Constructing Performance Model of JMS Middleware Platform

Tomáš Martinec, Lukáš Marek,
Antonín Steinhauser, Petr Tůma
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
last.name@d3s.mff.cuni.cz

Qais Noorshams, Andreas Rentschler,
Ralf Reussner
Chair Software Design and Quality
Karlsruhe Institute of Technology
Karlsruhe, Germany
last.name@kit.edu

ABSTRACT

Middleware performance models are useful building blocks in the performance models of distributed software applications. We focus on performance models of messaging middleware implementing the Java Message Service standard, showing how certain system design properties – including pipelined processing and message coalescing – interact to create performance behavior that the existing models do not capture accurately. We construct a performance model of the ActiveMQ messaging middleware that addresses the outlined issues and discuss how the approach extends to other middleware implementations.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*

General Terms

Performance

Keywords

Software Performance; Performance Analysis; Measurement; Modeling; JMS

1. INTRODUCTION

Software performance engineering (SPE) is a discipline that focuses on incorporating performance concerns into the software development process, aiming to reliably deliver software with particular performance properties [36]. Among the tools employed by SPE are predictive performance models. Constructed in the early phases of the software development process, the models help predict the eventual software performance and thus guide the development [3].

To deliver the expected guidance, the predictive performance models must capture all relevant system components.

For modern software applications, this may entail modeling complex system layers such as the virtual machine or the messaging middleware. Composing such complete performance models directly is necessarily expensive and inefficient. Instead, the abstract application model can be constructed first, with the models of standard system components added later [39, 8]. This gives rise to the need for composable performance models of standard system components.

Our work focuses on the construction of such performance models for messaging middleware, specifically messaging middleware that implements the Java Messaging Service (JMS) standard [37]. Although JMS performance models were published before [26, 14, 30, 9, 13, 11, 34], we illustrate that the existing models often fail to capture important elements of middleware behavior. In turn, this omission results in reduced performance prediction accuracy, especially where processor utilization and message latency are concerned. Our contribution is as follows:

- Using code analysis and experimental measurements of a mainstream JMS implementation, we illustrate situations where observed performance is not accurately predicted by common models.

- We provide a detailed technical analysis of the observed effects as an essential basis for further modeling.

- We design a performance model that captures these effects and validate the model using experimental measurements.

We have decided to organize our presentation in a way that familiarizes the reader with the necessary platform-specific background as soon as possible. This helps avoid potentially inaccurate generalizations in the introductory text. In Section 2, we introduce our modeling context and describe our experimental platform. Section 3 explains the issues that complicate accurate performance modeling of our platform. We show how to construct a performance model that addresses these issues in Section 4, and follow by evaluating and discussing the model results in Section 5. This is where we pay particular attention to explaining how our results, so far presented in a platform-specific context, can be generalized. Section 6 relates our modeling efforts to the existing research, and finally Section 7 summarizes our conclusions.

2. MODELING CONTEXT

The expectations put on a performance model are closely related to the intended model use. We therefore start by describing such uses, paying particular attention to the inputs that are available to the modeler and the outputs that the modeler would seek in each context.

As noted in the introduction, middleware performance models are needed as building blocks in application performance models. Such models are used in early stages of the software development process to guide important design decisions, or in software maintenance activities when a change impact analysis can be conducted to choose among multiple modification directions [21]. On the input side, the modeler can usually collect information about the timing (or more generally resource demands) of the operations used as atomic elements of the model. Restrictions on the ability to instrument particular operations may require using specialized microbenchmarks or deriving detailed information from aggregate statistics such as overall system throughput. On the output side, the modeler would require the model to accurately predict general design feasibility and overall scalability trends with respect to performance. The model should also suffice for comparing design alternatives.

Middleware performance models can also be understood as a description of the expected performance (rather than an approximation of the actual performance). Besides simple software documentation purposes, this use can also benefit software performance testing [5, 15]. In this context, the models can be provided with the same inputs as in the early stages of the software development process, with one important addition – the models can be automatically calibrated against the actual performance in selected benchmarks. Such calibration makes the question of absolute prediction accuracy mute, the modeler instead evaluates the ability to fit the model to the measurements with reasonable values of the calibrated parameters.

It is also possible to use the models at runtime to plan system adaptation [7]. Particular to this context is the need to maintain low overhead in both collecting the inputs and evaluating the model. The output of the model is used to make adaptation decisions, reliable estimation of trends or reliable comparison of alternatives is therefore preferred to absolute prediction accuracy.

In summary, the three modeling contexts all put emphasis on predicting trends, which are used to make relative comparisons or to assess system scalability. Where absolute prediction accuracy is important, model calibration is performed on the timing information collected through measurement.

2.1 Modeling Messaging Brokers

Our work focuses on the construction of performance models for JMS middleware [37]. The JMS architecture envisions multiple clients communicating by sending and receiving application specific messages. The messages travel either through queues in a point-to-point pattern or through topics in a publish-subscribe pattern. The JMS standard provides multiple quality-of-service settings, especially important from performance perspective is deciding whether the JMS middleware should keep messages in transient buffers or persistent storage and whether the message delivery should be subject to transaction processing.

The model we construct should be a suitable building block in application performance models. It must be able to predict basic performance metrics relevant for the JMS middleware – especially resource utilization, message throughput and message latency – that would be observed for a given workload on a given platform. The middleware model does not describe the workload itself, that is the task of the application performance models that would incorporate the middleware model.

2.2 Experimental Platform

In our experience, the process of building and validating a performance model is necessarily platform-dependent. Although the individual steps can follow a common overall approach, the modeling accuracy depends on multiple technical details that need to be considered. We therefore introduce our experimental platform and continue the presentation in a platform-specific context. Generalizations are discussed as appropriate.

Our code analysis and experimental measurements are performed on the ActiveMQ 5.4.2 messaging middleware [2], which implements the JMS standard [37]. Central to the middleware is the message broker, a process that manages messaging channels, which are either queues or topics. Message producers and message consumers connect to the broker using sockets. We isolate broker performance by executing it on a dedicated computer, a single-core 2.33 GHz Intel Xeon machine with 4 GB RAM running Fedora Linux with kernel 3.9.2-200 x86_64 and OpenJDK 1.6.0-24 x86_64. The producers and consumers run on two additional computers connected through a dedicated gigabit Ethernet network with accelerated Broadcom network adapters, chosen so that they can saturate the broker while at low load themselves – the producer is an eight-core (two chips four cores each) 2.30 GHz AMD Opteron machine with 16 GB RAM and the consumer is an eight-core (two chips four cores each) 1.86 GHz Intel Xeon machine with 8 GB RAM.

From the many quality-of-service settings available, we focus on the transient message passing mechanism with acknowledgments. This setting targets applications that require low-latency high-throughput reliable message delivery and is therefore a natural performance modeling subject. We do not model quality-of-service settings that require persistent message storage, because with such settings, the storage performance tends to dominate the observations. Existing storage performance modeling methods are then likely better suited for capturing the observed performance [38].

The transient message passing mechanism is implemented in four broker threads that process a message passing through a broker queue, as shown on Figure 1:

- The first thread blocks waiting for messages arriving through a network socket. On message arrival, the thread reads the message, selects the destination queue and stores the message in a container associated with this queue. This thread is blocked when the container is full.

