Auto-tuned Publisher in a Pub/Sub System: Design and Performance Evaluation

Sowmya Balasubramanian[1], Dipak Ghosal[1,2], Kamala Narayan Balasubramanian[1],
Eric Pouyoul[1], Alex Sim[1], John Wu[1], and Brian Tierney[1]
[1]Lawrence Berkeley National Laboratory
[2]University of California, Davis

***Abstract*—Pub/sub systems form the underlying framework for many distributed applications. It consists of one or more publishers that publish to a broker from which subscriber can retrieve the published content. Many large and small distributed applications including social networking applications use the pub/sub model. In this paper we consider a pub/sub system in which the publisher, the broker, and the subscriber are in different administrative domains. While general pub/sub systems provide reliability of message delivery, good end-to-end latency performance in a multi-domain environment require that the pub/sub system adapt to workload changes and bottlenecks in the different sub-systems. This study is motivated by two applications. First, a pub/sub based Simple Lookup Service (sLS) that is used in perfSONAR to provide information about network performance in R&E networks. Second, the pub/sub system that is used to distribute alerts generated in the data pipeline in the Zwicky Transient Factory (ZTF). We consider a publisher with a multi-threaded architecture that uses batching to coalesce messages over some variable polling period. We propose a control algorithm that auto-tunes the batch-processing parameters namely, the batch size and the polling interval, to the input message load and to any upstream broker-side congestion that minimizes end-to-end latency and maximizes throughput. Using a detailed simulation model, we demonstrate the performance of the control algorithm for different scenarios. We then study the performance using a real-trace obtained from the Simple Lookup Service (sLS). We show that the proposed algorithm quickly adapts to rapid changes in the workload and yields lower mean end-to-end delay performance when compared with the delays in the current deployment.**

Keywords: Pub/Sub System, Performance Auto-tunning, Simulation Analysis, perfSONAR, Lookup Service, Experimental Evaluation

I. Introduction

Publish-subscribe system (also written as pub/sub or sometimes pub-sub system for short) is a distributing computing communication paradigm that is widely used in many distributed applications [6]. It provides an asynchronous communication framework between producers of content (publishers) and consumers of content (subscribers). Systems built on the pub/sub model decouple the publishers and subscriber by having intermediate broker (also referred to as the message queue). Content generated by the publishers is sent to the broker which stores them in different message queues based on *a priori* defined channels. Subscribers register with the broker for messages from a subset of the channels. When new messages are available, the broker delivers messages to the subscribers based on their subscription.

The pub/sub model is the underlying framework for many distributed application. Broadly speaking, pub-sub systems can be categorized as content-based or channel-based [13]. In content-based pub/sub systems, each message has a tag which is a set of attribute/value pairs. Subscribers express their interests in content as logical rules over the attributes. The broker matches the subscriber interest to the message tags and forwards the content to the appropriate subscribers [8], [15]. On the other hand, in channel-based pub/sub systems, subscribers specify their interest in content by submitting subscriptions to the broker for specific topics which are referred to as channels [8], [7]. The publisher tags the content with the channel name and sends it to the broker. All subscribers who have indicated interest in the channel receive the content using a push or a pull model. Large social networking applications such as Facebook [18], LinkedIn [12], as well as companies like Google [1] rely on the channel-based pub/sub model.

With the growth of distributed applications that use the pub/sub model, methods to provision resources in the pub/sub system in a flexible manner has gained interest. This is particularly the case when these systems are deployed as a cloud service [9], [13], [8]. In [13] the problem of resource allocation in a scalable pub/sub system is formulated as an optimization problem with different objective functions. The pub/sub system is modeled using a multi-class open queuing network model which is solved to obtain the system performance measures. The study proposes a greedy algorithm to determine the resource allocation to the pub/sub system. An evaluation based on simulation on a real system shows that the proposed solution outperforms the baseline and is robust in dealing with high-volume and fast-changing workload.

In this paper, we consider a pub/sub system that operates in a multi-domain network. In particular, we consider the scenario where the publishers, the broker, and the subscribers are in different administrative domains and hence no guarantees can be made regarding the compute, storage, and networking resources that are allocated for these services. Consequently, differences in resources allocated for these services and in the inter-networking can lead to bottlenecks resulting in poor throughout and latency performance. Within this broader context, this work focuses on the publisher. We consider a multi-threaded publisher that receives requests from external sources

that need to published to the broker. As the message arrival rate can change and there can be broker-side congestion, the publisher needs to dynamically adapt processing the incoming requests and transferring them to the broker in such a way that optimizes throughput and latency performance. In a multi-threaded architecture this can be done by batching processing requests. In this paper, we propose an auto-tuning algorithm in the publisher that adapts the batch-processing parameters both to the request load and upstream broker-side congestion.

The main contributions of this paper are the following:

1) We consider a multi-threaded publisher and propose an auto-tuning algorithm that adapts the batching parameters namely, the batch size and the polling interval, to the input request load as well as to the broker-side bottleneck. Specifically, the control algorithm employs an additive-increase and multiplicative-decrease algorithm to determine the batching parameters that minimizes the latency.
2) We develop a detailed simulation model of the multi-threaded publisher. Using simulation analysis we demonstrate that for synthetic workloads following Poisson arrival process, the tuning algorithm adapts to fast changing workload and has a significantly better latency performance than the baseline case without auto-tuning.
3) We also study the performance of the algorithm using a real traffic trace from the pub/sub based Simple Lookup Service (sLS) that is used with perfSONAR [10]. We show that our proposed algorithm significantly reduces end-to-end delay compared to the delays experienced in current deployed system.

The remainder of this paper is organized as follows. In Section II, we present an overview of the pub/sub system and outline the scope of this study. We then briefly describe the two applications that motivated this study namely, the Simple Lookup Service (sLS) used with perfSONAR and the alert system used in ZTF. In Section III, we describe the system model. In Section IV we present the auto-tuning algorithm. In Section V we present the simulation results. In Section VI we present the results based on a real trace. In Section VII we present the related literature and finally, in Section VIII we present the conclusions and future work.

