2011 31st International Conference on Distributed Computing Systems

Split and Subsume*

Subscription Normalization for Effective Content-based Messaging

K. R. Jayaram and Patrick Eugster
Department of Computer Science, Purdue University
{jayaram, peugster}@cs.purdue.edu

***Abstract*—Content-based publish/subscribe networks (CPSNs) scale to large numbers of publishers and subscribers by having brokers summarize subscriptions from subscribers and downstream brokers based on coverage relationships ("subsumption") between subscriptions. A broker forwards the summary to brokers which are upstream on the routes to the publishers. Current summarization and event processing mechanisms induce heavy event processing load on brokers, leading to low event throughput and high latency and further sharp performance degradation under high rates of churn, i.e., addition, deletion, or modification of subscriptions.**

This paper describes Beretta, a novel CPSN that leverages a simple model of typed events, enabling a succinct and uniform normalized representation of subscriptions. This in turn supports highly effective subsumption and attribute-wise split filtering with matching complexity logarithmic in the number of subscriptions, and enables the systematic introduction of parameters into subscriptions to support both parametric and structural updates. We empirically demonstrate that our techniques significantly improve throughput and latency of event propagation and reduce response times to subscription updates.

***Keywords*—subscription; summarization; subsumption; normalization; content-based; messaging;**

I. Introduction

Designing distributed applications following a *publish/subscribe* model, i.e., as components interacting anonymously by publishing *events* and consuming events subscribed to, is a popular approach. Such a design limits coupling of components (supporting scalability in terms of participating components), and allows for communication to be streamlined through a dedicated middleware system thus avoiding redundant traffic (supporting scalability in terms of communicated events).

A. Content-based Publish/Subscribe (CPS)

In content-based publish/subscribe (CPS), the most expressive publish/subscribe variant, a subscription is a predicate on the data attributes in an event. CPS middleware systems typically construct *content-based publish/subscribe networks* (CPSNs) to connect publishers with the appropriate subscribers, performing content-based routing and filtering in a decentralized manner over intermediate *broker* nodes. Examples of CPS middleware are Siena [1][2], HERMES [3], REBECA [4], Gryphon [5], PADRES [6][7],GREEN [8], XSiena [9] and JEDI [10]. The current standard approach views event content as sets of *attribute-value* pairs, and subscriptions as logical combinations of elementary predicates of the form $attr\ op\ value$ where op is an operator for event attribute $attr$.

To avoid storing and evaluating all subscriptions at each broker, CPSNs "summarize" subscriptions based on similarities among them. *Subscription subsumption* [2][6] is a popular technique used to summarize subscriptions. When a broker receives a subscription predicate Φ from a subscriber or a downstream broker, it forwards the subscription to an upstream broker *unless* Φ is *subsumed (or covered)* by a previous subscription Φ' ($\Phi \preceq \Phi'$). We say that $\Phi \preceq \Phi'$ iff $\forall e\ , \Phi(e) \Rightarrow \Phi'(e)$. To facilitate addition and removal of predicates, these are stored as a *partially ordered set* (poset), ordered through $\preceq$.

Another popular, *divide and conquer*-style, summarization technique was introduced by Triantafillou and Economides [11]. Instead of handling whole subscriptions, this technique splits subscriptions into attributes, and computes per-attribute summaries efficiently using arrays.

B. Current Bottlenecks in CPSNs

Subscription summarization significantly reduces the event processing load at each broker, when compared to approaches like static broadcast groups. But, if a broker receives n subscriptions for a given type of event with k attributes, several operations performed at the broker for subscription storage, management and event forwarding exhibit $O(k\,n)^1$ complexity. If posets, implemented as $d - ary$ max heaps are used instead of arrays to summarize subscriptions, the complexity of subscription addition can be brought to $O(k\,log_d\,n)$, but the complexities of unsubscription and event forwarding remain $O(k\,n)$. Several other used data-structures such as binary decision diagrams (BDDs) have not improved this bound (see both [6] and [12] for BDDs).

C. Dynamic Content-based Publish/Subscribe

Another bottleneck in current CPSNs arises from new application demands. Emerging applications like high frequency algorithmic trading (HFT) which accounts for 73% of all US trading volume (2009 figures [13], [14]) or location-sensitive applications with mobile clients (e.g., mobile social networking, vehicular networks) inherently require subscription *updates*, as stock values (and thus thresholds) as well as locations

*Research partially funded by US NSF (grants 0644013 and 0834529).

^{1}Some authors ignore k, because k is usually small; it is hard to imagine events with more than 100 attributes.

of mobile devices (and thus zones of interest) change. *Re-subscriptions* consisting in issuing a new subscription and subsequently canceling the outdated one were perfectly suitable a decade ago but tend to bog down brokers under high *churn* (subscription addition, removal, *and update*) rates of modern-day applications by affecting the very data-structures that are used for event *forwarding*. *Parametric subscriptions* [15] [16], which allow for variables in subscriptions, have proven to be an effective technique for subscription updates but these only address the simpler part of the problem – *structural* subscription updates such as addition or removal of elementary predicates are still not possible through parametric subscriptions.

D. Contributions

This paper presents Beretta, a novel CPSN which sustains high throughput and low event-propagation latencies under high churn, whilst scaling to thousands of event types, publishers, and subscribers. More precisely, our contributions are:

- We propose a succinct and uniform model of events and subscriptions. Our model leverages *strong event typing* and represents all subscriptions in a *normalized form* as combinations of *value intervals* and *set inclusions* without compromising on expressiveness. 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.
- We describe a novel highly efficient broker algorithm, which reduces the number of elementary predicates evaluated for matching an event from $O(k\,n)$ to $O(k\,log\,n)$, with n the number of subscriptions for the event type and k the number of its attributes. In short, our algorithm relies on a divide and conquer strategy which we call "split and subsume". Event types and normalization are exploited to *split* subscriptions into predicates on *individual event types and attributes* and to efficiently regroup these in *interval trees* and *hash maps*. Normalization also supports the *systematic* introduction of variables into predicates and subscription summarization *approximation* thus accelerating updates in the set of subscriptions (addition, removal, parametric and structural updates) handled by a broker.
- We empirically evaluate Beretta, first through two case studies based on algorithmic trading and traffic management respectively, and then through scalability studies by workload generation. Our evaluations demonstrate very promising performance.

E. Roadmap

Section II overviews the state of the art in CPSNs. Section III describes the design of Beretta. Section IV presents our empirical evaluation. Section V presents related work. Section VI draws final conclusions.

II. Background on CPSNs

In this section, we recall relevant notions and pinpoint bottlenecks for throughput and latency in current CPSNs.

A. Notions and Principles

We assume that CPSNs use 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 publishers and subscribers.

As pointed out by [2], like IP Multicast [17], CPS 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, and (b) *upstream evaluation* where unwanted events are filtered away as close as possible to the publisher to avoid wasting bandwidth. An *advertisement* defines the types of events produced by a publisher. When a CPSN does not use advertisements, the path of an event from a publisher to a subscriber is determined only by subscriptions. Subscriptions then have to be routed from a subscriber to every other publisher in the CPSN, essentially forming a spanning tree from the subscriber to every publisher attached to the CPSN. When an event is published, it is routed along this tree from the publisher to the subscriber using the reverse path forwarding. When advertisements are used, a subscription for an event of type τ only has to be routed to those publishers that generated an advertisement of type τ.