- The second thread blocks waiting for messages arriving in the container filled by the first thread. On message arrival, the thread locates the message consumer and passes the message to the third thread, responsible for communicating with that consumer.

– The third thread blocks waiting for messages and sends them on to the consumer through a network socket.

– The fourth thread blocks waiting for acknowledgments arriving from the consumer through a network socket. On acknowledgment arrival, the corresponding message is recognized as processed.
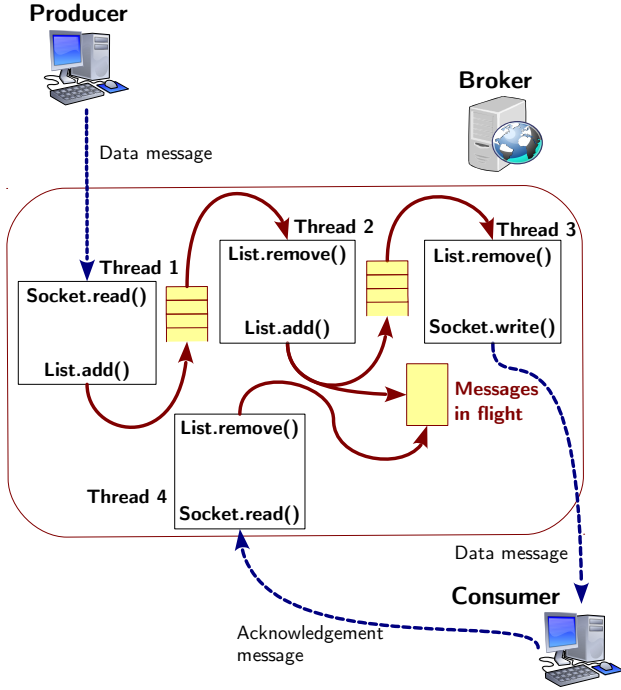

Figure 1: Transient message passing architecture in ActiveMQ 5.4.2.

3. MODELING ISSUES

Existing models of messaging middleware[1] typically belong to one of two broad classes, here called *models with queues* and *fitted models*:

– A typical model with queues relies on the fact that messaging channels resemble service queues. The model would represent resources such as processor or storage with service queues and approximate a message passing through a messaging channel with a single service request in each of the queues.

Models with queues were shown to achieve high accuracy especially in complex systems with multiple messaging channels, where the mean resource demands at the bottleneck resources determine the achievable throughput and the accumulated effects of queueing at the messaging channels dominate the observed latencies [18, 34].

– A fitted model is typically used when the observed performance is determined through interactions at the implementation level that are either not understood in

---

[1]We discuss the existing models in depth in the related work section. We avoid the discussion here to maintain text flow.

sufficient detail or simply too complex. After quantifying the workload properties that impact performance, the model would derive a function that predicts performance from the workload properties by fitting a function template to the observed measurements.

Fitted models were successfully used with workload properties such as message size or filter count [13, 11], whose impact is otherwise difficult to predict because it consists of many minuscule implementation effects.

Despite their many strong points, both model classes exhibit accuracy issues in certain situations inherent to our modeling context. We describe these issues next.

3.1 Pipelined Message Processing

The ActiveMQ broker processes messages in several phases that form a pipeline. When any of the phases limits concurrent processing – as is the case with the thread-per-connection and thread-per-destination patterns in our broker – messages may queue inside the pipeline. Such queueing has a relatively benign impact on throughput but a very significant effect on latencies, as illustrated on Figure 2.
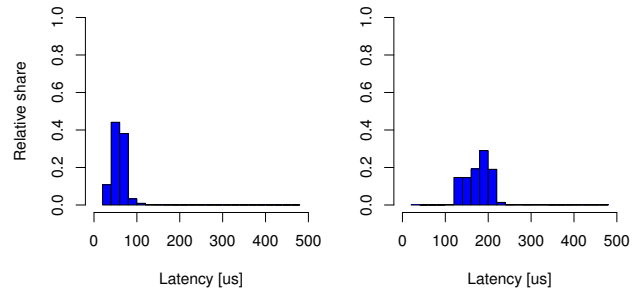

Figure 2: Impact of bursts on latency distribution. Constant throughput 5000 msg/s, left workload sending individual messages, right workload sending bursts of ten messages.

Figure 2 shows the distribution of message latencies observed at the throughput of 5000 msg/s in two workload configurations, regular and bursty. In the regular configuration, the producer emits one message every 200 µs. In the bursty configuration, the producer emits a burst of ten messages every 2 ms.
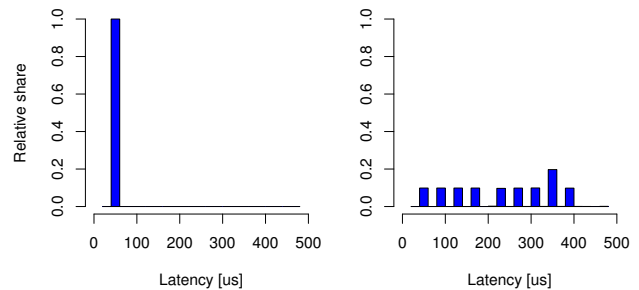

Figure 3: Predicting impact of bursts on latency distribution with G/G/1 queue. Constant throughput 5000 msg/s, left workload sending individual messages, right workload sending bursts of ten messages.

Figure 3 shows that approximating the broker with a single service queue – as a model with queues might do – is not enough when modeling the bursty workload latency. The model used the same distribution of the arrival times and the service times as Figure 2. For the regular workload, the predicted latency matches the measurement reasonably well. For the bursty workload, the predicted latency shows several regular clusters from 40 μs to 420 μs but the measurement forms a single cluster from 120 μs to 240 μs – the model not only failed to predict the absolute latency, it also failed to approximate the overall trend.

Section 5 shows how our model improves the prediction accuracy by reflecting the pipeline architecture in the model structure. A fitted model that would capture the latency would have to include the information quantifying the bursts in the workload properties. Unfortunately, adding new independent variables into the workload properties increases the cost of building a fitted model.

3.2 Thread Scheduling Overhead

The use of multiple threads in the ActiveMQ broker introduces the opportunity for context switching, that is, the act of handing control of the processor from one thread to another. Although the design intent is to make context switch a fast operation, the accumulated overhead of context switching can impact performance.

Two major reasons for a context switch are the scheduling policies enforced by the operating system and the blocking behavior exhibited by the executing threads. The scheduling policies are usually only enforced after a thread has run for some time – 750 μs on our platform – which makes them unlikely to impact relatively fast message processing – tens of microseconds on our platform. In contrast, the thread blocking behavior may trigger context switches arbitrarily fast.

The cost of a context switch can vary significantly [35, 23]. On our platform, a simple benchmark where two threads take turns blocking each other on a synchronization variable estimates the context switch duration to be 3.3 μs. The pipelined message processing, which involves four threads operating on each message, further multiplies the context switch overhead. Even more importantly, the amount of context switching per message varies. When messages arrive far from each other in time, the threads finish processing a message before the next one arrives and therefore block waiting once per message. But when messages arrive close to each other, the threads have a new message to process by the time they finish processing the previous one and therefore do not block waiting. This effect is shown on Figure 4.

