Parametric Subscriptions for Content-Based Publish/Subscribe Networks

K. R. Jayaram, Chamikara Jayalath, Patrick Eugster

Parametric Subscriptions for Content-based Publish/Subscribe Networks *

K. R. Jayaram Chamikara Jayalath Patrick Eugster

Department of Computer Science, Purdue University
{jayaram,cjalayat,peugster}@cs.purdue.edu

Abstract. Subscription *adaptations* are becoming increasingly important across many *Content-based publish/subscribe* (CPS) applications. In algorithmic high frequency trading, for instance, stock price thresholds that are of interest to a trader change rapidly, and gains directly hinge on the reaction time to relevant fluctuations. The common solution to adapt a subscription consists of a *re-subscription*, where a new subscription is issued and the superseded one canceled. This is ineffective, leading to missed or duplicate events during the transition. In this paper, we introduce the concept of *parametric subscriptions* to support subscription adaptations. We propose novel algorithms for updating routing mechanisms effectively and efficiently in classic CPS broker overlay networks. Compared to re-subscriptions, our algorithms significantly improve the reaction time to subscription updates and can sustain higher throughput in the presence of high update rates. We convey our claims through implementations of our algorithms in two CPS systems, and by evaluating our algorithms on two different real-world applications.

1 Introduction

By focusing on the *exchanges* among interacting parties rather than the parties themselves, the *publish/subscribe* interaction paradigm [1] is very attractive for building scalable decentralized applications. This dynamic interaction culminates in *content-based* publish/subscribe (CPS), where subscriptions are based on event *content* rather than on channels or topics.

1.1 Content-based Publish/Subscribe and Subscription Adaptations

Although current CPS systems are *dynamic* in the way they support the joining and leaving of publishers and subscribers, they fall short in supporting subscription *adaptations*, which are becoming increasingly important to many CPS applications. Consider high frequency trading (HFT), which as of 2009, accounts

* This research is supported, in part, by the National Science Foundation (NSF) under grant #0644013 and project #0834529. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

for 73% of all US equity trading volume [2]. A typical subscription to IBM stock quotes with values below a specific threshold could be expressed through a CPS API as CPS.subscribe("IBM", "price < 10.0"), and could be used to trigger purchases. But, HFT uses various techniques to determine and update price thresholds continuously during the trading day – from simple linear regression to game theory, neural networks and genetic programming. HFT typically thrives precisely on rapid adaptations in subscriptions such as rectifications of thresholds for issuing buying or selling orders [3,4,2]. Hence, the speed with which a CPS system reacts to subscription adaptations is vital to the HFT application using it.

Another emerging family of applications inherently requiring subscription adaptations are mobile location-aware applications (location-specific advertising, location-based social networks like loopt[1], etc.). In such applications, a subscription is a function of the subscriber location such as a perimeter surrounding the subscriber's location (GPS coordinates). Whenever the device moves, the subscription needs to adapt.

Current solutions for subscription adaptations can be categorized as follows:

Ad-hoc solutions: In location-based services, updates on locations (only) are typically handled in an ad-hoc manner by specific middleware solutions which handle context separately from event content [5], and by using location information along with time stamps [6] .

Wildcards: The simplest approach from a programmers' perspective to support adaptations on content-based subscriptions is to use wildcard matching for respective event attributes, leading to universal subscriptions reminiscent of topic-based subscriptions. In the HFT example, this simply means subscribing to all stock tickers for IBM or even to all stock tickers if the company of interest may vary. This wastes bandwidth – it may not matter for someone investing only in IBM stock, or even a few tech stocks, but is not an option for portfolio managers dealing with hundreds or even thousands of stocks and commodities.

Re-subscription: The common solution to adapt a subscription consists of a *re-subscription*, where a new, parallel, subscription is issued and the superseded one is canceled. This solution has several limitations. First, it is coupled with high overhead which may lead to missing many events in the transition phase. If the frequency of subscription adaptations is high, as in HFT, the bulk of the computational resources of event brokers in a CPS is spent on processing re-subscriptions rather than filtering events and routing them to interested subscribers. This leads to drastic drops in throughput and increased latency overall. Second, in the absence of synchronization of (un-)subscriptions in most CPS engines, the application must cater for duplicates if the old and new subscription overlap which is usually the case.

[1] `www.loopt.com`

1.2 Parametric Subscriptions

Since these solutions all have clear limitations, we propose the concept of *parametric subscriptions* — subscriptions with dynamically varying parameters — to capture the aforementioned subscription adaptations. Consider the HFT example. Intuitively, we would like to express subscriptions à-la CPS.subscribe("IBM", "price < "+ **ref** threshold) where the value of the variable threshold can be updated dynamically by the program and its most current value is considered whenever inspecting a stock quote event for that subscription.