B. Events and Subscriptions

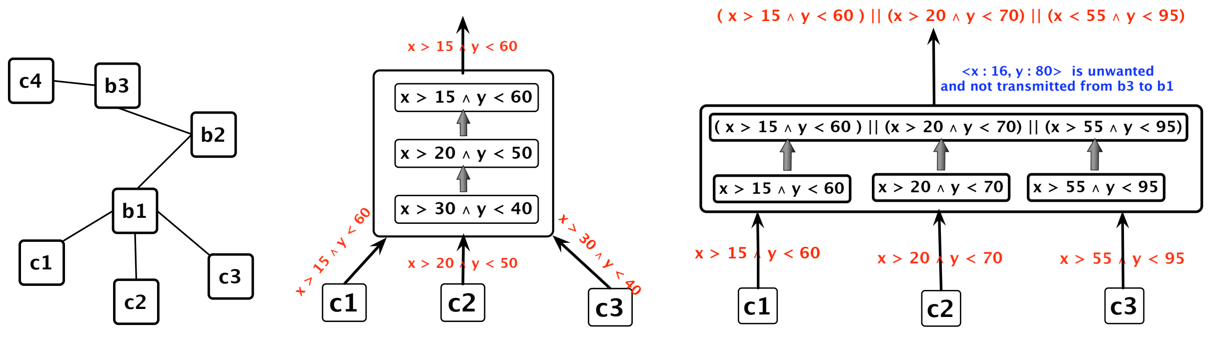
An event e can be viewed as a set of attribute-value pairs $\{a_1{:}v_1,\ldots,a_n{:}v_n\}$ [18], [5], [2]. A subscription is usually represented as a predicate Φ in disjunctive normal form with a grammar such as the following:

$$\begin{array}{llll} \mathit{Predicate} & \Phi & ::= & \Phi \lor \Psi \mid \Psi \\ \mathit{Conjunction} & \Psi & ::= & \Psi \land \Theta \mid \Theta \\ \mathit{Condition} & \Theta & ::= & a\ op\ v \\ \mathit{Operator} & op & ::= & \leq \mid < \mid = \mid > \mid \geq \end{array}$$

To decide on the routing of an event $e = \{a_1 : v_1, \ldots, a_n : v_n\}$, predicate Φ is evaluated on e by substituting v_i for a_i. This evaluation, written as $\Phi(e)$, yields a boolean value. As mentioned earlier, we say that predicate Φ' *covers (subsumes)* Φ, denoted by $\Phi \preceq \Phi'$, iff $\forall e, \Phi(e) \Rightarrow \Phi'(e)$. Disjoined conjunctions are usually handled internally just like separate subscriptions, with duplicate elimination. Therefore, for simplicity, we only consider predicates consisting of a single conjunction.

C. Subscription Summarization Through Subsumption

Figure 1a shows an example of a CPSN with four clients – three subscribers (c_1, c_2, c_3), one publisher (c_4) and three brokers (b_1, b_2, b_3). Consider a single event with two attributes `x` and `y` of integer type. Figure 1b shows a part of the CPSN, and how subscriptions propagate. c_1 subscribes to e with the predicate Φ_1 = `x>15∧y<60`. b_1 gets the subscription, stores it and propagates it to b_2. Then c_2 subscribes with predicate Φ_2=`x>20∧y<50`. b_1 gets this subscription, but does not forward it to b_2, because `x>15∧y<60` covers `x>30∧y<40` ($\Phi_2 \preceq \Phi_1$). Similarly, `x>30∧y<40` is received from c_3, and


c4
b3
b2
b1
c1
c2
c3
x > 15 ∧ y < 60
x > 15 ∧ y < 60
x > 20 ∧ y < 50
x > 30 ∧ y < 40
x > 15 ∧ y < 60
x > 20 ∧ y < 50
x > 30 ∧ y < 40
(x > 15 ∧ y < 60) || (x > 20 ∧ y < 70) || (x < 55 ∧ y < 95)
<x : 16, y : 80> is unwanted
and not transmitted from b3 to b1
(x > 15 ∧ y < 60) || (x > 20 ∧ y < 70) || (x > 55 ∧ y < 95)
x > 15 ∧ y < 60
x > 20 ∧ y < 70
x > 55 ∧ y < 95

(a) A CPSN (b) Subscription summarization on b_1 (I) (c) Subscription summarization on b_2 (II)

Fig. 1: Classic CPSNs and subscription summarization

not forwarded to b_2. In the scenario of Figure 1b, Φ_1 = $\texttt{x>15}\wedge\texttt{y<60}$ is the summary of Φ_1, Φ_2 and Φ_3.

To compute the summary of subscriptions *on a given event type*, a broker uses the covering relation $\preceq$ to construct a partially ordered set (poset) $\mathcal{P}$ of predicates. Figure 1b shows such a poset. The *least upper bound* (lub) of $\mathcal{P}$ is the predicate that covers all other predicates. If no lub inherently exists among the predicates, this predicate — dubbed $\text{LUB}(\mathcal{P})$ — is computed as the disjunction of all predicates that are not already covered by any other predicate. This poset can be stored as a tree and evaluated for an event e by letting e "trickle down" node-wise. Events that do not satisfy $\text{LUB}(\mathcal{P})$ are immediately discarded and events that satisfy individual subscriptions are forwarded to the corresponding subscribers.

D. Current Bottlenecks

Below we summarize the bottlenecks in current CPSNs:

1) Linear time operations: If an event type τ has k attributes, verifying $\Phi' \preceq \Phi$ is $O(k)$. Insertion of a new subscription into a poset $\mathcal{P}$ involves searching the poset to add the subscription while respecting the subsumption relation. When $\mathcal{P}$ is implemented as a $d-ary$ max-heap, this incurs a complexity of $O(k\,log_d|\mathcal{P}|)$=$O(k\,log_d\,n)$ where $n = |\mathcal{P}|$ is the number of existing subscriptions. But, processing an unsubscription is $O(k\,|\mathcal{P}|) = O(kn)$, because deleting a subscription may involve recreating the whole poset, e.g., when the subscription being removed is the root of the poset. Evaluation of $\mathcal{P}$ to forward an event to all interested subscribers is $O(k\,|\mathcal{P}|) = O(k\,n)$.

Many alternatives to posets have been explored including the Rete algorithm [19], used for instance in PADRES [7]. Rete yields a very effective event matching, because it uses a memory-intensive event flow graph. However, it does not inherently perform subscription summarization, and the complexities of subscription, unsubscription and event matching *remain* $O(k\,n)$. Another popular approach is based on binary decision diagrams (BDDs), but as shown by Campailla et al. [12] linear complexity is still observed (in fact $3kn$ operations [12] for many).

2) Opaque subscriptions: Many, especially early, systems advocated for *structural conformance* between events and subscriptions [18]. This leads to collapsing all subscriptions for all types of events into one single data-structure (e.g., poset) thus exacerbating bottlenecks. Most systems, even such described without event types, use types in practice, but only much later have the benefits of strong typing of events started to be fully exploited [20]. Similarly, only little work exists on the taking splitting a step further by systematically handling subscriptions *attribute-wise* (e.g., [1], [11]).

3) Large subscription summaries: The nodes of a poset can get arbitrarily complex if subscriptions do not subsume each other. Consider the scenario in Figure 1c. None of the subscriptions from the clients c_1, c_2, or c_3 subsume each other. 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, and these propagate throughout the network towards publishers (e.g., b_1 forwards its lub to b_2).

III. Split and Subsume in Beretta

This section presents the principles underlying Beretta, and the corresponding broker data-structures and algorithms.

A. Principles

Beretta involves a broker overlay network implemented on top of TCP/IP. While the algorithms are not bound to TCP/IP, we assume in the following pairwise FIFO reliable communication channels offering primitives SEND (non-blocking) and RECEIVE for presentation simplicity. For the same reason, we assume the absence of cycles in the broker network and a single broker or client per network node. This translates directly to FIFO guarantees on events and subscription updates. There are ways to operate in the absence of these assumptions but this makes the algorithms more complicated without invalidating our contributions. By the same token, we elide node failures; broker fault-tolerance is achieved by various means which are orthogonal to the contributions of this paper.

The scalability and efficiency of Beretta are based on six key ideas:

1. strong typing of events,
2. normalized subscriptions,
3. splitting of subscriptions by event types *and* attributes,
4. use of interval trees and hash maps,
5. systematic introduction of variables into subscriptions,
6. summarization approximation.

In short, (1) supports (2); (1) and (2) make (3) possible; (2) and (3) enable (4), (5) and (6).

B. Strongly Typed Events

In Beretta events are *strongly typed*. An event type τ denotes a set $\{\tau_1\ a_1,\ldots,\tau_k\ a_k\}$ of named attributes a_i which are typically of primitive types τ_i. An event e can thus be viewed as a set of attribute-value pairs $\{a_1{:}v_1,\ldots,a_k{:}v_k\}$ as before, but e carries an identifier for its type. This information can be used by a broker to efficiently dispatch an event e and to only evaluate subscriptions of the respective type on e. New event types can be added at runtime with only minimal propagation time through the network. For brevity, but unambiguously, we write $a_i \in \tau \Leftrightarrow \tau \doteq \{\ldots,\tau_i,a_i,\ldots\}$

C. Subscription Normalization

Subscriptions in Beretta are *normalized*, i.e., they are represented uniformly through a grammar modified as follows with respect to that of Section II-B:

$$Condition'\ \ \Theta ::= a\ \in\ [v,v]\ |\ a\ \in\ \{v,\ldots,v\}$$

Conditions are thus either interval queries or set inclusions. The former apply to totally ordered attribute types such as integer or floating point, and the latter apply to enumerable types such as strings. Our grammar improves the efficiency of CPS as we shall explain in Section III-E, yet is expressive enough to represent the subscriptions of the grammar presented in Section II-B. For example, a predicate `x>1000` where `x` is an integer attribute, can be expressed as `x`$\in$ [`1001`, `MAXINT`], and the predicate `z="IBM"` can be expressed as `z`$\in$ {`"IBM"`}. In fact, Beretta automatically normalizes subscriptions of the grammar of Section II-B. Assuming an event type $\tau \doteq$ {**int** `x`, **int** `y`, **string** `z`} with three attributes, a subscription `x`$\in$ [`45`, `78`] can be imagined as being transformed to `x`$\in$ [`45`, `78`]$\wedge$`y`$\in$ [`MIN_INT`, `MAX_INT`]$\wedge$`z`$\in$ $\{\star\}$, thus adding wildcard conditions for attributes without constraints. A key insight also for dealing with updates is that normalization ensures that a subscription to event type τ has *exactly one* condition for every attribute a_r of τ – be it a wildcard condition. Implementation-wise, such a condition does not actually have to exist at runtime, but if the application adds a corresponding condition to the subscription then the broker knows exactly where and how to create it based on the "new" bounds provided by the application.

