Subscription Normalization for Effective Content-Based Messaging

K.R. Jayaram, Weihang Wang, and Patrick Eugster

Abstract—Efficient subscription summarization and event matching is key to the scalability of content-based publish/subscribe networks (CPSNs). Current summarization and event matching mechanisms based on *subscription subsumption* induce heavy event processing load on brokers degrading the performance of CPSNs especially under high rates of churn, i.e., addition, deletion, or modification of subscriptions. Yet, many modern CPS applications such as location-based services or algorithmic trading inherently rely on high frequency subscription changes. This paper describes Beretta, a dynamic CPSN which sustains high throughput and low event-propagation latencies even under a high frequency of subscription changes. Beretta leverages *strong event typing* and represents all subscriptions in a *normalized form* as combinations of *value intervals* and *set inclusions* without compromising on expressiveness. Beretta's "split and subsume" broker algorithm reduces the complexity of matching an event from $O(K\,N)$ to $O(K \log N + |result|)$, with N being the number of subscriptions for the event type and K the number of its attributes. Event types and normalization are exploited to *split* subscriptions into predicates on *individual event types and attributes* and to efficiently regroup these in *segment trees* and *hash maps* which yield excellent subsumption properties and support attribute-wise split filtering during event matching. Normalization enables the *systematic* introduction of parameters into subscriptions to support both parametric and structural updates. This paper also empirically demonstrates the performance improvements due to our techniques through realistic algorithmic trading and highway traffic monitoring benchmarks.

Index Terms—Subscription, summarization, subsumption, normalization, content-based, messaging

1 Introduction

By focusing on the *exchanges* among interacting parties rather than the parties themselves, the *publish/subscribe* paradigm [1] is an appealing candidate for building scalable networked distributed systems. This dynamic interaction culminates in content-based publish/subscribe (CPS), where subscriptions are based on event *content* rather than on channels or topics. Content-centric communication abstractions have been more recently investigated in the context of future Internet design [2], [3]; the advent of software-defined networking (SDN) supports the implementation of such models.

effectively and efficiently route published events to subscribers with corresponding interests, existing application-level CPS systems typically construct *overlay networks* called "content-based publish/subscribe networks" (CPSNs). A CPSN consists of several event routers—commonly referred to as *brokers*—that interconnect publishers and subscribers often making use of *advertisements* of publishers in addition to subscriptions to transmit events to subscribers. That is, advertisements and subscriptions are propagated down- and up-stream respectively to set up connections such as to ensure that there is a path for the propagation of an event from any publisher to any subscriber with a potentially matching subscription [1]. Current CPSNs however present a number of limitations in their core constituents:

- *Matching algorithms.* Matching events to subscriptions, which occurs at each broker in a CPSN, is key to the efficient routing of events. Many matching algorithms, however, have a time complexity of $O(N)$, N being the number of subscriptions matched.
- *Subscription summaries.* Efficient subscription summarization is a key requirement for efficient routing of subscriptions and to avoid storing every subscription at every broker. Existing summarization algorithms, however, choose accuracy over efficiency thereby producing *complex subscription summaries*.
- *Subscription updates.* Several emerging CPS applications involve updates to their subscriptions at a high frequency [4], [5]. Examples include algorithmic trading and location-based services; examples for corresponding updates are changes to threshold values on prices for purchasing stock and updates to absolute geographical ranges of interest based on the change of vantage point induced by mobility respectively. In existing CPSNs, the typical way to update a subscription is through a *re-subscription*, which involves issuing a new subscription and canceling the old (stale) subscription. Re-subscriptions, however, typically involve $O(N)$ operations at least at one broker and potentially at many more brokers depending on the effectiveness of subscription summarization algorithm.

- *K.R. Jayaram is with IBM T.J. Watson Research Center, New York, NY. E-mail: jayaramkr@us.ibm.com.*
- *W. Wang and P. Eugster are with the Department of Computer Science, Purdue University, West Lafayette, IN. E-mail: wang1315@purdue.edu, p@cs.purdue.edu.*

Manuscript received 16 July 2013; revised 10 June 2014; accepted 18 June 2014. Date of publication 7 Sept. 2014; date of current version 7 Oct. 2015. Recommended for acceptance by D. A. Bader.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2355823

In this paper, we present Beretta, a novel CPSN which tackles the above-mentioned three problems through the following technical contributions:

1) *Subscription normalization* based on strong event typing without loss of expressivity of subscriptions. The normalized form is a combination of *value intervals* and *set inclusions*. Our use of event types in lieu of a structural approach allows for aggressive performance optimizations without hampering interoperability or the addition of new event types.
2) An efficient algorithm for matching events to subscriptions, which reduces the number of elementary predicates evaluated for matching an event from $O(K\,N)$ to $O(K\,log\,N)$, with N being the number of subscriptions for the event type and K the number of its attributes. Our algorithm relies on a divide and conquer strategy which we call "split and subsume"—event types and normalization are exploited to split subscriptions twice, first into predicates on individual event types and further based on attributes; and to efficiently regroup these in *augmented segment trees* and *hash maps*.
3) *Inherent parameterization* of normalized subscriptions to efficiently implement parameter-based [4], [5] as well as structural subscription updates.
4) *Subscription summarization approximation* to efficiently handle subscription updates as well as joining/leaving of subscribers in decentralized CPSNs.

While normalization has long been used in the management of data "at rest", we believe our work is the first to apply such concepts to queries on live data, and to leverage them for query updates. This paper extends a previous publication at ICDCS 2011 [6], yet includes a novel means of intersecting partial match sets efficiently based on augmented segment trees, and evaluates our approach through a new real-life benchmark.

The rest of this paper is organized as follows. Section 2 presents background information. Section 3 presents the state-of-the-art and its limitations. Section 4 introduces our subscription model. Section 5 describes our algorithm, and Section 6 presents an overview of our empirical evaluation. Section 7 draws conclusions. Due to space constraints, this paper also includes supplementary material consisting of three appendices. Evaluation results are presented in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2355823. Appendix B, available in the online supplemental material, analyzes the complexity of our algorithm. A detailed analysis of related work is available in Appendix C, available in the online supplemental material.

2 Background, Model, and Definitions

This section presents notation employed, assumptions made, and definitions of concepts used in this paper.

2.1 System Model

We consider CPSNs using decentralized, dedicated, interconnected, broker processes b_i to convey events between client nodes c_i, i.e., publishers and subscribers. Brokers which serve client processes are called *edge* brokers. We use the term *client* to refer to either publishers or subscribers. For presentation simplicity we assume CPSNs forming directed graphs.

We focus on CPS systems whose routing algorithms follow the principles of (a) *downstream replication* where an event is routed in a single copy as far as possible from the publisher and only cloned downstream as close as possible to the subscribers interested in receiving it, (b) *upstream evaluation* where unwanted events are filtered away as close as possible to the publisher to avoid wasting bandwidth, and (c) *subscription-based reverse-path forwarding*, where subscriptions are routed from the subscriber to the edge broker to which publishers are connected, thereby forming a spanning tree rooted at each subscriber, and events follow the reverse path along this tree to the subscriber. In this context, the CPSN can either (c.1) perform *subscription flooding*, i.e., route every subscription to every publisher in the CPSN, or (c.2) use *advertisements*. An advertisement defines the types of events produced by a publisher, and a subscription on an event type T has to be routed *only* to the edge broker with at least one publisher which has advertised T.