This immediately hints to the challenges in implementing parametric subscriptions. Simply passing the reference to threshold throughout the network means that nodes filtering events on behalf of the subscriber would access the variable, introducing failure - and performance dependencies.

1.3 Contributions

This paper tackles the problem of subscription adaptation in CPS broker overlay networks (CPSNs) through the following technical contributions:

- We introduce the concept of parametric subscriptions and discuss feasible and desired properties of corresponding solutions.
- We propose novel algorithms for updating routing mechanisms in CPSNs based on the original concept of *broker variables* to avoid global variable references (between publishers/subscribers) and thus global dependencies.
- To demonstrate the applicability and the efficacy of parametric subscriptions and our algorithms in CPSNs that use different algorithms for matching events to subscriptions, we evaluate two implementations of our algorithms, one in the well-known Siena [7] CPSN, and a second one in our own CPSN which uses the Rete algorithm for event matching. Our evaluation includes two benchmark applications, namely (1) algorithmic trading, and (2) a highway traffic control system, and a scalability analysis. Compared to re-subscriptions, our approach in both systems significantly improves the reaction time to subscription changes (up to 6×), reduces the load on subscribers by reducing the number of stale events delivered (up to 6×), and allows to sustain higher throughput (up to 8×) .

Roadmap. Section 2 presents background information and related work. Section 3 introduces parametric subscriptions and feasible properties. Section 4 describes our CPSN algorithms. In Section 5 we introduce two implementations and evaluate them. Section 6 concludes with final remarks.

2 Background and Related Work

Content-based publish/subscribe (CPS) promotes *content-based routing* to deliver events produced by *publishers* to *subscribers* with appropriate *subscriptions*. That is, the routing of an event in CPS is guided exclusively by its content.

Most CPS systems employ a network of interconnected *event brokers* to mediate events between *client processes*, i.e., to route events from the publishers to the appropriate subscribers. We refer to such a network as a *content-based publish/-subscribe network* (CPSN). Examples of existing CPS systems based on CPSNs are Siena [7], HERMES [8], REBECA [11], Gryphon [12], or PADRES [13].

2.1 Handling Subscriptions in CPSNs

Siena [7] introduces a covering-based scheme known as *subscription subsumption* – an elementary predicate (*attribute-value constraint*) in a subscription is said to be subsumed by that of another if the attributes are the same and the bound in the latter is more lax. *Subscription summarization* [14] builds on subscription subsumption by propagating only subscription summaries to brokers. New subscriptions are independently merged to their respective summary structures. Several systems use concepts similar to subsumption and summarization. REBECA [11] for instance uses subscription subsumption by *merging* filters in a way yielding a linear execution time irrespective of the number of subscriptions. In merging based routing, a broker merges the filters of existing routing entries and forwards them to a subset of its neighboring brokers. A perfect merging based algorithm generates perfect mergers and additionally ensures that the generated mergers are forwarded in a way such that only interesting notifications are delivered to a broker. Li et al [13] propose subscription covering, merging, and content matching algorithms based on binary decision diagrams (BDDs) in PADRES. HERMES [8] provides content-based filtering on top of type- and attribute-based routing and makes use of a distributed hash-table to orchestrate processes. Jafarpour et al. [10] present a new CPS framework that accommodates richer content formats including multimedia publications with image and video content. The work presented in [10] is orthogonal to this paper, though we anticipate future extensions of our approach to handle richer content. Jafarpour et al. [9] present a novel approach based on negative space representation for subsumption checking and provides efficient algorithms for subscription forwarding in CPSNs. The proposed heuristics for approximate subsumption checking greatly enhance the performance without compromising the correct execution of the system and only adding incremental cost in terms of extra computation in brokers.

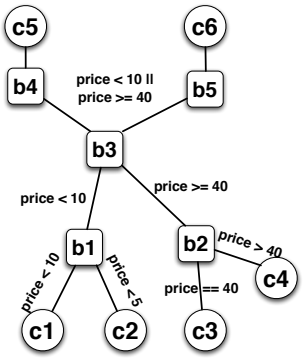

Fig. 1: Example of a CPSN.

Subscription summarization can attenuate the overheads of joining and leaving subscribers [14], but for updates the improvements are more a side-effect and insufficient. Our support proposed later-on is amenable to most CPS systems.

2.2 Alternative Implementation Strategies and Models

Astrolabe [15] is an example of an alternative category of CPS systems. With an emphasis on fault tolerance, processes in Astrolabe periodically exchange membership information with their peers. This information includes interests of processes, which is aggregated based on physical or logical topology constraints. Processes are selected to represent others based on the same criteria, leading to an overlay hierarchy reducing memory complexity on processes. This approach attempts to avoid dedicated brokers, but processes appearing high up in the hierarchy must handle high loads which probably exceed the capacities of regular desktop machines. The proactive gossiping about interests inherently propagates changes, but incurs a substantial overhead if none occur.

