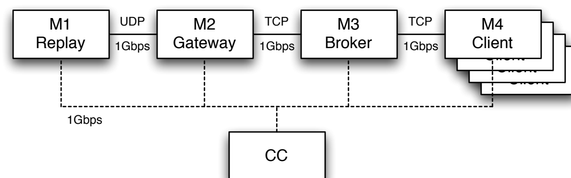

Figure 7: Experimental setup

To address this challenge, we setup an infrastructure that can deterministically replay a representative sample of traffic and feed it into different middleware configurations. In detail, the architecture used is depicted in Figure 7, showing the main computers used and their network connections. All computers had at least an Intel Core2 Duo CPU, 2 network interfaces, and 2 GB of RAM [2] and were running Scientific Linux 6.1 x86_64, a Red Hat Enterprise Linux clone, with kernel 2.6.32 (2.6.32-131.6.1.el6.x86_64). The two switches used, one for interconnecting the broker with the clients, and the other for the command-and-control network, are both gigabit. All links are gigabit with the exception of the replay machine (M1) command-and-control link that is only 100 Mbps.

Server M1 was responsible for impersonating the NYSE Euronext XDP feed. This impersonation was done by replaying traffic captured during a real Euronext session. The traffic, which has been previously captured, was replayed with the tcpreplay [7] tool, after being modified with tcprewrite [3] in order to change the frame source mac address and the source IP address. It basically replayed the multicast XDP packets (Ethernet/IPv4/UDP/XDP).

Server M2 had a simple thread-based software component implemented in C++ that received the feeder real-time multicast traffic (XDP data over UDP), dropped the duplicate packets, optionally splitted the packets in its messages, and injected them in the Qpid broker as AMQP messages (Eth/IPv4/TCP/AMQP/XDP). Note that this application ignored the Refresh multicast traffic, i.e., it didn't join the Refresh multicast groups. This application was linked against the Qpid client and the Boost [1] libraries.

Server M3 runned a Qpid broker, an open source implementation of the Advanced Message Queuing Protocol (AMQP) [4]. This particular Qpid broker was installed via yum using the RPMs available in the Scientific Linux repositories (qpid-cpp-server-0.10). All system components, in particular the kernel and the broker, had their default configurations, with the exception of the broker default queue size that was increased to 500 MiB. All published and subscribed data was handled over AMQP messages.

Hosts M4, M5, M6, and M7 executed the client application, that consumed or subscribed the feed information using either a direct or a topic exchange [4]. The client application was also done in C++, linked with the Qpid client and the Boost libraries. All subcribed data was transported in AMQP messages.

Finally, CC was used to start the tests and collect the system under test statistics. These operations were performed through a second and independent gigabit network.

[2]The gateway had 3 gigabit network interfaces; the broker had a quad core CPU (Intel i7), 5 gigabit network interfaces, and 8 GB of RAM

[3]Tcprewrite is part of the tcpreplay software suite.

[4]The topic exchange supports multiple words keys

4 Measurements

The latency introduced by the middleware system, an Apache Qpid broker, was measured in two major scenarios: *i)* a publish/ subscribe scenario where every non-duplicated real-time Euronext packet was independently published and *ii)* a publish/subscribe scenario where every Euronext message of non-duplicated real-time Euronext packets was independently published. Each of the previous scenarios was further subdivided in two separated experiments: *i)* using a broker direct exchange for the publish/subscribe operations and *ii)* using a broker topic exchange for the publish/subscribe operations.

Other important system considerations were:

- A subset of the captured Euronext session traffic was replayed at 40 Mbps from M1 system. The subset used, the captured traffic between 07:55 and 10:00, contained approximately 20.1 millions Euronext packets that represented roughly 9.99 millions non-duplicated real-time Euronext packets containing 50.6 millions Euronext messages.

- Every time a non-duplicated Euronext real-time packet arrived at the gateway (M2), it was pushed to its respective service queue [5]. Another thread, one for each real-time service, was used to consume the packets from the service queue, and publishing them on the Qpid broker. In the second scenario, this thread was also responsible for splitting the Euronext packets in its messages, suffixing them with a 7 byte tag (Service ID, PacketSeqNum, and MessageNum), before publishing them to the Qpid broker.

  In the first scenario, and for every odd PSN Euronext packet, an UDP control message containing the current packet tag was sent to the CC system where it was timestamped on arrival. In the second scenario, an UDP message was sent for every odd message of every odd PSN Euronext packet (roughly 1 control message for every 4 Euronext messages).