II. Pub/Sub System

Figure 1 shows the different components of a generic pub/sub system. It consists of publishers (Ps) that generate content that is consumed by subscribers (Ss). The publishers provide the interface to sources that generate the content and after some processing transfers the content to the Message Queue which provide different queues for different content. Subscribers register with the Broker for different subset of the content. When appropriate content is available, they are pushed to (or pulled by) the appropriate subscribers. In this paper, we consider a pub/sub system that operates in a multi-domain network. In particular, we consider the scenario where the publishers, the broker, and the subscribers are in different administrative domains. To set the context, below we briefly

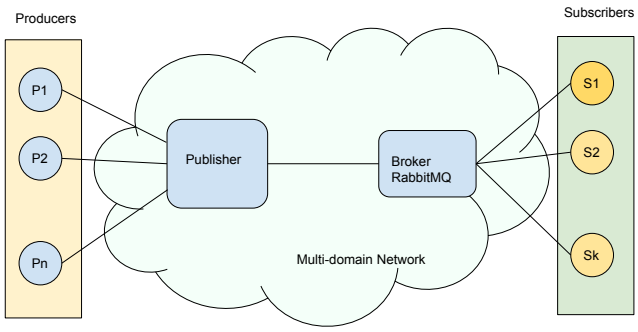

Fig. 1: The various components of a generic pub/sub system.

discuss two applications perfSONAR and Zwicky Transient Facility (ZTF) that require the use of a pub/sub system.

A. sLS for perfSONAR

The global Research & Education (R&E) network is comprised of hundreds of international, national, regional and local-scale networks. While these networks all interconnect, each network is owned and operated by separate organizations (called "domain") with different policies, customers, funding models, hardware, bandwidth and configurations. This complex, heterogeneous set of networks must operate seamlessly from "end-to-end" to support science and research collaborations that are distributed globally.

perfSONAR is an open source software project that enables seamless deployment of a network monitoring infrastructure across multiple administrative domains. perfSONAR serves two primary roles. The first role is to help set expectations on what the network is capable of providing. Often users blame the network when performance is slow, while in fact the bottleneck may be the disk, or a host tuning issue, or some other problem besides the network. The second perfSONAR role is to help identify "soft failures" in the network. A soft failure is where packets still arrive at their destination, but much slower than expected, based on available capacity. The perfSONAR "Toolkit" is a set of tools that include Network Monitoring Tools (iperf3, iperf, nuttcp, ping, traceroute, tracepath, and others), Test Scheduler to schedule test, Archive Manager to store test results, and Data Analysis Tools/Dashboard to visualize test results. perfSONAR is used by several Network Operations Centers worldwide to monitor for soft failures between their sites and key remote sites.

The Simple Lookup Service(sLS) based on the pub/sub model allows perfSONAR community to register all perfSONAR and some other non-perfSONAR services. The design of the Simple Lookup Service(sLS) does not define any particular architecture or deployment since it is intended to support different communities. Instead, the sLS defines only a single building block, the Simple Lookup Service(sLS) node. The sLS node provides persistent storage for the information and enables searching for information using a REST/JSON

API. An sLS node can be deployed as a centralized, single instance service. However, a common architecture is one that uses geographical localization and caching in order to minimize network latency and/or the effect of lossy long distance paths. Each region deploys an instance of the Simple Lookup Service or a topology of instances. Services from that region then register to it. The content of other regions is duplicated by deploying additional Simple Lookup Service using the pub/sub, streaming API. Those act as local cache of the services registered in other regions and, themselves, are registered as such, in the local Lookup Service or are published in the region for the client's consumption. Note that this model can be generalized to support community based deployments of the lookup service. In this architecture an sLS node may act 1) as a Core node which is the source of truth or 2) as a Cache node which is a read-only replicas that contain all or subset of information. The architecture may have one or

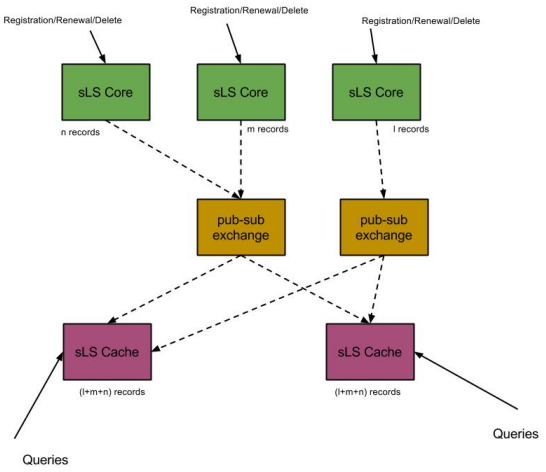

Fig. 2: sLS deployment with caches.

more core nodes that accept registration, renewal and delete requests (write requests). They contain only a portion of the data and may or may not process query requests. On the other hand, cache nodes are zero or more sLS nodes that contain a read-only view of a subset or the entire data set. They are exclusively for processing queries and do not accept registration, renewal and deletion requests.

Figure 3 shows a trace of the registration, renewal and delete requests (write requests) that are made to a core node in the sLS system.

B. Zwicky Transient Factory (ZTF)

The Zwicky Transient Facility (ZTF) is a newly commissioned optical synoptic survey that scans a large area of the night sky to identify changes [4]. It is capable of finding transients and variable stars an order of magnitude faster than the previous generation of synoptic surveys. Since these rare transients, such as exploding supernovae, are important for understanding cosmology and fundamental physics, a large

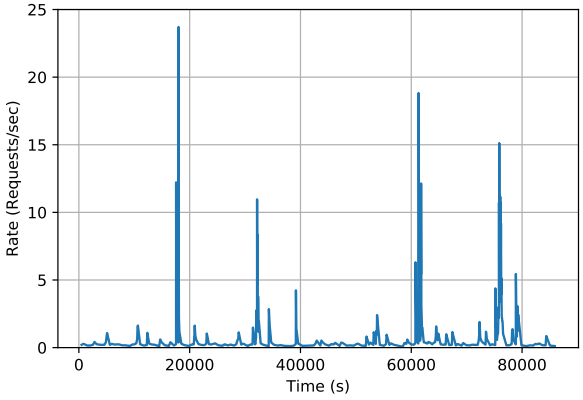

Fig. 3: Number of write requests that are made to a core node in the sLS system. The rate shown is the average rate over 100 samples.

community of scientists around the world are interested in the transients identified by ZTF. The ZTF project team is planning to publish the transients they identified through a publisher-subscriber model[1].