Meghdoot [16] is a CPS system that 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.

In *topic-based* publish/subscribe, topics represent the interests of subscribers that receive all events pertaining to the subscribed topic. Each topic corresponds to a logical channel that connects each publisher to all interested subscribers. Examples of topic-based publish/subscribe systems include SCRIBE [17], Bayeux [18], and Spidercast [19]. The topic-based publish/subscribe model provides less expressiveness than the content-based one.

3 Parametric Subscriptions

This section presents our model of parametric subscriptions as well as desired and feasible properties for corresponding support.

3.1 Model

An event e is of a certain type τ comprising a sequence of named attributes $[a_1, \ldots, a_n]$ which are typically of primitive types. An event e can thus be viewed as a record of values $[v_1, \ldots, v_n]$ for the attributes of its type. We consider subscriptions Φ represented in disjunctive normal form following a BNF grammar:

$$\begin{array}{llll} \textit{Subscription} & \Phi ::= \Phi \vee \Psi \mid \Psi & \textit{Predicate} & P ::= a\ op\ v \\ \textit{Conjunction} & \Psi ::= \Psi \wedge P \mid P & \textit{Operator} & op ::= \leq \mid < \mid = \mid > \mid \geq \mid \neq \end{array}$$

Intervals or set inclusion can be expressed above by a conjunction of two predicates or a disjunction of equalities respectively.

To decide on the routing of an event $e = [v_1, \ldots, v_n]$, subscriptions Φ are evaluated on e, written $\Phi(e)$. We assume type safety, meaning that a subscription has a type τ and is never evaluated on an event e of type $\tau' \neq \tau$. A predicate $P{=}a_k\ op\ v$ is evaluated as $P(e){=}v_k\ op\ v$.Obviously, satisfying a conjunction

($\Psi = P_1 \wedge \ldots \wedge P_m$) requires satisfying each of its predicates ($\Psi(e) = \bigwedge_{l=1}^{m} P_l(e)$), and a disjunction ($\Phi = \Psi_1 \vee \ldots \vee \Psi_s$) is satisfied by any of its conjunctions ($\Phi(e) = \bigvee_{r=1}^{s} P_r(e)$). We say that subscription Φ *covers* Φ', denoted by $\Phi' \preceq \Phi$, iff $\forall e\ \Phi'(e) \Rightarrow \Phi(e)$.

A *parametric* subscription, in addition, allows predicates to compare event attributes a to *variables* x local to the respective processes. This addition leads to the following extended definition of predicates substituting the one above:

$$\textit{Predicate} \quad P ::= a\ op\ v \mid \underline{a\ op\ x}$$

As variables x are time-sensitive, the evaluation of a subscription Φ is no longer only parameterized by an event e, but also by a time t: $\Phi(e, t)$. This evaluation takes place on variables at that point in time: $x(t)$.

3.2 Example

The expression and management of variables in parametric subscriptions can be made by the means of an API. Perhaps a more concise way of illustrating the use of variables is through a programming language. In EventJava [20] for instance, events are represented by specific, asynchronously executed, *event methods* preceded by the keyword **event**. Content-based subscriptions are defined by *guards* on these methods, following the **when** keyword. Guards can refer to event method arguments (event attributes a) and specific fields (variables x) of the subscriber object. Events can be published by invoking them like **static** methods on classes or interfaces declaring them. Consider the algorithmic trading scenario below. A stock quote can be published as StockMonitor.stockQuote(...). Now we can trigger a reaction when the stock price of IBM drops below the lowest previous value:

```
class StockMonitor {
  float lastBuy = ...;
  ...
  event stockQuote(String firm, float price)
    when (firm == "IBM" && price < lastBuy) {
      lastBuy = price;
      // e.g. issue purchase order
    }
}
```

Being a field of StockMonitor, lastBuy can be modified in other parts of the class than the body of StockQuote. Tracking such changes requires language support but mostly requires distributed runtime support for propagating them.

3.3 Desired Properties

Just like we represent parametric subscriptions with a temporal dimension, we can characterize events with a time of production. With Φ_i referring to the subscription of a process p_i, we can define the following guarantees on delivery of

events in response to parametric subscriptions. Assume a process p_i's subscription does not change after a time t_0, i.e., $\forall e, \forall t \geq t_0\ \Phi_i(e,t)$ or $\forall t \geq t_0\ \neg\Phi_i(e,t)$:

STRICTNESS: Process p_i delivers no event e published at time $t' \geq t_s \geq t_0$ if $\neg\Phi_i(e,t_0)$.