- The publish/subscribe control client (M4), either a direct or a topic exchange client, for every message with an odd PSN it received, extracted the message tag and sent it in an UDP control message to the CC system, where it was also timestamped on arrival. In the second scenario, an UDP control message was sent for every odd message of every odd PSN Euronext packet.

- The CC machine, by timestamping the messages it received from the gateway and from the control client, and matching their tags, was able to calculate the latency introduced on the system by the Qpid broker.

[5] Remember that every service receives data via two independent channels

4.1 Scenario 1 - Euronext packets as AMQP messages data

The results of this setup, where there is a one-to-one relation between the number of non-duplicated real-time Euronext packets and the number of AMQP messages published, are summarized in Tables 4 and 5 and in Figures 8, 9, and 10.

Clients	Mean	Min	Max	P(50%)	P(90%)
1	0.786	0.008	421.028	0.762	1.007
2	0.871	0.008	271.007	0.836	1.137
4	0.918	0.008	323.354	0.883	1.193
6	1.216	0.008	347.970	1.026	1.946
8	1.464	0.008	264.230	0.999	2.932
10	1.519	0.008	277.401	0.975	3.119

Table 4: Packets: direct exchange latencies (ms)

Clients	Mean	Min	Max	P(50%)	P(90%)
1	0.807	0.009	301.832	0.783	1.034
2	0.903	0.008	383.946	0.856	1.167
4	0.990	0.008	1595.953	0.931	1.303
6	1.326	0.008	265.208	1.048	2.327
8	1.516	0.008	253.650	1.002	3.097
10	1.566	0.009	404.169	1.000	3.201

Table 5: Packets: topic exchange latencies (ms)

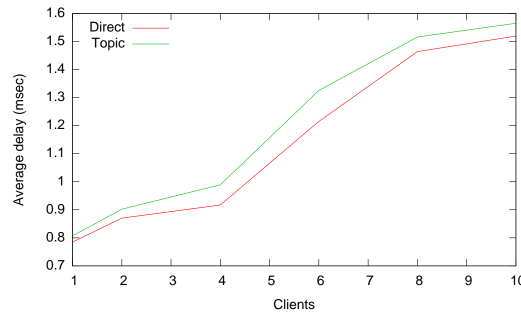

Figure 8: Euronext packets: latencies

During this setup runs, the machines CPU load, network load, and memory consumption were monitored using Dstat [8], and no resource related problems were detected in any of them. In particular, the system load of the broker (M3) oscillated between 2% and 12% while its user load oscillated between 8% and

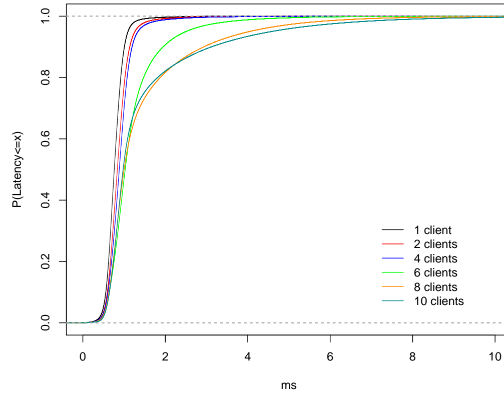

Figure 9: Packets: direct exchange latencies

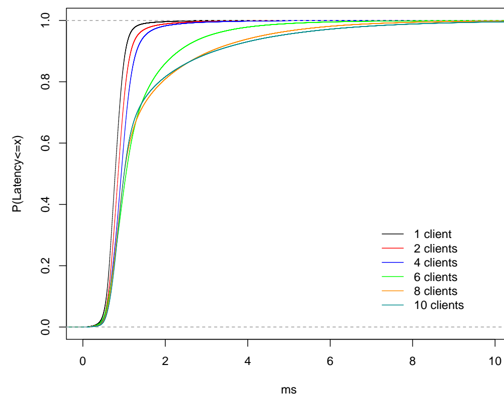

Figure 10: Packets: topic exchange latencies

30%; the replay system transmitted 5 MB/sec, the gateway transmitted 3 MB/s in its publishing link, and the client consumed data at 3.4 MB/s.