Figure 4 illustrates that on our broker, the relative amount of context switching changes from about 20 switches per message for low throughput to about 1 switch per message for high throughput. The peak throughput can be deduced from Figure 5, which shows the dependency between the target throughput and the actual throughput (the producer attempts to generate messages at the target throughput rate, but the broker flow control restricts the producer to avoid message loss). Also worth noting is the implied fact that peak processor utilization does not coincide with peak throughput – a practically important effect because high processor utilization is often taken to indicate a bottleneck.

To model this thread scheduling effect, a model with queues would require a special load dependent service queue. A fit-
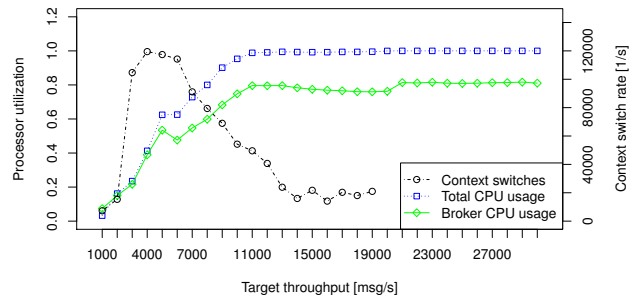

Figure 4: Dependency of broker processor utilization and context switch rate on target throughput.
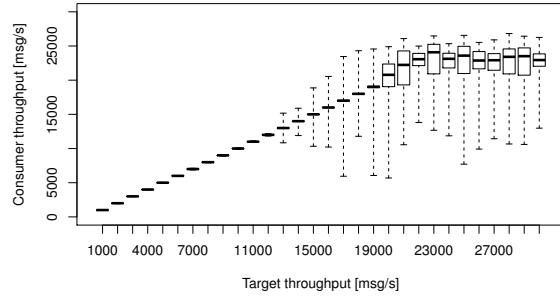

**Figure 5: Dependency of actual throughput on target throughput.**

ted model can probably capture the effect more easily, but we are not aware of any work doing so.

3.3 Message Coalescing

The performance impact of both the pipelined processing and the thread scheduling is more pronounced in bursty workloads than in regular workloads. Besides bursts that are inherent to the workload from the application perspective, more bursts can be introduced as the broker processes the messages, again influencing the observed performance. One source of such message bursts in our broker is the implementation of the TCP protocol in the network stack, which is used to transport messages between the producers and consumers and the broker. The protocol minimizes the processing overhead by coalescing smaller messages into larger packets, both in software and in hardware. Coalescing in software follows RFC 896 [16] and is disabled by default – because this is a sensible default, we leave it disabled in our experiments. Coalescing in hardware is done as a part of the Generic Receive Offload (GRO) [41] and Generic Segmentation Offload (GSO) [40] features.

Both GRO and GSO are enabled by default, and although they can also be disabled, keeping the default makes the experimental platform more realistic. We believe existing simulation tools such as the `ns` simulator [31] are more suitable for modeling the message coalescing behavior at the TCP protocol level than the performance models considered here. To avoid the need for modeling this behavior, we collect the information quantifying the bursts on the broker machine. Figure 6 illustrates message coalescing on our platform – the producer uses the `sendto` socket function to transmit 1030 B long messages at a rate of 20000 msg/s, the two graphs show the statistical distribution of packet sizes observed through

the `pcap` monitoring interface when departing the producer and when arriving at the broker. Without message coalescing, the graphs would show all messages having 1030 B plus the TCP protocol header.
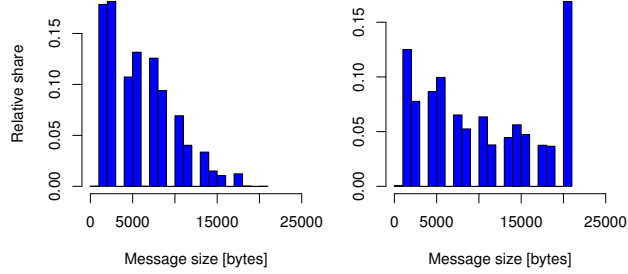
Figure 6: Packet sizes observed when departing the producer (left) and arriving at the broker (right).

Another opportunity for message coalescing arises in connection with the garbage collection. On our platform, the garbage collector occasionally stops the broker threads to free heap space. Messages received while the broker threads wait are held in the operating system buffers and processed by the broker as soon as the garbage collector finishes. From the perspective of the broker, this has the same effect as if the messages arrived in one burst. Figure 7 displays the latencies during a garbage collection pause. With pluses, we show latencies measured at the points where messages enter and leave the broker – the cluster of pluses at the end of the garbage collection pause shows the broker reading the messages held in the operating system buffers during the pause. With circles, we show latencies estimated at the points where messages enter and leave the operating system buffers – the slope of circles during the garbage collection pause shows how the messages accumulate. Section 4 explains how this effect is captured by our model.
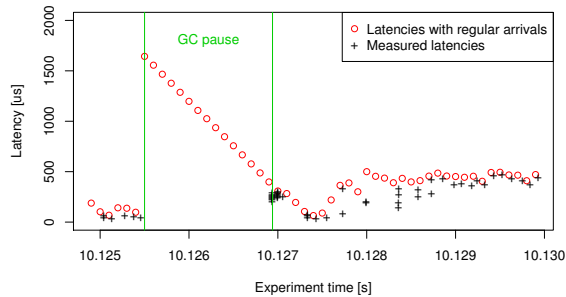
Figure 7: Effect of garbage collection pause on latency.

4. PERFORMANCE MODEL CONSTRUCTION

To address the modeling issues outlined in Section 3, we construct a performance model that directly reflects the broker structure as shown on Figure 1. We use Queueing Petri Nets (QPN) [20] as the modeling formalism, both because it offers modeling abstractions that match the architecture elements and because it has extensive tooling support [19].

QPN combines the modeling concepts of Petri Nets and Queueing Networks. The essential elements of a QPN model

are immediate and timed places and immediate and timed transitions. As usual, places can hold colored tokens, transitions consume tokens in input places and produce tokens in output places. Immediate places always make their tokens available to transitions, timed places only make tokens available after they pass an internal service queue. Tokens can also be subject to departure discipline that imposes ordering restrictions. Immediate transitions have weights and are considered to happen instantaneously, timed transitions have firing rates and are considered to happen after a random delay. QPN models can be nested, a timed place can represent a nested QPN model, tokens arriving at the nested place are submitted to the nested model, tokens departing the nested model are made available to transitions.

4.1 Broker Model

We model the broker by a QPN model shown on Figure 8, which is nested in the QPN model of the measurement harness. This nesting is the reason why the model has a single input place and a single output place – tokens representing all incoming network traffic arrive at the input place, tokens representing all outgoing network traffic depart from the output place. Colors are used to distinguish messages from acknowledgments.

The path a message takes through the broker, implemented by multiple threads described in Section 2.2, is modeled as follows:

– A new message is represented by a token of the `msg` color that arrives in the `input` place. The `msg` token immediately transitions to the `accept-msg` place, with another token deposited in the `queue` place to model the storage occupied by the message.