The ZTF camera is designed to capture an image every 45 seconds. From one image, about 1000 changes are expected to be identified, which leads to 1000 alerts. Each alert might contain about 20KB of data. Given that about 1 million such alerts are anticipated per night, about 20GB of alert data are expected. To avoid loss of data, such alert events are to be replicated among multiple alert databases and brokers.

The alerts are expected to be bursty as number of transients are not uniformly distributed in the sky. One common outcome of an alert is to trigger a large telescope to be used to perform follow up observations. In many cases, these follow ups can only be performed in the night, there is likely more pressure to process the alerts produced early in the night so that a follow up could be arranged within the same night. Such varying urgency in response time could place interesting demands on the alert distribution system.

III. System Model

Figure 4 show the various processes in the publisher. The main receive thread (MRT) receives requests from external sources; each request carries a message that needs to be published. The main publisher thread (MPT) coordinates publishing messages to the subscriber using the broker. MRT and MPT share a common pool of worker threads (WTs) that perform much of underlying functions on behalf of MPT and MRT. They also share a data store which temporarily store messages until they have been published and received by the subscriber. In the following paragraphs we describe the key functions of each component in detail.

[1]A mock stream system of alerts is available at https://github.com/lsst-dm/alert_stream.

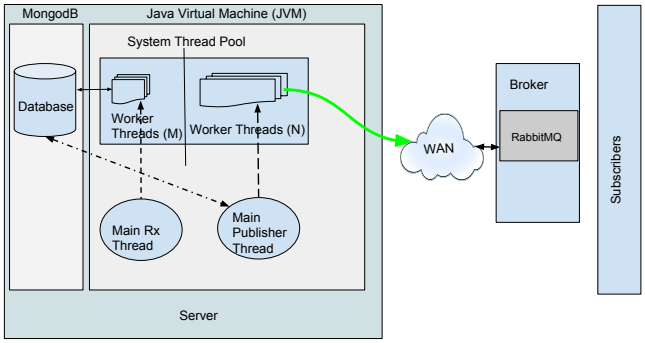

Fig. 4: The various components of the pub/sub system and the main threads in the publisher.

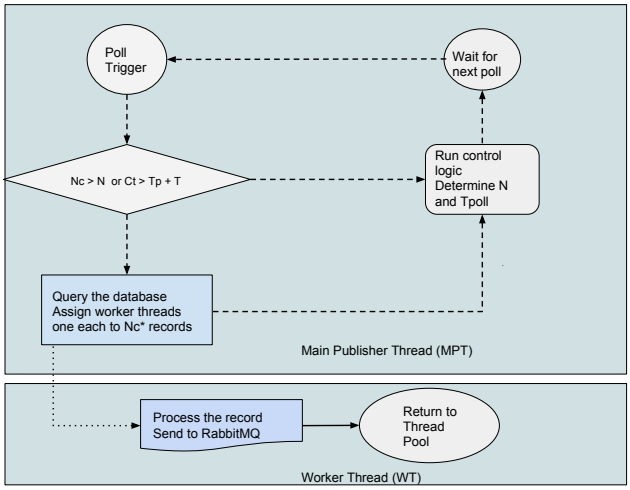

Fig. 5: The flowchart of the Main Publisher Thread (MPT).

a) Main Receive Thread (MRT):: The MRT thread is the external interfacing thread. Messages to be published generated from external sources are received by the MRT which assigns a WT to each request. To handle cases when requests arrive faster than what the MRT can process, the MRT maintains a finite buffer to queue the requests. When the buffer is full any incoming request is dropped.

b) Main Publisher Thread (MPT):: Functions of MPT are a) to read a number of messages from the data store and b) to assign these messages to WTs. The manner in which MPT reads messages from the data store is governed by three parameters N, T_{poll}, and T. The parameter N denotes the maximum number of requests that the MPT attempts to retrieve at a time from the data store. T_{poll} determines how often MPT checks to see if N messages have accumulated. Finally, T is the maximum time MPT waits for N messages to accumulate and is set to be multiple of T_{poll}. When T expires, the MPT retrieves however many requests there are in the data store and processes them. These three parameters impact the end-to-end delay and throughput performance of the pub/sub system. In this study we set T to a fixed value of T_{poll}^{max} and auto-tune the values of N and T_{poll}. The flowchart of MPT is shown in Figure 5.

c) Worker Thread (WT):: Worker Threads (WTs) perform tasks on behalf of MRT and MPT. When assigned on behalf of the MRT, a WT processes the message and writes it to the data store. When a WT is assigned a message by the MPT, it processes the message and then transfers the message to the broker. The WT returns back to the WT pool only once it receives an ACK indicating that the message has been received by the broker[2]. The communication between a publisher (WTs) and the broker is performed over a single TCP connection.

d) The Data Store: The data store can be a regular database or an in-memory database. When a WT is assigned a request by the MRT, it writes the corresponding message to the data store. MPT reads one or more messages from the data store. We assume that writes have non-preemptive priority over reads. Furthermore, the time to read messages comprises of a fixed cost for accessing the data store and a time that is proportional to the number of messages retrieved.

[2]The ACK does not imply that the request is received the subscriber

IV. Auto-tuning Algorithm

The goal of this study is to develop an auto-tuned publisher that adapts to changing workload (message arrivals) and broker-side congestion. The latter includes bottleneck in the communication link between the publisher and the broker, resource exhaustion within the broker, or a slow subscriber. Congestion in the communication link would result in network layer delay and/or packet drops which would lead to TCP congestion control resulting in longer time for a WT to complete its task. Resource exhaustion in the broker or a slow subscriber, would result in queue build-up in the broker which would eventually lead to TCP flow control. This would again lead to the WTs taking longer time to service a request. In this study, we do not distinguish between these different types of broker-side bottlenecks. We will model the impact of these in terms of a message-oriented flow control which results in longer service time for the WT to transfer a message to the broker. This is further explained in SectionV. With the above context, we discuss the various parts of the auto-tuning algorithm in the publisher

A. Priority Access to WT Pool

As mentioned before, MRT and MPT share a common WT pool. Rather than statically partitioning the WT pool between the MRT and the MPT, the entire WT pool is shared between by MRT and MPT. However, the MRT has non-preemptive priority over MPT in assigning tasks to WTs. The goal is to ensure that if the workload changes and the message rate increases, the MRT can use more and more of the WTs to process incoming requests and write message to the data store.