D. Subscription Splitting

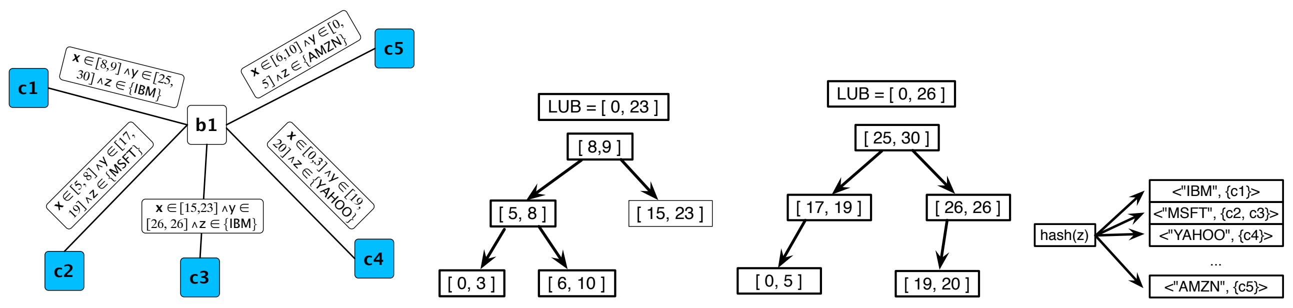
Each broker in the CPSN knows the set of neighboring brokers and the set of clients (publishers and subscribers) connected to it. Figure 3 presents the client algorithm of Beretta. Advertisements are omitted for brevity. A client subscribes to its edge broker by using event types (line 8). For presentation simplicity, the figure assumes a single subscription per client and type, and subscriptions to be already normalized. Thus a subscription is a conjunction of conditions ($\bigwedge_r \Theta_r$). Thanks to normalization the shape of any condition is known by the type of the respective attribute (totally ordered $\Rightarrow$ interval query; otherwise enumerable $\Rightarrow$ set inclusion), and thus the actual conditions do not have to be transmitted to the broker at all. As every node has a single subscription for a given event type, the algorithm does not need to actually denote any variables – the node identity together with the identifier r of the attribute a_r will allow to uniquely identify a variable based on the type of the attribute. More precisely, $\overline{v}_r$ represents the *vector* of values for a condition Θ_r on attribute a_r. In the case of an interval query $\overline{v}_r$'s size is 2 as it contains *both* bounds (lower bound first), while in the case of set inclusion the value set is transmitted.

The broker (see Figure 4) stores subscribers and publishers per types as well. (Advertisements and initial setup are again elided for brevity.) Subscriptions are stored in a *hash map* $\mathcal{S}$ indexed not only by type τ but also *by attribute* a_r (of τ). $\mathcal{S}[\tau][r]$ refers to a data-structure, detailed below, that represents all conditions from all subscriptions to type τ gathered from the broker's neighbors. $\mathcal{S}$ being a hash map enables constant time $O(1)$ lookup. Since published events are all tagged with their respective types this splitting allows to scale to any number of types, and does not impose restrictions on complexity of types.

E. Interval Trees and Hash Maps

Instead of maintaining subscriptions in a single poset as shown in Figure 1c, Beretta deals these up as mentioned. This finer granularity has benefits for all operations on and with subscriptions – addition, removal, updates, and of course evaluation. The modified subscription grammar and resulting normalization enable to further reduce complexity. More precisely, we organize conditions for totally ordered attribute types (interval queries) in *interval trees* [21]; for enumerated types (set inclusion) *hash maps* are used just like for subscription splitting. Consider the CPSN shown in Figure 1a where a broker b_1 is connected to five subscribers $c_1,...,c_5$, and the type $\tau \doteq$ {**int** `x`, **int** `y`, **string** `z`} introduced earlier. Figure 2a shows the subscriptions issued by each client attached to b_1, and Figure 2b and Figure 2c show the interval trees associated with attributes `x` and `y` respectively. b_1 receives five subscriptions on `z`. As shown in Figure 2d, Beretta uses a hash map whose key is `z` – each bucket stores a string v and the set of clients/brokers subscribed to `z`$=v$.

When an event $\{a_1{:}v_1,\ldots,a_k{:}v_k\}$ of type τ arrives, searching an interval tree for all intervals into which the value of an attribute a_r of totally ordered type falls is $O(log\ n)$ in the worst case if there are n intervals in the tree, i.e., n subscriptions for τ currently stored. For an attribute of enumerable type, retrieving the set of matching nodes can even be accomplished in constant time $O(1)$ due to the use of a hash map. For an event with k attributes, the worst-case



(a) Subscriptions (b) Interval tree for x (c) Interval tree for y (d) Hash map for z

Fig. 2: Examples of interval trees and hashing

complexity of event evaluation becomes $O(k\ log\, n)$. On the contrary, using a poset $\mathcal{P}$ as explained in Section II-D results in an event evaluation complexity of $O(k\,|\mathcal{P}|) = O(k\,n)$.

Figure 4 abstracts the handling of interval trees and hash maps for attribute conditions, for brevity. Individual conditions of an attribute a_r are stored as *c-nodes* (condition nodes) of the form $\langle \overline{v}_r, n_j \rangle$, where $\overline{v}_r$ are the current values for n_j's condition on attribute a_r. For the ordering within an interval tree, the two bounds $\overline{v}$ are obviously used. For a given data-structure and event, function MATCH() returns the set of clients and brokers with c-nodes in the data-structure whose values, plugged into the corresponding condition, match that of the event. At line 22, the intersection of these sets is taken across all attributes of the given type to determine the forwarding set. This would further increase the complexity of event evaluation. In Beretta the individual c-nodes are thus linked to each other which allows for this intersection to be created implicitly in a way similar to the *fast forwarding* in [1]. Thereby we also avoid *output-sensitivity*, i.e., that the complexity of an interval tree query is conditioned by the size of the output set. The LUB() function simply returns the lub of a given interval tree or hash map. Functions INSERT(), DELETE(), and UPDATE() abstract the insertion, removal, and update of a given c-node respectively; these functions implicitly trigger a reorganization of the data-structure and an update of the lub if necessary. Requests for new subscriptions (SUB), unsubscriptions (UNSUB), and variable updates (UPD) are all handled similarly: if the request alters the lub of the corresponding data-structure then this can trigger an update request (line 32) to parent brokers.

F. Systematic Variables for Updates

The normalization of subscriptions in Beretta lends itself well to the *systematic use of variables* for efficient updates. For an example of variable use, consider the condition `x>var` where `x` is the integer attribute of the type τ introduced earlier and `var` is a program variable (or identified by a handle as part of an API); the condition is normalized to `x∈ [var, MAXINT]`. An update to `var` can be directly propagated through the system. Systematically using variables involves viewing any condition on attribute `x` as `x∈ [var1, var2]`, and tracking the current values of `var1` and `var2`. Since we know the exact shape of conditions on a given attribute, a variable is

```
1:  init
2:      $b$                                                    {edge broker}
3:  to PUBLISH($e$) of $\tau$ do
4:      SEND(PUB, $\tau$, $e$) to $b$
5:  to SUBSCRIBE($\bigwedge_{r=1..n} \Theta_r$) to $\tau$ do
6:      for all $a_r \in \tau$ do
7:          $\overline{v}_r \leftarrow$ values in $\Theta_r$
8:      SEND(SUB, $\tau$, $\bigcup_r \overline{v}_r$) to $b$
9:  to UNSUBSCRIBE from $\tau$ do
10:     SEND(UNSUB, $\tau$) to $b$
11: upon RECEIVE(PUB, $\tau$, $e$) do
12:     if $\Phi(e)$ | $\Phi$ is subscription to $\tau$ then        {maybe changed}
13:         DELIVER($e$)
14: upon change of value in $\Theta_r$ for subscription to $\tau$ do
15:     SEND(UPD, $\tau$, $r$, new$\overline{v}_r$) to $b$
```

Fig. 3: Beretta client algorithm

uniquely identified in our algorithms by the node identifier n_i (and an index in the case of multiple subscriptions), event type τ, and attribute a_r. In fact, all the client algorithm of Figure 3 does is sending the edge broker "abstract" conditions, consisting in the *current* values of implicitly defined variables (for bounds in interval queries; sets in set inclusion) when issuing a subscription. Corresponding c-nodes are added to interval trees or hash maps.