2.2 Events and Subscriptions

An event e is a set of attribute/value pairs $\{a_1 : v_1, \ldots, a_k : v_k\}$ which are typically of primitive types τ_i including scalar types `int`, `float`, etc. and character `strings` (cf. [1]). A subscription is usually represented as a predicate Φ based on a grammar like the following:

$$\begin{array}{llll} \mathit{Predicate} & \Phi & ::= & \Phi \wedge p \mid p \\ \mathit{Condition} & p & ::= & a\ op\ v \\ \mathit{Operator} & op & ::= & \leq \mid < \mid = \mid > \mid \geq . \end{array}$$

We focus on equality in the context of `strings`. Evaluation of a subscription Φ on event e, written $\Phi(e)$ involves substituting the value v_i for a_i, for all $a_i \in e$. Obviously, satisfying a subscription ($\Phi = p_1 \wedge \cdots \wedge p_u$) requires satisfying each of its conditions ($\Phi(e) = \bigwedge_{r=1}^{u} p_r(e)$). Without loss of validity but for simplifying presentation and comparison with predating work, our subscriptions do not support *disjunctions*. Subscriptions are usually viewed to be in disjunctive normal form (DNF), where conjunctions are handled individually as subscriptions. Disjunctions can then be dealt with by matching events sequentially and memorizing the last event sent to a given process, avoiding re-sends in case multiple conjunctions in a DNF subscription match a same event.

2.3 Content-Based Matching

The content-based matching problem (cf. [7]) is formally defined as: given an event e and a set of subscriptions Θ, compute $\theta = \{\Phi \mid \Phi \in \Theta \wedge \Phi(e)\}$, where $\Phi(e)$ is short for $\Phi(e) = true$. Clearly $\theta \subseteq \Theta$. A naïve algorithm for content-based matching works as follows: given an event e, for each Φ in Θ, if $\Phi(e)$, add Φ to θ (which is initially set to $\emptyset$). The *matching complexity*, i.e., the number of evaluated constraints, of this algorithm is $O(K\,N)$, if $|\Theta| = N$ and there are K attributes in the system. The *time complexity* of this algorithm is also $O(K\,N)$. Note that a broker in a CPSN, which matches events to subscriptions has to perform $O(|\theta|)$ to multicast an event to the subscribers/downstream brokers.

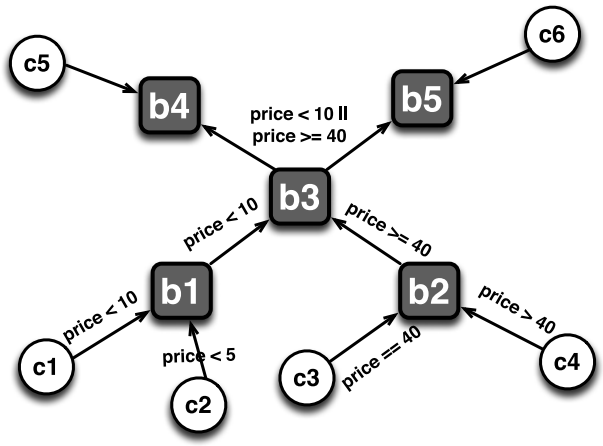

Fig. 1. Example of a CPSN.

2.4 Subscription Summarization and Routing

To avoid storing and evaluating all subscriptions at each broker, CPSNs perform *subscription summarization* based on covering relationships (subsumptions) among subscriptions. When a broker receives a subscription Φ from a subscriber or a downstream router, it forwards the subscription to an upstream router *unless* Φ is covered by a previous subscription Φ', that is, *unless* the set of events $\mathcal{E}$ satisfying Φ ($\mathcal{E} = \{\lceil \mid \Phi(\lceil)\}$) is a subset of the set of events $\mathcal{E}'$ satisfying Φ' ($\mathcal{E}' = \{\lceil \mid \Phi'(\lceil)\}$). The condition is expressed simply as $\mathcal{E} \subseteq \mathcal{E}'$, or put differently, $\forall e\, \Phi(e) \Rightarrow \Phi'(e)$. We say that predicate Φ' *covers (subsumes)* Φ, denoted by $\Phi \preceq \Phi'$, iff $\forall e, \Phi(e) \Rightarrow \Phi'(e)$. To facilitate the checking of subsumption relationships between predicates, these are typically stored in a *partially ordered set* (poset), ordered by $\preceq$.

Fig. 1 shows an example of a CPSN with six clients—four subscribers (c_1, c_2, c_3, c_4), two publishers (c_5, c_6) and five brokers (b_1, b_2, b_3, b_4, b_5). We focus on a single event type **`StockQuote`** with two attributes $a_1 =$ **`firm`** and $a_2 =$ **`price`** of **`string`** and **`float`** types respectively. We assume that all the clients subscribe to **`StockQuotes`** of the same **`firm`**, e.g., **`firm`** $=$ **`"IBM"`**. c_1 subscribes to **`StockQuote`** with $\Phi_1 =$ (**`price`** < 10). b_1 gets the subscription, stores it and propagates it to b_3. Then c_2 subscribes with $\Phi_2 =$ (**`price`** $<$ 5). b_1 gets this subscription, but does not forward it to b_3, as (**`price`** < 10) covers (**`price`** < 5), i.e., $\Phi_2 \preceq \Phi_1$. Fig. 2 illustrates subscription summarization throughout the overlay. Brokers b_1 and b_2 summarize subscriptions from $\{c_1, c_2\}$ and $\{c_3, c_4\}$ respectively, and b_3 further summarizes the summaries from b_1 and b_2.

3 Problem Description and State-of-the-Art

Beretta tackles several inefficiencies in content-based publish/subscribe systems. In this section, we describe each of them and why they arise in detail, while also placing them in their historical context, where relevant. Related work is further elaborated on in Appendix C, available in the online supplemental material.

3.1 Inefficient Event Matching

Early systems advocated mostly for *structural conformance* between events and subscriptions [8]. This leads to collapsing all subscriptions for all types of events into one single data-structure (e.g., poset) thus exacerbating bottlenecks. Some early CPS systems [1] reused the poset for matching events to subscriptions to avoid separate data-structures for subscription storage/summarization and event matching. This places a high load on brokers as all subscriptions end up being stored in the same data-structure with a worst-case depth equalling the number of nodes N in the poset. In turn, N is in the worst case equal to the total number of subscribers in the system though in practice it may be better. Assuming that there are K attributes in an event, and therefore up to $O(K)$ constraints in a subscription, the complexity of event matching is typically $O(K\,N)$, leading to low throughput and high end-to-end latency.

Most systems, even such described without event types, use types in practice, but only much later have the benefits of typing of events started to be exploited [9], [10]. Similarly, only little work exists on taking splitting a step further by systematically handling subscriptions *attribute-wise* (e.g., [11], [12]). Two prominent categories of smarter matching algorithms are those based on binary decision diagrams (BDDs—e.g. [13], [14]), and on Bloom filters (e.g. [15]). But, both these solutions have still $O(K\,N)$ matching complexity, unless assumptions are made, e.g., on the amount of common constraints in subscriptions. Bloom filters also require $O(2^B)$ space [15], where B is the number of bits used to store a subscription. Ineffective algorithms explain why, in spite of its more generic nature than the predating less dynamic *topic-based* multicast model, CPSNs

