Elastic Scaling of a High-Throughput Content-Based Publish/Subscribe Engine

Raphaël Barazzutti[*], Thomas Heinze[†], André Martin[‡], Emanuel Onica[*], Pascal Felber[*],
Christof Fetzer[‡], Zbigniew Jerzak[†], Marcelo Pasin[*] and Etienne Rivière[*]
[*]*Université de Neuchâtel, Switzerland;* [*]*SAP AG, Dresden, Germany;* [‡]*Technische Universität Dresden, Germany*

***Abstract*—Publish/subscribe (pub/sub) infrastructures running as a service on cloud environments offer simplicity and flexibility for composing distributed applications. Provisioning them appropriately is however challenging. The amount of stored subscriptions and incoming publications varies over time, and the computational cost depends on the nature of the applications and in particular on the filtering operation they require (e.g., content-based vs. topic-based, encrypted vs. non-encrypted filtering). The ability to elastically adapt the amount of resources required to sustain given throughput and delay requirements is key to achieving cost-effectiveness for a pub/sub service running in a cloud environment. In this paper, we present the design and evaluation of an *elastic* content-based pub/sub system: E-STREAMHUB. Specific contributions of this paper include: (1) a mechanism for dynamic scaling, both out and in, of stateful and stateless pub/sub operators, (2) a local and global elasticity policy enforcer maintaining high system utilization and stable end-to-end latencies, and (3) an evaluation using real-world tick workload from the Frankfurt Stock Exchange and encrypted content-based filtering.**

I. INTRODUCTION

Cloud computing provides processing infrastructures, platforms, or software solutions *as a service*. One of the key promises of cloud computing is its support for *elasticity* [5], i.e., the ability to dynamically adapt the amount of resources allocated for supporting a service. Thanks to elasticity, cloud users no longer need to statically provision their infrastructure. Instead, resources can be added (scale out) or removed (scale in) on-demand in response to variations in the experienced load [27]. This allows a service to accommodate unpredictable growth or decay in popularity.

To support elasticity, an application or service deployed on a cloud must provide three key features. First, it should be intrinsically *scalable*: the addition or removal of resources must result in an increase, respectively decrease, of the processing capacity of the service. Second, it should support the *dynamic allocation of resources*. This allows modifying at runtime the processing capacity of a service without halting or restarting that service. Finally, it should provide *decision support*, in order to drive the dynamic allocation based on the experienced workload and according to *elasticity rules*. These rules trigger the allocation of new resources and the deallocation and reallocation of existing resources based on the observed load.

Publish/subscribe (pub/sub) [13] is a decoupled communication paradigm that is particularly well suited for cloud-based deployments [1]. In pub/sub systems, information producers, or *publishers*, send data to the pub/sub service in the form of *publications*. The pub/sub service is then in charge of dispatching these publications to all the parties, called *subscribers*, that previously expressed interest in them by the means of *subscriptions*. In the *topic-based* model [1], [2], publications and subscriptions are associated to predefined topics. *Content-based* pub/sub supports subscriptions that express interest as filters on the content of publications, typically by means of predicates over the attributes associated to each publication [4], [9], [10]. We consider the more general content-based model in this paper.

The load experienced by a pub/sub system running as a service is by nature difficult to predict. The amount of subscribers and the volume of publications can vary significantly over time. A typical example is stock market monitoring [6]: stock exchanges (publishers) send real-time stock ticks or quotes, while clients (subscribers) register their interests in ticks and/or quotes related to their investment strategies, e.g., when the price for a given stock goes over or under a specific threshold. In such a setting, the volume of publications depends on the activity of the stock exchanges, which have specific periods of activity every work day and are closed on week-ends. Figure 1 shows a typical tick load recorded on the 18th of November 2011 at the Frankfurt Stock Exchange. The volume sharply rises when the trading opens at 9:00 and rapidly declines after market closes at 17:30. Static provisioning of cloud resources for a pub/sub system supporting the peak load of this application would be cost-ineffective. Conversely, the ability to elastically scale the amount of resources allows a better utilization of the

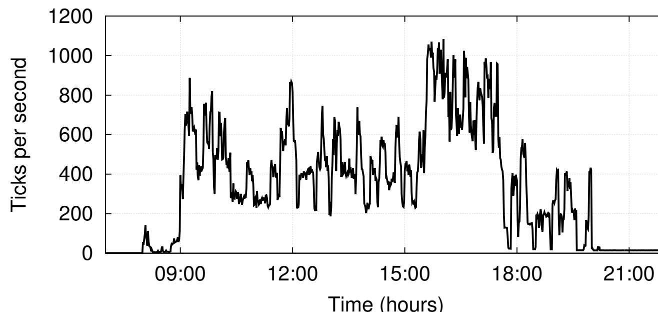

Figure 1. Typical volume of ticks (Frankfurt Stock Exchange, 2011/11/18).

infrastructure, translating into better return-on-investment for the service provider.

Contributions. In this paper we present the design, implementation, and evaluation of an elastic pub/sub middleware service supporting high-throughput and low-delay content-based filtering. Our system, named E-STREAMHUB, extends a scalable but static pub/sub engine, STREAMHUB [6], with comprehensive elasticity support. It supports arbitrary types of filtering operations, including encrypted filtering for privacy preservation in public cloud settings. It leverages a stream processing engine, STREAMMINE3G [8], and relies on a set of stream processing *operators*, each implementing a specific aspect of the pub/sub service, allowing the system to scale *horizontally* and *vertically* with the number of allocated nodes.

In this work, we make the following contributions: (1) support for slicing of the state and computations of a high-performance and massively parallel pub/sub engine; (2) migration mechanisms for dynamically allocating resources to each operator of a pub/sub engine with low impact on delays and throughput; (3) decision support for elasticity and resource allocation based on system observation and enforcement of global and local elasticity policies; and (4) implementation of these mechanisms in E-STREAMHUB and their evaluation on a private cloud of 240 cores, using real load evolution traces from the Frankfurt Stock Exchange. Our evaluation allows adapting the number of machines used to the actual workload experienced by the pub/sub service, while maintaining continuous operation and with minimal impact on notification delays.