Upon an update request (UPD), the corresponding c-node is first updated, and then the respective interval tree or hash map. If the lub c-node changes, function PROPAGATE() is invoked which issues an update request to any upstream brokers if any of the bounds changed. Note that such a recursive upstream update request — just like an upstream update request engendered by a subscription addition or removal — contains the identity of the broker and *not* of the node associated with the respective lub c-node, to avoid propagating references and thus creating dependencies. Our model implies that *any* addition, removal, or update of a subscription results in a set of variable updates, recursively through brokers. In practice, the different update requests for a same type sent at line 32 are bundled.

Observe that our model does not only support *variable-based* updates (e.g., changes of bounds on intervals). Since subscriptions are normalized by including wildcard subscrip-

```
1:  init
2:      $subs[]$                       {hashmap, indexed by event types $\tau$}
3:      $pubs[]$                       {hashmap, indexed by event types $\tau$}
4:      $\mathcal{S}[][]$     {hashmap, indexed by event types $\tau$ and attribute index}
5:  upon RECEIVE($\text{SUB}, \tau, \bigcup_r \overline{v}_r$) from $n_j$ do
6:      for all $a_r \in \tau$ do              {insert condition for attribute $r$}
7:          $subs[\tau] \leftarrow subs[\tau] \cup \{n_j\}$    {add node to subscribers of $\tau$}
8:          $c\text{-}node \leftarrow \langle \overline{v}_r, n_j \rangle$
9:          $c\text{-}node_0 = \langle \overline{v^0}_r, n_k \rangle \leftarrow \text{LUB}(\mathcal{S}[\tau][r])$          {old lub}
10:         INSERT($\mathcal{S}[\tau][r], c\text{-}node$)
11:         $c\text{-}node_\nu = \langle \overline{v^\nu}_r, n_l \rangle \leftarrow \text{LUB}(\mathcal{S}[\tau][r])$          {new lub}
12:         PROPAGATE($c\text{-}node_0, c\text{-}node_\nu, \tau, r$)
13: upon RECEIVE(UNSUB,$\tau$) from $n_j$ do
14:     $subs[\tau] \leftarrow subs[\tau] \backslash \{n_j\}$
15:     for all $a_r \in \tau$ do
16:         $c\text{-}node \leftarrow \langle \overline{v}, n_j \rangle \in \mathcal{S}[\tau][r]$      {get vals for $n_j$'s condition}
17:         $c\text{-}node_0 = \langle \overline{x^0}, n_k \rangle \leftarrow \text{LUB}(\mathcal{S}[\tau][r])$          {old LUB}
18:         DELETE($\mathcal{S}[\tau][r], c\text{-}node$)
19:         $c\text{-}node_\nu = \langle \overline{x^\nu}, n_l \rangle \leftarrow \text{LUB}(\mathcal{S}[\tau][r])$          {new LUB}
20:         PROPAGATE($c\text{-}node_0, c\text{-}node_\nu, \tau, r$)
21: upon RECEIVE($\text{PUB}, \tau, e$) from $n_j$ do
22:     for all $n_k \in \bigcap_{a_r \in \tau} \text{MATCH}(\mathcal{S}[\tau][r], e)$ do
23:         SEND($\text{PUB}, \tau, e$) to $n_k$
24: upon RECEIVE($\text{UPD}, \tau, r, \overline{(v)}$) from $n_j$ do
25:     $c\text{-}node_0 = \langle \overline{v^0, n_s} \rangle \leftarrow \text{LUB}(\mathcal{S}[\tau][r])$          {old LUB}
26:     $c\text{-}node_{upd} \leftarrow \langle \overline{v^{upd}}, n_j \rangle \in \mathcal{S}_\tau[r]$
27:     UPDATE($\mathcal{S}[\tau][r], c\text{-}node_{upd}, \overline{x}$)    {update the node in $\mathcal{S}[\tau][r]$}
28:     $c\text{-}node_\nu = \langle \overline{v^\nu}, n_q \rangle \leftarrow \text{LUB}(\mathcal{S}[\tau][r])$          {new LUB}
29:     PROPAGATE($c\text{-}node_0, c\text{-}node_\nu, \tau, r$)
30: procedure PROPAGATE($\langle \overline{v^0} \rangle, \langle \overline{v^\nu} \rangle, \tau, r$)
31:     if $\exists v_s^\nu \neq v_s^0$ then          {LUB change$\Rightarrow$update upstream}
32:         SEND($\text{UPD}, \tau, \overline{v^\nu}$) to all $b_k \in pubs[\tau]$
```

Fig. 4: Beretta broker algorithm as executed by b_i. Common processing of interval tree modifications (new subscriptions, unsubscriptions, updates) are regrouped in PROPAGATE

tions for all attributes without actual conditions, this holds equally for *structural* updates of subscriptions – it is as if behind the scenes a subscription to a type τ had a condition for every attribute of τ already. Condition addition (as well as removal) just boils down to an update of the corresponding variable(s). As mentioned earlier though, a wildcard condition does not actually have to exist in memory on a broker; if a variable update is received for such a condition, the broker knows exactly how to interpret the "update".

G. Summarization Approximation

When sending a summary to a parent, a broker *approximates the summarization*. Remember that a broker manages a separate data-structure $\mathcal{S}[\tau][r]$ for each attribute a_r of a given event type τ. 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 Figure 4 this conjunction arises implicitly, as an update request (line 32) contains the lub for the respective type and attribute. It is easy to see that this is an approximation. Consider the conditions of an attribute a_r of type τ for a number n of subscriptions equalling the number k of attributes in τ ($n = k$). Subscription r's condition on a_r can be the lub for a_r, while the condition for every other a_q, $q \neq r$, of subscription r can be covered by that of subscription q.

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.

IV. Evaluation

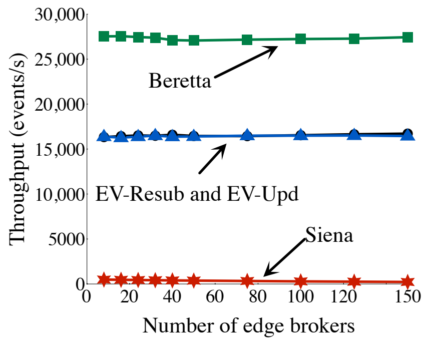
We first evaluate Beretta on two benchmarks modeled on two different real-life applications – highway traffic management and algorithmic trading. Then we evaluate the scalability of Beretta through a statistically generated workload.

A. Metrics

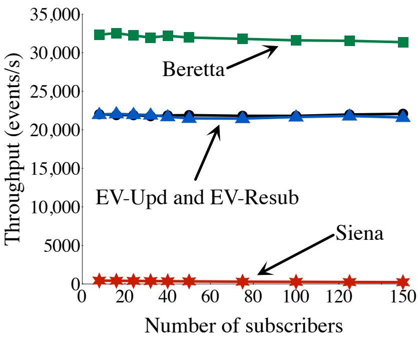
Similar to [16], our evaluation uses four metrics.

1) Throughput: The throughput of a CPSN is the average number of events delivered by a subscriber per s. Throughput thus depends on: (a) the number of publishers, (b) the production rate of events at each publisher, (c) the update rate for subscriptions of individual subscribers (updates can be structural or parametric) and, (d) the selectivity of the subscription of a subscriber. Selectivity is the probability that a event published by a publisher satisfies the subscription. A selectivity of 1.0 implies that every published event satisfies the considered subscription and 0.0 implies that none do.