– The `accept-msg` place represents the thread that reads the message from the network socket and stores it in a destination container. After processing, the `msg` token transitions to the `process` place.

– The `process` place represents the thread that reads the messages from the destination container and locates the message consumer. After processing, the `msg` token transitions to the `dispatch` place.

– The `dispatch` place represents the thread that sends the messages to the consumer through the network socket. After processing, the `msg` token transitions to the `system` place.

– The `system` place represents processing done by the operating system outside the broker, which does not count towards latency measured as messages enter and leave the broker, but still contributes to processor utilization. After processing, the `msg` token departs the broker network.

An acknowledgment will eventually confirm the reception of the message. The path the acknowledgment takes through the broker is modeled as follows:

– A new acknowledgment is represented by a token of the `ack` color that arrives in the `input` place. The `ack` token immediately transitions to the `accept-ack` place.
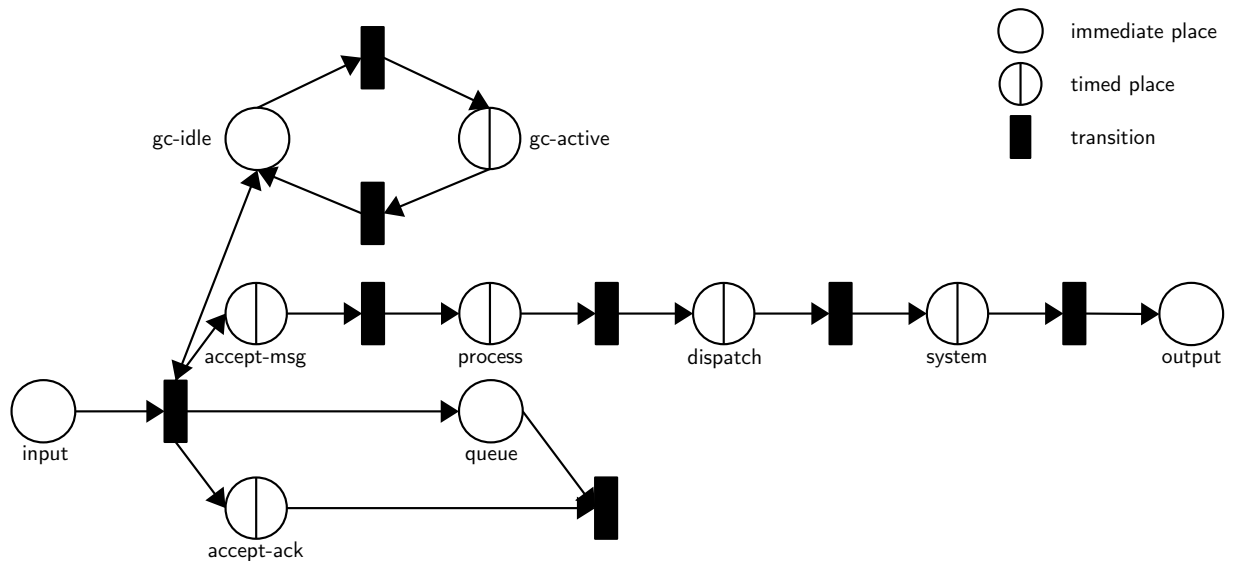
Figure 8: Nested broker QPN model

– The `accept-ack` place represents the thread that reads the acknowledgment from the network socket and recognizes the corresponding message as processed. After processing, the `ack` token is discarded by a transition that also removes one token from the `queue` place to indicate no storage is occupied by the message anymore.

4.2 Garbage Collection

Section 3.3 explains how garbage collection causes message coalescing in the operating system buffers. To model this behavior, we need to represent the garbage collection pauses. To do this, we observe how the broker allocates memory.

The objects maintained by the broker are primarily concerned with clients and messages and destinations. Individual instances of the objects represent individual clients and messages and destinations. The lifetime of these objects is necessarily related to the lifetime of the concepts they represent, simply because keeping them around for longer would cause memory leaks. This arrangement makes messages most important from the garbage collection perspective – objects related to messages have high allocation rates (on par with throughput rates) and short lifetimes (on par with roundtrip times).

Our experimental platform uses a generational garbage collector that will never promote message-related objects beyond the young generation (except if the young generation lifetime was shorter than the message roundtrip time, which is not common [24]). We can therefore imagine that each message passing through the broker will require allocating objects of certain average size. When the accumulated size of these objects reaches the young generation size, a young generation collection will be triggered and all these objects will be collected.

In the model on Figure 8, the garbage collector state is modeled using the `gc-idle` and `gc-active` places and a single `collector` token. The transition from the `input` place is enabled only when the `collector` token resides in the `gc-idle` place. Once the `collector` token transitions into the

`gc-active` place, no tokens transition to the `accept-msg` and `accept-ack` places, simulating a garbage collection pause. Multiple `garbage` tokens are used to represent allocated objects. The `garbage` tokens accumulate in the `gc-idle` place with each message, the transition from the `gc-idle` place to the `gc-active` place requires that enough `garbage` tokens accumulate.

4.3 Context Switching

Section 3.2 explains how the thread scheduling overhead impacts performance. We model this effect by introducing a new processor scheduling strategy into the QPN formalism. The strategy assumes each timed place represents a thread that keeps executing until no more work remains or until the scheduler executes another thread instead. In this context, we mimic two elements of a typical thread scheduler behavior – the overhead of switching from one thread to another and the limit on the time a thread is allowed to execute when other threads wait.

The strategy accepts the context switch duration c and the quantum duration q as parameters. Tokens from one timed place are processed until the accumulated execution time reaches q. At that moment, the strategy switches to executing tokens from another timed place, extending the execution time of the first token in that place by c.

4.4 Model Calibration

Before use, the model must be populated with a number of parameters. These are the resource demands of the processing performed by the broker threads, the resource demands related to processing outside the broker, and additional constants – the quantum duration, the context switch duration, the garbage collection threshold.

To collect the processor demands of the broker threads, we insert measurement probes into the broker source code, collecting time needed to execute the relevant code fragments. As a technical complication, the collected time may include passive waiting, which is not a processor demand. In our case, excluding passive waiting by the usual means (mea-

suring and subtracting the waiting duration or using clock that stops while waiting) was burdened by excessive overhead. We have therefore decided to measure the broker when near saturation and discard the upper decile of the processor demand measurements. Running near saturation makes passive waiting rare and because the times we measure are short, measurements that are distorted by waiting are easily identified by their extreme value. Our outlier filtering choice may have a slight systematic impact on modeled latencies.

To measure the processor demand related to processing outside the broker, we look at the difference between the overall processor utilization and the processor utilization due to the broker threads. From the data used in Figure 4, we estimate the processor demand of the `system` place to be 20% of the total processor demand used in the other timed places in the model.

The collected processor demands are necessarily burdened by measurement overhead. To assess and compensate, we scale the average processor demand per message to match the peak throughput. The data used in Figure 5 place the peak throughput at $22400\,\mathrm{msg/s}$, this gives us an average processor demand per message of $1/22400$ or $45\,\mu s$, of which 20% or $9\,\mu s$ is related to processing outside the broker. Without overhead compensation, the average total demand of the timed places in the model is $46\,\mu s$, we compensate by multiplying each broker demand by 0.96 to give the average total demand of $45\,\mu s$.