COVERAGE: An event e published at time $t' \geq t_c \geq t_0$ is eventually delivered by p_i if $\Phi_i(e,t_0)$.

Intuitively, STRICTNESS captures a possible *narrowing* underlying a subscription update: if the conditions become tighter in one place there is a time t_s after which no more events falling exclusively into the outdated broader criteria will be delivered. COVERAGE captures a *broadening*: after some time t_c no more events of interest are missed. A subscription which "switches", such as an equality '=' for which the target value changes, can be viewed as a combination of a broadening (include the new value) and a narrowing (exclude the old value).

3.4 Practical Considerations

STRICTNESS and COVERAGE represent safety and liveness and may compete which each other. A system which never delivers any event to any process trivially ensures STRICTNESS for $t_s{=}t_0$ but fails to ensure COVERAGE. Conversely, a system which delivers every event to every process ensures COVERAGE for $t_c{=}t_0$ but not STRICTNESS. STRICTNESS can be achieved by the means of local filtering mechanisms. In fact, we can get t_s arbitrarily (making use of local synchronization) close to t_0 by fully evaluating a subscription $\Phi_i(e,t)$ locally on a subscriber process p_i at the last instance before possibly delivering any event e to it. Relying *solely* on such a mechanism for filtering leads to many *spurious events* being routed all the way to p_i and thus does not constitute an ideal solution. More interesting are solutions which filter en route, like CPSNs. Yet, in *asynchronous* distributed systems it is impossible for a process to inform another one of new interests in bounded time, so there is no bound on t_c-t_0 in a CPSN. However, we can investigate solutions which *in practice* yield small values for t_c-t_0.

In practice, subscriptions that change over time may of course change more than once. In a sequence of successive changes, intermittent values might get skipped or their effects might not become apparent because no events arrive during their (short) period of validity. This can not be systematically avoided in the absence of lower bounds on transmission delays. A particularly interesting case arises if a variable switches back and forth between two values v_1 and v_2 (or more), e.g., $v_1 \cdot v_2 \cdot v_1 \ldots$. Events *delivered* in response to the second epoch with v_1 might very well have been *published* during the epoch of v_2 but before the first switch to v_2 had successfully propagated throughout the network. An important property which may be masked by such special cases is that any visible effects of changes in subscriptions appear in the order of the changes.

4 Algorithms

This section outlines a simple algorithm based on subscription subsumption/-summarization and then presents our algorithms for parametric subscriptions.

4.1 CPSN Model

We assume in the following a CPSN which uses dedicated broker processes b_i to convey events between client processes c_i. Brokers are interconnected among themselves. Brokers which serve client processes are called *edge* brokers. For simplicity we assume the absence of cycles in the broker network and a single process p_i per network node. Processes communicate via pairwise FIFO reliable communication channels offering primitives SEND (non-blocking) and RECEIVE. We assume failure-free runs; fault tolerance can be achieved by various means which are largely orthogonal to our contributions. Client processes PUBLISH events and DELIVER events corresponding to their subscriptions (SUBSCRIBE, UNSUBSCRIBE). For presentation simplicity, clients issue at most one subscription.

4.2 Static Subscriptions

The client primitives are illustrated in the simple client algorithm for the case of static subscriptions, i.e., without any variables x, in Figure 2. Figure 3 outlines the corresponding broker process algorithm. All primitives (e.g., **upon**) execute atomically and in order of invocation.

Algorithm. A broker stores processes that it perceives as subscribers in *subs*, and those that it acts as subscriber towards in *pubs*. It uses the covering relation ($\preceq$) to construct a *partially ordered set* (poset) $\mathcal{P}[\tau]$ of predicates of type τ received. The algorithm uses two elementary operations:

- INSERT$(\mathcal{P}[\tau], \Phi)$ is used to insert Φ into the poset $\mathcal{P}[\tau]$, which is ordered with respect to $\preceq$.
- DELETE$(\mathcal{P}[\tau], \Phi)$ is used to remove Φ from poset $\mathcal{P}[\tau]$.

The *least upper bound* (LUB) of $\mathcal{P}[\tau]$ is the predicate that covers all other predicates. If no LUB exists, this predicate — dubbed LUB$(\mathcal{P}[\tau])$ — is computed as a disjunction of all predicates that are not already covered by another predicate. All events of type τ that don't satisfy LUB$(\mathcal{P}[\tau])$ are discarded by the broker and events that satisfy individual subscriptions are forwarded to the corresponding subscribers. In practice, it is the poset that is "evaluated" on the event to avoid repetitive evaluation among predicates ordered in the poset.