This is achieved by ensuring that when a WT finishes its task whether on behalf of MPT or MRT, it first checks to see if there are any remaining requests in the MRT queue before returning to WT pool. This priority scheme will minimize the number of requests that are dropped at the finite input buffer of the publisher.

B. Auto-tuning N and T_{poll}

The values of N and T_{poll} should adapt to the workload and any broker-side bottleneck. When the incoming message rate is low, both N and T_{poll} should be set to small values so that requests are handled almost as soon as they arrive. When the incoming request rate is high, N and T_{poll} can be increased to achieve the benefit of batching. This is similar to the interrupt coalescing that occurs in high-speed network interfaces [17]. In this paper we adopt a control algorithm that adapts the value of N and T_{poll} depending on the request load and the broker-side bottleneck. This is explained below.

TABLE I: A list of the parameters and their typical values used in the simulation analysis.

T_{poll}	Current value of polling interval
T_{poll}^{min}	Minimum value of the polling interval. This is set to 50 ms.
T_{poll}^{max}	Maximum value of the polling interval. This is set to 200 ms.
T	The maximum wait time for N messages to accumulate
N	Current value of the number of requests that will be batched
N^{max}	Maximum number of requests that will be batched
N^{min}	Minimum number of requests that will be batched. In this study it is set to 1
T_{mpt}	The time required by MPT to assign the requests to the WTs
λ_s	Recent sample of the load
λ_{est}	Estimate of the load.
α, γ	These are filter gain parameters for estimating the load and broker side capacity. These are set to 0.5 and 0.75, respectively.
β	This parameters is used to set T_{poll}. It is set to 1.2.
RTT	The roundtrip time between the publisher and the broker. Set at average value of 140 ms.

The different variables for auto-tuning algorithm are shown in Table I. The auto-tuning is based on the following heuristics:

1) If the load is very low, N should be set to N_{min} and T_{poll} should be set to T_{poll}^{min}. If $N_{min} = 1$, setting T_{poll}^{min} small would amount to processing the message almost as soon as it arrives. This will be similar to an interrupt driven processing which would minimize the latency.
2) As the load increases both N and T_{poll} should be increased. Coalescing messages has multiple benefits. It allows a number of requests to be read from the data store in one read. This amortizes the fixed cost of a data store read over a number of messages. Furthermore, messages can be assigned to WTs which can process them in parallel and transfer them to the broker.
3) However, coalescing too many messages by increasing T_{poll} has multiple disadvantages. First, it increases latency. Second, if enough WTs are not available, then MPT will get blocked which if it becomes long enough may cause the MPT to miss the poll timer. Another, important issue relates to communication between the publisher and the broker. Typically, there is a single TCP connection between the publisher and the broker. If a large number of WT simultaneously push data through the connection then the broker may become bottlenecked and flow control the publisher. This will lead to WT threads taking longer time to complete their transfers.

*a) **Estimating the Load**::* At each T_{poll}, we have a sample of the load in the previous polling interval denoted by λ_s. We apply an Exponential Weighted Moving Average (EWMA) filter to estimate the load for next polling interval. Specifically, for the nth polling period the estimated load λ_{est}^n is given by

$$\lambda_{est}^n = \alpha\lambda_{est}^{n-1} + (1-\alpha)\lambda_s \quad (1)$$

Note that the polling interval is not fixed since T_{poll} changes as explained below.

*b) **Setting** T_{poll}::* Each time MPT reads a batch of messages from the data store, it assign them to WTs. The MPT returns only when all the messages have been assigned. The time from when the MPT read from the data store and completes assigning messages to WTs is T_{mpt}. Ideally, T_{poll} should be set so that it closely tracks T_{mpt}. Specifically, we set $T_{poll} = max(min(\beta T_{mpt}, T_{poll}^{max}), T_{poll}^{min})$ where β is set to a factor greater than 1 and the max and the min functions bound the value of T_{poll} within T_{poll}^{max} and T_{poll}^{min}.

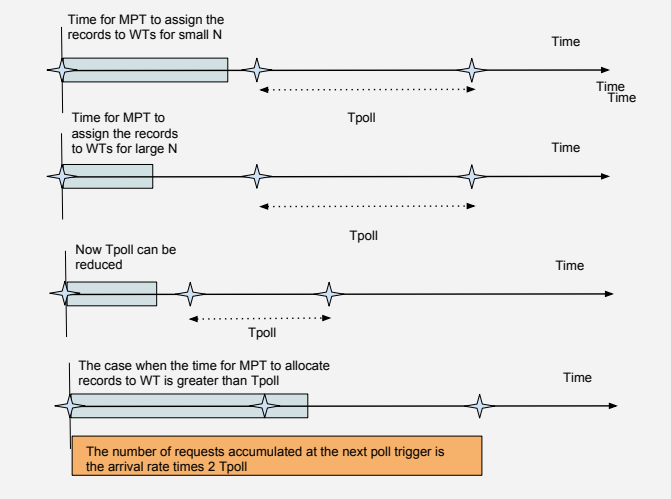

Fig. 6: Timing diagram illustrating the impact of setting T_{poll}.

