StreamHub: A Massively Parallel Architecture for High-Performance Content-Based Publish/Subscribe

Raphaël Barazzutti,[1] Pascal Felber,[1] Christof Fetzer,[2] Emanuel Onica,[1],
Jean-François Pineau,[1] Marcelo Pasin,[1] Etienne Rivière,[1] and Stefan Weigert[2]
1. University of Neuchâtel, Switzerland 2. TU Dresden, Germany
first.last@{unine.ch, tu-dresden.de}

ABSTRACT

By routing messages based on their content, publish/subscribe (pub/sub) systems remove the need to establish and maintain fixed communication channels. Pub/sub is a natural candidate for designing large-scale systems, composed of applications running in different domains and communicating via middleware solutions deployed on a public cloud. Such pub/sub systems must provide high throughput, filtering thousands of publications per second matched against hundreds of thousands of registered subscriptions with low and predictable delays, and must scale horizontally and vertically. As large-scale application composition may require complex publications and subscriptions representations, pub/sub system designs should not rely on the specific characteristics of a particular filtering scheme for implementing scalability.

In this paper, we depart from the use of *broker overlays*, where each server must support the whole range of operations of a pub/sub service, as well as overlay management and routing functionality. We propose instead a novel and pragmatic *tiered approach* to obtain high-throughput and scalable pub/sub for clusters and cloud deployments. We separate the three operations involved in pub/sub and leverage their natural potential for parallelization. Our design, named StreamHub, is oblivious to the semantics of subscriptions and publications. It can support any type and number of filtering operations implemented by independent libraries. Experiments on a cluster with up to 384 cores indicate that StreamHub is able to register 150 K subscriptions per second and filter next to 2 K publications against 100 K stored subscriptions, resulting in nearly 400 K notifications sent per second. Comparisons against a broker overlay solution shows an improvement of two orders of magnitude in throughput when using the same number of cores.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*

Keywords

Publish/subscribe; Scalability; Performance

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

1. INTRODUCTION

Content-based publish/subscribe (pub/sub) [20] is a strong contender for offering an efficient, yet *natural* communication paradigm to developers of large-scale applications. It supports decoupled interactions between the producers (*publishers*) and the consumers (*subscribers*) of information by the means of messages (*publications*). Decoupling occurs both in terms of space and time: publishers and subscribers do not need to know the existence or identity of one another, and no particular synchronization between them is necessary. They only communicate indirectly through a *pub/sub system.* It is the responsibility of this system to *route* publications from the publishers to interested subscribers. Routing is based on *subscriptions* registered by the subscribers to express their interest in specific content. The operation of *matching* the content of the publications against the subscriptions stored in the system is called *content filtering.*

A typical use of pub/sub systems is for composing a collection of independent applications running on different administrative domains or geographical locations. Communication between these applications takes place via a common pub/sub service running on a set of *dedicated servers*, typically set up in a *public cloud* or a *cluster equipped with a public address*, interconnected through a local area network and exposing access points to client applications.

The decoupled and data-centric nature of the pub/sub communication model allows for seamless integration and evolution of large-scale applications. A typical example is *QoS Monitoring as a Service* [36], where an application running on a private cloud is monitored and key performance indicators (KPIs) are generated as publications. These KPIs are propagated to a third-party monitoring service, based on subscriptions generated from a service level agreement (SLA) in order to detect violations of this SLA. Communication takes place via a pub/sub service deployed on a public cloud accessible by both parties. Other applications include e-Health systems [26] that bridge several medical and healthcare institutions sharing information about patients cases, or the canonical example of stock trading [24]. We note that for all these applications, the use of a third-party infrastructure for communication may raise concerns about privacy and data security: publications and subscriptions represent sensitive data that should not be leaked to a third party. As a result, *encrypted content filtering schemes* have gained interest in the recent years [6, 18, 25, 26, 34] as they support filtering of encrypted publications against encrypted subscriptions without needing decryption. Such approaches suffer, however, from a high computational cost and disallow

some optimizations, in particular those based on containment relationships between subscriptions (i.e., the fact that a subscription will match a subset of the publications matching another subscription) or on the aggregation of a set of subscriptions into a single one.

Objectives. We argue that the key properties of a pub/sub system running on a public cloud or cluster and supporting large-scale application composition should be as follows.

(1) High throughput and low, predictable delays. The raw performance of the pub/sub service deployed on a public cloud or cluster must be sufficient to support demanding applications, such as high-frequency trading or network monitoring. This requires exploiting parallel processing of incoming subscriptions and publications as much as possible. Since the filtering operation itself is costly, the design must avoid filtering an incoming publication against a given subscription multiple times, which typically happens in overlay brokers systems. Furthermore, delays between the generation of a publication and its dispatching to interested subscribers must remain of the same order as the delay a coupled communication between the producer and consumer of information would take. As a corollary, there should not be significant deviation in the notification time for all subscribers interested in a given publication.

(2) Scalability. The ability to support increasing numbers of publishers/publications, subscribers/subscriptions, and notifications, as well as more computationally intensive filtering schemes, requires several levels of scalability. *Vertical scalability* is required to take advantage of additional resources available on a given node, notably multi- and many-core architectures that can process the pub/sub traffic in parallel. *Horizontal scalability* allows supporting a higher load by adding more nodes to the cluster. Ideally, a linear increase in the number of nodes should result in a linear increase in maximum supported throughput.

(3) Filtering scheme agnosticism. The design and architecture of distributed pub/sub systems should not be dependent on a particular filtering scheme and in particular on the semantics and representations of publications and subscriptions. Most existing distributed pub/sub systems [11, 14, 17, 27, 40] support filtering schemes based on conjunctive predicates ($<$, $\leq$, $=$, ...) over discrete attribute values (integers, strings, ...), and their designs are closely tied to the nature of this particular representation. This applies, for instance, to the construction and maintenance of routing tables between brokers that drive the flow of publications. To minimize inter-broker traffic, these systems typically rely on the ability to determine containment relationships between subscriptions and/or to construct aggregated subscriptions. Yet, such features are not available with all content-based filtering schemes, notably with encrypted approaches [6, 18, 25, 26, 34]. As a matter of fact, there exist no fundamental reasons why content-based routing should be restricted to attribute- and predicate-based filtering: a pub/sub service should be able to integrate virtually any filtering scheme operating on the content of exchanged data using stored filters, as required by the application. Examples include not only encrypted filtering for privacy preservation, but also statistical methods such as Bayesian filtering [37] or even template matching for digital images (for instance, for face recognition) [10]. The architecture of the pub/sub system should be independent of the nature of the filtering scheme, while still allowing for specific optimizations at the level of a single node.

Contributions. In this paper, we revisit the design of a distributed content-based pub/sub engine for supporting high throughput, low latency, and horizontal and vertical scalability. We propose a novel approach based on a tiered architecture and inspired by dataflow programming techniques, which exploits parallelism in ways similar to MapReduce [19] and stream processing engines inspired from it [2, 5, 9, 33]. A set of independent operators, each spanning an arbitrary number of servers and taking advantage of multiple cores on individual servers, implement the three fundamental operations of content-based pub/sub: *subscription partitioning*, *publication filtering*, and *publication dispatching*. Interactions with the pub/sub system are managed by a set of independent *data converters and connection points* (DCCP) that maintain persistent connections with clients (publishers and subscribers).

We implement our approach in StreamHub, a pub/sub engine designed for operating on a public cluster or cloud. StreamHub leverages the runtime support of an existing stream processing engine such as S4 [33], Storm [2], or StreamMine [9]. We use the latter engine in our prototype implementation.