Unadvertisements, the analogous to unsubscriptions, are omitted for brevity. They are simpler to handle than unsubscriptions as posets remain unchanged.

CPSN client algorithm. Executed by client c_i

```
1: init
2:     b                                   {edge broker}
3:     for all published type τ do
4:         SEND(AD,τ) to b

5: to PUBLISH(e) of type τ do
6:     SEND(PUB,τ,e) to b

7: to UNSUBSCRIBE(Φ) from type τ do
8:     SEND(USUB,τ, Φ) to b

9: to SUBSCRIBE(Φ) to type τ do
10:    SEND(SUB,τ, Φ) to b

11: upon RECEIVE(PUB, τ, e) do
12:    DELIVER(e)
```

Fig. 2: Simple client algorithm. The client is instantiated with an edge broker. Updating a subscription goes through unsubscribing the outdated subscription and issuing a new one (or vice versa).

CPSN broker algorithm. Executed by broker b_i

```
1: init
2:     pubs[]              {Indexed by event types τ}
3:     P[]                 {Indexed by event types τ}
4:     subs[][]            {Indexed by τ and Φ}

5: upon RECEIVE(AD, τ) from p_j do
6:     pubs[τ] ← pubs[τ] ∪ {p_j}
7:     SEND(AD, τ) to
           all b_k ∈ ⋃_Φ subs[τ][Φ] ∪ pubs[τ]\{p_j}

8: upon RECEIVE(PUB, τ, e) from p_j do
9:     if LUB(P[τ])(e) then
10:        for all Φ ∈ P[τ] do
11:            if Φ(e) then
12:                SEND(PUB, τ, e) to all p_k ∈ subs[Φ]

13: upon RECEIVE(SUB, τ, Φ) from p_j do
14:    subs[τ][Φ] ← subs[τ][Φ] ∪ {p_j}
15:    Φ_old ← LUB(P[τ])
16:    INSERT(P[τ], Φ)
17:    if Φ_old ≠ LUB(P[τ]) then
18:        SEND(SUB, LUB(P[τ])) to all b_k ∈ pubs
19:        SEND(USUB, LUB(P[τ])) to all b_k ∈ pubs

20: upon RECEIVE(USUB, τ, Φ) from p_j do
21:    subs[Φ] ← subs[Φ] \ {p_j}
22:    Φ_old ← LUB(P[τ])
23:    DELETE(P[τ], Φ)
24:    if Φ_old ≠ LUB(P[τ]) then
25:        SEND(SUB, LUB(P[τ])) to all b_k ∈ pubs
26:        SEND(USUB, LUB(P[τ])) to all b_k ∈ pubs
```

Fig. 3: Algorithm for event processing in a CRN with subscription summarization. $\mathcal{P}[\tau]$ is the predicate poset ordered by $\preceq$. $pubs[\tau]$ stores the advertising peers. $subs[\tau][\Phi]$ stores peers that subscribe to Φ. $subs[\tau][\Phi]$ avoids the need to duplicate Φ in $\mathcal{P}[\tau]$, if more than one peer subscribes with Φ.

Illustration. Figure 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 IBMStockQuote with one attribute a=price. Figure 4 shows a part of the CPSN, and how subscriptions propagate. c_1 subscribes to IBMStockQuote with the predicate Φ_1=(price < 10)[2]. b_1 gets the subscription, stores it and propagates it to b_3. Then c_2 subscribes with predicate Φ_2=(price < 5). b_1 gets this subscription, but does not forward it to b_3, because (price < 10) covers (price < 5) (since $\Phi_2 \preceq \Phi_1$). Figure 1 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 summarizes the "summaries" from b_1 and b_2.

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($\mathcal{P}$[IBMStockQuote]) changes to (price < 30), b_1 unsubscribes from (price < 5) and subscribes to (price < 30). Figure 4 shows how the poset of predicates changes at b_1 and b_3. Calculating LUB($\mathcal{P}$[IBMStockQuote]) is shown in Figure 1

[2] Predicates are wrapped in parentheses for clarity.

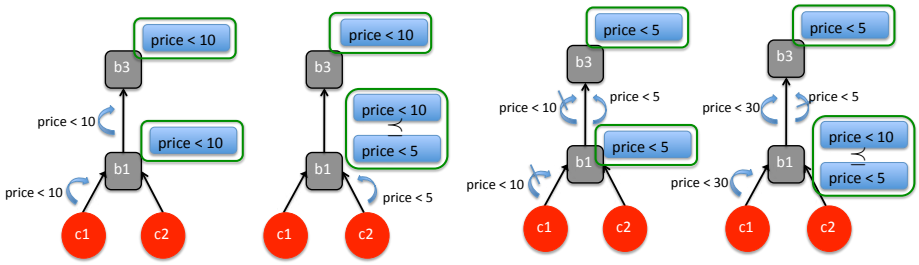

Fig. 4: Update propagation with re-subscriptions.

through an example. Subscriptions are routed to all brokers that have at least one publisher with a matching advertisement.