*c) **Setting** N::* If the load is low then N should be set to $N_{min} = 1$. When the load increases, N should increase upto a maximum of N_{max}. The key design issues are 1) how should N be increased (decreased) as the load changes and 2) what is the maximum value of N_{max}. The overall algorithm is shown Figure 7. We use an algorithm that is similar to the additive increase multiplicative decrease (AIMD) algorithm used in TCP congestion control. It has been shown that these type of distributed control algorithms have good stability and

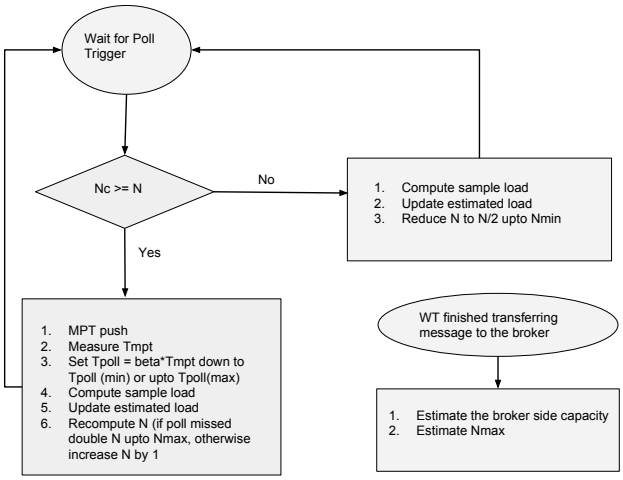

Fig. 7: The control algorithm for auto-tuning N and T_{poll}.

fairness properties [11]. When T_{poll} expires, we check if N messages are pending to be published. If enough messages have not accumulated we decrease N multiplicatively (by a factor of 2) and restart the poll timer. This ensures that the MPT quickly adapts with rapid changes in the workload (in this case rapid decrease in the message arrival rate). If on the other hand, enough messages do accumulate, then MPT starts a publish event by reading the messages from the data store and assigning them to WTs. If the time to perform this publish event finishes before the next poll timer expires then the value of N is increased by 1. This is to probe if there is available broker-side capacity so that additional WTs can be allocated to publish messages. If on the other hand, the time required by MPT to assign the messages to WT takes longer than the current poll timer, then N is decreased by 1. This reflects that the aggregate capacity demand due to number of assigned WTs is more than the broker-side capacity.

The value of N_{max} will limit how much N can increase and should depend on the broker-side capacity. We estimate the broker-side capacity as follows. When a WT completes transferring a message to the broker, we record the time it took. We denote this by s_i. We also keep track of the number of WTs that are currently transferring messages to the broker. This is denoted by n_i. Given these two values, we estimate the current available message transferring capacity (in messages/sec) to the broker by $\eta(i) = n_i/s_i$. Using this as the current sample of the capacity, we estimate the expected capacity by an exponential weighted moving average filter. Specifically, the estimated capacity $\eta_{est}(i+1)$ is given by

$$\eta_{est}(i+1) = \gamma\eta_{est}(i) + (1-\gamma)\eta(i) \quad (2)$$

where $0 < \gamma < 1$. We set N_{max} to $\eta_{est}(i+1)$. As the service time s_i increases, the message throughput will decrease which would decrease the estimated η_{est} which in turn would decrease N_{max}. Note that s_i will increase whenever there is congestion in the link between the publisher and broker, or congestion in the broker. Conversely, when the service time s_i decreases it will result in a higher value of the sampled capacity, which will result in a higher estimated capacity, and hence higher value of N_{max}.

V. Simulation Analysis

The auto-tuning algorithm described in the previous section was evaluated using simulation analysis. In this section we first describe the simulation tool and then we present the results.

A. Simulation Tool

We implemented the system model described in Section V using SimPy[3]. SimPy provides a simple and yet powerful platform for discrete event simulation. The multi-threaded architecture of the publisher was accurately implemented in the simulator. The broker is implemented as a simple queue with occupancy threshold that trigger flow control between the broker and publisher. The subscriber operates in a pull mode and is implemented as a gated vacationing server. Specifically, it alternates between work mode and vacation mode and upon returning from vacation reads all the messages that are at the broker at that instant. Messages arriving during the work mode are outside the gate and hences read in the next cycle. The subscriber is characterized by two parameters - 1) time to read a single message, and 2) vacation time and both of these are drawn from a negative exponential distribution with different rate parameters.

The main approximation in the simulation pertains to the communication between the publisher and the broker. In the real system, message transfers between the publisher and the broker is performed using a single TCP connection that is shared among the WTs. The flow control between the broker and the publisher is implemented using the TCP flow control which is a byte-level flow control. Specifically, the TCP receiver sends the available buffer space (in bytes) with the acknowledgements (ACKs) to the TCP transmitter. In the simulator, however, the flow control is performed at the message level. The broker is single server queue with a finite buffer. When it receives a message from the publisher it sends back an ACK in which it indicates the remaining number of messages it can accept. Thus the flow control is implemented at the granularity of a message rather than the byte-level flow control implemented in TCP.

B. Results and Discussions

We consider three important scenarios for our simulation study. We first consider different input request loads (requests/sec) and compare the response times with auto-tuning on and off. Second, we study how the publisher reacts to rapid changes in the input workload. In both of the above two cases, we set the parameters such that there is no broker-side congestion. In the third scenario, we consider the case when there is broker-side congestion due to a slow subscriber. The request arrivals follow a Poisson process and the values of the key parameters are shown in Table I.

[3]https://simpy.readthedocs.io/en/latest/

1) Adapting to Input Load: Figure 8 show the end-to-end latency for the messages with auto-tunning off. Results are shown for different values of N with T_{poll} set to 50 ms. With auto-tuning on different N values correspond to initial value. The four plots correspond to four different load levels, namely (clockwise from top left) 1.75, 17.5, 35, and 55 requests/sec. With auto-tuning off, when the load is very low (1.75 requests/sec) the minimum response time is achieved when N is set to 1. As N is increased, the response time increases as more and more time is spent to accumulate the required number of requests. As the load is increased, small values of N has high response times as there are not enough WTs to serve the request load. As N is increased, the response time decreases. However, beyond a certain value which is load dependent (≈ 10 for the request load of 35 requests/sec) the response time increases as more time is spent in accumulating larger number of requests.

As shown in Figure 9 with auto-tuning on we see that the response time is the same irrespective of what is the initial value of N. As expected, there is a slight increase in the average response time as the load is increased. The additive increase and multiplicative decrease algorithms is able to search appropriate value of N that optimizes the end-to-end latency.