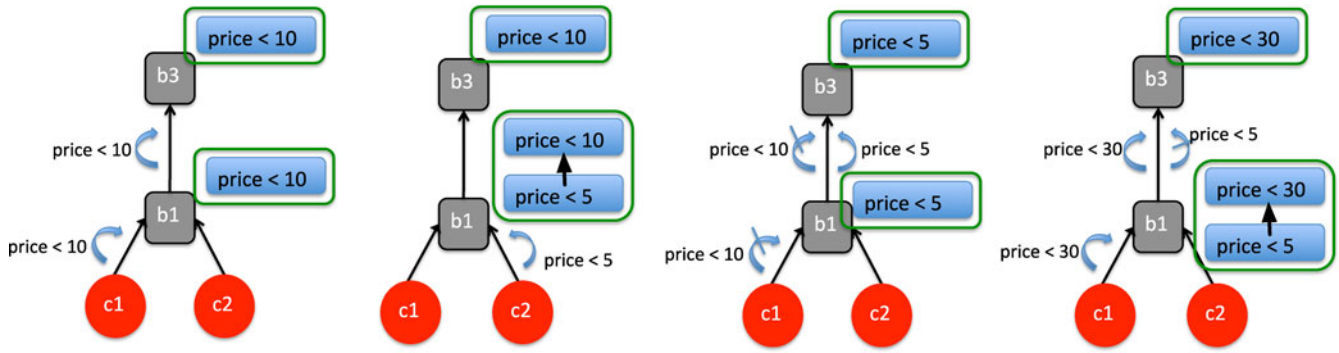

Fig. 2. Example of re-subscriptions, and cascading re-subscriptions. When c_1 unsubscribes from (**`price`** < 10), b_1 forwards (**`price`** < 5) to b_3. Then, when c_1 subscribes to (**`price`** < 30), b_1 reconstructs the poset. Since the LUB changes to (**`price`** < 30), b_1 unsubscribes from (**`price`** < 5) and subscribes to (**`price`** < 30).

like Siena [11], [16], HERMES [17], REBECA [18], Gryphon [7], PADRES [14], [19] and JEDI [20] have not dethroned topic-based multicast systems like ActiveMQ [21], ZeroMQ [22] and FioranoMQ [23]. Topic-based routing is easy to implement—routing a message only involves hashing a string (its topic) to forward the message to interested downstream nodes.

3.2 Inefficient Subscription Updates

Several emerging CPS applications such as high frequency trading (HFT) or mobile location-based services need to update their subscriptions at a high frequency [4], [5]. HFT hinges on rapid subscription updates, using several mathematical techniques to determine and change price thresholds during the "trading day". In location-aware applications (location-based advertising and social networks like `loopt` [24], etc.), a subscription is a function of the subscriber location such as a perimeter surrounding the location. Whenever the device moves, the subscription needs to change.

Re-subscriptions—the typical solution for subscription changes, where a new subscription is issued and the superseded one is canceled—have several limitations:

- *High computational cost*, due to operations at each affected broker on the poset storing subscriptions. Deletion and insertion of new subscriptions into the poset has a time complexity of $O(KN)$. Thus *two* $O(KN)$ *operations* have to be performed at a broker. As shown empirically [5], under high rates of subscription updates, the bulk of the computational resources of event brokers in a CPSN is spent on processing re-subscriptions rather than filtering and forwarding events. This leads to drastic drops in throughput and increased latency overall.
- *Cascading re-subscriptions* at upstream brokers: when the poset's least upper bound (LUB) changes as a result of un- or re-subscription, then the posets at upstream brokers also have to be updated. Hence, re-subscriptions can lead to *two* update *messages* being propagated per update the path from a subscriber to a publisher, as illustrated by Fig. 2.
- In the absence of synchronization of UNSUBSCRIBE and SUBSCRIBE messages during re-subscriptions, and of guarantees by the CPSN on the time taken for re-subscriptions to reach relevant brokers, the application must cater for duplicates if the old and new subscriptions overlap—the common case.

Parametric subscriptions [4], [5] are subscriptions referring to variables of subscribers. A typical subscription to IBM stock quotes with values below a specific threshold expressed through a CPS API as **`CPS.subscribe`**`("StockQuote", "firm == 'IBM' and price < 100.0")`. Updating the price threshold to \$150 would typically require a re-subscription. A correspond parametric subscriptions can be expressed as **`CPS.subscribe`**`("StockQuote", "firm == 'IBM' and price < " + ref threshold)`, where **`threshold`** is an internal subscriber variable. Instead of re-subscribing with the new price of \$150, the subscriber just sends the new value of **`threshold`**. Note that solutions for specific applications, such as *context-aware publish/subscribe* [25], [26] do not apply to HFT.

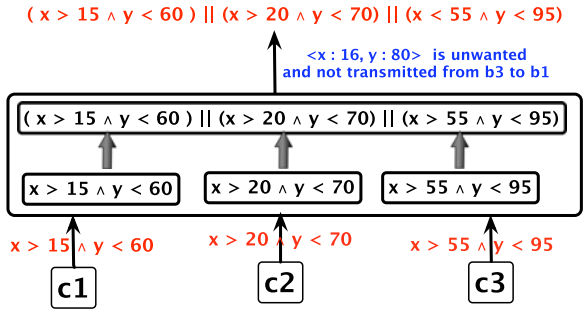

Fig. 3. Subscription summarization when none of the subscriptions subsume each other.

The decentralized implementation of parametric subscriptions described in [5], [27] still exhibits considerable limitations in application scenarios requiring fast, frequent subcription updates. First off, being "tacked on to" existing (poset-based) matching algorithms, it still incurs a high reaction time to updates. In addition parametric subscriptions do not handle structural updates, e.g., they can not handle updates from `"price < 100"` to `"price > 100 && price < 150"`. For scenarios like HFT which are governed by non-trivial algorithms additional flexibility is needed for structural subscription changes.

3.3 Large Subscription Summaries

The nodes of a poset can get arbitrarily complex if subscriptions do not subsume each other. Consider the scenario in Fig. 3. None of the subscriptions of clients c_1, c_2, or c_3 subsumes any others. Hence, the LUB is the *disjunction* of Φ_1,Φ_2 and Φ_3. In the worst case, if N clients connect to a broker, the LUB is the disjunction of N subscriptions, which propagate through the network towards publishers (e.g., b_1 forwards its LUB to b_2).

A beneficial "side-effect" observed with parametric subscriptions [5] is that many times (un-)subscriptions translate transitively in a CPSN to parameter updates. Together with the above-mentioned observations, this hints to an intriguing potential for CPSN systems that are fundamentally based on dynamic subscriptions.

Algorithm 1. Beretta client algorithm

```
1:  init
2:      b_j                                              {edge broker}
3:  to PUBLISH(e) of type τ do
4:      SEND(PUB, τ, e) to b_j
5:  to SUBSCRIBE(Φ) to type τ do
6:      χ ← NORMALIZE-TO-WELL-FORMED(Φ)
        {create interval predicates}
7:      SEND(SUB, τ, χ) to b_j
8:  to UNSUBSCRIBE from type τ do
9:      SEND(UNSUB, τ) to b_j
10: upon RECEIVE(PUB, τ, e) do
11:     if Φ(e) | Φ is subscription to τ then          {maybe changed}
12:         DELIVER(e)
13: upon change of value in δ_r for subscription to type τ do
14:     v̄_r ← new values for a_r in δ_r {δ_r = (a_r ∈ [v_1, v_2] or δ_r =
        (a_r ∈ {v_1, ..., v_n})}
15:     SEND(UPD, τ, ⟨r, v̄_r⟩) to b_j
```

4 Subscription Normalization

Beretta builds on *interval subscriptions*—subscriptions where the only boolean operator is $\in$. To avoid ambiguities, we refer to subscriptions Φ following the grammar introduced in Section 2.2 as *regular* subscriptions. Beretta normalizes regular subscriptions into *well-formed* interval subscriptions. An interval subscription is represented as an interval predicate χ with the following grammar:

$$\begin{array}{lllll} \textit{Predicate} & \chi & ::= & \chi \wedge \delta \mid \delta \\ \textit{Condition} & \delta & ::= & a \in [v, v] \mid a \in \{v, \ldots, v\}. \end{array}$$