Section 3.2 explains how the amount of context switching per message changes between rates that generate peak utilization and peak throughput – the data used in Figure 4 shows these rates to be $11000\,\mathrm{msg/s}$ and $22400\,\mathrm{msg/s}$. Our model is constructed to involve five context switch penalties at rates close to peak utilization and zero context switch penalties at rates close to peak throughput, we can therefore calculate a single context switch penalty to be $(1/11000 - 1/22400)/5$ or $11\,\mu s$. This is a model parameter only, more context switches with shorter duration actually happen in reality. The other parameter related to scheduling – the quantum duration – is a part of the operating system settings.

Finally, we measure the number of messages that trigger garbage collection by looking at the garbage collection log. To avoid interference due to the virtual machine ergonomics [17], we fix the young generation size.

5. PERFORMANCE MODEL RESULTS

We show the behavior of our performance model on the same workloads that were used to illustrate the modeling issues in Section 3. We use transient message passing mechanism with acknowledgments to transport $975\,\mathrm{B}$ long byte array messages between the producer and the consumer. We vary the throughput rate, generating messages either in a regular pattern (producing one message every $1/r$ for throughput r) or in a bursty pattern (producing ten messages every $10/r$ for throughput r), and observe (and model) processor utilization and message latency at the given throughput rate. The model is fed the same distribution of the arrival times as the broker in the measurement experiments.

To measure message latency, we use dynamic library wrappers that intercept calls to the `recvfrom` and `sendto` socket functions at the points where messages enter and leave the broker. We use unique message identifiers embedded in the message body to associate the calls with individual mes-

sages. Our measurements indicate the overhead of wrapping the socket calls does not influence the achievable throughput noticeably, however, we collect the latency information separately from other measurements as a precaution.

The broker processor utilization information is collected through the `cpu` controller of the control group subsystem [29]. While more accurate than other sources, this method does not include the network processing part of the workload that occurs inside the kernel rather than the broker. We therefore also plot the information in the `proc` pseudo file system, which includes the kernel interrupt processing.

Our measurement harness, based on the performance tests included with the messaging middleware, uses dedicated producer and consumer machines to generate message traffic. We check throughput and utilization at both machines, making sure no bottlenecks limit the traffic. We collect the essential measurements for 10 minutes at each throughput rate and discard observations distorted by the warmup and shutdown phases (some measurements are timed and inspected manually).

5.1 Processor Utilization

Figure 9 shows how our model approximates the processor utilization. The measured values are the same as shown on Figure 4.
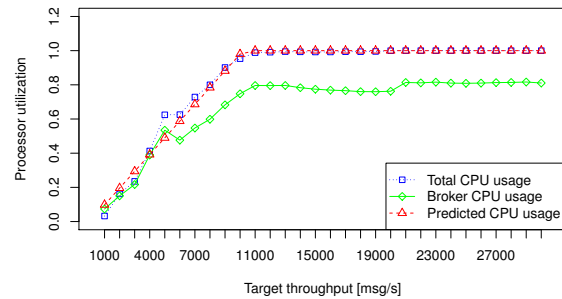

Figure 9: Prediction of broker processor utilization.

The fact that the model captures the linear increase of utilization with throughput is relatively mundane. As a more important contribution, the model also captures the fact that processor utilization peaks much sooner than at maximum throughput – in our measurements, the processor utilization exceeds 95% at $10000\,\mathrm{msg/s}$, but the maximum throughput is around $22400\,\mathrm{msg/s}$.

Compared to measurements, the model does not explain the increase of processor utilization around $5000\,\mathrm{msg/s}$. To explain this effect, we show the outbound network traffic information on Figure 10. We see that although the broker transmits an almost constant amount of bytes per message, at $5000\,\mathrm{msg/s}$ it suddenly uses about 25% more packets per message than at $4000\,\mathrm{msg/s}$. This increase in network traffic is reflected directly in the increase of processor utilization.

The reason for the network traffic increase is related to detailed behavior the TCP protocol, which can be observed by capturing the network traffic between the broker and the consumer. At $4000\,\mathrm{msg/s}$, the delay between sending a message and receiving an acknowledgment is smaller than the delay between sending two messages – at the TCP protocol level, packets carrying messages from broker to consumer and packets carrying acknowledgments from consumer to broker therefore alternate and each acts as a TCP ACK for
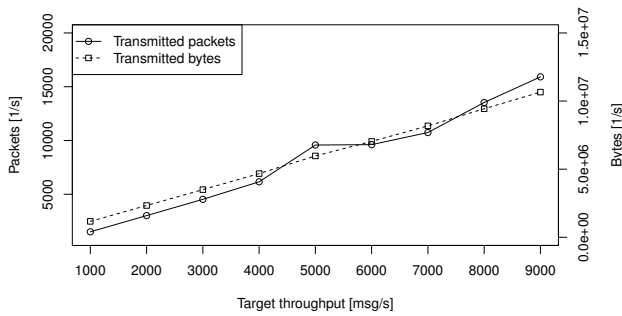
Figure 10: Network traffic from broker to consumer measured in packets and bytes.

the previous packet. At 5000 msg/s, the delay between sending a message and receiving an acknowledgment is close to the delay between sending two messages, which means that the broker sometimes manages to send two messages and then receive two acknowledgments – at the TCP protocol level, this means packets in the two directions no longer alternate and the flow control mechanism mandates sending extra TCP ACK packets, causing the increase in network traffic. While an interesting phenomenon per se, we believe this increase is outside a reasonable scope of performance modeling.
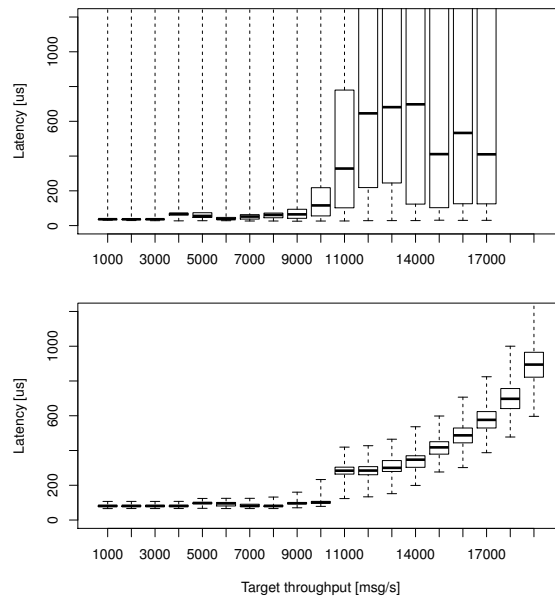
5.2 Message Latency

Message latency consists of the time spent processing and the time spent waiting. At low throughput, processing tends to dominate and latencies are relatively low. At high throughput, waiting tends to dominate and latencies are relatively high. To avoid losing detail due to scale, we examine several ranges separately.
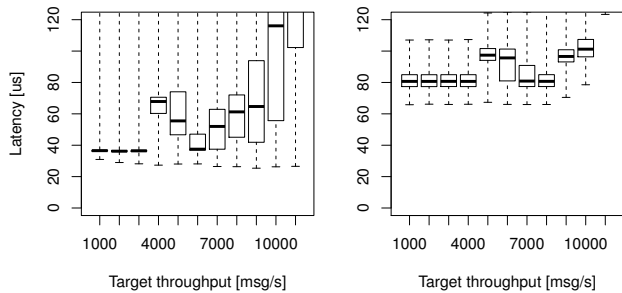
Figure 11: Measured (left) and predicted (right) message latencies at low throughput with regular workload.