Subscription updates. When a subscriber c wants to update its subscription, it unsubscribes and re-subscribes. Unsubscription on a broker involves searching the poset $\mathcal{P}[\tau]$ for Φ, removing Φ from it, and readjusting it with respect to $\preceq$. If the poset is implemented as a $d-ary$ max-heap ordered with respect to $\preceq$, with d the maximum degree of the heap, readjusting is $O(|\mathcal{P}[\tau]|)$ [21]. The worst case occurs when Φ is the root of the poset and all other nodes are its children. Searching $\mathcal{P}$ is $O(|\mathcal{P}[\tau]|)$. Hence processing an unsubscription is $O(|\mathcal{P}|)$. Similarly, subscription (SUBSCRIBE(Φ)) involves searching $\mathcal{P}$ to check whether Φ already exists, in this case, c is simply added to the list of subscribers of Φ. If not, Φ is inserted into the $\mathcal{P}$. Insertion is $O(log_d|\mathcal{P}[\tau]|)$ [21]. If LUB($\mathcal{P}$) changes as a result of subscription/unsubscription, then the broker unsubscribes the old LUB($\mathcal{P}[\tau]$) and issues a fresh subscription with the new LUB($\mathcal{P}[\tau]$).

Note that a client might also want to issue a new subscription first, before unsubscribing, and filter any duplicates in the interim. In common CPSNs, both subscription and unsubscription operations are asynchronous though, providing no information on their penetration into the CPSN. A practical solution consists in canceling the outdated subscription upon reception of the first event which does *not* match the outdated subscription.

4.3 Supporting Parametric Subscriptions

We now outline a solution to supporting variables x in subscriptions.

Algorithm. Figure 5 describes the new client algorithm as extension to that of Figure 2. Besides the addition of a reaction to changes of variables appearing in a subscription Φ, the algorithm performs additional local evaluation of Φ on a client to enforce STRICTNESS, as the view of it's end broker may be lagging.

The broker algorithm shown in Figure 6 follows the same structure as the previous broker algorithm. The main differences are that nodes in the poset are now

CPSN client algorithm with parametric subscriptions for c_i. Reuses lines 1-8 of algorithm in Figure 2

```
 9: upon RECEIVE(PUB, τ, e) do
10:     if Φ(e, current time) | Φ is on τ then
11:         DELIVER(e)

12: to SUBSCRIBE(Φ) to τ do
13:     x̄, v̄ ← vars in Φ and respective vals
14:     SEND(SUB,τ, Φ, x̄, v̄) to b

15: upon change of variable x to v in Φ do
16:     SEND(UPD,τ, Φ, x, v) to b
```

Fig. 5: Client algorithm with support for parametric subscriptions.

tuples of the form $(\Phi^{\mathrm{V}}, \Phi, \overline{(x, v)})$ where Φ is the original predicate without values substituted for variables, and Φ^{V} after substitution (e.g. line 11 of Figure 6). $\overline{(x, v)}$ is a set of mappings of values v_i for variables x_i. Furthermore, poset additions (INSERT) and removals (DELETE) are now parameterized by nodes. These changes lead to two new primitives being used in the algorithm:

- SUBSTITUTE(v, x, Φ) denotes the substitution of v for x in Φ. This primitive is also used by brokers to substitute variables of neighbors against their own.
- UPDATE$(\mathcal{P}[\tau], node, x, v)$ (see line 45) updates *node within* the poset, by adopting v as new value for x in the substitution to v, re-performing the variable substition, storing the updated predicate in the node, and re-ordering the poset if needed. Poset ordering is based on Φ^{V}; if two predicates need to be disjoined, the corresponding variable mappings are merged.

lookup[...][x] stores identifiers of nodes containing respective variables x for fast lookup and modification upon incoming update messages. Since such variables are always specific to a single predicate, they are introduced by one node. Disjunctions created for summarization will indirectly be modified by updates to such introducing nodes. Similarly, due to variables in subscriptions, there is now never more than one subscriber stored for a given predicate Φ in *subs*[...][Φ]. This can be overcome in practice by variable substitution.

Procedure PROPAGATE captures the common part of all subscription modifications – new subscriptions, unsubscriptions, updates. It compares the root node of the poset ($node_0$, e.g. line 36) with the root node after modification ($node_\nu$, line 34), and initiates corresponding transitive updates. Hence, subscriptions/unsubscriptions are reduced, and when an update message arrives, a hash table based index can for example be used to guarantee a $O(1)$ bound on updates with *lookup*.