A constraint $[v, v']$ represents an interval of permissible values for an attribute of an *ordered* type such as integers. Floating point values also fit this model. A constraint $\{v, \ldots, v\}$ represents a set of permissible values for an *enumerated* (discrete) type such as **`strings`**. A well-formed interval subscription is straightforwardly one which, for any given attribute, has *exactly one condition* on that attribute. While interval subscriptions improve the efficiency of content-based matching as we will show, they are as expressive as the subscriptions presented in Section 2.2 (a formal account of the expressivity of interval subscriptions is given in Appendix, available in the online supplemental material). For example, a predicate $\mathbf{x} > \mathtt{1000}$ where $\mathbf{x}$ is an integer attribute, can be expressed as $\mathbf{x} \in [\mathtt{1001}, \mathtt{MAX\_INT}]$. An equality (e.g., $\mathbf{x} = \mathtt{1000}$) can be modeled as a inclusion in a set with a single element. So the first step in subscription normalization is to convert constraints involving relational operators ($<, \leq, =, >, \geq$) into constraints involving the inclusion operator ($\in$) as we described informally above. This step yields an interval subscription which has at most one constraint on each attribute of an event type. It is important to note that a generic set inclusion with n elements corresponds to n disjoined conjunctions (i.e., subcsriptions) in regular syntax.

To normalize the interval subscription further and make it well-formed, i.e., to ensure that it contains exactly one constraint for each attribute, we add wildcard constraints to the subscription depending on the types of constraint-less attributes. An (implicit) wildcard constraint for an ordered type τ is $[MIN(\tau), MAX(\tau)]$, and $*$ for an enumerated type.

5 Beretta Algorithms

This section presents a novel divide and conquer matching algorithm for CPSNs called FASTINT, which is a key component of Beretta. FASTINT consists in a client-side component and a broker component which deals with event matching and routing.

5.1 Overview

Assuming events have up to K attributes and that the total number of subscriptions is N, many existing summarization and matching algorithms have $O(K\,N)$ matching complexity because they store "entire" subscriptions in a single data-structure. Then, since, subscriptions contain multiple constraints on different attributes, the chances of a subscription covering *entirely* another can become small, especially as the number of attributes increases. The key strategy adopted by FASTINT is therefore to *split* well-formed *interval subscriptions* by *event type* τ, and then *by attribute* a, thereby using one data-structure for each attribute of every event type (see Fig. 4).

Algorithm 2. FASTINT broker algorithm as executed by b_i. Common processing of data-structure modifications (new subscriptions, unsubscriptions, updates) are regrouped in PROPAGATE. $\oplus$ represents concatenation

```
 1: init
 2:   subs[]          {subscribed clients/brokers by event types τ}
 3:   pubs[]          {publishing clients/brokers event types τ}
 4:   vals[][][]      {current constraint values by event types τ, attribute, node id}
 5:   S[][]           {per-attribute data-structures by event types τ and attribute}

 6: upon RECEIVE(SUB, τ, ⊕_r v̄_r) from n_j do
 7:   updates ← ∅
 8:   subs[τ] ← subs[τ] ∪ {n_j}      {add node to subscribers of τ}
 9:   for all a_r ∈ τ do
10:     vals[τ][r][n_j] ← v̄_r
11:     v̄^0 ← LUB(S[τ][r])                                  {old LUB}
12:     INSERT(S[τ][r], v̄_r, n_j)
13:     v̄^v ← LUB(S[τ][r])                                  {new LUB}
14:     updates ← updates ∪ {⟨r_l, v̄^0, v̄^v⟩}
15:   PROPAGATE(τ, updates) end upon

16: upon RECEIVE(UNSUB,τ) from n_j do
17:   updates ← ∅
18:   subs[τ] ← subs[τ]\{n_j}
19:   for all a_r ∈ τ do
20:     v̄ ← vals[τ][r][n_j]                  {get vals for n_j's condition}
21:     vals[τ][r][n_j] ← ⊥
22:     v̄^0 ← LUB(S[τ][r])                                  {old LUB}
23:     DELETE(S[τ][r], v̄, n_j)
24:     v̄^v ← LUB(S[τ][r])                                  {new LUB}
25:     updates ← updates ∪ {⟨r_l, v̄^0, v̄^v⟩}
26:   PROPAGATE(τ, updates)

27: upon RECEIVE(UPD, τ, {⟨r_1, v̄_1⟩, ..., ⟨r_m, v̄_m⟩}) from n_j do
28:   updates ← ∅
29:   for all l ∈ 1..m do
30:     v̄^0 ← LUB(S[τ][r_l])                                {old LUB}
31:     if typeof(r_l, τ) = string then
32:       ⟨v̄_{l_1}, v̄_{l_2}⟩ | v̄_l = v̄_{l_1} • v̄_{l_2}
33:     DELETE(S[τ][r_l], v̄_{l_1}, n_j)     {del from hash map S[τ][∇↕]}
34:     INSERT(S[τ][r_l], v̄_{l_2}, n_j)     {add to hash map S[τ][∇↕]}
35:       vals[τ][r_l][n_j] ← vals[τ][r_l][n_j]\{v̄_{l_1}} ∪ {v̄_{l_2}}
36:     else
37:       UPDATE(S[τ][r_l], vals[τ][r_l][n_j], v̄_l, n_j) {update S[τ][∇↕]}
38:       vals[τ][r_l][n_j] ← v̄_l
39:     v̄^v ← LUB(S[τ][r_l])                                {new LUB}
40:   updates ← updates ∪ {⟨r_l, v̄^0, v̄^v⟩}
41:   PROPAGATE(τ, updates)

42: procedure PROPAGATE(τ, ⟨r_1, v̄_1^0, v̄_1^v⟩, ..., ⟨r_m, v̄_m^0, v̄_m^v⟩)
43:   updates ← ∅
44:   for all l ∈ 1..m do
45:     if v̄_l^v ≠ v_l^0 then          {LUB change⇒update upstream}
46:       if typeof(r_m, τ) = string then
47:         updates ← updates ∪ {⟨r_l, v̄_l^0\v̄_l^v • v̄_l^v\v̄_l^0⟩}
48:       else
49:         updates ← updates ∪ {⟨r_l, v̄_l^v⟩}
50:   if updates ≠ ∅ then
51:     SEND(UPD, τ, updates) to all b_k ∈ pubs[τ]
```

For presentation simplicity, in the following, we assume that 1) attributes are named uniquely in event types and are ordered in events (e.g., based on names), and that 2)

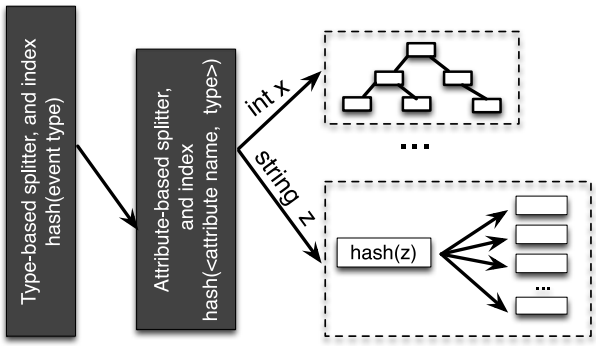

Fig. 4. Overview of FASTINT. It uses novel SV tree s for numeric attributes and hash maps for string attributes.

subscription identifiers are contiguous. $\overline{x}$ represents a sequence $x_1, \ldots, x_n$.