Figure 11 shows the measured and predicted message latencies for low throughput rates generated in the regular pattern. Both the measurement and the model show the same trend, which starts with mostly constant latencies and gradually introduces variation. In absolute terms, the model overestimates the latency roughly by a factor of two. One reason for this difference is our calibration procedure, which removes outliers and scales the remaining values to maintain throughput – because throughput is sensitive to outliers in resource demands, removing outliers requires scaling the remaining values towards higher resource demands, which yield higher latency estimates.

Figure 12: Measured (upper) and predicted (lower) message latencies at low to medium throughput with regular workload.

Figure 12 shows the measured and predicted message latencies for low to medium throughput rates. Again, both the measurement and the model show the same trend, with latencies increasing by about an order of magnitude around the point where the throughput rate exceeds 10000 msg/s, which also happens to be the point where the processor utilization nears the peak. In absolute terms, the model does not exhibit the variation apparent in the measurements. We attribute this to the differences between our scheduler model and the real scheduler. While our scheduler model handles timed places in a round-robin fashion, the real scheduler on our platform enforces strict fairness. More complex scheduler models may help here [10].

Figure 13 completes the latency prediction information for all throughput rates generated in the regular pattern. When the producer attempts to generate messages at rates above peak throughput, the broker flow control restricts the producer to avoid message loss. In this situation, the message latency is determined by the storage threshold that triggers flow control – approximated by the maximum capacity of the `queue` place in our model. The fact that the model successfully estimates the very high latency is therefore due to a trivial model parameter, more important is the fact that the model estimates the throughput at which the flow control is triggered.

We point out that the behavior of the broker near peak throughput is unstable, with long periods of degraded performance. At high throughput rates, there is only little spare capacity to deal with backlog that may form due to minor disruptions. The broker therefore takes a long time to recover from such disruptions, which leads to large accumulated impact on latencies. Figure 14 illustrates this lack of stability.

As the sole exception to the rule that the model is fed the same distribution of the arrival times as the broker in the measurement experiments, Figure 13 uses modeled arrival
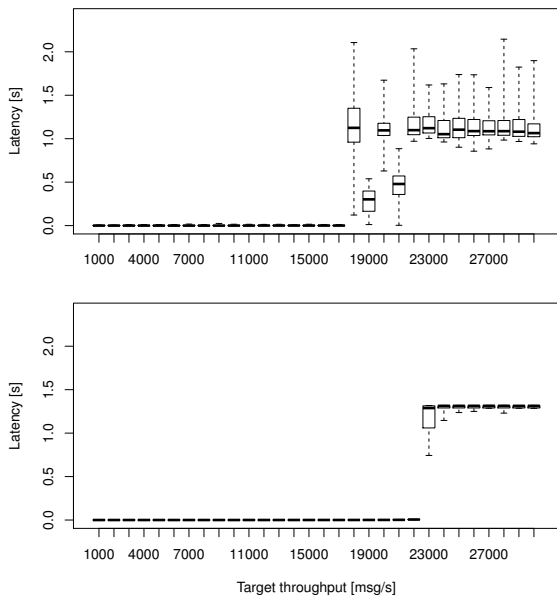
Figure 13: Measured (upper) and predicted (lower) message latencies at low to high throughput with regular workload.
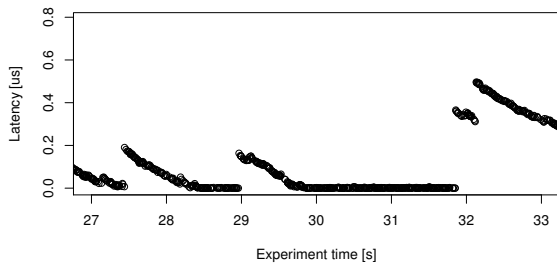

Figure 14: Unstable broker latencies at 19000 msg/s.

Figure 15: Measured (upper) and predicted (lower) message latencies at low to medium throughput with bursty workload.

Figure 16: Predicting impact of bursts on latency distribution. Constant throughput 5000 msg/s, left workload sending individual messages, right workload sending bursts of ten messages.

times that match the target throughput. This is necessary because at high throughput rates, the measured arrival times include broker flow control and therefore reflect the observed throughput rate rather than the target throughput rate.

Figure 15 shows the measured and predicted message latencies for low to medium throughput rates generated in the bursty pattern. Similar to the regular workload results, the bursty workload results show the same trend, with some overestimation of latency and some underestimation of variation. As an important factor, the model correctly predicts that introducing burstiness results in shifting the cluster of observed latencies en bloc, rather than creating multiple clusters as Section 3.1 illustrates. Compare Figure 16 with Figures 2 and 3.

5.3 Discussion

The results show that our model is capable of addressing the issues outlined in Section 3 as far as the trends are concerned – we predict that pipelined processing of message bursts results in a tight cluster of latencies, we show that varying thread scheduling overhead leads to utilization and throughput peaking at very different rates, and we do both
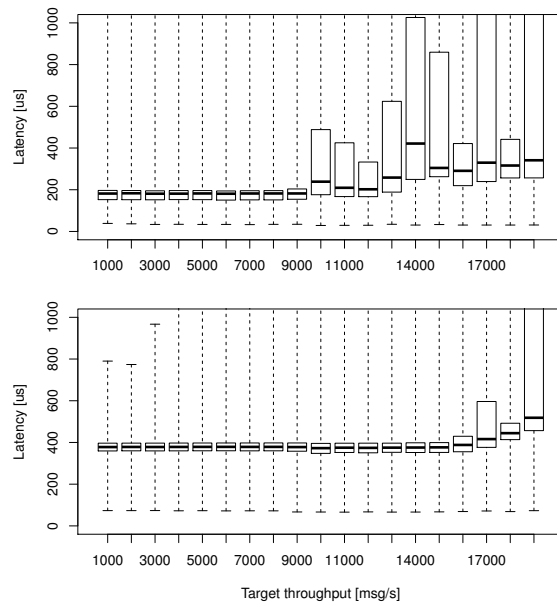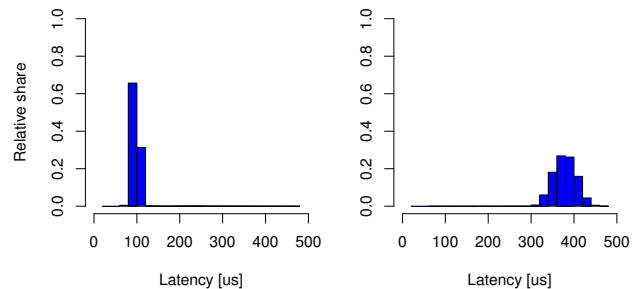
in presence of realistic message coalescing. To our knowledge, these effects were not captured by JMS models before.