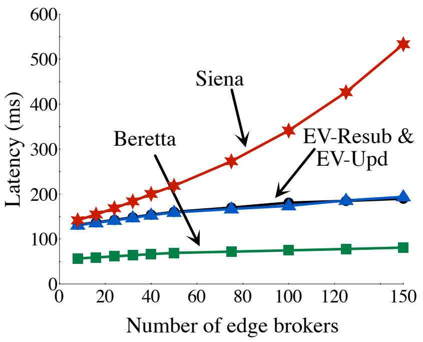
2) Latency: Latency refers to event dissemination latency: if an event e is produced at time t_1 and is received by a subscriber at time t_2, then the dissemination latency of e is $t_2 - t_1$. Latency of e measures the timeliness of the subscription subsumption, update, and event forwarding algorithms, and is



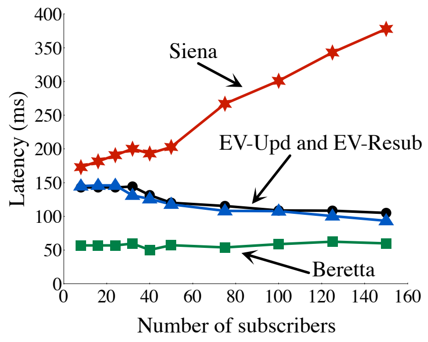
(a) Throughput (0 updates/s) (HTM benchmark)



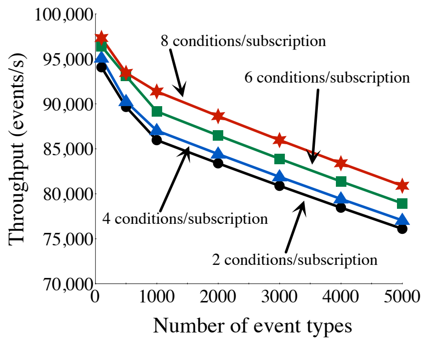
(b) Throughput (0 updates/s) (AT benchmark)



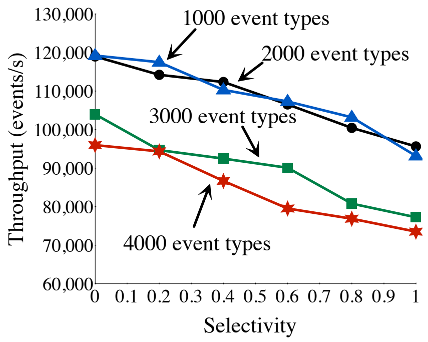
(c) Latency (0 updates/s) (HTM benchmark)



(d) Latency (0 updates/s) (AT benchmark)



(e) Broker throughput vs subscription complexity



(f) Broker throughput vs subscription selectivity

Fig. 5: Comparing Beretta's performance with EV-Resub, EV-Upd and Siena. Figs. (a) to (d) show improvements in throughput and latency when the update frequency is 0 and the number of subscribers in AT and edge brokers in HTM is varied in the HTM and AT benchmarks. Figs. (e) and (f) show per-broker throughput on statistically generated workloads

a function of: (i) the time taken by a broker to decide whether e has to be discarded or sent to a downstream broker (ii) the bandwidth of the network links used to connect brokers and (iii) network traffic and congestion on broker-broker links. Although no algorithms described in this paper require or directly benefit from synchronized clocks, we ensured that clocks of publishers and subscribers were synchronized while conducting experiments that measured latency.

3) Update delay: Update delay is the time it takes for a CPSN to react to subscription updates (either structural or parametric). If a subscriber changes its subscription Φ_i to Φ_i' at time t_1 and the first event matching Φ_i' but not Φ_i is delivered at time t_2, then t_2-t_1 yields an approximation of the update delay at the subscriber.

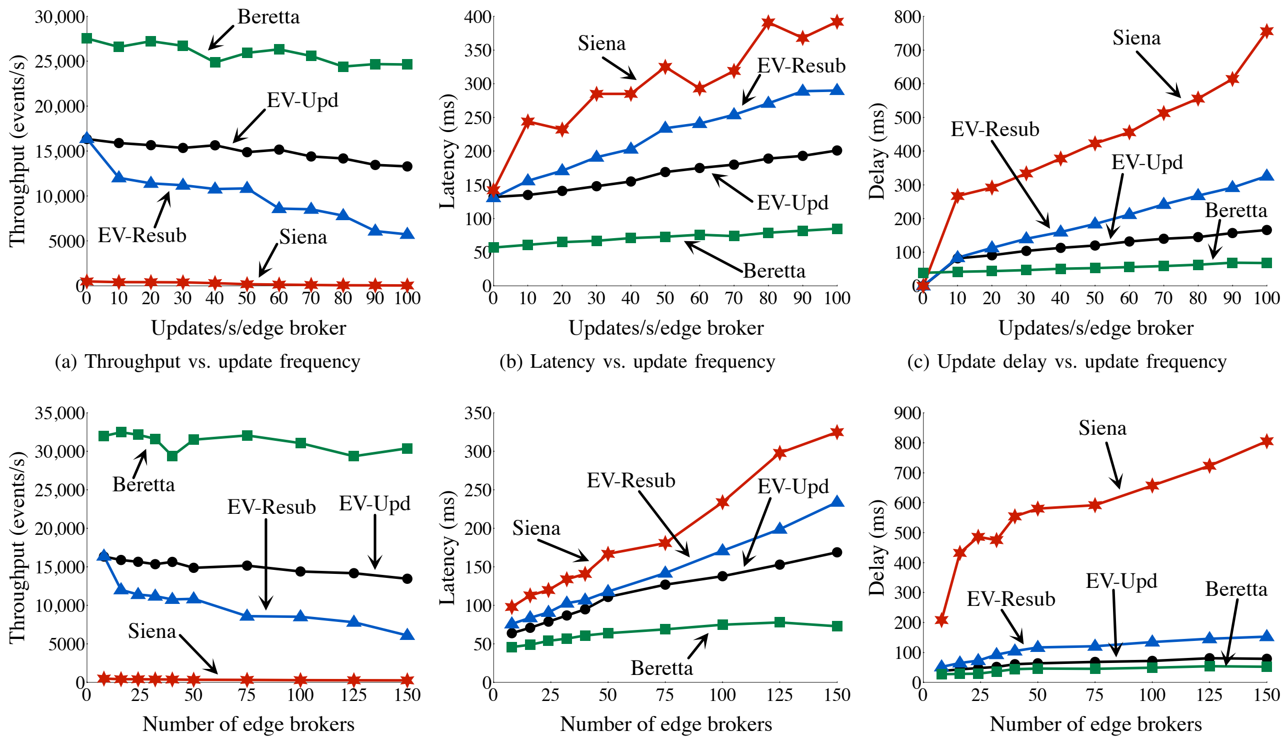
4) Spurious events : The effect of inefficient updates might be absorbed if brokers are powerful dedicated servers or individual clients are only interested in few events . Otherwise, increased stress might manifest especially on resource-constrained clients. To gauge this, we measure the amount of spurious events delivered by clients. If a subscriber changes its subscription Φ_i to Φ_i' at time t_1, then spurious events are those matching Φ_i but not Φ_i' and received by the client *after* t_1 and filtered out locally to it (see line 12 in Figure 3). These capture the overhead imposed on clients.

B. Systems for Comparison

To characterize performance gains due to our novel routing mechanisms, we compare Beretta against three CPSNs:

1) **Siena** [1], [2], a seminal, open-source, CPSN.
2) **EV-Resub**, i.e., EV (no subscription updates): EV is a publish/subscribe middleware based on the Rete algorithm [7], [22]. The Rete algorithm is highly effective and highly touted for matching events to subscriptions, and has been used also, e.g., in PADRES [7] and JBoss messaging middleware [23]. It has been shown to sustain high throughput also in *complex* event detection (patterns of events) [22]. EV uses posets for subscription management and the Rete algorithm for event matching.
3) **EV-Upd**, i.e., EV with parametric updates: To separate the performance benefits of dynamic updates from the performance benefits of replacing posets for event matching, we evaluate a version of EV supporting parametric subscription updates.

For systems 1) and 2) without support for subscription updates we emulate updates by resorting to the standard solution of re-subscriptions: issuing a new subscription before canceling the outdated one.



(a) Throughput vs. update frequency

(b) Latency vs. update frequency

(c) Update delay vs. update frequency

(d) Throughput vs. number of edge brokers

(e) Latency vs. number of edge brokers

(f) Update delay vs. number of edge brokers

Fig. 6: Comparing Beretta's performance with EV-Resub, EV-Upd and Siena for the HTM benchmark by varying the update frequency (keeping number of edge brokers fixed at 150) in Figures (a)–(c) and the number of edge brokers (keeping update frequency fixed at 50 updates/s/edge broker) in Figures (d)–(f).