Our evaluation on a cluster with up to 384 cores on 48 physical machines indicates that StreamHub is able to sustain high-throughput workloads: up to 150 K subscriptions registered per second; and up to almost 2 K publications filtered per second with a population of 100 K stored subscriptions, resulting in an output flow of nearly 400 K notifications per second to interested subscribers.

We note that our contribution is not on the actual filtering scheme itself, which is supported by an independent library that can be chosen arbitrarily as long as it implements a simple and schema-oblivious API. We demonstrate the performance of StreamHub using the well-established counting algorithm of SIENA [12], and we leave the integration and comparison of other filtering libraries, and in particular those providing privacy-preserving encrypted matching [6, 18, 25, 26, 34], for future work. Similarly, while StreamHub is designed with elastic scalability in mind (i.e., the ability to dynamically adapt the number of servers associated with each operator according to the experienced workload), we leave the implementation of elastic server provisioning for future work and concentrate on the performance and scalability of the architecture with various static configurations.

Outline. The remainder of this paper is organized as follows. We survey previous work on distributed pub/sub systems in Section 2, and we present and motivate our proposed architecture in Section 3. We describe the implementation of StreamHub in Section 4, as well as the libraries used in our evaluation for filtering publications and clustering subscriptions. We evaluate our approach and compare it to a broker-based pub/sub system in Section 5, before concluding in Section 6.

2. RELATED WORK

We start by reviewing related work on distributed content-based pub/sub systems. We focus on high-efficiency middleware operating on dedicated machines and do not specifically elaborate on peer-to-peer approaches. Similarly, we do not discuss work targeting the simpler topic-based filtering model.

2.1 Publish/Subscribe Engines

Most earlier work on scalable pub/sub has relied on networks of *brokers*, which are dedicated machines, each performing the whole range of operations that compose the content routing task: (1) management of subscriptions from users and other brokers, (2) filtering of incoming publications against stored subscriptions and dispatching to local interested subscribers, and (3) filtering of incoming publications against routing tables for dispatching to other brokers. Brokers are typically organized in a *broker overlay*, with subscriptions and publications flowing between brokers according to its logical structure, typically a tree or a mesh.

Well-known examples of broker-based pub/sub middleware are SIENA [11], Gryphon [3], and PADRES [27]. In these systems, a client (publisher or subscriber) connects to one of the brokers, which then acts as its single point of contact. Brokers forward subscriptions registered by their clients towards neighboring brokers. These systems are based on a filtering scheme where subscriptions are defined as conjunctions of predicates ($<$, $\leq$, $=$, ...) on a set of discrete attributes values (integers, strings, ...). They rely on the ability to (1) determine containment relationships between subscriptions, and (2) construct aggregated subscriptions representing the interests of sets of subscriptions. Subscriptions are aggregated along the way from consumers to producers of information, taking advantage of containment relationships between subscriptions: a single aggregated subscription may represent the interests of many downstream subscriptions, thus reducing the number of subscriptions managed by the broker and improving filtering performance (see for instance [28]). This approach works well with few publishers and with subscriptions that have certain locality properties (e.g., subscribers with similar interests connect to the same broker). However, it requires complex algorithms for maintaining the consistency of forwarding tables and provides only limited benefits when information flows from many sources or when the subscription representation does not allow for containment or aggregation [6, 10, 18, 25, 26, 34, 37].

In broker overlays, under some workloads, publications may have to traverse a number of brokers that have no local interested subscriber but still have to filter the publication against stored subscriptions. These are called *forwarder-only* brokers. While techniques for rewiring the broker overlay have been proposed to tackle this problem [31], the presence of such forwarder-only brokers is intrinsic to a design where communication flows depend on the filtering scheme and on the current workload of stored subscriptions. Since all brokers play all roles in the pub/sub operation, the allocation of publishers and subscribers to brokers has a strong impact on the balance of load and on the overall filtering efficiency. A bad placement may result in a high number of messages being propagated between brokers. Some optimizations were proposed to address this problem by connecting subscribers with similar subscriptions to the same brokers [15], or by linking publishers and their expected subscribers to the same brokers [16, 32]. Cheung *et al.* [17] proposed to use similar techniques to rewire the PADRES overlay in order to reduce the environmental footprint of the pub/sub system. Again, these mechanisms are dependent on the filtering scheme and require the ability to determine proximity relations between subscribers and publishers. This would not be possible, for instance, with encrypted filtering approaches.

In contrast to systems based on overlay of brokers, the logical connections between the elements in our proposed architecture are independent of the nature of the subscription and publication workloads and of the nature of the filtering scheme. We support scaling each of the pub/sub operations independently by simply adding more processors to the set of nodes that support this operation. We do not require any specific optimization support from the filtering scheme, though we can leverage their existence for improving single-node performance inside filtering libraries. Note that our approach is readily applicable to architectures like Google's GooPS [35], where pub/sub is implemented by regional data centers consisting of clusters of brokers and interconnected by dedicated network links.

2.2 Filtering Mechanisms

Our architecture supports *pluggable* filtering mechanisms. As the design of new such mechanisms is not the focus of this paper, we only briefly discuss below a few well-known algorithms that can be readily used in our StreamHub implementation (see Section 4.2.1). Note that this list is far from being exhaustive.

SIENA uses a counting algorithm [12] for efficiently matching publications against subscriptions. Individual predicates are stored in a forwarding table and a subscription is detected as matching when all its predicates have been encountered. We use an implementation of this counting algorithm as the default filter in StreamHub for non-encrypted matching. Additional details on its operation are given in Section 4.2.1. Encrypted filtering can be supported for instance by *asymmetric scalar-product preserving encryption* [18], combined with pre-filtering [6] for efficiency. Gryphon [3] inserts the set of subscriptions into a matching tree: leaves contain subscriptions, non-leaf nodes contain tests, and outgoing edges represent the results of the tests. A publication traverses down the tree by following all matching paths and reports a matching subscription for each leaf node reached. PADRES uses a scalable filtering engine [22] that can leverage multiple cores on a shared memory architecture. By splitting the state of subscriptions and using multiple threads synchronized using either locks or transactional memory, the filtering throughput is significantly improved. We also exploit all the cores available on a machine and provide synchronization mechanisms for concurrent accesses to a shared state.

Other examples of filtering mechanisms that can be leveraged in the context of StreamHub include, but are not limited to, the following. RAPIDMatch [30] is a tree-based filtering mechanism that takes into account the sparseness of criteria definitions over the whole attribute set in some pub/sub workloads for greater efficiency. TAMA [41] trades accuracy and space complexity for efficiency by clustering range-based subscriptions in predefined sets based on discrete cuts of the definition range of each attribute. The use of discrete cuts leads to the presence of false positives, while the presence of subscriptions in multiple buckets leads to higher memory consumption, in return for faster filtering. Fabret *et al.* [21] use schema-clustering to minimize the number of filtering operations performed, along with techniques to improve cache performance of the algorithm. Filtering mechanisms also exist for boolean expressions [8, 23], XML documents and XPath expressions [4, 13], and compact data representation using Bloom filters [29].