II. RELATED WORK

In this section we present an overview of related work with specific focus on elastic platforms, event processing systems, and pub/sub middleware.

A. Elastic Platforms and Infrastructures

Elastic infrastructures and platforms as a service (IaaS and PaaS) support ad hoc addition and removal of virtual machines (VMs) to/from a virtual environment (VE). A typical example is the Amazon EC2 Auto Scaling service [3]. It relies on basic elasticity policies by setting simple thresholds on resource utilization. The *elastic site* mechanism for Nimbus and EC2 clouds proposes more elaborate policies, for workloads composed of independent tasks under batch scheduling [23]. AutoScale [16] is an example of an adaptive IaaS elasticity policy, where the goal is to maintain the minimum amount of spare capacity to handle sudden load increases while keeping the total number of servers as low as possible.

All elastic scaling solutions at the IaaS level require that the VMs composing the VE, as well as the jobs that run on these VMs, are stateless or independent. These requirements are not fulfilled by the nodes of a pub/sub system, which require application-level elasticity mechanisms. Elastic applications interact with elastic IaaS using the VM allocation and deallocation APIs of the IaaS elasticity manager.

B. Elastic Complex Event Processing

Both complex event processing systems (CEP) and stream processing engines (SPE) process queries composed of directed acyclic graphs of stateful or stateless *operators* processing data in the form of *event streams*.

Fernandez et al. [15] propose application-level scale out for stateful operators, integrated with passive replication (checkpointing). The integration is made possible by explicit operator state management, an approach similar to that of E-STREAMHUB. The operator state is dynamically split to form new partitions deployed over multiple machines when scaling out, potentially requiring application support for partitioning. E-STREAMHUB supports full elasticity, i.e., scale *out* and *in*, while optimizing the overall system utilization, and without requiring specific application support. Its underlying runtime engine can support both passive [26] and active [25] replication.

In the context of IBM's System S [31], Schneider et al. [28] propose an extension of the SPADE domain-specific language to support parallel operators on a single host. E-STREAMHUB supports elastic parallelization within a single host as well as elastic distribution across multiple hosts.

ElasticStream [20] proposes to outsource parts of a System S SPE deployment to a public cloud. It periodically adjusts the deployment based on decision from a linear programming formulation of the allocation problem, where the goal is to minimize the monetary cost of the cloud usage. E-STREAMHUB elastically scales based on the *current* state of the system, immediately addressing load variations.

StreamCloud [17] monitors the average load per cluster in order to detect under- and overload situations. When an overload is detected, StreamCloud triggers pair-wise rebalancing between most and least loaded hosts. STREAMHUB elasticity policies share this goal of rebalancing the load but with the additional goals of minimizing the amount of migrations and the support of migrations with minimal interruption of the flow.

C. Elastic Publish/Subscribe

Topic-based pub/sub engines used in the industry, such as Apache Kafka [21] or Hadoop HedWig [2], typically support *incremental scalability*. This refers to the ability to add/remove support servers in order to achieve linear scalability in throughput or number of topics. None of these systems supports automated addition and removal of servers based on the experienced workload, and therefore cannot be classified as elastic. E-STREAMHUB supports *incremental scalability* and *elasticity*, and considers the more general and challenging content-based model.

EQS [30] is a message queue architecture that can be classified as topic-based pub/sub. Elasticity in EQS is achieved by monitoring the load on each topic and migrating topics between hosts. This imposes a potential cost in terms of service interruption. In contrast to EQS, E-STREAMHUB targets content-based filtering where the destination of events can only be determined at runtime. It also migrates parts of the load to new hosts when overload is detected, but with the objective of minimizing service interruption thanks to an application-level migration mechanism.

Hoffert et al. [18] consider the problem of predicting the load requirements for an adaptive topic-based pub/sub service running in a cloud environment. Using supervised machine learning, the approach is able to dynamically configure the service (e.g., the transport protocols used) for different QoS requirement, but does not consider the dynamic addition or removal of hosts for supporting the service.

BlueDove [22] targets attribute-based pub/sub deployments on public cloud services. The message flow in BlueDove is strictly dependent on the attribute-based filtering model used for subscriptions: the attribute space is split in regions, and subscriptions and publications are dispatched to the matching servers that are in charge of an overlapping region. This disallows the use of filtering schemes that are not based on low-dimensionality attribute spaces or that do not allow examining the content of the subscriptions at the server side, such as with encrypted filtering. Finally, BlueDove supports only scale out, therefore, unlike E-STREAMHUB, it cannot be described as fully elastic. Fang et al. [14] follow a similar approach to BlueDove and only scales out.

III. THE STREAMHUB SCALABLE PUB/SUB ENGINE

We present in this section the high-level architecture of the pub/sub engine that we extend for elasticity support, STREAMHUB [6]. Its components are shown in Figure 2. The design of STREAMHUB targets high throughput filtering and notification pub/sub. It also aims for low and stable latencies. For any given publication, the delay for notifying users shall not vary more than a small fraction of its average value.

The matching of incoming publications against stored subscriptions is performed in STREAMHUB by means of external filtering libraries. Deploying a content-based pub/sub system in a shared (public) cloud infrastructure poses security challenges. The values of the publications' attributes and the subscriptions' predicates may reveal confidential information that must be protected from potential attackers [29]. Encrypted filtering techniques [7], [11] allow matching encrypted publications against encrypted subscriptions, without revealing their original content. In the context of this paper and in our evaluation, we focus on computationally intensive encrypted filtering using the ASPE algorithm [11]. Due to the independence of the architecture from the filtering scheme, our results would apply to any other scheme.