C. *Infrastructure*

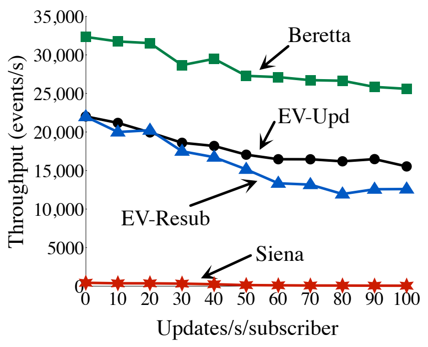
All brokers were executed on commodity dual core Intel Xeon 3.2Ghz machines with 4GB RAM running Linux, with each machine executing exactly one broker. Subscribers were deployed on a cluster, where each node is an eight core Intel Xeon 1.8Ghz machine with 8GB RAM running Linux, with 8 subscribers deployed on each node (one subscriber per core). Publishers were deployed on dual core Intel Pentium 3Ghz machines with 4GB RAM, with no more than 2 publishers per machine (one publisher per core). Deploying publishers, subscribers and brokers on different nodes ensured that all relevant communication (publisher-broker, broker-broker and subscriber-broker) was over a network, and across LANs. 10 ms delays were added to each network link to simulate wide area network characteristics as is done in EmuLab [24].

D. *Real-life Benchmarks*

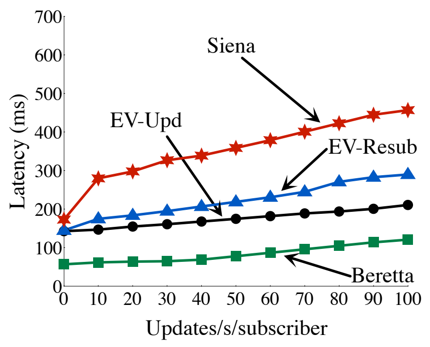
We now evaluate Beretta by comparison against three CPSNs based on two real-life benchmarks.

1) Highway Traffic Management (HTM): As in [8], to evaluate our algorithms, we used a traffic management system based on [25], [26] which controls an area, with ten peripheral highways and four highways intersecting at the middle. The system we used consists of a CPSN where the publishers are several wireless sensors and cameras located along the highway, monitoring road conditions, traffic density, speeds, temperature, rainfall, snow etc. Each sensor connects wirelessly to a base station which serves as the broker – the base stations are connected to each other by a wired network. Subscribers are navigation systems and other location-based applications (e.g., daemons which serve location-based ads) in vehicles, computers and mobile phones. Subscribers subscribe to events in a certain geographical range around their current location (GPS coordinates), based on a driver (or passenger's) input and hence the system issues both structural and parametric updates.

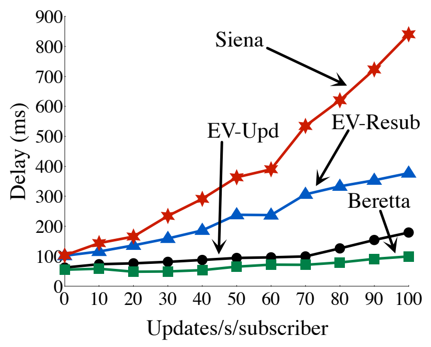
The system uses a non-hierarchical CPSN (highways in an urban area are not hierarchical) with 200 brokers (150 edge brokers and 50 "non-edge" brokers), and 5 publishers per edge broker, resulting in a total of 750 publishers. The distribution of operators *op* in subscriptions was 37% $\geq$, 41% $\leq$, and 22% $=$. On this benchmark, we evaluate our algorithms with update frequencies ranging from 10 updates/s to 100 updates/s/edge broker. Update frequencies of 83.33 updates/s/edge broker are pretty common in traffic management – consider a traffic density of 500 cars/mile (250 cars in either direction) on a 10 mile stretch of six lane highway controlled by an edge broker. Assuming an update frequency of 1 update/min/*subscriber*, up-



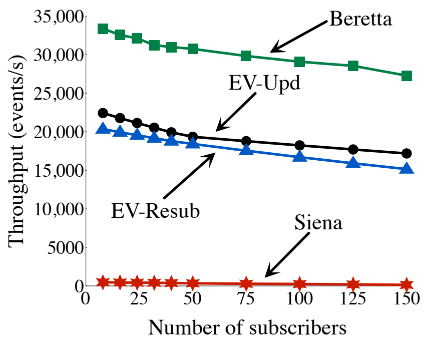
(a) Throughput vs. update frequency



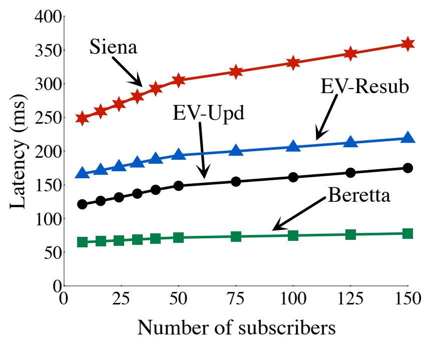
(b) Latency vs. update frequency



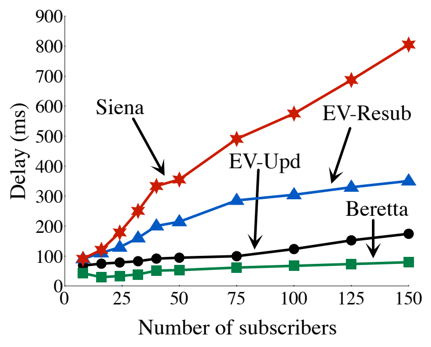
(c) Update delay vs. update frequency



(d) Throughput vs. number of subscribers

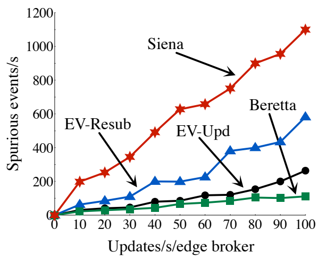


(e) Latency vs. number of subscribers

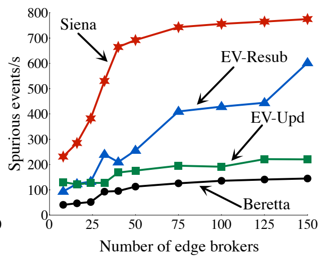


(f) Update delay vs. number of subscribers

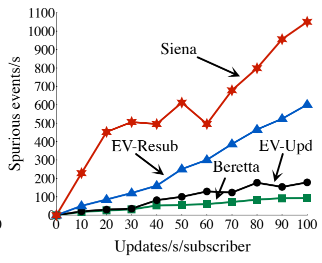
Fig. 7: Comparing Beretta's performance with EV-Resub, EV-Upd and Siena for the AT benchmark by varying the update frequency (keeping number of subscribers fixed at 150) in Figures (a)–(c) and the number of subscribers (keeping update frequency fixed at 50 updates/s/subscriber) in (d)–(f)



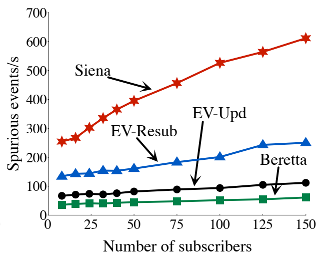
(a) Spurious events – HTM



(b) Spurious events – HTM



(c) Spurious events – HTM



(d) Spurious events – HTM

Fig. 8: Comparing the performance of Beretta with EV-Upd, EV-Resub and Siena with respect to the number of spurious events

date frequency at the corresponding edge broker is 500*10/60 = 83.33.

Each publisher generates events (traffic conditions, weather conditions, commercial advertisements, etc) at the rate of 150 events/s. Each subscriber (vehicle) subscribes to less than 40% of events generated, and updates its location to the edge broker.

2) Algorithmic Trading (AT) : We consider the monitoring component of an algorithmic stock trading system. We use a CPSN disseminating commodity prices with 20 brokers, 10 publishers and 150 subscribers. In AT, the number of publishers is small – stock quotes are published by stock exchanges only. We assume that a subscriber is a computer at an AT firm. Our benchmark had 300 event types, which includes the quotes of 153 stocks, analyst predictions, etc. In the experimental setup, we employed a hierarchical broker overlay network, which is typical in stock quote dissemination – newswires and market data systems at the top, large clearing houses at the second level, large brokerages and trading firms at the next level to which small brokerages and consumers connect.