Figure 10 shows the changes in N for request load of 35 requests/sec. We observed that our proposed algorithm adapts to the value of N with the input load and the average value is close to the value that achieves the minimum response time as shown in Figure 8 for the request load of 35 requests/sec. While the average N shown by the dark line is the right value, there is some amount of variability. This is due to the fact that we have used a Poisson arrival process. The growth and decay of N is guided by the estimated load which uses exponentially weighted moving average to estimate the load. Since the Poisson arrivals are independent and hence there are no short- and long-term dependencies, estimated load can be inaccurate resulting in inaccurate setting of the value of N.

2) Adapting to Load Changes: We also investigated the impact of step change in the workload. Table II shows the input workload profile considered for this experiment. Figure 11

Start Time	End Time	Rate (Req/s)
0	100000	1.75
100000	300000	55.0
300000	500000	17.5
500000	750000	35.0
750000	1000000	17.5

TABLE II: The load profile. The step changes in the rate at times 100000, 500000, 7500000.

shows how N adapts as the load is changed in a stepwise manner. The red lines correspond to the request loads over the corresponding time intervals. From the figure we observe that with auto-tuning, the value of N quickly adapts to the changes in the request load and the average value correlates well with the value that optimizes the latency (.

3) Adapting to Broker-side Congestion: We simulated broker-side congestion by reducing the rate at which the subscriber pulls messages from the broker. As mentioned before, the subscriber is modeled by two parameters a) time between message pulls and b) the time to pull a message. The time to pull a message is drawn from an exponential distribution with rate 25 messages/sec. The time between pulls is also drawn from an exponential distribution with rate parameter changed to model a slow subscriber. When the subscriber takes longer time to pull the messages from the broker, its queue grows which causes it to flow control the publisher. The net effect is that it takes longer for a WT to complete the transfer of a message from the publisher to the broker. Without auto-tuning more and more of the WTs are busy transferring messages until there are not enough WTs to handle the incoming requests. This results in requests being dropped at the input to the publisher. With auto-tuning, MRT has non-preemptive priority in assigning WTs to incoming requests. Furthermore, the allocation of WTs by MPT is reduced when the publisher is flow controlled. This back-pressure in the allocation of WTs ensures that there are WTs that can handle the incoming requests and hence there are no losses. Figures depicting these results are omitted due to space considerations.

VI. Implementation Results

We next consider the impact of the auto-tuning algorithm on a real trace obtained from sLS implemented for perfSONAR. The request rate from the trace file is shown in Figure 3. While in the real implementation, T_{poll} is set at 1000 ms, in the following simulation analysis, we set it to 50 ms. Figure 12 shows the response times with and without auto-tuning for different configurations corresponding to different values of N. Since the load is low, low values of N gives the minimum response time without auto-tuning. As N is increased to 10 and 50, the response time increases as more time is required to accumulate the larger number of requests. With auto-tuning turned on we achieve the same performance irrespective of what value is set for N.

As the original load was low, we created a higher load trace by scaling down the packet inter-arrival times by an factor of 50. As result we created a workload that was 50 times the real workload but with the same statistical feature as the original trace. Figure 13 shows the response times of the messages with and without auto-tuning for different values of N. Since load is high, the response time is very high for $N = 1$ without auto-tuning. With auto-tuning we achieve a good performance irrespective of what initial value is set for N. Figure 14 shows the adaptation of N which correlates well with the changes in the load.

VII. Related Work

There is a rich body of literature on pub/sub systems. The overall design objectives of content based pub/sub system, channel-based pub/sub systems, and event notification systems both for general as well as P2P systems are similar in nature [2], [3], [16], [20], [23]. These systems use intermediate

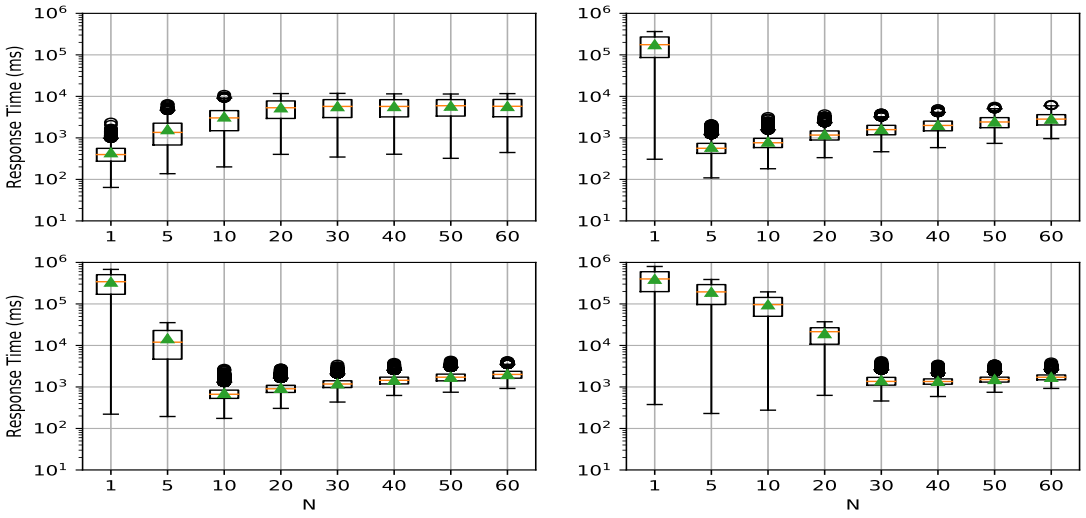

Fig. 8: End-to-end latency without auto-tuning for different values of N and different request loads. Load values going clockwise are 1.75, 17.5, 35, 55 requests/sec.