Most of the previous distributed content-based pub/sub approaches were based on an overlay of brokers, where each broker would implement all operations of the pub/sub service [9], and brokers would have to maintain routing tables between them whose efficiency and construction cost directly depend on the subscription model and workload. STREAMHUB departs from the use of broker overlays and adopts a tiered architecture with dedicated components for each operation. This yields better performance and allows the system to scale independently of the filtering scheme(s) being used. STREAMHUB splits the pub/sub service into three fundamental operations. Each of these operations is mapped to an *operator* spanning over an arbitrary number of processing entities, called *operator slices*.

The first operation is *subscription partitioning*. It is supported by an *Access Point* (AP) operator and splits the workload of subscriptions in non-overlapping sets, one for each slice of the next operator, using modulo hashing on subscription identifiers. This introduces data parallelism and allows processing of several incoming publications in parallel against all subscriptions. Incoming publications are broadcast to reach all slices of the next operator.

The second operation is *publication filtering*. Each slice of the corresponding *Matching* (M) operator is associated with an instance of the filtering library that stores incoming subscriptions in the state associated with the slice. Upon receiving a publication, the library returns the list of subscribers that have registered a matching subscription. The publication and this list are forwarded to the next operator. Note that there might be several M operators, one per filtering scheme supported by the platform (e.g., encrypted and non-encrypted).

Finally, the third operation is *publication dispatching*. It is supported by an *Exit Point* (EP) operator whose goal is to collect and combine the lists of matching subscribers sent by each of the M operator slices, for each publication. These lists are dispatched to EPs using modulo hashing on publication identifiers. As a result, the lists for a particular publication will all reach the same EP operator slice. Once all lists are collected by that slice, a notification message is prepared and sent to interested subscribers through the connection point(s).

STREAMHUB leverages the runtime support of a stream-based processing engine, STREAMMINE3G [8]. *Events* flow through a directed acyclic graph (DAG) of operators. All slices of an operator use the same processing code for handling incoming events. Synchronized access to operator state can take place *within a given slice* by using read (R) and read/write (R/W) locks. A slice has no access to the state of other slices, even within the same operator. Each physical node (*host*) supports vertical scalability for locally-deployed operator slices by using a thread pool whose size is adapted to the number of available cores. Each slice can be supported by multiple cores operating

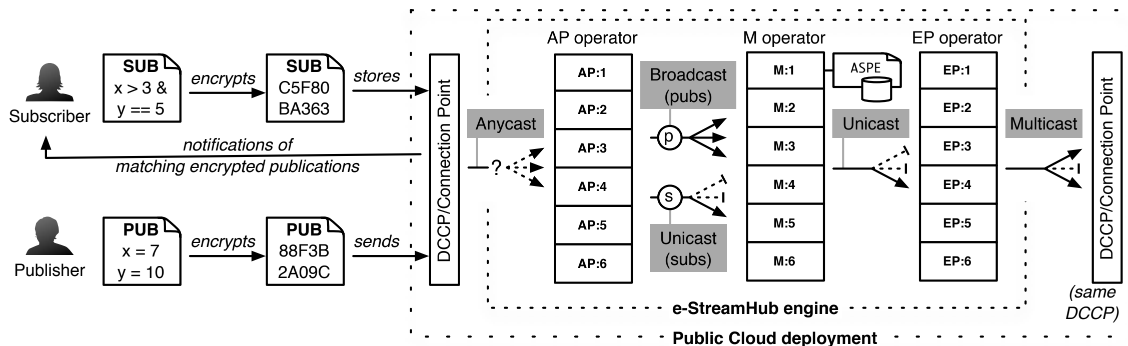

Figure 2. Example of a STREAMHUB engine deployed on a public cloud. Each operator is supported by 6 slices. Events flow from left to right.

in parallel when the processing is stateless or requires only an R lock. STREAMMINE3G supports dependability through passive [26] or active [25] slice replication. This requires no modification to the application but the evaluation of these mechanisms is out of the scope of the present paper.

The original STREAMHUB design assumes a static configuration: each operator slice is allocated to a single host and the number of slices for each operator must be determined manually based on the expected load for each of the pub/sub operations. Such manual provisioning is a challenging task as the workload is not necessarily known in advance. Even if the expected throughput could be estimated in certain cases, the intrinsic characteristics of its impact on the load each operator will have to support are hard to determine. Over-provisioning the operators is not desirable as the number of stored subscriptions and the rate of incoming publications will vary over time, thus leading to poor cost-effectiveness.

While scalability is a primary and necessary aspect of such a middleware solution, it is not sufficient *per se*. One also needs the ability to dynamically adapt the number of hosts allocated to each of the operators, based on the observed load on their slices. We present in the remaining of this paper the mechanisms (Section IV) and policies (Section V) required to provide elasticity within the pub/sub engine. These solutions provide minimal service interruption: they maintain a high filtering throughput and low delays even under dynamic reconfigurations.

IV. ELASTICITY MECHANISMS

Elasticity in E-STREAMHUB is supported by migrating operator slices across a varying number of hosts. The total number of active slices used by a single operator across all these hosts is fixed: we thus perform static partitioning of the operator state. A static partitioning is preferred as it does not require the application to be aware of scaling operations, avoiding specific up-calls for state management. Furthermore, using static partitioning relieves from using a restrictive storage model to allow state splitting and coalescing [15].

In the remainder of this section we describe the mechanisms for operator slice migration in STREAMMINE3G and the E-STREAMHUB manager component that orchestrates elasticity and system configuration.

A. Slice Migration with Low Impact on Delays

Migration of slices is supported at the level of STREAMMINE3G. The requirement is that the interruption of service must be as short as possible. We minimize the delay introduced in the processing of events during migrations by using slice duplication and in-memory logging/buffering of events.