The conclusion is that it is possible to use the proposed middleware as the backbone for dissemination, even if topic subscribtion is being performed instead of direct exchange. Note however that if the application is highly sensitive to delay, the long tails in Figures 9 and 10 indicate that, even in this case in which broker resources are far from being exhausted, there are some packets that are delayed significantly more than the average delay.

4.2 Scenario 2 - Euronext messages as AMQP messages data

The results of this setup, where there is a one-to-five relation between the number of non-duplicated real-time Euronext packets and the number of AMQP messages published, are summarized in Tables6 and 7 and in Figures 11, 12, and 13.

Clients	Mean	Min	Max	P(50%)	P(90%)
1	0.762	0.009	505.393	0.709	1.024
2	0.809	0.021	589.011	0.702	1.081
3	0.974	0.018	567.455	0.754	1.183
4	2.230	0.022	592.593	1.001	3.763
5	60.652	0.088	705.414	53.199	125.689

Table 6: Messages: direct exchange latencies (ms)

Clients	Mean	Min	Max	P(50%)	P(90%)
1	0.775	0.018	530.459	0.713	1.033
2	1.062	0.009	596.690	0.753	1.340
3	2.185	0.022	613.629	0.969	3.512
4	10.850	0.043	503.512	3.985	29.263
5	120.456	0.221	720.153	120.128	146.817

Table 7: Messages: topic exchange latencies (ms)

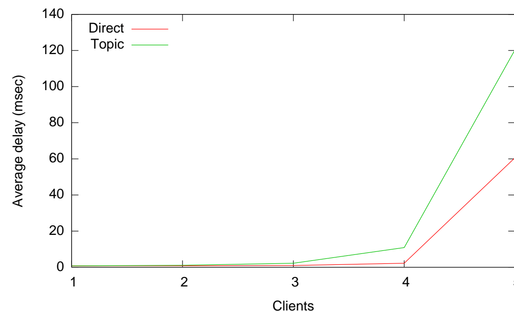

Figure 11: Euronext messages: latencies

During this setup runs, the machines CPU load, network load, and memory consumption were monitored using Dstat [8], and some resource related problems start to appear. While the broker (M3) maintained its resources under control, where its system load oscillated between 2% and 10% and its user load oscillated between 15.5% and 38%, the gateway started to experience some CPU and memory stress: the broker producer flow-control and what appears to be a single-thread per connection publishing started to increase the memory consumption and CPU load.

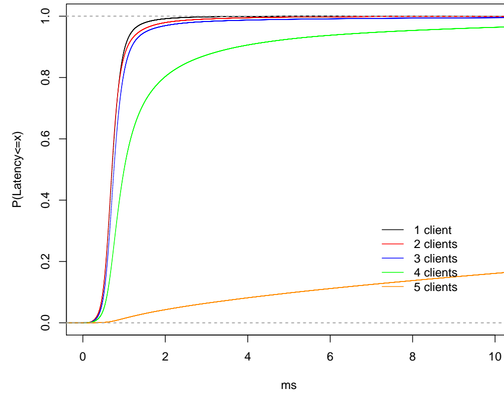

Figure 12: Messages: direct exchange latencies

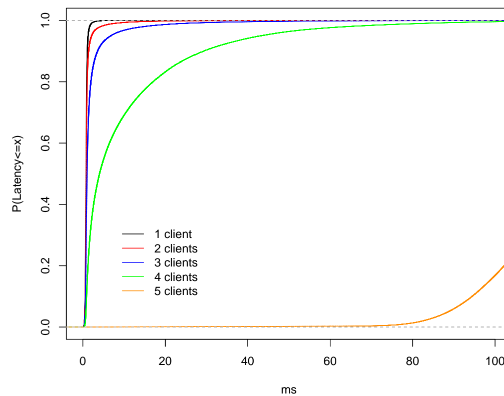

Figure 13: Messages: topic exchange latencies

In this setup the replay system transmitted 5 MB/sec, the gateway transmitted 6.1 MB/s in its publishing link, and the client consumed data at 7.8 MB/s.

In this case, in which the broker has to manage a substantially higher number of messages, even if smaller in size, it becomes clear that it becomes a bottleneck fairly quickly. In fact, with as little as 5 subscriptions, there is significant queuing happening with dramatic impact in latency.

5 Lessons learned

These experiments, where the data was replayed at a constant bit rate and every client subscribed to all data published, allowed us to uncover the following lessons.