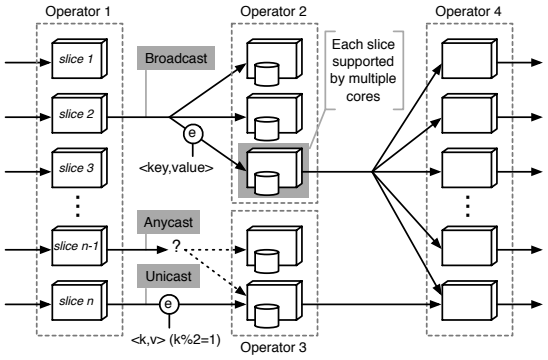

Figure 1: Architectural principles.

2.3 Clustering Subscriptions

Similarly to filtering mechanisms, we support subscription clustering by the means of pluggable libraries. Subscription clustering, as described in Section 4.2.2, splits the whole set of subscriptions maintained by the pub/sub system into clusters according to similarity (if a proximity metric is available on the subscription). This typically increases the level of containment and the potential for aggregation, which in turn improves the performance of the filtering operation. Classical clustering algorithms include K-means [39], Event Space Partitioning (ESP) [38], or R-trees [7].

3. ARCHITECTURE

Our design choices aim at maximizing pipeline, task, and data parallelism in order to support high-throughput and scalable pub/sub. The resulting architecture shares these design principles with big-data processing systems such as MapReduce [19], and in particular with the online models of computation on data *streams* that were inspired from it [2, 5, 9, 33].

Our architecture uses the base construction principles illustrated by Figure 1. It is composed of a set of *operators*, sharing the same code and supporting pipeline parallelism. Operators are organized as a directed acyclic graph (DAG). Communication takes place in the form of *events* flowing through the DAG of operators. Each operator can scale horizontally by using an arbitrary number of operator *slices*, each running on a different server and managing an independent state. Slices scale vertically by partitioning the received event load between all available cores on each server. There is no communication or shared state between the slices of an operator. Event forwarding between operators can use one the three primitives: *anycast* (sending to a random slice of an operator), *broadcast* (sending to all slices of an operator), or *unicast* (sending to a slice of an operator chosen according to a key).

We use a set of three operators. Each implements a different aspect of the pub/sub service: subscription *partitioning*, publication *filtering*, and publication *dispatching*. Thanks to the scalability properties of operators, one can easily adapt the number of physical machines and cores to the load experienced by each of these three operations. This load varies with the nature of the workload, such as the number, complexity, or selectivity of subscriptions. The load also varies with the nature of the filtering schemes. For instance, encrypted filtering requires more processing power than non-encrypted

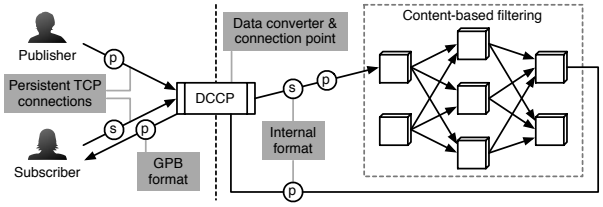

Figure 2: User view of StreamHub.

filtering. To sustain the same publication throughput, one should allocate more slices (servers) to the publication *encrypted filtering* operator.

We present in this section our operators and support mechanisms. We start by describing the endpoints used by external clients to access StreamHub. Afterwards, we present the operators that support content-based filtering, as well as the partition of the load onto different slices at each operator. The filtering operation itself is delegated to one or more *filtering libraries*. StreamHub also provides optional support for *clustering libraries*, which can partition the state of subscriptions in elaborate ways and speed up the filtering operation. As these libraries are pluggable components whose algorithms do not represent novel contributions of this paper, we describe them in Section 4.

3.1 Connection To and From Clients

The pub/sub operators are typically deployed on a cluster or a cloud, i.e., a set of machines with limited hardware heterogeneity. In our implementation, StreamHub, operators are implemented using the same language (C++). As a result, the internal communication and serialization formats between the elements forming the architecture can be selected based on performance criteria. Our implementation uses the efficient binary format provided by Boost libraries[1] for internal propagation of events. In contrast, clients may execute on different platforms and use a variety of languages. The choice of the external format is thus driven by its hardware and language independence. Our implementation uses Google Protocol Buffers (GPB),[2] which provide efficient serialization primitives for subscriptions, unsubscriptions, and publications while hiding language and platform heterogeneity.

Publishers and subscribers need a persistent and public connection point to the cluster or cloud supporting the pub/sub service. Connecting to any of the nodes supporting the pub/sub service is impractical in clouds (due to VM migrations) and often impossible in clusters (as most nodes do not have a public IP address). Our design features components external to the operators implementing the pub/sub service, that act as such persistent connection points. These are also in charge of translating between the external and internal representation format, and henceforth named *Data-Converter & Connection-Points* or DCCPs. Figure 2 presents a user-centric view of the system. Clients connect to a DCCP via a *persistent* TCP connection to enable low end-to-end delay for communication with the pub/sub service and, more importantly, to support asynchronous notifications of matching publications, as clients may not be directly reachable

[1] `http://www.boost.org/`
[2] `http://code.google.com/apis/protocolbuffers/`

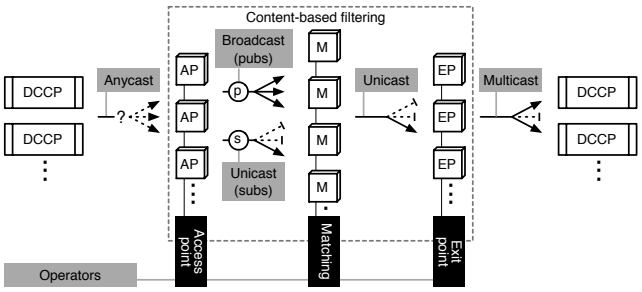

Figure 3: StreamHub processing operators (libraries and states not shown for clarity).

(for instance, they may be located behind a NAT or firewall). We note that this practical impossibility to reach clients directly limits the applicability of rewiring schemes for solutions based on brokers overlays [15–17, 32].

Several DCCPs can be used for the same StreamHub deployment, e.g., when the number of opened connections or the necessary bandwidth becomes too high for a single machine, when the cost of conversion creates a bottleneck, or on the same host when several network adapters are available.

3.2 Content-Based Routing Operators

In this section, we present the three operators that form the *core engine* of our scalable pub/sub architecture. These three operators are organized as a pipeline. They are listed in Table 1 and illustrated by Figure 3, together with the communication primitives used for propagating events between them. A detailed example of the path taken by subscriptions and publications within the StreamHub engine is shown by Figure 4.

3.2.1 Access Point Operator

The *Access Point* (AP) operator plays the role of the input operator. It receives events from any of the DCCPs. The selection of an AP operator slice by a DCCP is done at random to guarantee good load balancing properties. The role of the AP operator is to *partition* incoming subscriptions among all slices of the *Matching* (M) operator as follows.

Each incoming event has a *key*, which is a data structure that indicates the type of the client request (i.e., a new subscription, an unsubscription, or a publication). Subscriptions are not stored by the AP operator slices but are instead forwarded to the M operator that implements the filtering operation as we describe next. Our architecture can simultaneously support different filtering schemes (such as flat vs. structured data, encrypted publications and/or subscriptions, declarative vs. executable filters). Each filtering scheme is supported by a separate M operator. The choice of the destination operator depends on the *filtering scheme identifier* embedded in subscriptions. The same applies to publications.

Only one of the slices of the M operator holds any given subscription.[3] AP slices hence use unicast communication to select the appropriate M operator slice that will be responsible for an incoming subscription. The default mechanism relies on unicast and routes the subscription based on the hashing of the *subscription identifier* specified in event keys. We note that this selection mechanism is *stateless* and *reproducible*: an unsubscription will be routed from the AP to the M operator using unicast and will arrive at the M operator slice that actually holds the subscription.