The prediction of processor utilization is also very accurate in absolute terms. The same cannot be said about latency, where our predictions at low throughput are somewhat pessimistic and predictions at high throughput do not fluctuate as much as measurements – as we explain, this is in part due to model calibration and in part due to realistic scheduling being more complex than the scheduling disciplines of our model. The accuracy of latency prediction is very reasonable for uses outlined in Section 2. We should note that we do not use measured latencies to calibrate the model and still predict latencies of individual messages at very high resolution. Again, we believe this was not done in JMS models before.

An important question that we address in this discussion is whether our results can be generalized beyond our experiments. We present arguments for why we believe our work is not strictly limited to our experimental platform. We also

provide the source code and the data we have collected and used, so that more experiments are possible [1].

The most visible concern in generalizing our results is the range of workloads used in the experiments. While we vary both the throughput rate and the distribution of message arrival times, we use messages of equal size and type exchanged between a single producer and a single consumer. This contrasts especially with work that experiments on complex workloads such as SPECjms2007 [34].

The existing body of work on JMS performance provides a reliable summary of how individual workload parameters influence performance, and in fact suggests that extending the workload along many parameter axes would bring little principal benefit. Work such as [33, 13] shows there usually is a linear dependency between the message size and the associated processor demand, in contrast there usually is almost no dependency on the number of clients and destinations as long as messages are not replicated. Our model can be extended to support multiple message sizes and types by using multiple token colors with different associated processor demands, as used in [34]. Support for multiple clients and destinations should not require principal changes to our model either – the relevant message processing paths in our broker are reasonably similar to the message processing path of our workload. On the other hand, workloads that require persistent message storage would represent a challenge, due to the dominating nature of storage latencies in the model that otherwise deals in microseconds.

Experiments with limited workloads provide an important benefit in that they help isolate individual modeling concerns. Tracking the performance issues that we focus on in a complex workload is virtually impossible – although they are still likely to exist (there is no reason why context switching or garbage collection would go away with more complex workloads), their performance impact is combined with the performance impact of workload variability.

As one item, our work covers the impact of thread scheduling overhead on performance. The exact impact is both workload-dependent and platform-dependent – in general, we can expect the need for context switching to increase with more clients and destinations (because clients and destinations are served by separate threads) and to decrease with more cores (because threads will not compete for cores as much). As long as there are more clients and destinations (and therefore internal broker threads) than cores, the thread scheduling overhead should be present. The performance impact of individual context switches is also likely to increase with heavier workload, because such workload is associated with heavier memory cache traffic and context switches may flush memory cache content.

As another item, our work describes pipelined message processing. This is an architectural decision that concerns the broker implementation, one that is apparently reasonable but certainly not the only one possible. Brokers that use different architectures may require different models – unfortunately, determining the broker architecture for performance modeling purposes is a demanding endeavor even when broker sources are available, and not likely to get easier with closed source brokers.

Finally, our work requires measuring durations of operations that occur inside the broker. This is again easier when broker sources are available, but with current instrumen-

tation techniques [28, 27], instrumenting major control flow locations such as network communication or thread synchronization inside closed source brokers is also possible.

6. RELATED WORK

Performance modeling of distributed systems based on messaging is a frequent research subject. Some authors choose to work at a relatively high abstraction level, modeling complex networks of computers that communicate through messaging. At this level, details of individual node performance are typically simplified and the modeling efforts investigate important high level properties such as system capacity limits. Some high level modeling work is very close to our research, for example [18] proposes a method of constructing models that approximates communicating nodes with M/M/1 queues and uses QPN for experimental evaluation. Our model can improve this approximation – the possibility is actually mentioned by the authors, but there is not enough technical information in the paper to estimate the contribution of such model change to accuracy.

In [34], the SPECjms2007 benchmark is modeled with QPN, using G/M/8 queues to approximate processors and G/M/1 queues to approximate storage. The authors achieve significant modeling accuracy on a variety of workloads – in contrast with our work, the authors cover a wide variety of message sizes and types and quality-of-service settings, but keep the broker processor utilization below 80 %. The authors use a nested QPN model with three timed places in tandem representing the processor, the storage and the network resources – our model can again improve this approximation when exploring workloads that lead to high broker utilization, provided it is extended with more quality-of-service settings.

In [26], the broker is approximated with an M/M/* queue, similar queues are used to model a component container and a database. The authors predict throughput and latency in a closed workload with zero think time – a situation which exercises the ability of the broker to serve individual clients fairly, leading to a linear dependency between the number of clients and the latency.

In a broader context, other formal tools are used to model messaging networks – for example, probabilistic timed automata are used to capture behavior in presence of message loss in [12]. We observe that high level modeling is considered valuable even when validation against a real system is not done.

Some studies focus on explorative evaluation of broker performance. Among early examples is [6], where performance of two JMS brokers were evaluated. The measurements focus on maximum sustainable throughput with various quality-of-service settings. A thorough study of JMS performance is [33], where one JMS broker is examined using the SPECjms2007 benchmark. Although these studies do not construct performance models (and sometimes do not even name the examined brokers due to licensing restrictions), they are a valuable source of common performance trends that can be observed across brokers. One typical observation is that message size is an important factor, increase in message size causes linear increase in processor demand. In contrast, the number of clients and destinations does not seem to be important when the total traffic re-

mains constant. These observations support our discussion on including additional validation workloads in our work.

Explorative evaluation of broker performance can help create fitted models. This is the case in a large range of experiments summarized in the doctoral thesis [13]. In a number of separate publications, these experiments investigate parameters such as throughput [14] or latency [30] and construct fitted models that approximate the measurements. Interestingly, some of the experiment parameter ranges are chosen with the assumption that peak processor utilization implies peak throughput, which we show is not necessarily true.

A thorough process of building a fitted JMS model through explorative experiments is described in [11]. The experiments are carried on the ActiveMQ 5.3 messaging middleware, which makes the results even closer to ours. Again, the choice of experiment parameter ranges equals peak utilization workload with peak throughput workload. The work also demonstrates the difficulties of building an accurate model for the range of workloads we consider – the processor utilization in the experiments used to create the fitted model never exceeds 50 %, and the parameter dependencies are collected in experiments that assume no resource contention, which may limit suitable parameters.

Another work that creates a fitted JMS model is [9], the authors show how the model can be integrated into a larger performance model that captures particular SPECjms2007 interactions. The focus is on the integration process, technical details of the JMS model are not investigated. Similar approach in the context of component systems was investigated in [25].

Our work also touches on the issue of constructing a performance model with limited knowledge of the modeled system. Other authors have tackled this problem, in [4] an enterprise application model is constructed from partial architectural information and collected execution traces.

The problem of determining resource demands with limited measurement ability in the context of workload with multiple request types was addressed in [32] and [22], the authors of [42] estimate and adjust performance model parameters by tracking the prediction error. Using similar techniques in combination with artificial workloads crafted to exercise particular elements of the broker architecture can likely provide enough information to calibrate the performance model even when measurements based on instrumentation are not available.