Last but not least, PROPAGATE illustrates the concept of *broker variables* (*brokervars*, see line 24). These limit the scope of variables to a client and its edge broker or to a broker and its immediate neighbors thus avoiding global dependencies. When a new subscription is sent to a neighbor broker, variables in the root predicate Φ of the poset $\mathcal{P}$[...] are substituted by freshly chosen ones.

Illustration. The main difference to a CPSN without parametric subscriptions is illustrated in Figure 7, which contrasts with Figure 4. In Figure 7, updating a subscription involves unsubscribing the old one (price < 10) and issuing a new

CPSN broker algorithm supporting parametric subscriptions. Executed by broker b_i.

```
1: init
2:     pubs[]                    {Indexed by event types τ}
3:     P[]                       {Indexed by event types τ}
4:     subs[][]                  {Indexed by τ and Φ}
5:     brokervars[]              {Indexed by τ}
6:     lookup[][]                {Indexed by τ and var x}

7: upon RECEIVE(AD, τ) from p_j do
8:     pubs[τ] ← pubs[τ] ∪ {p_j}
9:     SEND(AD, τ) to
       all b_k ∈ ⋃_Φ subs[τ][Φ] ∪ pubs[τ]\{p_j}

10: upon RECEIVE(SUB, τ, Φ, (x, v)) from p_j do
11:     Φ^V ← SUBSTITUTE(v̄, x̄, Φ)        {Var subst}
12:     subs[τ][Φ] ← p_j                  {At most 1}
13:     node ← (Φ^V, Φ, (x, v))
14:     for all (x, v) ∈ (x, v) do
15:         lookup[τ][x] ← ref node       {Store ref}
16:     node_0 = (Φ_0^V, Φ_0, (x^0, v^0)) ← LUB(P[τ])
17:     INSERT(P[τ], node)
18:     node_ν = (Φ_ν^V, Φ_ν, (x^ν, v^ν)) ← LUB(P[τ])
19:     PROPAGATE(node_0, node_ν)

29: upon RECEIVE(USUB, τ, Φ) from p_j do
30:     subs[τ][Φ] ← ∅
31:     node ← (Φ^V, Φ, (x, v)) ∈ P[τ]
32:     for all x ∈ x̄ do
33:         lookup[τ][x] ← ⊥
34:     node_0 = (Φ_0^V, Φ_0, (x^0, v^0)) ← LUB(P[τ])
35:     DELETE(P[τ], node)
36:     node_ν = (Φ_ν^V, Φ_ν, (x^ν, v^ν)) ← LUB(P[τ])
37:     PROPAGATE(node_0, node_ν)

38: upon RECEIVE(PUB, τ, e) from p_j do
39:     for all node = (Φ^V, Φ, (x', v)) ∈ P[τ] do
40:         if Φ^V(e) ∧ subs[τ][Φ^V] ∉ {⊥, p_j} then
41:             SEND(PUB, τ, e) to subs[τ][Φ^V]

42: upon RECEIVE(UPD, τ, x, v) from p_j do
43:     node_0 = (Φ_0^V, Φ_0, (x^0, v^0)) ← LUB(P[τ])
44:     node_upd ← deref lookup[τ][x]
45:     UPDATE(P[τ], node_upd, x, v)
46:     node_ν = (Φ_ν^V, Φ_ν, (x^ν, v^ν)) ← LUB(P[τ])
47:     PROPAGATE(node_0, node_ν)

20: procedure PROPAGATE((Φ_0^V, Φ_0, (x^0, v^0)), (Φ_ν^V, Φ_ν, (x^ν, v^ν)))
21:     if Φ_ν^V ≠ Φ_0^V then                {Different concrete subscriptions}
22:         if Φ_ν = Φ_0 then                {Same structure and variables}
23:             for all v^ν ≠ v^0 do         {Can be regrouped}
24:                 SEND(UPD, τ, brokervars[τ], v^ν) to all b_k ∈ pubs[τ]
25:         else
26:             brokervars[τ] ← fresh x_1 ... x_n | x^ν = x'_1 ... x'_n
27:             SEND(SUB, τ, SUBSTITUTE(brokervars[τ], x^ν, Φ_ν), (x, v^ν)) to all b_k ∈ pubs[τ]
28:             SEND(USUB, τ, Φ_0) to all b_k ∈ pubs[τ]
```

Fig. 6: Broker algorithm for parametric subscriptions. Common handling of poset updates (new subscriptions, unsubscriptions, updates) are regrouped in PROPAGATE.

subsciption (price < 30). In a CPSN with parametric subscriptions, Φ contains binary predicates, some of which involve local variables. Each subscription message sent to a broker now must include the values of the variables used in the subscription. However, changing a subscription doesn't necessarily lead to an unsubscription and a re-subscription. The subscriber (c_1 in Figure 7, for example) merely specifies the name of the variable and its new value.