Alternatively to this default mechanism, the user can decide to defer selection to a library, denoted by `libcluster` in Figure 4, which supports more complex forms of subscription clustering (see Section 4.2.2). A clustering algorithm can maintain a slice-supported state, which allows for deciding on subscription placement based on the content of that subscription. This incurs additional costs at the AP operator level, in return for a better performance at the M operator level. In case the selection mechanism is not deterministic and reproducible, unsubscriptions need to be broadcast to all slices of the corresponding M operator.

Publications need to be matched against all subscriptions. They are thus broadcast from the AP operator to all slices in the corresponding M operator. Note that our architecture targets deployments in clouds or clusters, which are typically supported by dedicated, high-performance network infrastructures. The broadcast operation of publications in our architecture is designed to take advantage of the availability of IP multicast in such settings for dispatching publications, although our current evaluation does not exploit this feature.

3.2.2 Matching Operator

The *Matching* (M) operator supports publication *filtering*. An M operator is associated with a library, denoted by `libfilter` in Figure 4, operating on the independently-maintained state at each of its slices. This library matches incoming publications against registered subscriptions. Different filtering implementations can be used as `libfilter` for different M operators, but they must comply with a simple API supporting two main operations: (1) storing/removing subscriptions based on their identifiers; and (2) processing a publication and returning a list of matching subscriber identifiers. At this stage, the content of the subscriptions and publications themselves is only accessed by the filtering library, making our architecture oblivious to the nature of the matching operation. The default filtering library provided with StreamHub is based on the SIENA counting algorithm [12] and is described in Section 4.2.1. Privacy-preserving filtering can be easily implemented using asymmetric scalar-product preserving encryption [18] or other mechanisms [25, 26, 34]. Recent proposals to reduce the cost of privacy-preserving encrypted filtering through the use of a pre-filtering stage [6] can also trivially be integrated to `libfilter` libraries.

Subscriptions are stored in the state maintained for each slice. This state can be accessed concurrently using read and read-write locks (see Section 4 for implementation details). As filtering only requires reading the subscription set, and since most pub/sub workloads are dominated by publications, this allows for vertical scaling of the filtering operation for each slice of the M operator on multiple cores.

An M operator slice calls its `libfilter` for each incoming publication and generates an output event composed by the publication p and a list of matching subscriber identifiers, $s_1, s_2, \ldots, s_n$. When this list is empty, an output event indicating the lack of matching subscription is generated. The event is then sent to the next operator, the *Exit Point* (EP), using unicast. The routing key for selecting the slice of the EP operator is the identifier of the publication p. As

[3]Resilience of subscriptions in the presence of nodes faults can be handled at the level of the underlying stream processing engine, for instance using active replication or checkpoint/replay techniques.

Operator	Role	Description
AP Access Point	Subscription *partitioning*	– Receives subscription events and dispatches each to a single slice of an M operator. Optionally applies subscription clustering using a `libcluster` library. – Receives publications, forwards them to all slices of an M operator.
M Matching	Publication *filtering*	– Receives subscriptions and forwards them to the `libfilter` library. The `libfilter` library stores the subscription and corresponding subscriber identifier in the operator slice state. – Receives publication events and forwards them to the `libfilter` library, which returns a set of matching subscriber identifiers. The M operator slice forwards each publication and list of matching subscriber identifiers to the EP operator by unicast, using the publication identifier as key.
EP Exit Point	Publication *dispatching*	– Receives a publication and list of matching subscriber identifiers. When all lists are received, prepares the notifications, splits the list of matching identifiers, and dispatches them to corresponding DCCPs.

Table 1: Operators supporting scalable CBR.

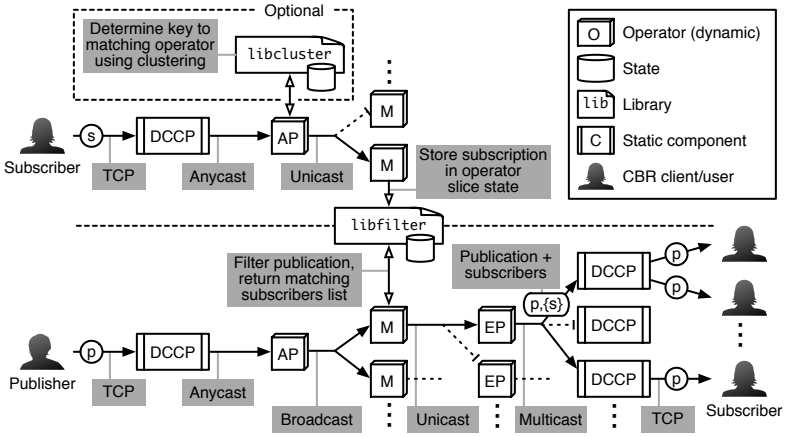

Figure 4: Path taken by subscriptions (top) and publications (bottom) in the StreamHub architecture.

a result, each slice of the M operator processing p will send its list of matching identifiers to the same slice of the EP operator.[4]

3.2.3 Exit Point Operator

The *Exit Point* (EP) operator acts as the output operator of the engine. It shares similarities with the *reduce* phase in the MapReduce terminology [19]. Each incoming publication will be filtered at all slices at the M operator level, but will be processed by a single slice at the EP operator level. An EP operator slice receives the publication and lists of matching subscriber identifiers from all slices of the M operator (or notifications of empty matching lists). Once lists have been received from all M operator slices (or after a timeout to avoid slow M operator slices to delay notifications), the EP operator proceeds with *publication dispatching*: it contacts each DCCP maintaining a connection to at least one interested subscriber and sends it a notification message together with the identifiers of matching subscribers connected to that DCCP. The latter is then in charge of propagating the notification to the actual subscribers.

4. IMPLEMENTATION

We implement our architecture on top of a stream processing engine, StreamMine [9]. Other frameworks also present the features and abstractions our implementation requires, such as S4 [33], Storm [2], or Continuous-MapReduce [5].

[4]If publications are of significant size, it is possible to have a single slice of the M operator send p and the others sending only their lists.

We present an overview of StreamMine in this section, as well as the `libfilter` and `libcluster` libraries supported by our prototype StreamHub.

4.1 StreamMine stream processing engine

StreamMine is a framework that targets scalable processing of information flows in the form of streams of *events*. Its architecture follows the design principles presented at the beginning of Section 3 and illustrated in Figure 1. StreamMine allows defining operators organized in a DAG. Operators are composed of a number of *slices* that typically reside on separate machines in a processing cluster or a cloud, and have unique identifiers (*id*). Slices of the same operator share the same code, in the form of an *event handler*, a function that is called whenever a new event is received and may emit new events for operators downstream in the DAG (for instance, in Figure 1, slices of operator 1 may generate events for operator 2 or 3). The final operator is responsible for propagating the stream of resulting events to the clients of the event-based application.