Each publisher generates events (stock quotes, analyst reports, insider trading information etc) at the rate of 6000 events/s. Each subscriber subscribes to less than 20% of events generated – which is typical in the case of stock brokers that

Metric	Increase in throughput		Decrease in latency		Decrease in delay		Decr. in spurious events	
Benchmark	HTM	AT	HTM	AT	HTM	AT	HTM	AT
Beretta vs. EV-Upd	Fig. 6a up to 1.85×	Fig. 7a up to 1.65×	Fig. 6b up to 2.36×	Fig. 7b up to 1.74×	Fig. 6c up to 2.44×	Fig. 7c up to 1.80×	Fig. 8a up to 2.36×	Fig. 8c up to 1.90×
Beretta vs. EV-Resub	Fig. 6a up to 4.29×	Fig. 7a up to 2.03×	Fig. 6b up to 3.41×	Fig. 7b up to 2.40×	Fig. 6c up to 4.8×	Fig. 7c up to 3.79×	Fig. 8a up to 5.21×	Fig. 8c up to 6.42×
Beretta vs. Siena	Fig. 6a more than 10×	Fig. 7a more than 10×	Fig. 6b up to 4.61×	Fig. 7b up to 3.78×	Fig. 6c up to 11.11×	Fig. 7c up to 8.45×	Fig. 8a up to 9.85×	Fig. 8c up to 11.23×

TABLE I: Comparing Beretta's performance with EV-Resub, EV-Upd and Siena – this table shows improvements in throughput, latency, delay, and spurious events when the update frequency is varied yet number of subscribers in AT and edge brokers in HTM is fixed at 150.

Metric	Increase in throughput		Decrease in latency		Decrease in delay		Decr. in spurious events	
Benchmark	HTM	AT	HTM	AT	HTM	AT	HTM	AT
Beretta vs. EV-Upd	Fig. 6d up to 2.26×	Fig. 7d up to 1.59×	Fig. 6e up to 2.32×	Fig. 7e up to 2.24×	Fig. 6f up to 1.5×	Fig. 7f up to 2.19×	Fig. 8b up to 2.34×	Fig. 8d up to 1.83×
Beretta vs. EV-Resub	Fig. 6d up to 4.98×	Fig. 7d up to 1.8×	Fig. 6e up to 3.20×	Fig. 7e up to 2.81×	Fig. 6f up to 2.9×	Fig. 7f up to 4.40×	Fig. 8b up to 9.71×	Fig. 8d up to 4.09×
Beretta vs. Siena	Fig. 6d more than 10×	Fig. 7d more than 10×	Fig. 6e up to 4.45×	Fig. 7e up to 4.60×	Fig. 6f up to 15.34×	Fig. 7f up to 10.13×	Fig. 8b up to 12.5×	Fig. 8d up to 10×

TABLE II: Comparing Beretta's performance with EV-Resub, EV-Upd and Siena – this table shows improvements in throughput, latency, delay, and spurious events when the update frequency is fixed and the number of subscribers in AT and edge brokers in HTM is varied. Update frequency is 50 updates/s/subscriber for AT and 50 updates/s/edge broker for HTM

consume a fraction of the information.

E. Results and Analysis – 0 Updates/s

Figures 5a,5b,5c and 5d and Table III describe the performance of Beretta vis-a-vis EV-Resub, EV-Upd and Siena when there are no subscription updates, either structural or parametric.

	Increase in throughput		Decrease in latency	
Benchmark	HTM	AT	HTM	AT
Beretta vs. EV-Resub/EV-Upd	1.48× Fig. 5a	1.69× Fig. 5b	2.39× Fig. 5c	2.48× Fig. 5d
Beretta vs. Siena	117× Fig. 5a	132× Fig. 5b	6.57× Fig. 5c	7.34× Fig. 5d

TABLE III: Performance of Beretta vs. EV-Resub and EV-Upd when there are no subscription updates

The throughput of Beretta is more than 100× the throughput of Siena, which demonstrates the effect of reducing event matching from $O(k\,n)$ to $O(k\,log\,n)$. This also decreases event dissemination latency by more than 5× in both benchmarks. Figures 5a and 5b show that the throughput of Beretta is up to 1.48× the throughput of EV-Resub or EV-Upd for the AT benchmark and 1.69× the throughput for the HTM benchmark. When there are no updates, the throughput of EV-Resub and EV-Upd are almost equal, as expected. This proves that Beretta improves on the highly effective Rete algorithm by more than 48%, despite Rete's construction of a separate event flow graph. We also note that Rete is memory intensive but Beretta is not.

F. Results and Analysis (Broker-level)

Figures 5e and 5f show that Beretta retains a high per-broker throughput irrespective of the selectivity of the subscriptions and the complexity (number of conditions) of the subscription. Figures 5e and 5f measure per-broker throughput under various statistically generated workloads, and they do not correspond to either the HTM or AT benchmarks. In Figure 5e, we measured the throughput of a Beretta broker while simultaneously increasing the number of event types and the number of conditions per subscription (predicate). The total number of subscriptions was 100,000, which were uniformly distributed over all event types i.e. when there are 100,000 subscriptions and 5000 event types, there were 20,000 subscriptions per event type. Though the throughput of Beretta decreases by ∼25% when the number of event types involved is increased from 100-1000, the throughput stabilizes and decreases only by ∼6% when the number of event types is increased from 1000-5000. Similarly, Figure 5f shows that throughput decreases as the selectivity of the subscription *decreases* (from 0 to 1). Recall that selectivity of a subscription refers to the probability that a subscription matches an event. When a subscription is highly selective, more events are filtered out by Beretta's algorithms, and consequently more events can be "pushed through" a broker, resulting in higher throughput. When selectivity decreases and reaches 1, every event is matched to some subscription, which leads to more conditions being evaluated on each event, thereby decreasing throughput.

G. Results and Analysis (With Updates)

To gauge the performance of Beretta under high rates of churn, i.e., the modification and removal of subscriptions, we evaluated it against the three other CPSNs, under two scenarios. First, we varied the update frequency at each subscriber (in AT) and edge broker (in HTM) from 0 updates/s to 100 updates/s, while keeping the number of subscribers (in AT) and the number of edge brokers (in HTM) constant at 150. The updates were 40% parametric updates and 60% structural updates. Table I summarizes the improvements in performance of Beretta vis-a-vis EV-Upd, EV-Resub and Siena under increasing update frequencies. Then, we measured the performance of Beretta with an increase in the number of subscribers (in AT) or edge brokers (in HTM) while keeping update frequency constant at 50 updates/s, and the performance improvements under this scenario are summarized in II. From Figures 6, 7 and 8, we observe the following:

1) Increased throughput: From Figures 6 and 7 we observe that update support significantly increases event processing throughput of a CPSN with an increasing frequency of subscription updates at a subscriber or edge broker respectively. Resubscriptions lead to *thrashing* – where a CPSN expends more time and resources tearing down and rebuilding posets and other data-structures used in routing subscriptions thereby severely limiting resources available at brokers for filtering and forwarding events. Thrashing is especially evident in the drastic drop of Siena's throughput (Figures 6a, 6d, 7a, and 7d). The $O(k\,n)$ processing time for subscription management does not help either. We also observe that the choice of the forwarding algorithm does affect throughput, and the extent to which it decreases.

The throughput of Siena is low because it uses the poset for forwarding events. But the throughput of EV-Resub and EV-Upd are higher because they use the Rete algorithm. The decrease in the throughput of EV-Resub, inspite of using Rete can be explained by thrashing. It can also be observed that throughput decreases with an increase in the number of subscribers (and consequently, an increase in the number of subscriptions).

2) Reduced latency: Event dissemination latency increases with an increase in update frequencies in all systems (Figures 6b, 7b, 6e and 7e). This is due to the sharing of resources, between routing events and processing subscription updates, of each broker on the path of an event from publisher to a subscriber at increased update frequencies. But, in both benchmarks, the latency of a CPSN that uses resubscriptions to update subscriptions increases at a much faster rate than that of a CPSN with inherent support for updates.

3) Reduced update delay: Figures 6 and 7 illustrate the strong benefits in terms of decreased delay in responding to subscription updates with inherent update support. Beyond the raw figures on the decrease in delay (up to 2.44× vs. EV-Upd, up to 4.8× vs. EV-Resub and up to 15.34× vs. Siena), we observe that this decrease is more pronounced as the update frequency increases or as the number of subscribers increases. This is because resubscriptions can potentially involve two operations on the poset – INSERT and DELETE, whereas systematically introducing variables into intervals and updating them only involves one operation. Since both INSERT and DELETE have to acquire a lock on $\mathcal{P}[\tau]$ (in both versions of EV) or interval tree (in Beretta), replacing two operations by one reduces the contention on either of these data structures by half, thereby significantly decreasing delay. The decrease in delay is greater than 2× because update messages not only reduce contention at edge brokers but also at upstream brokers on the path to the publisher.