The migration mechanism is illustrated by Figure 3. The slice to migrate initially runs on a host 1 (Figure 3-❶). The E-STREAMHUB manager (described later in this section) orchestrates the migration of that slice to host 2 when requested by the E-STREAMHUB elasticity enforcer (described in the next section). The STREAMMINE3G runtime creates a new slice on host 2, which is initially inactive. The operator DAG is rewired in order to duplicate all incoming events for that slice (❷). Events still reach the original slice on host 1 where they are processed normally, but they also reach the new slice on host 2 where they are queued for later processing. There is one queue per originating slice of the previous operator. The copy of the state takes place when all queues contain events with sequence numbers that are lower than or equal to those of events already processed on host 1 for the same source. Before copying the state, processing is stopped on host 1 (❸). The state is associated with a timestamp vector. Processing resumes on host 2 using events following the migrated state's timestamp vector, filtering obsolete events and preventing duplicate processing (❹), and the original slice on host 1 is removed (❻).

The slice migration time, and hence the duration of the service interruption, depends on the state size. AP operator

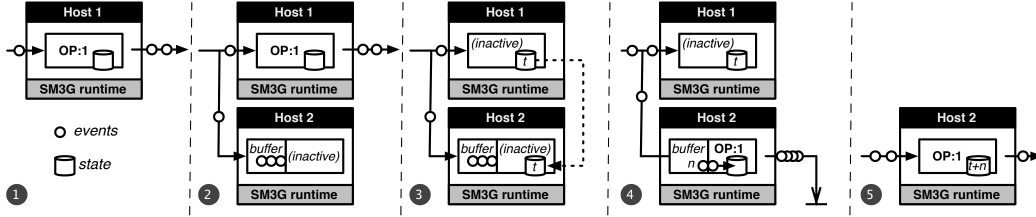

Figure 3. The five main steps of a slice migration between two hosts.

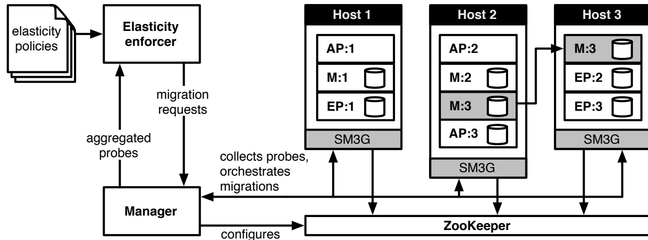

Figure 4. Interaction of the E-STREAMHUB components. Each host runs an instance of the STREAMMINE3G runtime (SM3G) and several operator slices. The manager collects probes about the utilization of the hosts for each individual slice and maintains the configuration in ZooKeeper. Probes are dispatched to the elasticity enforcer, which takes as input a set of elasticity policies and issue migration requests to the manager.

slices are stateless and there is no copy phase, therefore limited latency is expected during a slice migration. M operator slices have a persistent state consisting of the stored subscriptions. The migration delay is expected to mostly depend on this state size. EP operator slices also maintain a state, but it is transient and expected to be small: it consists of the lists of matching subscriber identifiers for publications being processed. Therefore, migrating EP operator slices is expected to have a small impact on the notification delay.

B. The E-STREAMHUB *Manager*

The interaction of the components of the system is illustrated in Figure 4. The manager is in charge of the system configuration. It orchestrates the migrations according to the flow of operations described above, and updates the configuration accordingly. Migrations are not initiated by the manager itself but requested by the elasticity enforcer component, described in the next section. The manager collects probes from all participating hosts via heartbeats. Probes indicate for each slice the CPU utilization, memory utilization, and network usage. They are then aggregated on a per-slice and per-host basis, and sent to the elasticity enforcer in order to trigger changes to the slice placement according to the input elasticity policies.

The operation of E-STREAMHUB requires that all hosts supporting the system share a common configuration. At the application level, the static configuration allows, for instance, EP operator slices to know the amount of M operator slices from which they must await matching subscriber lists. At the runtime level, the configuration is dynamic and includes the location of the slices on the hosts, which is updated upon migrations. The orchestration of the migration and the update of the configuration must be reliable to tolerate, in particular, a failure of the master. To this end, the shared configuration and migration orchestration leverage an instance of the ZooKeeper [19] coordination kernel. ZooKeeper maintains a simple filesystem-like hierarchy of shared objects used to reliably store the configuration. At the core of ZooKeeper, a reliable atomic broadcast with ordering guarantees on multiple support servers can tolerate failures and maintain configuration availability. The whole state of the manager is stored in ZooKeeper. This allows to easily restart the manager in case of failure.

V. ELASTICITY ENFORCER AND POLICY

The elasticity enforcer decides on the placement of operator slices on hosts by requesting migrations to the manager (e.g., migration of M:3 from host 2 to host 3 in Figure 4). It bases its decisions on input *elasticity policies* and on probes from the manager. The goal of the enforcer is to match policies requirements while minimizing the number and cost of slices migrations and thus the impact on service degradation.

The E-STREAMHUB elasticity policy combines *global* and *local* rules. Rules are evaluated as soon as a set of probes for all slices has been received since their last evaluation. Violations of global rules cause the system to scale in or out, by adding or removing hosts. Scaling out implies moving slices from overloaded hosts to newly added ones. Scaling in implies re-dispatching slices from the least loaded hosts to other hosts, and releasing the former. Local rules violations result in a re-allocation of slices among existing hosts. Global rules have the highest priority: local rules are only evaluated when no global rule is violated.

In this work, we us as primary metric for the elasticity policy the CPU utilization. Network bandwidth and memory constraint are only used as constraint during the migration decisions.

The global rule used in our evaluation states that the average CPU load for existing hosts must remain in the

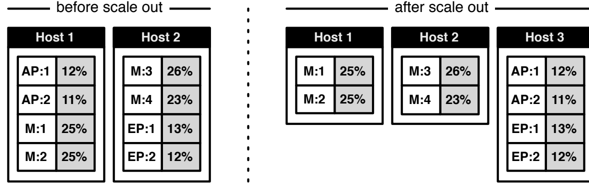

Figure 5. Example of a slice placement decision.