StreamMine provides support for communication among operators and for management of the state that may be maintained by operator slices. Communication between operators is exclusively conducted based on events. Each event is a <key,value> pair. StreamMine supports *unicast*, *anycast*, and *broadcast* communication primitives. The operator to which events are sent is chosen upon their creation. StreamMine communication primitives are oblivious to the content of events, which may only be used inside event handler functions. All communications take place on pre-established and persistent TCP connections: all slices of one operator are

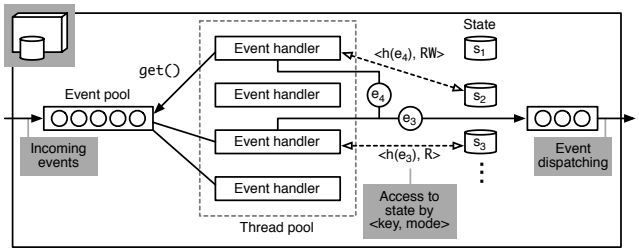

Figure 5: Details of an operator slice from Figure 1 supported by 4 threads.

persistently connected to all slices of the next operator(s) in the DAG.

The *unicast* primitive determines the *id* of the slice in the next operator by using a key and a hash-function (a simple modulo by default). A specific hash-function can be specified by the event handler function, which may implement more sophisticated stream partitioning mechanisms. The *anycast* primitive sends the event to a random slice of the next operator. The *broadcast* primitive sends the event to all slices of the next operator.

StreamMine operators slices can be either stateless or stateful, and can be composed of several independent processing threads to support vertical scalability when running on a multi-core processor. All threads of all slices of a given operator support the execution of the same event handling function. Figure 5 presents a detailed view of the state of the greyed slice from Figure 1. Threads from a pool process incoming events from an input queue using a part of the state determined according to the event identifiers. The state of a slice (such as a window of events or summary information from previously processed events) is managed by StreamMine, and can be partitioned using the same keys that are used for unicast routing between operators. Each thread accesses the state corresponding to the event to process using the appropriate read-only (`R`) or read-write (`RW`) mode. An event with key k_1 can be processed in parallel with another event with key k_2 as long as $k_1 \neq k_2$ or, if $k_1 = k_2$, only when both accesses are read-only.

4.2 Filtering and Clustering Libraries

In the following, we present the filtering and clustering libraries (`libfilter` and `libcluster`) that StreamHub currently supports. While these libraries are based on known algorithms and do not represent novel contributions *per se*, they contribute to the overall performance of StreamHub as studied in Section 5.

4.2.1 Filtering Libraries

StreamHub can support any filtering scheme if implemented through an appropriate `libfilter` library. We listed variants of filtering schemes in Section 2.2. We note that our architecture also supports filtering schemes that need to maintain a state for each of the subscription they store, across the processing of several publications. For instance, a subscriber might wish to receive only the n^{th} publication that matches a given subscription, or be able to send subscriptions on the statistical evolution of publications attributes (e.g., over a window of publications). The corresponding state can be maintained by the `libfilter` in the slice-supported state.

StreamHub currently features an attribute-based filtering scheme library (`libfilter`) that uses a counting algorithm similar to that of SIENA [12]. It organizes predicates for received subscriptions in a forwarding table. An incoming publication will traverse and match in this forwarding table the predicates organized in conjunction sets. Each subscription is associated with a counter that specifies how many of its predicates have been matched so far. When a predicate is satisfied, the counters of all associated subscriptions are increased, and the whole subscription is marked as matched when all predicates of a subscription have been satisfied. Numerical predicates are indexed according to their type ($=$, $<$, $>$) and sorted by values in order to speed up traversals. Therefore, typically only a small part of the graph is traversed by publications. The algorithm generally scales sublinearly in the number of evaluated conjunction sets.

4.2.2 Clustering Libraries

When using multiple slices in the matching operator, each of these slices holds a subset of all subscriptions and filters publications concurrently with other matchers. By default, we partition subscriptions in a simple and deterministic way using a hash. This splits the load among all M operator slices. However, for many filtering schemes, filtering performance can be improved when subscriptions are partitioned in a content-aware manner. These types of subscription clustering are more costly than hash-based partitioning but they result in gains for the publication filtering performance. This is the case of the attribute-based filtering scheme described previously. As similar subscriptions are stored in the same M operator slice, the filtering algorithm may be able to better factorize common predicates and achieve higher filtering performance (this typically depends on the ability of the filtering operator to support containment determination between subscriptions). For pub/sub systems that process more publications than subscriptions, the relative gain can be significant as we show in our evaluation.

StreamHub supports various clustering algorithms by the means of `libcluster` libraries, that can optionally maintain state about previous subscriptions. When multiple filtering schemes are supported by multiple M operators, each slice of the AP operator supports a different `libcluster` (or default hash-based unicasting) for each such M operator. Deterministic clustering allows unicasting unsubscriptions while non-deterministic clustering require broadcasting unsubscriptions. StreamHub features the two clustering libraries described below.

K-Means [39]: This clustering algorithm performs a partitioning of the subscriptions into K groups and a repetitive re-assignment based on the distance between subscriptions and groups until convergence. The algorithm is stateful and non-deterministic. We implement it in an online manner (*sequential* K-Means) for the dynamic clustering of subscriptions.

Event Space Partitioning (ESP) [38]: The space of subscriptions is represented as a d_s dimensional space, where d_s is the number of attributes. Each M operator slice is responsible for subscriptions that fall within its portion of the space. Subscriptions that intersect multiple domains are managed by the M operator slice that hosts the first attribute in lexicographic order. As the value d_s cannot be known in advance with content-based routing, it will increase when encountering subscriptions with unknown attributes. This clustering mechanism is stateful but deterministic.

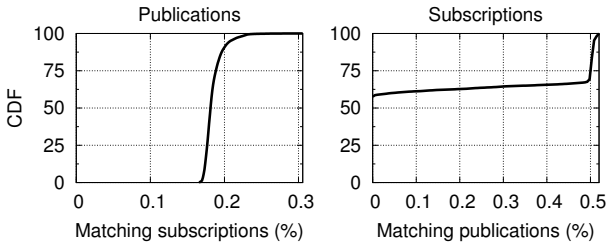

Figure 6: Workload characteristics: cumulative distribution of matching set sizes for publications (left) and matching ratios for subscriptions (right).

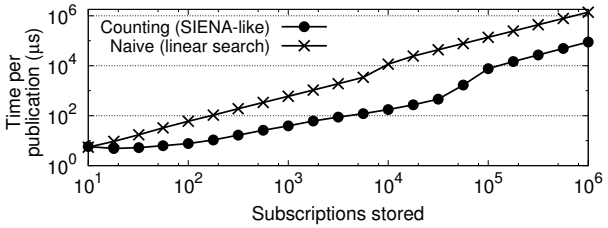

Figure 7: Performance of the counting `libfilter` for filtering incoming publications with respect to the size of the stored subscriptions set.

5. EVALUATION

In this section, we present the experimental validation of StreamHub on a cluster of 48 nodes, each with two quad-core Intel Xeon (E5405) 2 GHz processors and 8 GB of RAM (384 cores total), interconnected with full-duplex 1 Gbps Ethernet. Our implementation uses the C++ language. We configure StreamMine to use batching between operators. Up to 16 KB of events can be stored in output buffers for each operator, and sent in batches or after a time limit. Batching allows increasing maximal supported throughput but has an impact on delays, as we demonstrate at the end of this Section.

We first present the characteristics of the pub/sub workload used for the evaluation, followed by the baseline performance of the counting algorithm (`libfilter`). We then proceed to a operator-by-operator evaluation of the StreamHub architecture, highlighting performance and scalability of each of the operators. We describe the impact of subscription clustering (`libcluster`) and evaluate how the system scales when using an increasing number of nodes. Finally, we present a comparison of our approach with a broker overlay solution running on the same cluster.