A broker b_j in a CPSN receives well-formed interval subscriptions from subscribers c_i connected to it as well as downstream brokers. Subscriptions on an event type with k attributes are of the form $\chi = \delta_1 \wedge \cdots \wedge \delta_k$. A client (see Algorithm 1) communicates with its edge broker b_j. When creating a normalized subscription χ (line 6), wildcard constraints are automatically added for unconstrained attributes. Upon receiving an event e, a client c_i verifies if $\chi(e)$ holds, as updates might have occurred in the meantime. Updates to a predicate δ_r trigger the sending of an update message to b_j.

5.2 Subscription Indexing

Next we focus on the FASTINT broker algorithm. As illustrated in Fig. 4 data-structures for storing constraints are accessed by a double index, namely on event type τ and attribute. In Algorithms 2 and 3, $\mathcal{S}$ represents this index. In addition the algorithm uses three associative maps $subs$, $pubs$ and $vals$. The former two store sets of subcribing clients or downstream brokers, and sets of publishing clients or upstream brokers, respectively, indexed by event type τ. For presentation simplicity operations on these data-structures are represented as assignments $\leftarrow$ and operations on the sets that they store. To perform subscription updates efficiently, $vals$ is used. $vals$ reflects the normalized nature of subscriptions, and their *inherent parameterization*: $vals[\tau][r][n_j]$ contains a client/downstream broker's (with id n_j) current bounds on the interval query or set inclusion constraint on attribute a_r of event type τ. Each well-formed subscription χ on τ namely contains exactly one constraint on any attribute a_r. The constraint either consists in upper and lower bounds (two values) or in a set of permissible values depending on the type $typeof(r, \tau)$ of the attribute a_r in τ. Either of these can represented by a sequence of values $\overline{v}$. In the absence of actual constraints, $MIN(\tau)$ *and* $MAX(\tau)$ are the values for an attribute of ordered type τ and $*$ the single value for an attribute of **`string`** type. Note that this model also captures *structural updates*: for example, an update from $[MIN(\tau), v]$ to $[v', MAX(\tau)]$ for an attribute a reflects the change of a constraint $a \leq v$ to a constraint $a \geq v'$.

5.3 SV Tree and Hash Maps

While $vals$ serves for fast updates, FASTINT stores constraints for actual matching in a *SV tree* for an a of an ordered type (see **`x`** in Fig. 5), and in a *hash map* for an attribute a of an enumerated type (see **`z`** in Fig. 5). The potential drawback of a matching algorithm based on the divide and conquer strategy is that when an event with values for k attributes $e = \{a_1 : v_1, \ldots, a_k : v_k\}$ is received, each of the corresponding k data-structures has to be queried and the results combined. If querying data-structure i produces $result_i$, the set of subscriptions $result$ matching e is the intersection $\cap_{i=1}^{k} result_i$ whose naïve computation takes $O(max_i(|result_i|))$ time. This is undesirable because $|result|$ in many cases will be smaller than any of the $|result_i|$.

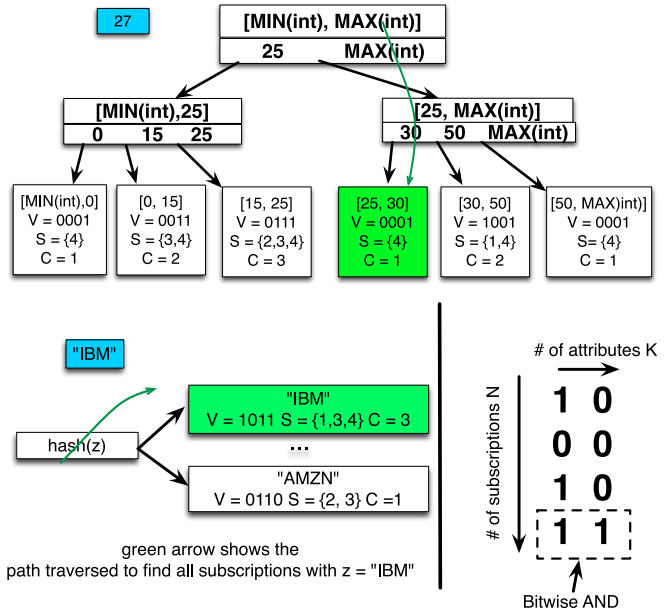

Fig. 5. Illustration of FASTINT. We assume subscriptions χ_1 = (**`x`** $\in [30, 50] \wedge$ **`z`** $\in$ {`"IBM"`}), χ_2 = (**`x`** $\in [15, 25] \wedge$ **`z`** $\in$ {`"AMZN"`}), χ_3 = **`x`** $\in [0, 25]$, χ_4 = **`z`** $\in$ {`"IBM"`}. Nodes with matching constraints are shaded green. Matched event is {**`x`**:27,**`z`**:`"IBM"`}.

To avoid this, we precompute three pieces of information for the relevant nodes in the attribute data-structures—S, C and V. S is the set of identifiers of subscriptions whose respective constraints on attribute a satisfy ("include" in the case of interval subscriptions) the value(s) represented by that specific node. C is simply $|S|$, and V is a bit vector of size N, indexed by the identifiers of subscriptions. Bit $V[i]$ is set to 1 iff $i \in S$. We use S, C and V for computing intersections efficiently.

When a is of the **`string`** type, an entry in the hash map for attribute constraint $a \in \{v\}$ contains a tuple $\langle S, C, V \rangle$ as outlined above with S being the set of identifiers of subscriptions with constraints on a including v.

A segment tree [28] is a data-structure for storing line segments or intervals (represented as axis-parallel line segments). In this paper, we introduce an SV tree , which is a segment tree optimized for queries on points (rather than intervals) and augmented with information such as $\langle S, C, V \rangle$ mentioned above for efficient intersection. Our SV tree is implemented as a B-tree with a fanout of f, i.e., each node in the B-tree has at least f and up to $2f - 1$ children. Assuming that there are n subscriptions to be processed, let the *canonical intervals* (i.e., intervals specified in the subscriptions) corresponding to an ordered attribute a be $[v_1, v_2], \ldots, [v_{2n-1}, v_{2n}]$. Like traditional segment tree

construction algorithms, FASTINT partitions the real line induced by the values v_i, i.e., FASTINT sorts the endpoints of the canonical intervals, obtaining $v'_1, \ldots, v'_{2n}$, and stores the following disjoint *elementary intervals* $[MIN(\tau), v'_i], [v'_1, v'_2], [v'_2, v'_3] \ldots [v'_{2n-1}, v'_{2n}], [v'_{2n}, MAX(\tau)]$ in the leaves of the segment tree. Each leaf node is also augmented with $\langle S, C, V \rangle$ just like hash map entries. An intermediate node contains simply the minimal interval covering all segments of its sub-nodes (either intermediate or leaf nodes). While matching an attribute of an event against the corresponding SV tree , matching occurs recursively one level at a time with at most one sub-node's interval matching at any level. After reaching a leaf node, $\langle S, C, V \rangle$ is returned. $\langle S, C, V \rangle$ is similarly stored for every key in a hash map and returned upon a corresponding query.