[30%:70%] range. E-STREAMHUB scales out if the average load exceeds 70% for at least 30 seconds and scales in if it drops below 30% for the same duration. The local elasticity policy states that the individual CPU utilization for a given host shall remain in the [20%:80%] range, also measured over a 30 seconds period. Finally, the policy sets a target (ideal) average CPU utilization of 50% and specifies a grace period of at least 30 seconds between migration requests in order to reduce the probability of thrashing.

The elasticity enforcer executes a two-step resolution algorithm when a global or local rule is violated. First, it decides on the set of slices to migrate. Second, it decides on a new placement (destination hosts) for these slices.

We start by describing the case of a scale out operation, upon a violation of the 70% average CPU utilization threshold of the global policy. The slice selection step consists in identifying a set of slices, whose summed CPU utilization is larger than, or equal to, the absolute difference between the current utilization and the average utilization target (50% in our case), starting from overloaded hosts. This corresponds to the *subset sum* problem [24], which is solved using dynamic programming with pseudo-polynomial complexity. This returns a set of valid solutions, among which we select the one incurring the minimal amount of state transfer (as reported by memory utilization probes). This allows us to minimize the cost and duration of migrations and to reduce service degradation.

The second step is the placement of the slices selected for migration. The enforcer uses a *First Fit* bin packing approach [12]. Hosts are modeled as bins with capacities reflecting the available CPU resources. Each slice is modeled as an item with weight equal to the CPU usage of the slice. A slice can only be assigned to a host when its load is smaller than the remaining CPU capacity of the host. The bin packing approach starts from the current placement of slices without the slices selected for migration. It greedily assigns migrated slices to hosts, in decreasing order of CPU utilization. Furthermore, the enforcer automatically derives allocation decisions for new hosts if the spare system capacity is not sufficient to accommodate a migrating slice without violating a local rule. The cost of the second step is linear in the number of hosts and slices in the system.

Figure 5 illustrates slice placement. The system starts with an average load of $\frac{74+73}{2} = 73.5\%$, violating the global rule and requiring the system to scale out. In order to reach a target utilization that is less than or equal to 50% for all hosts, the first step selects slices AP:1 and AP:2 for host 1 and slices EP:1 and EP:2 for host 2, among possible sets, as these have the lowest memory usage. The second step identifies that a new host is required for a placement that does not break the 50% max utilization rule. All selected slices are migrated to that new host.

When a global rule violation requires scaling in, the first step is to calculate the required number of hosts as the difference between the current number of hosts and the minimal number of hosts required for an average load equal to or higher than 50%. In the second step, the enforcer marks the least loaded host for release. Slices from that host are reassigned onto already running hosts and the host is released when it becomes empty. This procedure is repeated until the required number of hosts has been released.

Local policy enforcement uses the same two steps, except that only slices from the underloaded or overloaded host are considered by the bin packing algorithm.

VI. EVALUATION

We present in this section the experimental evaluation of E-STREAMHUB. We first evaluate the baseline performance without elasticity support. Then, we measure the cost of migrations and their impact on notification delays. We finally observe the behavior of E-STREAMHUB in terms of hardware resource utilization and notification delays, under synthetic and trace-based load evolution patterns.

A. Experimental Setup

We deploy E-STREAMHUB in a private cloud of 30 hosts running Debian Linux 6.0.7 Squeeze (kernel 2.6.32-48). Each host features two quad-core Xeon E5405 2.0 GHz processors and 8 GB of RAM. Hosts are interconnected by a 1 Gbps switched network.

We use a fixed number of 8 slices for the AP and EP operators, and 16 slices for the M operator. The ZooKeeper service, the elasticity enforcer, and the E-STREAMHUB manager run on separate hosts. In addition to the AP, M and EP operators, we deploy on 4 dedicated hosts two convenience operators, the source and the sink, each with 4 slices. The source operator pushes subscriptions and publications to the system from pre-encrypted events stored on disk. The sink operator receives the notifications. We place the source and sink operator slices two-by-two onto the same nodes. We measure the notification delays between one source operator slice and the sink operator slices on the same host, to avoid the effect of potential clock drifts on delay measurements. The number of notifications is large enough for the measured average and standard deviation to be statistically significant.

B. Workload

Our evaluation focuses on the support of elastic pub/sub as a service running on a untrusted public cloud. We therefore use the ASPE [11] encrypted filtering scheme, and workloads of pre-encrypted subscriptions and publications. While the performance of plain-text filtering may depend on the characteristics of the workload, such as the possibility to leverage containment between subscriptions [4], [7], [9], [10], [13], [14], [22], encrypted filtering such as with ASPE requires filtering each incoming publication against all stored subscriptions. Each individual filtering operation cost is quadratic in number of attributes (O(d^2)). The static performance of E-STREAMHUB is thus impacted by two factors. First, the number of attributes d has an impact on the cost of each individual filtering operation at the M operator level. We use a ASPE schema with $d = 4$ attributes. Second, the matching rate is the probability that a publication will match each subscription. It impacts on the number of notifications an incoming publication generates, and therefore on the load at the EP operator level. We generate subscriptions with an average matching rate of 1%. Unless explicitly noted, we use a workload of 100,000 subscriptions. Each publication thus generates an average of 1,000 notifications. Since we use encrypted filtering, where each publication has to be matched against each one of the stored subscriptions, our experiments are independent of the nature of the workload as there is no support for containment in the ASPE scheme. We therefore do not need to use traces and rely on synthetic subscriptions and publications.

Our experiments always begin with a subscription storage phase where subscriptions are stored in the system prior to sending any publication. Then, the rate of publications is either the maximal one supported by the system (for the baseline evaluation in a non-elastic setting), based on synthetic descriptions of the rate evolution, or based on the tick trace from the Frankfurt stock exchange described in our introduction (Figure 1).