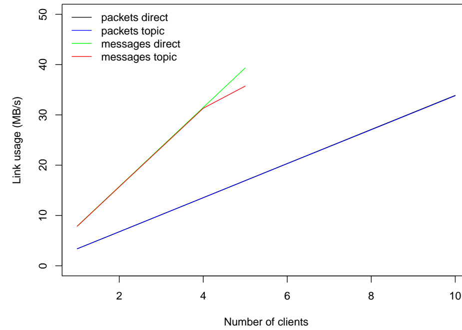

Figure 14: Broker network bandwidth used by the subscribers

First, splitting the XDP packets too soon in the dissemination path causes several problems: *i)* the broker CPU load (as seen in figure 15) increased sharply with the higher number of smaller messages being published, *ii)* the broker network bandwidth usage in the link where the subcribers were connected (as seen in figure 14) more than duplicated; this also show us how critical the protocols overheads were and how easy it would be to saturate a 1 Gbps network link even with a small number of clients, *iii)* the end-to-end latencies increased rapidly (Tables 6 and 7 vs Tables 4 and 5) making the SUT become unusable with only 5 clients.

Second, the fine grained pub/sub also caused problems in the gateway system, making its memory usage grow steadly (due to queueing of messages) as the broker started to throttle back the producer message rate (Qpid producer flow control feature). This behaviour was observed in the Messages/Topic experiment of the second scenario (Figure 16). This also made the gateway a prime contributer to the end-to-end latency.

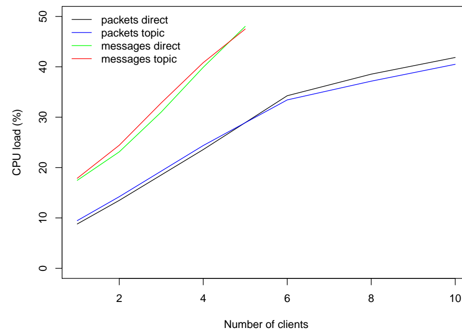

Figure 15: Broker CPU load

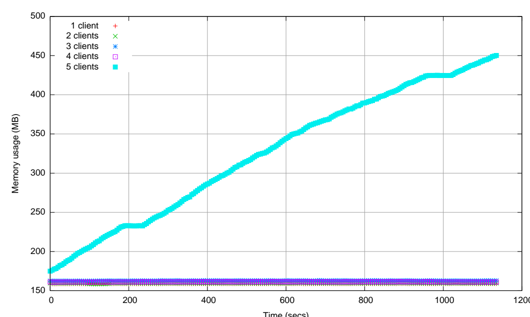

Figure 16: Scenario 2 Msg/Topic - gateway memory usage

6 Conclusions

In this paper we describe an experiment with an event processing and dissemination infrastructure for handling a typical financial market data feed within a financial institution. The first contribution is thus a characterization of this workload, that should be useful when researching event dissemination systems, for instance, when generating realistic workloads for testing.

Although our setup handles the load while consuming only a fraction of available CPU and memory bandwidth of the broker, we observed that additional clients, in particular with fine grained publication/subscription, quickly cause an increasing share of events to be delayed for an increasingly larger period. This is worrisome and thus future work should explore the scalability limits of the system. Moreover, the large number of related events close together in the event stream found by this study should also be considered in future research.

7 Acknowledgments

Partially funded by FCT through grant SFRH/ BDE/ 33304/ 2008 and by ParadigmaXis - Arquitectura e Engenharia de Software, S.A.

References

[1] Boost C++ Libraries. `http://www.boost.org/`.

[2] Wireshark - network protocol analyzer. `http://www.wireshark.org/`.

[3] T. Apache Software Foundation. Apache Qpid: Open Source AMQP Messaging. `http://qpid.apache.org/`.

[4] A. W. Group. Advanced Message Queuing Protocol. `http://www.amqp.org/`.

[5] NYSE Technologies, http://www.nyxdata.com/doc/32342. *NYSE Euronext European Cash Markets Exchange Data Publisher Client Specification*, version 3.6 edition, September 2012.

[6] V. Paxson. Strategies for sound internet measurement. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, IMC '04, pages 263–271, New York, NY, USA, 2004. ACM.

[7] A. Turner. tcpreplay. `http://tcpreplay.synfin.net/`.

[8] D. Wieers. Dstat: versatile resource statistics tool. `http://dag.wieers.com/home-made/dstat/`.