5.4 Basic Operations

Assuming that the hash map implementation uses a good hashing function, load factors less than 0.5, and chaining and rehashing, all elementary hash mapoperations have a time complexity of $O(1)$ [29]. In the case of a set of values $\overline{v} = v_1 \ldots v_l$ this time is multiplied by l. Note that for complexity comparisons we can consider $l = 1$, because $l > 1$ would require a *disjunction of l* conjunctions in a regular DNF subscription: $\bigvee_{k \in 1..l} a = v_1$. In the formal description of FASTINT in Algorithms 2 and 3, the standard operations on hash maps and SV tree s are represented in a uniform manner:

INSERT$(D, \overline{v}, id)$ creates of an association between $\overline{v}$ and id in data-structure D. In the case of a hash map, $\overline{v}$ represents a set of keys (values from the perspective of matching) for which id (a process identifier n_j) is added to the set of corresponding values. For an SV tree , $\overline{v} = v_1, v_2$ represents the upper and lower bounds of an interval which is of interest to id. In the case of a hash map, insertion of each key of $\overline{v}$ has a time complexity of $O(1)$ due to hashing. But, assuming that the hashmap contains n subscriptions, i.e. n keys, the vector V associated with each key is of size n. Hence updating n vectors has a time complexity of $O(n)$, which becomes the time complexity of INSERT$(D, \overline{v}, id)$. The same is true in the case of an SV tree. Inserting an interval $[v_1, v_2]$ into the SV tree only involves traversing $log\, n$ nodes of the tree to find the spots to insert v_1 and v_2, but up to n vectors have to be updated, with a time complexity of $O(n)$.

DELETE$(D, \overline{v}, id)$ removes the association between $\overline{v}$ and id from data-structure D. In the case of a hash map, id is removed for all keys $\overline{v}$. For an SV tree , the interval denoted by $\overline{v} = v_1, v_2$ associated with id is removed. In the case of a hash map, deletion of each key of $\overline{v}$ has a time complexity of $O(1)$ due to hashing. But, assuming that the hashmap contains n subscriptions, i.e. n keys, the vector V associated with each key is of size n. Hence updating n vectors has a time complexity of $O(n)$, which becomes the time complexity of DELETE$(D, \overline{v}, id)$. In the case of an SV tree, deleting an interval $[v_1, v_2]$ from the SV tree only involves traversing $log\, n$ nodes of the tree to find the tree nodes containing v_1 and v_2, but up to n vectors have to be updated, with a time complexity of $O(n)$.

RETRIEVE(D, v) queries data-structure D for all ids associated with value v. For a hash map, the return set includes all values stored for key v; for an SV tree the query returns the ids of all intervals which contain v.

LUB(D) returns the "covering constraint" for data-structure D. For a hash map H this includes all keys, unless the key set includes the wildcard $*$, in which case only that value is needed. For an SV tree , the covering interval stored in the root node is returned.

UPDATE$(D, \overline{v}, \overline{v'}, id)$ changes the association between $\overline{v}$ and id in data-structure D to $\overline{v'}$ for id. This can be naïvely implemented by DELETE and INSERT. This operation is only used for $\mathcal{S}$, where both bounds for a constraint are replaced atomically for simplicity even if only one has changed; in the case of hash maps used for string attributes, updates focus on only adding new elements and removing obsolete ones.

Handling of new subscriptions and unsubscriptions follow similar structures. Changes are made to all relevant data-structures. For any such data-structure D, the LUB before update is stored. After all updates have been performed, the set of previous LUBs and respective new LUBs are passed to PROPAGATE which determines all *actual* changes. In the case of interval queries, it suffices that one of the bounds has changed. In the case of set inclusions the obsolete elements $\overline{v_1}$ and new elements $\overline{v_2}$ are determined; the two sets are concatenated by inserting a special token value $\bullet$: $\overline{v_1} \bullet \overline{v_2}$. When handling a corresponding update message UPD, the two sets are removed from and added to the appropriate hash map respectively. After handling such an update, any transitive updates are similarly handled via PROPAGATE. This illustrates the fundamental nature of updates in Beretta: except when a broker b_1 connects to another broker b_2 for the first time or disconnects from b_2, all changes in broker connections take place via update messages.

5.5 Matching

When an event e with k attributes is received (Lines 52-65 in Algorithm 3), at most k corresponding data-structures are "evaluated" to determine subscriptions that match e. If an attribute a_i is of an ordered type, then $\mathcal{S}$ is queried in $O(1)$ to retrieve the corresponding SV tree T (Line 59). If e's a_i value is v_i, then T is searched with v_i to determine the set of intervals (and thereby the subscriptions $S[i]$) that match v_i. If a_i is of **`string`** type, $\mathcal{S}$ is queried in $O(1)$ to retrieve the corresponding hash map H. The subscriptions $S[i]$ that match v_i are obtained by querying H for v_i. GET on hash maps and SV tree s returns a tuple with three elements, $\langle S, C, V \rangle$.

In Algorithm 3, we use a variable Vec to refer to a precomputed bit matrix whose dimensions are $k \times N$. When an event $e = \{a_1 : v_1, \ldots, a_l : v_k\}$ is received, $Vec[i]$ points to the vector encoding subscriptions matching v_i. Since every subscription matches an unconstrained attribute, $Vec[i]$ points to a bit vector whose bits are 1 if a_i is unconstrained. Since $result \subseteq min_{i=1}^{k}(result_i)$, Algorithm 3 keeps track of the index (min_result_index) of the smallest $result_i$, and the set of subscriptions contained in it. To decide whether e matches any of the subscriptions in $result_{min\_result\_index}$, a bitwise AND operation is performed on the k bits of $Vec[j]$ for all $j \in Vec[min\_result\_index]$. This yields a complexity of $O(K \log N + K\, min_{i=1}^{K}(|result_i|))$, because $k \leq K$.

5.6 Summarization Approximation

When sending a summary to a parent, a broker *approximates the summarization*. Remember that a broker manages a separate data-structure $\mathcal{S}[\tau][\nabla]$ for each attribute a_r of a given event type T. The LUB of such a data-structure covers all conditions on the respective attribute for all subscriptions known to the broker. A broker summary for a given event type simply consists in the conjunction of these LUBs. In Algorithm 2 this conjunction arises implicitly, as update messages (line 51) contain the LUBs for the relevant types and attributes.

Algorithm 3. FASTINT – Matching

```
52: upon RECEIVE(PUB, τ, e = {a_1 : v_1, ..., a_k : v_k}) from n_j
53:   Subs ←new 2-D array of size k × |subs[τ]|
54:   Count ←new 1-D array of size k
55:   Vec ←new 2-D array of size k × |subs[τ]|
56:   min_result_count ← |subs[τ]|
57:   min_result ← subs[τ]
58:   for i = 1..k do
59:     ⟨Subs[i], Count[i], Vec[i]⟩ ← RETRIEVE(S[τ][a_i], v_i)
60:     if Count[i] ≤ min_result_count then
61:       min_result_count ← Count[i]
62:       min_result_index ← i
63:   for all index ∈ Subs[min_result_index] do
64:     if BITWISE−AND(Vec[index][1, ..., k]) = 1) then
65:       result ← result ∪ {index}
66:   for all n_k ∈ result\{n_j} do
67:     SEND(PUB, τ, e) to n_k
```

This approximation can be disabled in Beretta, e.g., for sets of summarized subscriptions below a threshold size, and a disjunction (created like in traditional approaches) sent to parent brokers instead. In this scenario, a subscription sent by a broker b_i to its parent broker b_j consists in fact in a set of (logically disjoined) normalized subscriptions, say of size u, necessary to cover all subscriptions of b_j. Upon a change at b_i to its data-structures, e.g., induced by the addition, removal, or update of a subscription, there are two scenarios: (a) the set of disjoined subscriptions in this root subscription remains the same, or (b) it changes. In the former case, if the values of certain variables have changed only the updates need to be transmitted to b_j. In the latter case we can further distinguish three cases, based on whether the number of disjoined subscriptions in the root (b.1) grows to v, (b.2) remains u (some subscriptions in the disjunction may have changed though), or (b.3) shrinks to v. In all cases, we identify $min(u, v)$ subscriptions—preferably such that were in the previous set already—with the previous ones, and send corresponding variable updates where necessary. If we have more subscriptions now ($v > u$) or fewer ones ($v < u$) then b_i additionally informs b_j of the new ones to be added or of those to be removed. The latter actions are handled differently from regular new subscriptions on unsubscriptions, as brokers take note of multiple subscriptions that are logically linked to a same peer or client to avoid multiple transmissions of the same event.