C. Baseline STREAMHUB Performance

We start by evaluating the baseline performance of E-STREAMHUB with 100 K subscriptions. We consider static configurations of 2 to 12 hosts dedicated to the E-STREAMHUB engine and supporting AP, M, and EP operator slices. We used twice as many hosts for the M operator than for each of the two others. With 8 hosts, we deploy AP slices on 2 hosts (4 processors, 16 cores), M slices on 4 hosts (32 cores), and EP slices on 2 hosts (16 cores). Slices from two different operators can be placed on the same host (e.g., with 2 hosts, one host runs all AP and EP operators slices). This placement is not guaranteed to be optimal in terms of CPU utilization of the hosts but allows us to simply illustrate the linear scaling behavior.

Figure 6.up presents the maximal throughput supported by each configuration, before events start accumulating at the input of the AP operator. The throughput is perfectly linear as expected [6]. 12 hosts support a flow of 422 publications per second, corresponding to 42.2 millions encrypted filtering operations and 422,000 notifications sent per second.

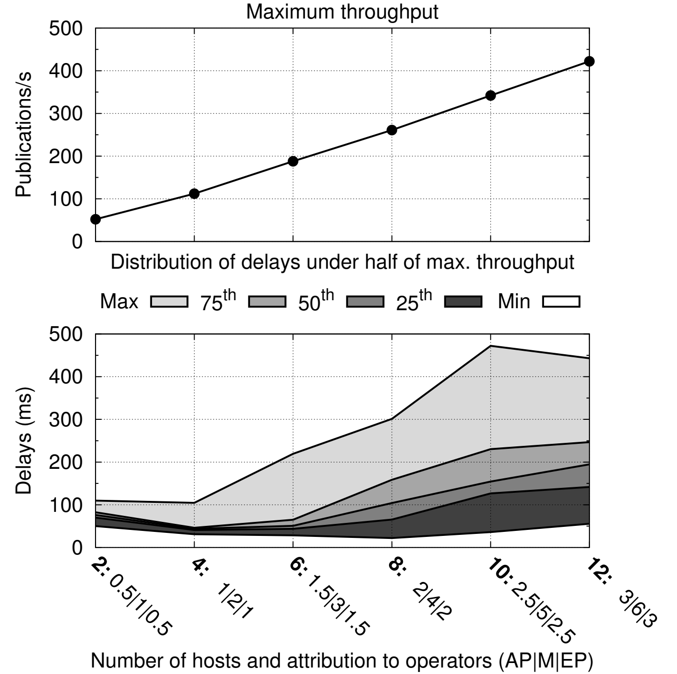

Figure 6. Performance of static E-STREAMHUB (100 K subscriptions).

	AP	**M** (12.5 K)	**M** (50 K)	**EP**
average	232 ms	1.497 s	2.533 s	275 ms
std. dev.	31 ms	354 ms	1.557 s	52 ms

Table I
MIGRATION TIMES, 100 PUBLICATIONS/S, 12.5 K OR 50 K SUBSCRIPTIONS STORED PER M OPERATOR SLICE, FOR A TOTAL OF 100 K AND 500 K SUBSCRIPTIONS, RESPECTIVELY.

We evaluate the delays by submitting to each configuration an incoming publication rate of half the maximal throughput, corresponding to the target load set in our elasticity policy. Figure 6.bottom presents the evolution of delays between the sending of each publication by the source operator and the reception of the last notification by the sink operator. For each configuration, stacked percentiles are represented as shades of grey. For instance, with 12 hosts, the minimal delay is 55 ms, while for 75% of the publications the notification is received by the last notified subscriber in less than 247 ms.

D. Operator Slice Migration Performance

We now proceed to the evaluation of the performance of slice migration. Table I presents the average and standard deviation of the migration time for the three operators, over 25 migrations. We consider larger slices: 4, 8, and 4 slices

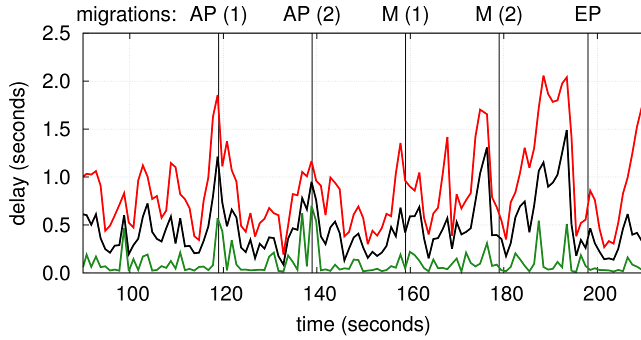

Figure 7. Impact of migrations on delay (min, average, max).

for the AP, M and EP operators, respectively deployed on 2, 4, and 2 hosts as the initial placement. Each migration picks a random slice of the corresponding operator and migrates it to another host. The system is under a constant flow of 100 publication/s, slightly less than half its maximal filtering throughput (Figure 6), and we consider the storage of 100 K subscriptions (12.5 K per M operator slice) and 500 K subscriptions (50 K per M operator slice). Migration times are very small for the AP operator, which is stateless, and for the EP operator, which has a very small state. The standard deviations are also small compared to the average. Migrations of M operator slices are more costly, as expected. The state size impacts on the migration time, but delays remain reasonable even for the very large number of subscriptions considered. We deliberately evaluated small deployments (with 4 M operators) and large numbers of subscriptions (up to 500 K), to show that even disproportionately large migrations take place within a few seconds.

Our next experiment evaluates the impact of migrations on the experienced delays. We use the same configuration as previously, with 100 K stored subscriptions. Figure 7 indicates the delay average, deviation, and min/max values, and points the time when we perform migrations, respectively, for two AP operator slices consecutively, for two M operator slices consecutively, and finally for one of the EP operator slices. We observe that the notification delay increases from the steady state of 500 ms to a maximal value of less than two seconds after migrations take place, while the average notification delay remains below a second for the most part.