5.1 Experimental Workload

We constructed an experimental workload similar to the one used for the evaluation of Meghdoot [24]. It targets an attribute-based filtering scheme. We gathered five years of quotes for 200 randomly selected stocks from Yahoo! Finance [1]. This corresponds to over 250,000 publications. We built synthetic subscriptions based on the same categories as in [24]. These subscriptions contain a variety of ranges and equality predicates on the attributes of stock quotes, namely the symbol, date, exchanged volume, and daily statistics on their price (open, close, high, low). The characteristics of the workload are detailed in Figure 6. They represent a moderately selective type of pub/sub workload: a publication needs to be dispatched to a median of 0.18% of all subscriptions (with 100,000 subscriptions, each publication generates a median of 180 notifications), while a large part of subscriptions do not find publications of interest in the workload but yet need to be processed by the M operator.

5.2 Baseline Filtering Performance

We first evaluate the raw performance of the `libfilter` filtering library based on the SIENA-like [12] counting algorithm. We compare it to a naive linear-search filtering mechanism acting as a baseline. Figure 7 indicates that the filtering operation cost evolves sublinearly and is at least an order of magnitude better than the naive algorithm above 500 stored subscriptions. Nonetheless, the cost of filtering can grow quite high with large sets of subscriptions, as can be observed on the right side of the graph. This highlights the importance of scaling the processing of incoming publications horizontally and vertically to sustain a high filtering throughput, and to process an incoming publication against subsets of the overall set of subscriptions to reduce dispatching delays.

5.3 Performance of Operators

We now proceed to an operator-by-operator evaluation of StreamHub. Our methodology is to add one operator at a time, replacing the operators downstream the DAG by *sink* operators that receive the events but do not process them further. We denote such sink operators as S(AP), S(M), and S(EP). We focus our evaluation on the scalability aspects of each operator. All experiments are based on 180-seconds runs of StreamHub, during which the system is fed with subscriptions and/or publications as fast as possible to observe the maximal achievable throughput. When observing the performance of the publication filtering operation, subscriptions are registered before starting the measurement.

In our evaluation, the DCCPs are replaced by *generators* that inject the workload into the AP operator. We first verified that these generators can provide the system with a sufficient throughput of publications and subscriptions and do not represent a bottleneck. Our results (not shown) indicate that the generators are able to nearly saturate the input bandwidth capacity of the nodes that host the AP operator slices. This indicates they will not impair the remainder of the evaluation.

We present the complete operator-by-operator evaluation results in Figure 8. We look primarily at the throughput in terms of bandwidth and events processed.

5.3.1 AP Operator Scalability

The first column of two plots in Figure 8 presents the AP operator scalability. It depicts the maximal input and output throughput of the operator with a publications-only workload. This corresponds to the worst case scenario since publications, unlike subscriptions, need to be broadcast by each AP operator slice to all sink S(M) operator slices. As expected, the input throughput of the AP operator is inversely proportional to the number of sink S(M) operator slices: a copy of each publication needs to be made for every S(M) operator slice and the bottleneck becomes the output bandwidth of AP operators. The planned support for IP multicast between the AP and M operators would boost

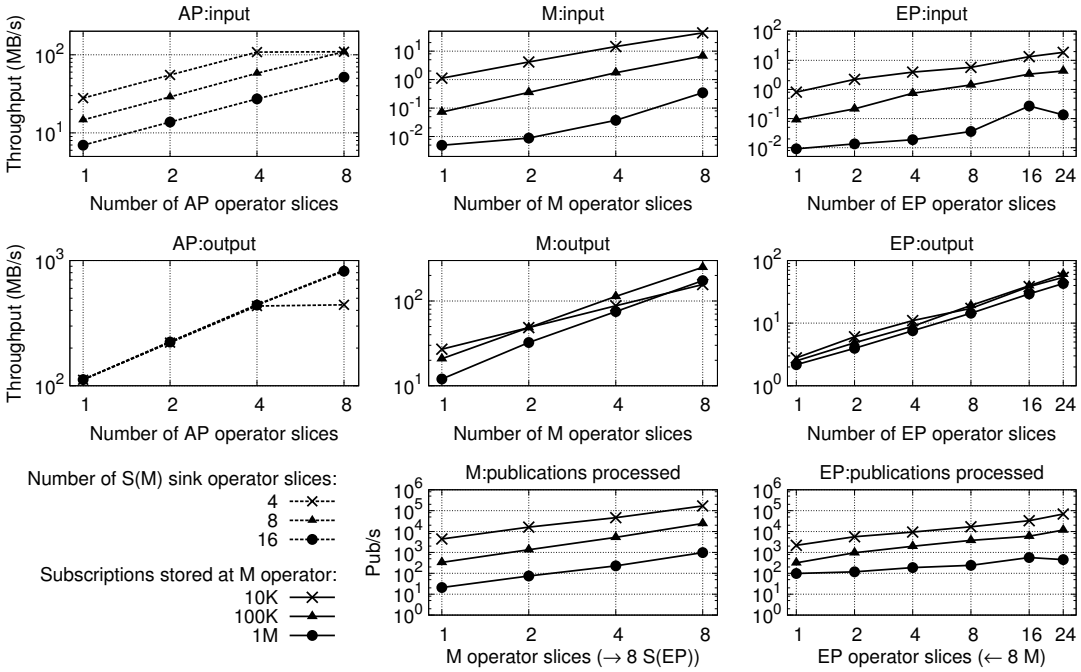

Figure 8: Scaling of StreamHub operators: Input and output throughput for all operators when varying the number of slices and subscriptions. Each operator is evaluated with the downstream operator replaced by a sink. We use one physical machine per slice.

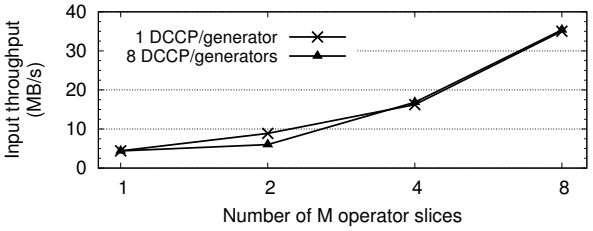

Figure 9: Scaling of the M operator receiving and storing a subscriptions-only workload. The throughput corresponds to the traffic between the DCCP/generators and the AP operator, with 8 AP operator slices.

performance for this operation. We observe nonetheless that this output throughput nearly saturates the cross-bandwidth of the connections between the AP and S(M) operator slices, and is able to saturate the input bandwidth of the S(M) operator slices (serving 104 MB/s of publications to each of them). This indicates that the AP operator will not hinder scalability when the S(M) sink operator slices are replaced by real M operator slices that need to perform the computationally-intensive filtering operation, as confirmed by our next experiment.

5.3.2 M Operator: Subscription Storage Scalability

We start the evaluation of the M operator by assessing the scalability of the subscription storage with a subscriptions-only workload. Figure 9 presents the average bandwidth that the generators are able to push through the AP operator for storage at the M operator level. We use a set of 8 AP operator slices so that the AP operator does not form a bottleneck. We use 1 to 8 M operator slices. We observe that the scalability of the subscription storage is almost linear and StreamHub is able to register a flow of 35.4 MB/s with 8 M operator slices, which corresponds to a constant flow of 150,000 subscriptions stored per second. We observe a slight degradation of the throughput when using 8 generators and only 2 M operator slices. The reason was tracked down to overflows in input buffers of the M operator slices, leading to retransmissions of messages from the AP operator and some loss of bandwidth.