5.7 Initialization

Initialization on a given node with a set of subscribers (downstream nodes) is simple and thus only outlined informally here. After identifying the set of attribute/type pairs across all subscriptions an empty SV tree is created for every attribute of ordered type encountered in some subscription, and an empty hash map is likewise created for any attribute of enumerated type. These are indexed in $\mathcal{S}$ by their respective attribute/type pairs.

Then the data-structures are populated from subscriptions, one-by-one. In contrast to the handling of a new subscription in Algorithm 2, (any) new LUBs are only determined once all subscriptions have been added.

6 Summary of Evaluation

In this section we present a summary of our empirical evaluation. Please refer to Appendix A, available in the online supplemental material, for a detailed description. We use two benchmarks for empirical evaluation, based on: (1) Marketcetera algorithmic trading, and (2) Highway Traffic Management (HTM). We compare Beretta against (1) Beretta-NoApprox, which disables summarization approximation in Beretta, (2) EV, a CPSN which uses the Rete algorithm [30] for event matching, posets for subscription summarization and supports parametric subscriptions [5], (3) EV-Resub, which is the same as EV, except for the use of re-subscriptions for subscription updates, (4) the seminal Siena [1] CPSN, and (5) Apache ActiveMQ [21]. We use four main metrics to evaluate the performance—throughput, event propagation latency (EPL), delay in propagating updates from subscribers to CPSNs, and the percentage of stale events that are delivered to a subscriber during subscription updates. In addition we measure the main memory (RAM) usage of the various CPSNs as well as bandwidth usage.

Tables 1 and 2 present an overview of our results. Details of the topology used, benchmarks, systems being evaluated, configurations and a detailed analysis of the results, along with graphical illustrations of the same are presented in

TABLE 1
HTM Benchmark

	Increase in Throughput	Decrease in EPL	Decrease in Delay	Decrease in Spurious events	Parameter varied
Beretta vs.	1.14 – 1.28×	1.12 – 1.19×	1.09 – 1.21×	1.38 – 1.55×	# subscribers
Beretta-NoApprox	1.19 – 1.46 ×	1.05 – 1.23×	1.13 – 1.31 ×	1.07 – 3.17 ×	upd. frequency
Beretta vs.	1.58 – 1.85×	2.16 – 2.44×	1.07–1.29×	2.24 – 4.48×	# subscribers
EV	1.67 – 1.97	2.16 – 2.43 ×	1.95 – 2.44 ×	1.61 – 3.57 ×	upd. frequency
Beretta vs.	3.08 – 5.98×	3.12 – 4.4×	2.05 – 5.7×	3.38 – 7.43×	# subscribers
ActiveMQ	3.2 – 7.2 ×	3.14 – 4.7 ×	2.4 – 6.5 ×	2.13 – 6.21 ×	upd. frequency

TABLE 2
Marketcetera Benchmark

	Increase in Throughput	Decrease in EPL	Decrease in Delay	Decrease in Spurious events	Parameter varied
Beretta vs. Beretta-NoApprox	1.15 – 1.29×	1.14 – 1.17×	1.12 – 1.19×	1.08 – 1.35×	# subscribers
	1.13 – 1.25 ×	1.08 – 1.22×	1.15 – 1.58 ×	1.04 – 1.35 ×	upd. frequency
Beretta vs. EV	2.36 – 2.47×	1.74 – 2.48×	1.26–1.81×	1.83 – 3.47×	# subscribers
	1.67 – 1.56	2.4 – 3 ×	1.98 – 4.29 ×	1.68 – 3.4 ×	upd. frequency
Beretta vs. ActiveMQ	5.12 – 6.67 ×	2.88 – 3.6×	3.52 – 7.94×	8.45 – 15.52×	# subscribers
	5.23 – 7.91 ×	3 – 4.06 ×	2.07 – 4.49 ×	8.18 – 14.56 ×	upd. frequency

Appendix A, available in the online supplemental material. In the case of the Marketcetera benchmark, we observe that the use of summarization approximation increases the throughput of Beretta by up to 1.29× with an increase in the number of subscribers and by up to 1.25× with an increase in update frequency, both of which are non-trivial. EPL, delay and the percentage of spurious events are lower in Beretta by up to 1.22×, 1.58× and 1.35× respectively. With respect to Siena, the improvements in performance are very substantial, by up to more than 100× in the case of throughput. This is because of (1) the use of posets for both summarization and matching, (2) inefficiencies due to not aggressively using types, and (3) the use of re-subscriptions for handling subscription changes. Similar trends are also seen in the case of EPL and delay (more than 5× lower) as well as the percentage of spurious events (more than 20× lower). As the differences in performance are easily observed in the case of Siena, we do not quantify them in more detail in Tables 1 and 2.

In the case of Marketctera, when compared to EV, the performance benefits are much more substantial—up to 2.47× higher throughput, and up to 3×, 4.29× and 3.47× lower EPL, delay and spurious events respectively. Since EV already supports re-subscriptions, these benefits are purely due to the efficiency of FASTINT over the Rete matching algorithm due to type-based splitting, indexing and efficient computation of intersections. In Appendix A, available in the online supplemental material, we also observe a comparison of EV versus EV-Resub as well as Beretta versus EV-Resub—EV significantly outperforms EV-Resub which characterizes the performance benefits of parametric subscriptions. We also observe that performance benefits of Beretta are much more substantial when compared to ActiveMQ—up to 7.91× higher throughput as well as up to 4.06×, 7.94× and 15.5× lower EPL, delay and spurious events respectively. This is both because of the absence of a summarization-based overlay network, as well as inefficient matching.

As hinted to by Table 1, performance benefits similar to Marketctera are also seen in the case of HTM. The trends are very similar, even though the two benchmarks having different numbers of publishers, subscribers and event types. The causes of these performance trends between the different systems also remain the same, as contained in the discussion of Marketcetera above.

7 Conclusions

This paper has presented Beretta, a new content-based publish/subscribe system. Beretta introduces types, and a simplified predicate grammar which allows it to normalize all subscriptions, leading to implicit parameterization and thus fast, localized, *subscription updates*.

Many algorithms for CPS matching have been presented in literature, most without analysis of their complexity. FASTINT is to the best of our knowledge the first algorithm to achieve matching of events with a number of constraint matches logarithmic in the total number of subscriptions. FASTINT also lends itself well to *subscription summarization* [31], by allowing for the most generic constraints on individual attributes to be extracted easily from respective data structures and conjoined to form an over-approximized summary of size $O(K)$. For precise summarization, a poset can also be used [32].

This paper, in addition to presenting FASTINT, also presents a detailed empirical evaluation of Beretta by varying several parameters—number of event types, update rate at subscribers, number of subscribers as well as selectivity. In all cases, Beretta outperforms several other state-of-the-art publish/subscribe systems.

Acknowledgments

Research partially funded by US National Science Foundation (NSF) (grants 0644013 and 0834529) and by DARPA (grant N11AP20014).

References

[1] A. Carzaniga and D. S. Rosenblum and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, pp. 332–383, Aug. 2001.

[2] N. Fotiou, D. Trossen, and G. C. Polyzos, "Illustrating a publish-subscribe internet architecture," *Telecommun. Syst.*, vol. 51, no. 4, pp. 233–245, 2012.