E. Elastic Scaling under Varying Workloads

We now evaluate the complete E-STREAMHUB with elasticity support. We start with a synthetic benchmark. Results are presented by Figure 8. The first plot depicts the load evolution. We start with a single host that initially runs all 32 slices (AP and EP: 8 slices, M: 16 slices). The system is initially fed with 100 K encrypted subscriptions. We gradually increase the rate of encrypted publications, until we reach 350 publications per second. After a rate stability period, we decrease the rate gradually back to no activity.

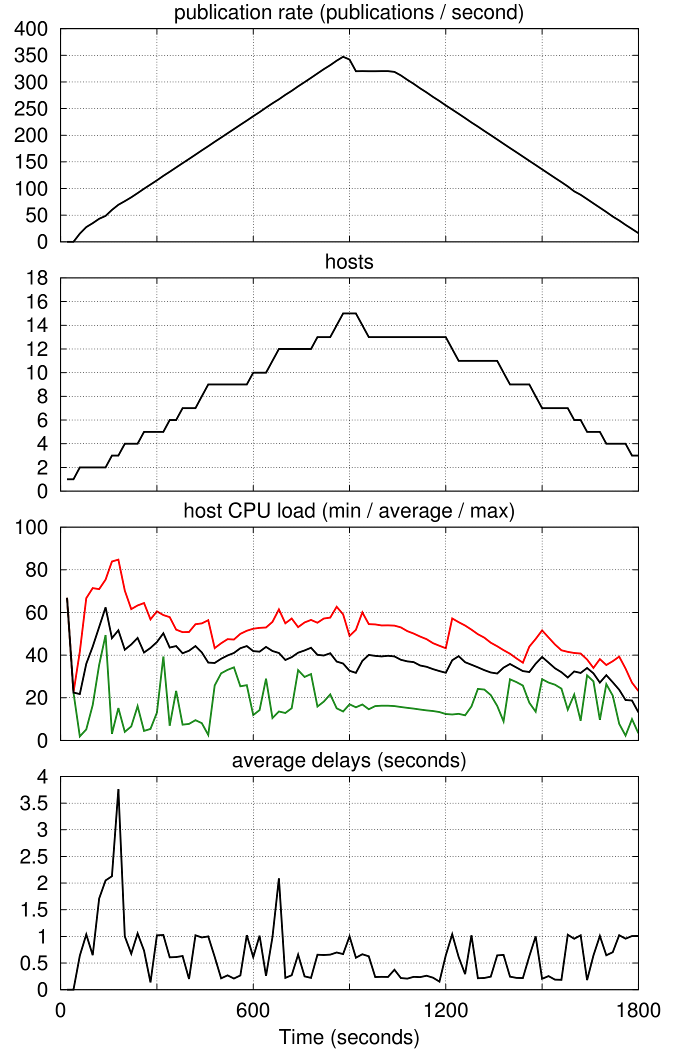

Figure 8. Elastic scaling under a steadily increasing and decreasing synthetic workload and 20 K encrypted subscriptions.

We observe the number of hosts used by the system, as dynamically provisioned by the enforcer, in the second plot. The minimum, average and maximum CPU load observed at these hosts are presented by the third plot. Finally the average delays and standard deviations are shown on the fourth plot. For all plots, we present averages, standard deviations, minimum, or maximum values observed over periods of 30 seconds.

We clearly observe that the system meets the expectations set by the elasticity policy. The number of hosts gradually increases towards 15 hosts. This corresponds to the range of static performance expected (note that Figure 6 presents the maximal achievable throughput with more subscriptions, hence the deviation). As the publication rate decreases, hosts are released and we return to the initial configuration. The elasticity policy enforcement and corresponding 246

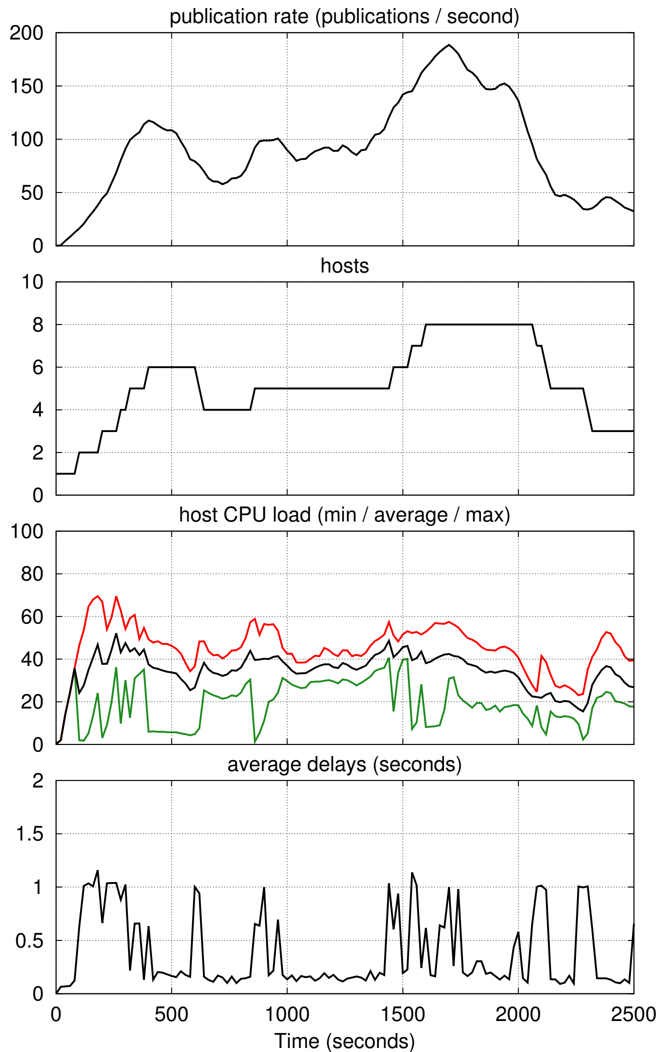

Figure 9. Elastic scaling under load from the Frankfurt stock exchange.

migrations during the experiment result in the load of the hosts remaining within an envelope of 40% to 70%, the average being close to the 50% target. Finally, we observe that the delays remain small despite migrations, which confirms the ability of the E-STREAMHUB approach to achieve elastic scale out and scale in with minimal degradation of the service performance. The initial migration from one to two hosts is the one that impacts most the delay. Indeed, the relative augmentation of load per host is more important in this case and roughly half of the operator slices have to be migrated to the new host, and the delays at the three operators may sum up if the three types are migrated as a same operation.