5.3.3 M Operator: Publication Matching Scalability

We now investigate the scalability of the filtering operation at the M operator level, i.e., matching each publication against the set of stored subscriptions. We use a set of 8 sink S(EP) operator slices as the downstream operator and 8 AP operator slices for the upstream operator. The second column of three plots in Figure 8 presents the achieved input/output throughput and the number of publications that the M operator filters per second, including transmission to the downstream S(EP) operator. We clearly observe that the architecture scales: the addition of new operator slices to the M operator results in linear increase of its processing capacity. Note that, as expected from the workload characteristics (median matching set of 0.18% of stored subscriptions), the bandwidth requirements are higher for output than for input because the publications are augmented with a potentially large list of matching identifiers.

5.3.4 EP Operator Scalability

We complete the operator-by-operator scalability evaluation by replacing the S(EP) sink operator slices with their real counterparts that perform publication dispatching. We

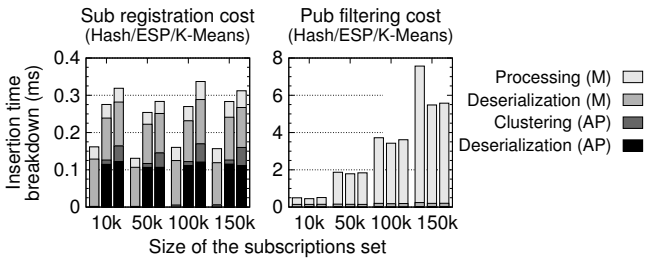

Figure 10: Overhead of clustering for storing subscriptions (left) and impact on filtering efficiency (right). Each group of stacked bars shows the breakdowns of average costs for one subscription or one publication matched against the corresponding subscription set. Each group has three bars for hash-based (no clustering), ESP, and K-Means.

use 8 generators, 8 AP, and 8 M operators slices. We observe in the third column of three plots of Figure 8 the input/output throughput of the EP operator and the number of publications that are effectively dispatched. With a configuration of 8 EP operator slices, StreamHub is already able to filter and dispatch from 3.8 K to 17 K publications per second, for respectively 100 K and 10 K stored subscriptions (corresponding to 684 K and 306 K notifications sent out to subscribers per second, respectively).

5.3.5 Discussion

The results of the operator-by-operator evaluation clearly show that the subscription registration and publication filtering operations scale by adding more slices (and thus physical machines) to the operator that supports them. Adequately provisioning the architecture allows handling an arbitrary number of publications and subscriptions. One should point out that the specific workload considered in our evaluation is costlier for the operators that deal with publication dispatching (EP) and matching (M) than for handling and forwarding incoming publications (AP).

5.4 Impact of Clustering

We now investigate the impact of using a `libcluster` library at the AP operator level for clustering subscriptions. Our objective is to evaluate if the additional cost for registering a subscription in the system using content-aware clustering is compensated by the subsequent performance gain when filtering publications against stored subscriptions. We present in Figure 10 the time required to store a subscription at the M operator level (left) and process an incoming publication against the set of stored subscriptions (right). Times are obtained by averaging over 10,000 events. The breakdown distinguishes between the different operations at the AP and M operators: deserializing the event at the AP operator level (for subscriptions when using clustering) and processing it (clustering, storing, or matching) at both the AP and M operators levels.

We observe that the cost of subscription insertion increases due to the additional deserialization and treatment at the AP operator level. On the other hand, the use of clustering yields significant performance gains when matching publications against a large set of subscriptions: for 150 K subscriptions matching is 25 to 27% faster when using K-Means or ESP. This supports our claim that using a `libcluster` library, when applicable to the filtering scheme being used, may significantly increase the filtering performance or reduce the number of M operator slices required to sustain a given publication filtering throughput requirement. At the same time, the use of a `libcluster` library does not break the separation of concerns and filtering schema agnosticism that underpins the complete architecture.

5.5 Overall Performance

Our last experiment with StreamHub relates to the overall provisioning and scaling of the complete architecture. We consider the case where a cluster or a cloud virtual environment needs to be scaled up to increase the throughput of the pub/sub process, with the objective of offering a linearly increasing performance as more physical nodes are added to support the service, and to sustain low and predictable end-to-end delays. As previously demonstrated, the optimal assignment of slices (and thus, physical machines) to operators depends on the nature of the workload. For the purpose of this evaluation, we determined the best configuration for a given budget of machines based on the operator-by-operator experiments. Our architecture is designed to easily support dynamic scaling by migrating slices between physical machines. We leave the integration of such mechanisms and the appropriate decision-making systems to future work. Figure 11 presents the evolution of the publication throughput with clusters of 8 to 32 machines (our other machines are used for 8 generators and 16 sink DCCPs). We observe that the scalability objectives of StreamHub are met: there is a linear gain in performance between 8 to 32 machines. The maximal supported throughput between the smaller and the larger configuration when using the ESP partitioning is actually 4.26x higher, which is mostly due to the reduced contention on the AP operator slices. We note that the impact of clustering is consistent with what we observed in Figure 10: throughput is from 10% to 28% better with clustering (see Figure 8). Table 2 presents the average end-to-end delays (between the reception of a publication and the reception of the corresponding notifications by the subscribers) observed by clients, and their variations. In this case, we selected the throughput to be around half of the maximal supported throughput in the 16 and 32 nodes configurations. We consider two settings, with batching and without. Batching increases throughput but also introduces extra delays. Disabling it divides the maximal supported throughput by approximately a factor of 2. In both cases though, the delays are low (around a second with batching, or a fraction of a second without it) and predictable as they show only slight variations between publications.

5.6 Comparison with PADRES

For completeness, we have also performed experiments with the same set of subscriptions and publications using the most recent version of PADRES [27] (v2.0). As detailed in our introduction, PADRES is based on different design choices than StreamHub: it establishes and maintains a network of brokers that collectively implement pub/sub functionality and is specific to a particular filtering scheme, while our design dedicates different machines to each operation and is independent from the actual filtering scheme that is used.

Our PADRES setup consists of one publisher, one subscriber, and a varying number of brokers. We verified that

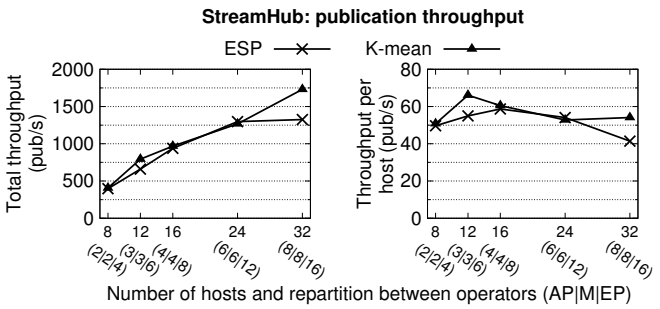

Figure 11: Throughput of StreamHub with 100,000 subscriptions, using `libcluster` and the workload-optimal configurations for each number of available machines. The number of slices at each operator is indicated within parentheses.