4) Reduced number of spurious events: Figure 8 shows that Beretta significantly decreases the number of spurious events received by a subscriber compared to Siena, EV-Resub or EV-Upd in both benchmarks. In fact, Beretta delivers up to 2.34× fewer spurious events than EV-Upd, which supports parametric updates. This demonstrates the agility of Beretta's subscription management and event forwarding algorithms.

V. Related Work

To the best of our knowledge, Beretta is the first CPSN with provable logarithmic matching complexity and demonstrating that it can sustain high throughput under a high frequency of parametric as well as structural subscription updates. Several seminal CPS systems have already been discussed earlier in this paper and thus we refrain from repeating those here. In general, many CPSN systems have been proposed and enumerating all of them is impossible in the allocated space.[2]

Gryphon [5] constructs an overlay graph such that individual attribute matches only occur on one node. This requires knowledge of all subscriptions and updates if any of those change, and achieves $O(n^{\lambda})$ complexity on brokers where λ depends on actual subscriptions, coming close to $1/2$ at times [5]. These support only equality comparisons. Li et al [6], [7] propose subscription covering, merging, and content matching algorithms based on a modified BDD representation in PADRES. XSiena [9] uses Bloom filters for subscription management and event forwarding, but as proven in [9], the complexity of subscription addition/removal is $O(2^m)$, where m is the number of *bits* required to store a condition in a subscription (i.e., filter in XSiena terminology). Bloom filters also require $O(2^m)$ space [9]. XSiena became openly available very recently and we plan to compare it empirically to Beretta. Meghdoot [27] uses a distributed hashtable (DHT) to determine the location of subscriptions and to route events to the subscribers. The partitioning of the DHT across peers allows Meghdoot to eliminate the need of brokers, however, the design is inflexible when the schema is dynamic as it requires the complete cartesian space to be reconstructed. HERMES [3] provides type- and attribute-based routing. GREEN [8] is a highly configurable and re-configurable publish/subscribe middleware for pervasive computing applications. Adaptation to varying underlying infrastructures, which the authors focus on, is very important in such environments. These efforts are complementary to Beretta, which focuses on adapting to application changes, i.e., subscriptions.

Topic-based publish/subscribe (TPS) provides limited expressiveness compared to CPS, as a subscriber receives all messages in a topic. Beretta is able to encode even hierarchical topics. Examples of TPS systems are SCRIBE [28],

[2]http://event-based.org/link-collection/ provides an exhaustive view of literature on the topic.

Bayeux [29], and Spidercast [30]. [31] is a recent divide and conquer approach for TPS.

Astrolabe [32] can be instantiated as a CPSN. With an emphasis on fault tolerance, nodes periodically exchange membership information. This information includes interests, which are aggregated based on topological or logical constraints. Nodes are selected to represent others based on the same criteria, leading to an overlay hierarchy. Interests are expressed like SQL queries, which can be applied to multicast events as well as to data stored on the overlay network. In that sense it is more like a *content distribution network* (CDN) (e.g. Coral CDN [33], Akamai [34] and Amazon CloudFront [35]). A CDN is a system of computers containing copies of data, placed at various points in a network so as to maximize bandwidth for access to the data from clients throughout the network. A client accesses a copy of the data near to the client, as opposed to all clients accessing the same central server, so as to avoid bottlenecks. CDNs consist of web caches (that store the most frequently accessed objects), software load balancers, hardware load balancers (layer 4-7 switches) and request routing infrastructures. CDNs are orthogonal to CPSNs – they solve different problems.

Recently, there have been proposals for a clean-slate content-centric redesign of the Internet such as Project CCNx [36] and Data-Oriented Network Architecture (DONA) [37]. Both CCNx and DONA combine the best aspects of CDNs and CPSNs, along with inherent support for confidentiality, data- persistence, availability and authenticity.

VI. Conclusions

We have presented Beretta, a highly scalable and efficient CPSN with inherent support for subscription updates (parametric as well as structural). The benefits of Beretta hinge on a combination of several key ideas, rooting in strong message typing and a novel model of normalized subscriptions. We have illustrated the benefits of our techniques through real-life and statistical workloads. We are currently extending Beretta with support for message *pattern* detection.

References

[1] A. Carzaniga and A. Wolf, "Forwarding in a Content-based Network," in *SIGCOMM '03*.

[2] A. Carzaniga, M. J. Rutherford, and A. L. Wolf, "A Routing Scheme for Content-based Networking," in *INFOCOM '04*.

[3] P. Pietzuch, B. Shand, and J. Bacon, "A Framework for Event Composition in Distributed Systems," in *Middleware '03*.

[4] L. Fiege, F. Gärtner, O. Kasten, and A. Zeidler, "Supporting Mobility in Content-Based Publish/Subscribe Middleware," in *Middleware '03*.

[5] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching Events in a Content-Based Subscription System," in *PODC'99*.

[6] 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 *ICDCS '05*.

[7] G. Li, V. Muthusamy, and H.-A. Jacobsen, "Adaptive Content-based Routing in General Overlay Topologies," in *Middleware '08*.

[8] T. Sivaharan and G. S. Blair and G. Coulson, "GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing," in *OTM Conferences (1), 2005*.

[9] Z. Jerzak and C. Fetzer, "Bloom Filter Based Routing for Content-based Publish/Subscribe," in *DEBS '08*.

[10] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE TSE*, vol. 27, no. 9, 2001.

[11] P. Triantafillou and A. A. Economides, "Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems," in *ICDCS'04*.

[12] A. Campailla and S. Chaki and E. Clarke and S. Jha and H. Veith, "Efficient Filtering in Publish-Subscribe Systems Using Binary Decision Diagrams," in *ICSE '01*.

[13] Moving Markets: Shifts in Trading Patterns are Making Technology Ever More Important. *http://www.economist.com/business-finance/displaystory.cfm?story_id=E1_VQSVPRT*, The Economist, 2006.

[14] Algorithmic Trading: Hype or Reality? *http://www.aitegroup.com/reports/20050328.php*, Aite Group, 2005.

[15] Y. Huang and H. Garcia-Molina, "Parameterized Subscriptions in Publish/Subscribe Systems," *Data and Knowledge Eng.*, vol. 60, pp. 435–450, 2007.

[16] K. R. Jayaram and C. Jayalath and P. Eugster, "Parametric Subscriptions for Content-based Publish/Subscribe Networks," in *MIDDLEWARE 2010*.

[17] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM TOCS*, vol. 8, no. 2.

[18] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The Information Bus: an Architecture for Extensible Distributed Systems," in *SOSP '93*.

[19] C. .L. Forgy, *On the Efficient Implementation of Production Systems*.

[20] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic Support for Distributed Applications," *IEEE Computer*, vol. 33, no. 3.

[21] T. H. Cormen, R. L. Rivest, C. Leiserson, and C. H. Stein, *Introduction to Algorithms*, 2010.

[22] P. Eugster and K. R. Jayaram, "EventJava: An Extension of Java for Event Correlation," in *ECOOP'09*.

[23] Red Hat Inc., "JBoss Messaging Middleware," 2010.

[24] Emulab – Total Network Testbed. *http://www.emulab.net*.

[25] S. Schneider, "DDS and Distributed Data-centric Embedded Systems. *http://www.drdobbs.com/embedded-systems/196601852*."

[26] D. Barnett, "Publish-Subscribe Model connects Tokyo Highways. *http://www.industrial-embedded.com/articles/barnett/*."

[27] A. Gupta, O. Sahin, D. Agrawal, and A. Abbadi, "Meghdoot: Content-based Publish/Subscribe over P2P Networks," in *Middleware '04*.

[28] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "SCRIBE: The Design of a Large-Scale Event Notification Infrastructure," in *Networked Group Communication 2001*.

[29] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: an Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination," in *NOSSDAV '01*.

[30] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "SpiderCast: a Scalable Interest-aware Overlay for Topic-based Pub/Sub Communication," in *DEBS '07*.

[31] C. Chen, H.-A. Jacobsen, and R. Vitenberg, "Divide and Conquer Algorithms for Publish/Subscribe Overlay Design," in *ICDCS 2010*.

[32] R. van Renesse, K. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining," *ACM TOCS*, vol. 21, no. 2, 2003.

[33] The Coral Content Distribution Network. *http://www.coralcdn.org*,

[34] Akamai HD Network.*http://www.akamai.com/html/solutions/hdnetwork.html*, Akamai Inc.

[35] Amazon CloudFront. *http://aws.amazon.com/cloudfront/*, Amazon.com Inc.

[36] Project CCNx. *http://www.ccnx.org/*, Palo Alto Research Center.

[37] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. Kim, S. Shenker, and I. Stoica, "A Data-oriented (and Beyond) Network Architecture," in *SIGCOMM 2007*.