Our next and final experiment uses the same presentation as the previous, but instead of using a synthetic publication workload, we replay the trace-based publication activity from the Frankfurt stock exchange described in our introduction and in Figure 1. Results are presented with the same format as for Figure 8. The publication rate from the source operator is based on the stock exchange ticks, while the set of 100,000 subscriptions is fixed. We consider a shorter time frame for the experiment. We speed up the tick trace by 10 times: one hour in the original trace corresponds to 3 minutes in the experiment (for a total duration of 40 minutes). We also reduce the maximum amount of publications per second to account for the limited size of our experimentation cluster: the flow is scaled down from 1,200 on Figure 1 to 190 publications per second (for a peak load of 19 millions evaluations and 19,000 notifications sent per second, 1,800 seconds after the beginning of the experiment). We observe that the number of machines for supporting the workload ranges from 1 to 8. The system reacts to changes in load, in particular at the beginning of day and for the spike observed in the afternoon, while lowering to 3 nodes for the lower load in the evening. We observe as with the previous experiment that the load remains for all machines within the requested envelope. The effect of scale out operation (e.g., around 500 seconds) is to reduce the average and max load but may also be to let machines under-used: the subsequent scale-down experiment chooses the underloaded machine as a candidate for release, raising the minimal load observed. The average notification delay remain below one second for entire duration of the experiment despite migrations. This final trace-based experiment conveys the ability of E-STREAMHUB to quickly adapt to the experience load while keeping notification delays small and stable.

VII. CONCLUSION

We presented the design, implementation and evaluation of an elastic content-based pub/sub engine, E-STREAMHUB. Pub/sub middleware eases the composition and integration of multiple applications, services and users across different administrative domains. Pub/sub is particularly suited as a cloud-provided service. The load experienced by such a service heavily depends on the application, the filtering scheme(s) used, and fluctuations of the service popularity. The unpredictability of the load poses a challenge for the appropriate dimensioning of the support infrastructure and for its evolution over time. We addressed this problem by describing mechanisms and techniques for building an elastic pub/sub engine that automatically scales out and in based on observation of the load experienced by the system. Elasticity takes place for each of the three operators forming the engine independently, allowing to adapt to the nature of the workload. Elastic scaling takes place through migrations of operator slices with minimal service degradation. The elasticity enforcer matches elasticity policies while reducing the cost and number of necessary migrations. Our evaluation using synthetic and trace-driven benchmarks indicates that E-STREAMHUB is able to react to dynamic changes in load, automatically adding and removing hosts as required by the elasticity policy.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 257843 (SRT-15 project).

References

[1] http://aws.amazon.com/sns/.

[2] https://cwiki.apache.org/ZOOKEEPER/hedwig.html.

[3] http://aws.amazon.com/autoscaling/.

[4] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB*, 2000.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[6] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, M. Pasin, J.-F. Pineau, E. Rivière, and S. Weigert. StreamHub: A massively parallel architecture for high-performance content-based publish/subscribe. In *DEBS*, 2013.

[7] R. Barazzutti, P. Felber, H. Mercier, E. Onica, and E. Rivière. Thrifty privacy: Efficient support for privacy-preserving publish/subscribe. In *DEBS*, 2012.

[8] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker de Brum, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with StreamMapReduce. In *CloudCom*, 2011.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM TCS*, 2001.

[10] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11, 2002.

[11] Sunoh Choi, Gabriel Ghinita, and Elisa Bertino. A privacy-enhancing content-based publish/subscribe system using scalar product preserving transformations. In *DEXA*, 2010.

[12] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

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

[14] Wenjing Fang, Beihong Jin, Biao Zhang, Yuwei Yang, and Ziyuan Qin. Design and evaluation of a pub/sub service in the cloud. In *CSC*, 2011.

[15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[16] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, 2012.

[17] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, 23(12), 2012.

[18] Joe Hoffert, Douglas C. Schmidt, and Aniruddha Gokhale. Adapting distributed real-time and embedded pub/sub middleware for cloud computing environments. In *Middleware*, 2010.

[19] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[20] Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *IEEE CLOUD*, 2011.

[21] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *NetDB*, 2011.

[22] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *IPDPS*, 2011.

[23] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGRID*, 2010.

[24] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[25] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. In *SRDS*, 2011.

[26] A. Martin, T. Knauth, S. Creutz, D. Becker de Brum, S. Weigert, A. Brito, and C. Fetzer. Low-overhead fault tolerance for high-throughput data processing systems. In *ICDCS*, 2011.

[27] Peter Mell and Timothy Grance. The NIST definition of cloud computing. *NIST special publication*, 800(145):7, 2011.

[28] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS*, 2009.

[29] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *CCSW*, 2011.

[30] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimányi. Eqs: An elastic and scalable message queue for the cloud. In *CLOUDCOM*, 2011.

[31] Kun-Lung Wu, Kirsten W. Hildrum, Wei Fan, Philip S. Yu, Charu C. Aggarwal, David A. George, Buğra Gedik, Eric Bouillet, Xiaohui Gu, Gang Luo, and Haixun Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *VLDB*, 2007.