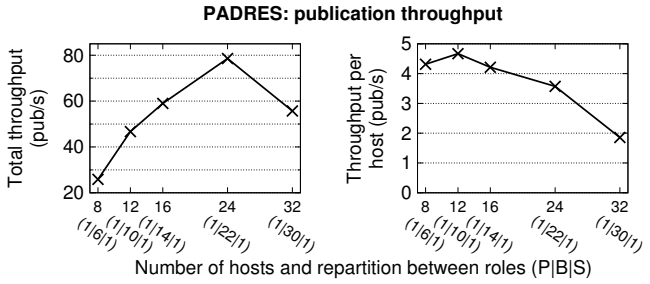

Figure 12: Throughput of PADRES with 100,000 subscriptions. The number of hosts is indicated within parentheses: a single publisher (P) and a single subscriber (S) were sufficient to fully load the brokers (B).

neither the publisher nor the subscriber represent a bottleneck in the experiments. The publisher and the subscriber are connected to every broker. Subscriptions and publications are randomly partitioned among brokers. Every machine executes 4 broker instances, which corresponds to half of its available cores. Using all cores on each machines yielded lower performance figures, probably due to contention on resources. Results are averaged over 100 publications, measured after an initial warm-up phase of 2,000 messages to enable JIT optimizations.

Figure 12 shows the throughput achieved with the same 100,000 subscriptions as for Figure 11 and various numbers of brokers. We observe that PADRES scales well for up to 88 brokers (i.e., 22 machines, each running 4 broker processes) but seems to suffer when adding more brokers. Actually, each publication has to be filtered by multiple brokers for propagation to other brokers due to the use of routing tables that are constructed according to the filtering schema that is used. This adds to the overall load and reduces the contribution of each broker to overall throughput (Figure 12, right). We finally observe that the maximal raw throughput achieved in our cluster is two orders of magnitude higher with our architecture than with PADRES. While a part of this difference can be accounted to language and implementation differences, the higher parallelism and independence of operations in our architecture clearly helps improving the filtering throughputs.

Configuration	(AP **4**|M **4**|EP **8**)		(AP **8**|M **8**|EP **16**)	
Batching	16 K	None	16 K	None
Publications/s	500	200	1,000	500
Average delay	1.06 s	0.36 s	0.98 s	0.22 s
Std. dev.	0.28 s	0.17 s	0.3 s	0.15 s

Table 2: End-to-end delays (settings as Figure 11).

6. CONCLUSION

We presented a novel design for high-throughput pub/sub services. We focused on the support of large-scale applications communicating through a managed environment providing the pub/sub service, such as a publicly available cluster or a public cloud deployment. Our architecture is highly parallel and scalable, and can readily support arbitrary complex filtering schemes, including encrypted or state-based filtering. We do so by departing from previous approaches based on broker overlays and by decoupling the architecture and communication flows of the pub/sub system from the filtering scheme(s) and the subscriptions workload. Our implementation, StreamHub, splits the pub/sub service into fundamental operations, allocated to horizontally and vertically scalable operators supported by a scalable stream processing engine. The evaluation of StreamHub on a cluster with up to 384 cores indicates that it can sustain high throughputs of subscription registrations and publication filtering: we filter thousands of publications against hundreds of thousands of registered subscriptions, resulting in hundreds of thousands notifications sent to clients every second.

This work opens several research perspectives that will be part of our future work on high-throughput pub/sub services for large-scale application compositions. The use of privacy-preserving encrypted filtering is a growing requirement for applications running on multiple private clouds and supported by untrusted public cloud infrastructures providing the pub/sub service. We plan on integrating such encrypted filtering approaches [18, 25, 26, 34]. We recently proposed techniques for lowering their computational cost [6], that we can easily integrate as `libfilter` libraries. Publications and subscriptions will need to be encrypted and decrypted as they enter and leave untrusted public clouds. This can be supported in a parallel and scalable manner at the AP and EP operator levels, and supported by pub/sub-specific key management techniques.

We also plan to support *elastic scaling* of the StreamHub architecture. This requires observing the workload experienced by each of the three operators and adapting the number of physical machines for each operator according to the requirements of the corresponding pub/sub operation. To that end, we intend to extend StreamHub with a resource provisioner for stream processing.

Acknowledgments. 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). We are grateful to Abhishek Gupta and Amr El Abbadi for their insights on using Yahoo! finance publication workloads.

7. REFERENCES

[1] `http://finance.yahoo.com/`.

[2] Storm: Distributed and fault-tolerant realtime computation. `http://storm-project.net`.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.

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

[5] N. Backman, K. Pattabiraman, R. Fonseca, and U. Cetintemel. C-MR: continuously executing MapReduce workflows on multi-core processors. In *MapReduce workshop*, 2012.

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

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[8] S. Bittner and A. Hinze. The arbitrary boolean publish/subscribe model: making the case. In *DEBS*, 2007.

[9] 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.

[10] R. Brunelli. *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley, 2009.

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

[12] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.

[13] C.Y. Chan, P. Felber, M.N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.

[14] R. Chand and P. Felber. Scalable distribution of XML content with XNet. *IEEE TPDS*, 19, 2008.

[15] A. Cheung and H.-A. Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Trans. Comput. Syst.*, 28(4), 2010.

[16] A. Cheung and H.-A. Jacobsen. Publisher placement algorithms in content-based publish/subscribe. In *ICDCS*, 2010.

[17] A. Cheung and H.-A. Jacobsen. Green resource allocation algorithms for publish/subscribe systems. In *ICDCS*, 2011.

[18] S. Choi, G. Ghinita, and E. Bertino. A privacy-enhancing content-based publish/subscribe system using scalar product preserving transformations. In *DEXA*, Lecture Notes in Computer Science, 2010.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[20] P.T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2003.

[21] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[22] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *DEBS*, 2009.

[23] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *SIGMOD*, 2010.

[24] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Middleware*, 2004.

[25] M. Ion, G. Russello, and B. Crispo. Supporting publication and subscription confidentiality in pub/sub networks. In *SecureComm*, 2010.

[26] Mihalea Ion, Giovanni Russello, and Bruno Crispo. An implementation of event and filter confidentiality in pub/sub systems and its application to e-health. In *CCS*, 2010.

[27] H.-A. Jacobsen, A. Cheung, G. Lia, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES publish/subscribe system. In *Handbook of Research on Adv. Dist. Event-Based Sys., Pub./Sub. and Message Filtering Tech.*, 2009.

[28] K. R. Jayaram and P. Eugster. Split and subsume: Subscription normalization for effective content-based messaging. In *ICDCS*, 2011.

[29] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS*, 2008.

[30] S. Kale, E. Hazan, F. Cao, and J. P. Singh. Analysis and algorithms for content-based event matching. In *DEBS*, 2005.

[31] R. S. Kazemzadeh and H.-A. Jacobsen. Opportunistic multipath forwarding in content-based publish/subscribe overlays. In *Middleware*, 2012.

[32] W. Li, S. Hu, J. Li, and H.-A. Jacobsen. Community clustering for distributed publish/subscribe systems. In *CLUSTER*, 2012.

[33] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *KDCloud*, Sidney, Australia, 2010.

[34] C. Raiciu and D. S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrastructures. In *Securecomm*, 2006.

[35] J. Reumann. Pub/Sub at Google. CANOE and EuroSys Summer School, 2009.

[36] L. Romano, D. De Mari, Z. Jerzak, and C. Fetzer. A novel approach to QoS monitoring in the cloud. In *CCP*, 2011.

[37] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk e-mail. In *AAAI Workshop on Learning for Text Categorization*, 1998.

[38] Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription partitioning and routing in content-based publish/subscribe networks. In *DISC*, 2002.

[39] T. Wong, R. Katz, and S. McCanne. An evaluation of preference clustering in large-scale multicast applications. In *INFOCOM*, 2000.

[40] Y. Yoon, V. Muthusamy, and H.-A. Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *ICDCS*, 2011.

[41] Y. Zhao and J. Wu. Towards approximate event processing in a large-scale content-based network. In *ICDCS*, 2011.