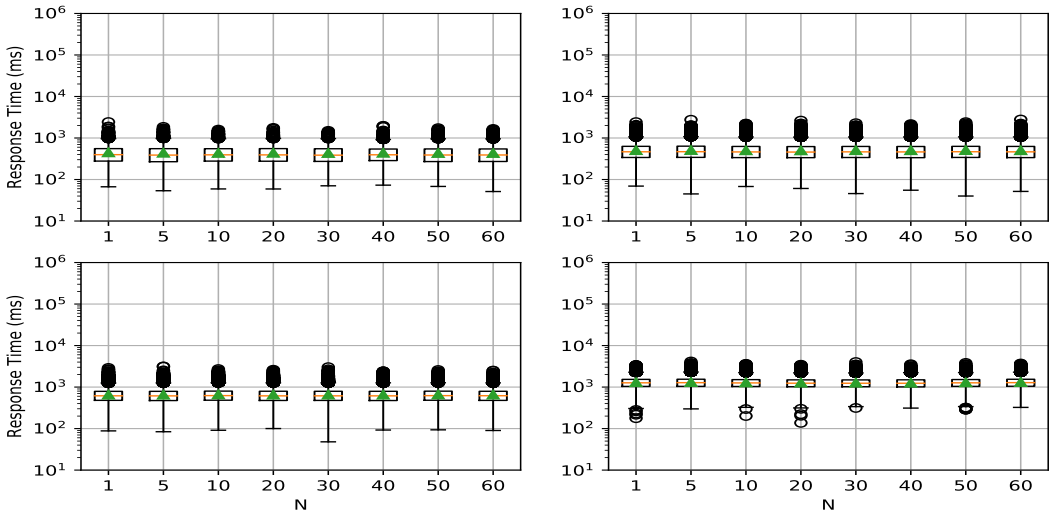

Fig. 9: End-to-end latency with auto-tuning for different values of N and different request loads. Load values going clockwise are 1.75, 17.5, 35, 55 requests/sec.

brokers that store and forward updates to subscribers. They can provide reliability by buffering updates for slow subscribers, while providing low latency to fast subscribers. While the basic framework of pub/sub systems have been studied over past couple of decades new interest in these systems have spurred by their use in some of the most well-know social networking applications.

Thialfi [1] is a notification service that Google uses to ensure the freshness of client data for applications that rely on cloud infrastructure to store and share data. SIENA [5] is also a similar scalable event notification system that allow distributed event based application to be deployed over the Internet. Kafka [12] is LinkedIns topic-based pub-sub system that is maintained by Apache. Finally, Wormhole [18] is a pub/sub system that Facebook uses to reliably replicate changes among several services that maintain data in geographically replicated datacenters. Host services for message buses such as IronMQ [14] and Amazon SQS [22] are hosted

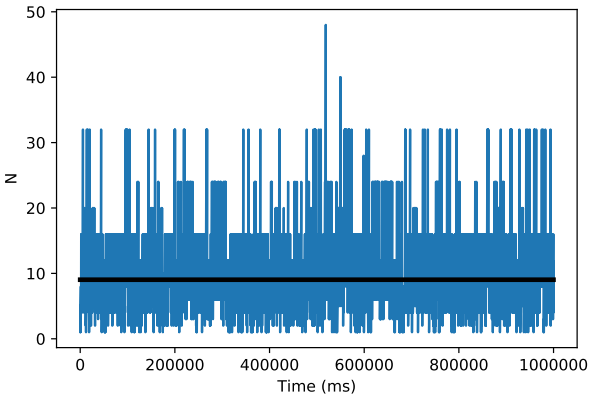

Fig. 10: The figures shows the adjustments of N with auto-tuning. The request load is 35 requests/sec. The dark line show the average value of N to be slightly less than 10 which matches with the value that minimizes the response time in Figure 8 with request rate 35 req/sec (lower left bottom).

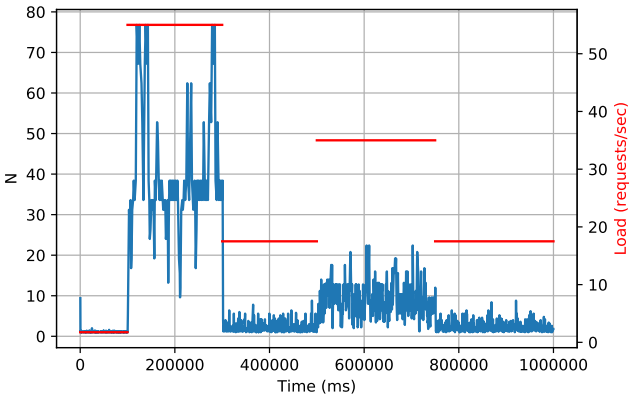

Fig. 11: The auto-tuned adjustments to N with step changes in the request load described in Table II. The red lines show the load in requests/sec over the corresponding time interval. The value of N is averaged over 5 samples.

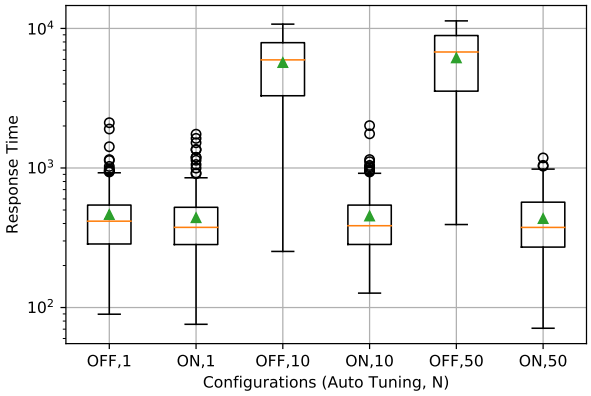

Fig. 12: Boxplots of the response times (in ms) with different configurations for the real traffic trace. The different configurations corresponding to Auto Tuning OFF and ON with different values of N.

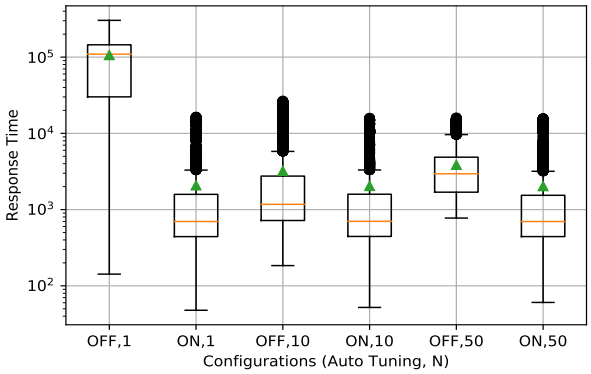

Fig. 13: Boxplots of the response times (in ms) with different configurations for the real traffic trace scaled by a factor of 50. The different configurations corresponding to Auto Tuning OFF and ON with different values of N.

and have scalability issues addressed in this paper. The most popular open-source message queues include Beanstalkd [19] and RabbitMQ [21].