As a summary to our related work survey, we believe our model can provide accuracy improvement in the context of existing modeling work, which mostly acknowledges that broker performance is implementation specific and provides mechanisms for plugging detailed broker models into platform independent application models. Where fitted models are used, our work highlights important effects related to pipelined processing and message coalescing that should be considered when selecting the model parameters. We also believe our work is the first to attract attention to the significant impact of pipelined processing and message coalescing in the context of broker performance modeling.

7. CONCLUSION

Our work is based on observing performance of the ActiveMQ messaging middleware. We attract attention to the fact that pipelined processing (the act of handling messages in stages by multiple broker threads) and message coalescing (the act of processing several adjacent messages together at some stage) can interact even with very simple workloads to create performance effects of significant magnitude that the existing performance models do not capture. We provide technical explanation for these effects and design a broker model that describes them.

We show that our model provides a reasonably accurate approximation of the identified effects. As an important distinction – where the existing JMS models may capture the effects by calibrating for a particular workload, our JMS model is built by analyzing and reflecting the reasons behind the effects. Our work therefore touches upon a broader question of how calibrating and validating the model against the same workload – something that is regularly done in model validation experiments – contributes to perceived model accuracy.

Although our work has used a specific platform and specific workloads, we argue that the effects we observe can reasonably occur on other platforms. We provide the source code and the data we have collected and used to make more experiments possible [1].

Acknowledgement

8. REFERENCES

[1] Complementary material.
    http://d3s.mff.cuni.cz/papers/jms-modeling-icpe.
[2] Apache Software Foundation. *Apache ActiveMQ*.
    http://activemq.apache.org.
[3] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Architecture-Based Reliability Prediction with the Palladio Component Model. *Transactions on Software Engineering*, 38(6), 2011.
[4] F. Brosig, S. Kounev, and K. Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proceedings of ROSSA 2009*, 2009.
[5] L. Bulej, T. Bures, J. Keznikl, A. Koubková, A. Podzimek, and P. Tuma. Capturing performance assumptions using stochastic performance logic. In *Proceedings of ICPE 2012*. ACM, 2012.
[6] S. Chen and P. Greenfield. QoS Evaluation of JMS: An Empirical Approach. In *Proceedings of HICSS 2004*. IEEE, 2004.
[7] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model Evolution by Run-Time Parameter Adaptation. In *Proceedings of ICSE 2009*. IEEE, 2009.
[8] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner. Parametric Performance Completions for Model-Driven Performance Prediction. *Performance Evaluation*, 67(8), 2010.
[9] J. Happe, H. Friedrich, S. Becker, and R. Reussner. A Pattern-Based Performance Completion for Message-Oriented Middleware. In *Proceedings of WOSP 2008*. ACM, 2008.

[10] J. Happe, H. Groenda, M. Hauck, and R. Reussner. A Prediction Model for Software Performance in Symmetric Multiprocessing Environments, 2010.

[11] J. Happe, D. Westermann, K. Sachs, and L. Kapova. Statistical Inference of Software Performance Models for Parametric Performance Completions. In *Proceedings of QOSA 2010*. Springer, 2010.

[12] F. He, L. Baresi, C. Ghezzi, and P. Spoletini. Formal Analysis of Publish-Subscribe Systems by Probabilistic Timed Automata. In *Proceedings of FORTE 2007*. Springer, 2007.

[13] R. Henjes. Performance Evaluation of Publish/Subscribe Middleware Architectures, 2010.

[14] R. Henjes, M. Menth, and C. Zepfel. Throughput Performance of Java Messaging Services Using WebSphereMQ. In *Proceedings of ICDCS 2006 WORKSHOPS*, 2006.

[15] V. Horky, F. Haas, J. Kotrc, M. Lacina, and P. Tuma. Performance Regression Unit Testing: A Case Study. In *Proceedings of EPEW 2013*. Springer, 2013.

[16] Internet Engineering Task Force. *Congestion Control in IP/TCP Internetworks.* http://tools.ietf.org/html/rfc896.

[17] R. Jones and R. Lins. Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html.

[18] S. Kounev, K. Sachs, J. Bacon, and A. Buchmann. A Methodology for Performance Modeling of Distributed Event-Based Systems. In *Proceedings of ISORC 2008*. IEEE, 2008.

[19] S. Kounev, S. Spinner, and P. Meier. QPME 2.0 - A Tool for Stochastic Modeling and Analysis Using Queueing Petri Nets. In *From Active Data Management to Event-Based Systems and More*, 2010.

[20] S. Kounev, S. Spinner, and P. Meier. Introduction to Queueing Petri Nets: Modeling Formalism, Tool Support and Case Studies (Tutorial Paper). In *Proceedings of ICPE 2012*. ACM, 2012.

[21] H. Koziolek, B. Schlich, C. Bilich, R. Weiss, S. Becker, K. Krogmann, M. Trifu, R. Mirandola, and A. Martens. An Industrial Case Study on Quality Impact Prediction for Evolving Service-Oriented Software. In *Proceedings of ICSE 2011*. ACM, 2011.

[22] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating Service Resource Consumption from Response Time Measurements. In *Proceedings of VALUETOOLS 2006*. ACM, 2006.

[23] C. Li, C. Ding, and K. Shen. Quantifying The Cost of Context Switch. In *Proceedings of ExpCS 2007*. ACM, 2007.

[24] P. Libič, P. Tůma, and L. Bulej. Issues in Performance Modeling of Applications With Garbage Collection. In *Proceedings of QUASOSS 2009*. ACM, 2009.

[25] Y. Liu, A. Fekete, and I. Gorton. Design-Level Performance Prediction of Component-Based Applications. *IEEE Transactions on Software Engineering*, 31(11), 2005.

[26] Y. Liu and I. Gorton. Performance Prediction of J2EE Applications Using Messaging Protocols. In *Proceedings of CBSE 2005*. ACM, 2005.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of PLDI 2005*. ACM, 2005.

[28] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In *Proceedings of AOSD 2012*. ACM, 2012.

[29] P. Menage. Linux Control Groups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[30] M. Menth and R. Henjes. Analysis of the Message Waiting Time for the FioranoMQ JMS Server. In *Proceedings of ICDCS 2006*, 2006.

[31] NS-3 Project. *NS-3*. http://www.nsnam.org/.

[32] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. Dynamic Estimation of CPU Demand of Web Traffic. In *Proceedings of VALUETOOLS 2006*. ACM, 2006.

[33] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance Evaluation of Message-Oriented Middleware Using the SPECjms2007 Benchmark. *Performance Evaluation*, 2009.

[34] K. Sachs, S. Kounev, and A. Buchmann. Performance Modeling and Analysis of Message-Oriented Event-Driven Systems. *Journal of Software and Systems Modeling*, 2012.

[35] B. Sigoure. How Long Does It Take To Make A Context Switch ? http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[36] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[37] Sun Microsystems. *Java Message Service Specification Version 1.1*, 2002.

[38] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and Challenges in the Performance Analysis of Real Disk Arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), 2004.

[39] T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester. Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models. *IEEE Transactions on Software Engineering*, 31(8), 2005.

[40] H. Xu. GSO: Generic Segmentation Offload. http://lwn.net/Articles/188489/.

[41] H. Xu. net: Generic Receive Offload. http://lwn.net/Articles/311357/.

[42] T. Zheng, C. M. Woodside, and M. Litoiu. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering*, 2008.