[3] *Project CCNx*. (2014). Palo Alto Research Center, Palo Alto, CA, USA [Online]. Available: http://www.ccnx.org/

[4] Y. Huang and H. Garcia-Molina, "Parameterized subscriptions in publish/subscribe systems," *Data Knowl. Eng.*, vol. 60, pp. 435–450, Mar. 2007.

[5] K.R. Jayaram and C. Jayalath and P. Eugster, "Parametric content-based publish/subscribe," *ACM Trans. Comput. Syst.*, vol. 31, May 2013, http://dl.acm.org/citation.cfm?doid=2465346.2465347

[6] K. R. Jayaram and P. Eugster, "Split and subsume: Subscription normalization for effective content-based publish/subscribe messaging," in *Proc. 31st Int. Conf. Distrib. Comput. Syst.*, 2011, pp. 824–835.

[7] M. K. Aguilera and R. E. Strom and D. C. Sturman and M. Astley and T. D. Chandra, "Matching events in a content-based subscription system," in *Proc. 18th Annu. ACM Symp. Principles Distrib. Comput.*, 199, pp. 53–61.

[8] B. Oki and M. Pfluegl and A. Siegel and D. Skeen, "The information bus: An architecture for extensible distributed systems," in *Proc. 14th ACM Symp. Operating Syst. Principles*, 1993, pp. 58–68.

[9] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic support for distributed applications," *IEEE Comput.*, vol. 33, no. 3, pp. 68–76, Mar. 2000.
[10] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of XML documents with XPath expressions," *VLDB J.*, vol. 11, pp. 354–379, 2002.
[11] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *Proc. SIGCOMM*, 2003, pp. 163–174.
[12] P. Triantafillou and A. Economides, "Subscription summarization: A new paradigm for efficient publish/subscribe systems," in *Proc. 24th Int. Conf. Distrib. Comput. Syst.*, 2004, pp. 562–571.
[13] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," in *Proc. 23rd Int. Conf. Softw. Eng.*, 2001, pp. 443–452.
[14] G. Li, S. Hou, and H.-A. Jacobsen, "A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams," in *Proc. 25th Int. Conf. Distrib. Comput. Syst.*, 2005, pp. 447–457.
[15] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in *Proc. 2nd Int. Conf. Distrib. Event-Based Syst.*, 2008, pp. 71–81.
[16] A. Carzaniga, M. J. Rutherford, and A. L. Wolf, "A routing scheme for content-based networking," in *Proc. INFOCOM*, 2004, pp. 918–928.
[17] P. Pietzuch, B. Shand, and J. Bacon, "A framework for event composition in distributed systems," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2003, pp. 62–82.
[18] L. Fiege, F. Gärtner, O. Kasten, and A. Zeidler, "Supporting mobility in content-based publish/subscribe middleware," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2003, pp. 103–122.
[19] G. Li, V. Muthusamy, and H.-A. Jacobsen, "Adaptive content-based routing in general overlay topologies," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2008, pp. 1–21.
[20] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Trans. Softw. Eng.*, vol. 27, no. 9, pp. 827–850, Sep. 2001.
[21] Apache Software Foundation. (2010). *ActiveMQ* [Online]. Available: http://activemq.apache.org/
[22] *ZeroMQ.*, ZeroMQ Inc. (2010) [Online]. Available: http://www.zeromq.org
[23] Fiorano Software Inc. (2010). *FioranoMQ JMS Server* [Online]. Available: http://www.fiorano.com/products/Enterprise-Messaging/JMS/Java-Message-Service/FioranoMQ.php
[24] CrunchBase (2013). Loopt [Online]. Available: http://www.crunchbase.com/company/loopt
[25] G. Cugola, A. Margara, and M. Migliavacca, "Context-aware publish-subscribe: Model, implementation, and evaluation," in *Proc. IEEE Symp. Comput. Commun.*, 2009, pp. 75–881.
[26] R. Meier and V. Cahill, "On event-based middleware for location-aware mobile applications," *IEEE Trans. Softw. Eng.*, vol. 36, no. 3, pp. 409–430, May/Jun. 2010.
[27] K. R. Jayaram, C. Jayalath, and P. Eugster, "Parametric subscriptions for content-based publish/subscribe networks," in *Proc. ACM/IFIP/USENIX 11th Int. Conf. Middleware*, 2010, pp. 128–147.
[28] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. New York, NY, USA: Springer-Verlag, 2008.
[29] T. H. Cormen, R. L. Rivest, C. Leiserson, and C. H. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2010.
[30] C. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," in *Expert Systems*. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990, pp. 324–341.
[31] P. Triantafillou and A. A. Economides, "Subscription summarization: A new paradigm for efficient publish/subscribe systems," in *Proc. 24th Int. Conf. Distrib. Comput. Syst.*, 2004, pp. 562–571.
[32] A. Carzaniga, D. Rosenblum, and A. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *Proc. 19th Annu. ACM Symp. Principles Distrib. Comput.*, 2000, pp. 219–227.
[33] R. Agostino. (2011). *The Marketcetera Trading Platform* [Online]. Available: www.marketcetera.org
[34] Activ Financial. (2012). *ACTIV Data Feed* [Online]. Available: http://www.activfinancial.com/
[35] FIX Protocol Limited. (2012). *The Financ. Inf. eXchange (FIX) Protocol* [Online]. Available: http://www.fixprotocol.org/
[36] NYSE Euronext. (2012). *NYSE Euronext Press Release* [Online]. Available: http://www.nyse.com/press/1245924443893.html
[37] S. Schneider, "DDS and distributed data-centric embedded systems," *Dr. Dobb's J.* [Online]. Available. http://www.drdobbs.com/embedded-systems/196601852
[38] D. Barnett, "Publish-subscribe model connects Tokyo highways," in *Ind. Embedded Syst.*, Mar. 2007, http://industrial.embedded-computing.com/article-id/?2072=
[39] T. Sivaharan, G. S. Blair, and G. Coulson, "GREEN: A configurable and re-configurable publish-subscribe middleware for pervasive computing," in *Proc. Confederated Int. Conf. Move Meaningful Internet Syst.*, 2005, vol. 3760, pp. 732–749.
[40] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "SCRIBE: The design of a large-scale event notification infrastructure," in *Netw. Group Commun. 2001*.
[41] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proc. 11th Int. Workshop Netw. Operating Syst. Support Digital Audio Video*, 2001, pp. 11–20.
[42] M. Sadoghi and H.-A. Jacobsen, "BE-Tree: An index structure to efficiently match boolean expressions over high-dimensional discrete space," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 637–648.
[43] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in *Proc. 2nd Int. Conf. Distrib. Event-Based Syst.*, 2008, pp. 71–81.
[44] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei, "A scalable and elastic publish/subscribe service," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1254–1265.
[45] Y-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. Wang, "Subscription partitioning and routing in content-based publish/subscribe systems," presented at the 16th Int. Symp. Distributed Computing Systems, Toulouse, France, 2002.

K.R. Jayaram received the MS and PhD degrees from Purdue University. He is a research staff member with IBM Research at the Thomas J. Watson Research Center in Yorktown Heights, NY. Until recently, he was a postdoctoral researcher at HP Labs in Palo Alto, CA. He is interested in distributed systems and programming languages.

Weihang Wang is currently working toward the PhD degree in computer science at Purdue University. She is interested in distributed algorithms and data center systems.

Patrick Eugster received the MS and PhD degrees from EPFL. He is an associate professor in computer science at Purdue University interested in distributed systems, algorithms, and programming. He received the National Science Foundation (NSF) CAREER award in 2007. He is a member of US Defense Advanced Research Projects Agency (DARPA)'s 2011 computer science study group, and received a ERC Consolidator award in 2013.