Some recent studies have started to investigate the application of pub/sub systems for distributed and latency constrained application. Dynamoth [8], is a dynamic, scalable, channel-based pub/sub that proposes a software layer to load balance large number of publishers that publishes messages across multiple brokers which can be deployed as a cloud service. Multipub [9] is another study that attempts to scale dynamically depending on the current communication demands with the multiple-brokers deployed in the cloud. MultiPub proposes a flexible pub/sub system for latency-constrained, globally distributed applications. It dynamically reconfigures the communication layer to achieve latency guarantee for the messages. The study presents experimental results that demonstrate the achieved latency and cost savings compared to traditional approaches.

The need for flexibly provisioning resources for pub/sub systems deployed as cloud service has been addressed in [13]. The problem of the resource management in a elastic pub/sub system is formulated as an optimization problems using different objectives functions. It models the elastic pub/sub system as a multiple-class open queuing network which is solved to derive system performance measures. They then propose greedy algorithms to efficiently solve the optimization to determine the resource allocation. The evaluation based on simulation of real system shows that the proposed solution outperforms the baseline and is robust in dealing with high-volume and fast-changing workload.

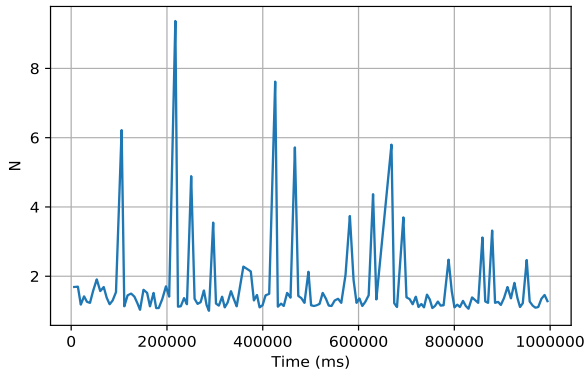

Fig. 14: Adaptation of N with Auto Tuning on with the real trace scaled by a factor of 50.

The above work is closest to the study in this paper and addresses the problem of fast-changing workload. Rather, than developing queueing models to predict the performance measure, we develop control algorithm for resource management based on system introspection. Specifically resource allocation is based real-time introspection of load and the broker-side bottleneck.

VIII. Conclusions and Future Work

In this paper we studied a pub/sub system in which the publisher, the broker, and the subscriber are in different administrative domains. We considered a a multi-threaded publisher which uses a control algorithm to auto-tune the batching parameters to optimize the end-to-end latency. Using a simulation model, we demonstrated the performance of the control algorithm. We also studied the performance of the algorithm using real trace from the Simple Lookup Service (sLS) that is used with perfSONAR. The results show significant improvement in latency compared to latency in existing deployment. In this study, we considered a single publisher. In a multi-domain network with multiple publishers we expect that our control algorithm will still be stable and fair. This is based on the fact that the control algorithm is similar to an additive-increase and multiplicative-decrease algorithm used in TCP congestion control which has reasonable fairness characteristics. As future work, we will implement our algorithm with multiple publishers and evaluate its performance.

References

[1] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. Thialfi: a client notification service for internet-scale applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 129–142. ACM, 2011.

[2] Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 437–446. IEEE, 2005.

[3] Roberto Baldoni and Antonino Virgillito. Distributed event routing in publishsubscribe communication systems: a survey. *DIS, Universita di Roma La Sapienza, Tech. Rep*, 5, 2005.

[4] E. C. Bellm, S. R. Kulkarni, and ZTF Collaboration. The Zwicky Transient Facility. In *American Astronomical Society Meeting Abstracts*, volume 225 of *American Astronomical Society Meeting Abstracts*, page 328.04, January 2015. text available at https://arxiv.org/abs/1410.8185.

[5] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227. ACM, 2000.

[6] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[7] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[8] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 486–496, June 2015.

[9] J. Gascon-Samson, J. Kienzle, and B. Kemme. Multipub: Latency and cost-aware global-scale cloud publish/subscribe. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2075–2082, June 2017.

[10] Andreas Hanemann, Jeff W Boote, Eric L Boyd, Jérôme Durand, Loukik Kudarimoti, Roman Łapacz, D Martin Swany, Szymon Trocha, and Jason Zurawski. Perfsonar: A service oriented architecture for multi-domain network monitoring. In *International Conference on Service-Oriented Computing*, pages 241–254. Springer, 2005.

[11] Go Hasegawa, Masayuki Murata, and Hideo Miyahara. Fairness and stability of congestion control mechanisms of tcp. *Telecommunication Systems*, 15(1-2):167–184, 2000.

[12] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[13] P. Nguyen and K. Nahrstedt. Resource management for elastic publish subscribe systems: A performance modeling-based approach. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 561–568, June 2016.

[14] Dharmit Patel, Faraj Khasib, Iman Sadooghi, and Ioan Raicu. Towards in-order and exactly-once delivery using hierarchical distributed message queues. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 883–892. IEEE, 2014.

[15] P. R. Pietzuch and J. M. Bacon. Hermes: a distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618, 2002.

[16] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *NSDI*, volume 6, pages 2–2, 2006.

[17] Khaled Salah, K El-Badawi, and F Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Computer Communications*, 30(17):3425–3441, 2007.

[18] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, et al. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *NSDI, May*, 2015.

[19] Luis Stevens, Curtis Andrus, and Vince Schiavone. Optimization for real-time, parallel execution of models for extracting high-value information from data streams, March 21 2017. US Patent 9,600,550.

[20] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. *arXiv preprint cs/9810019*, 1998.

[21] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[22] Hobin Yoon, Ada Gavrilovska, Karsten Schwan, and Jim Donahue. Interactive use of cloud services: Amazon sqs and s3. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 523–530. IEEE Computer Society, 2012.

[23] Shelley Q Zhuang, Ben Y Zhao, Anthony D Joseph, Randy H Katz, and John D Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20. ACM, 2001.