In Figure 7, when client c_1 subscribes to price $< c_1.x$, the variable $c_1.x$ is shared between c_1 and b_1. When b_1 propagates the subscription to b_3, $c_1.x$ is mapped to $b_1.x$, which is shared between b_1 and b_3. Note that, in Figure 7, updating the value of $c_1.x$, doesn't change the structure of the predicate involved. Also, new variables are introduced (by the variable mapping algorithm) only at those binary predicate containing variables. If a predicate has binary predicates with constants, a change to a constant will result in an unsubscription and a re-subscription instead of an update. To avoid this, we can go a step further and replace all values in binary predicates by variables (omitted for simplicity). A single update message can then be used instead of two messages (subscription/unsubscription) in further cases.

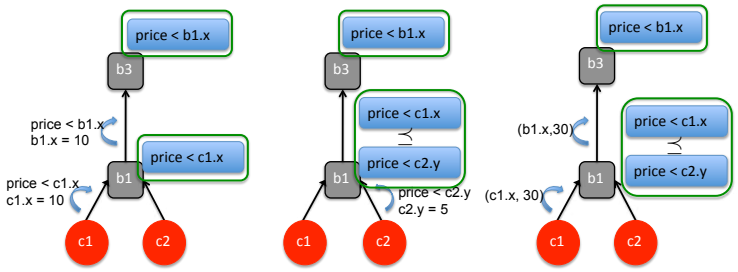

Fig. 7: Update propagation with support for parametric subscriptions.

5 Evaluation

In this section, we first introduce two implementations of our algorithms, namely as an extension to Siena [7] and in our own CPSN implemented in EventJava based on Rete [22]. While our own CPSN yields much higher throughput than Siena, it requires more resources which is why we compare both implementations against their respective extensions on both benchmarks. Siena was chosen instead of other systems because it is the only publicly available open source CPSN with acceptable performance. The source code was necessary because we had to implement our algorithms in existing systems to measure the gains in performance due to our proposal.

5.1 Implementation

The goal of our experimental evaluation is to demonstrate improvements in performance due to parametric subscriptions. Hence, for both benchmarks, we compare two "bare" CPSN — making use of re-subscriptions — against respective extensions following our proposal.

EventJava. EventJava is an extension of Java for generic event-based programming which supports the expression of event correlation, multicast, asynchronous event consumption (subscriptions) as well as synchronous consumption (message queuing) in an integrated manner. EventJava is implemented as a framework, with substitutable runtime components for event propagation, filtering, and correlation. Parametric subscriptions are supported naturally in EventJava as expressed in the example in Section 3.2, by allowing fields of subscriber objects to be used in event method guards.

We have extended the EventJava [20] compiler to track changes in the values of variables used in parametric subscriptions. The compiler translates EventJava to standard Java together with calls to the framework components, instrumenting assignments to relevant fields in order to issue UPD messages. It relies on a specialized static analysis, leading to the following steps:

1. Identify all fields used in subscriptions, all assignments to such fields.
2. Inject code to issue an UPD message after the assignment.
3. Protect this assignment together with the sending of the UPD message by a field-specific lock added to the respective class. This ensures that the update occurs in mutual exclusion with respect to other instrumented assignments to the same field, preventing race conditions/lost updates.

To ensure completeness of the static analysis, fields that can be used in guards are currently limited to **protected** and **private** fields of primitive types, e.g. **float**.

UPDSiena. We extended the Java Siena implementation to support a new message type named UPD (update) sent from subscribers to edge brokers and from edge brokers to the their neighboring brokers. When defining a predicate a user can optionally specify a variable for each of his binary predicates which will be later used to update the predicate in the broker network. This API can be used directly without EventJava. The class HierarchicalSubscriber implementing broker functionality was modified to create a new set of variables once a new predicate gets added to the root of a poset analogously to what is described in Figure 6. These can be used to update the subscription with the parent broker. Other classes modified include Poset, Filter, and ThinClient. Java applications can exploit our parametric subscription in UPDSiena as well as in our EventJava CRN through APIs, i.e., independently of EventJava.

5.2 Metrics

To assess COVERAGE and STRICTNESS (see Section 3.3) we use three metrics:

Delay: To approximate COVERAGE we measure the *delay* between an update and the reception of the first corresponding event. If a subscriber c_i changes its subscription Φ_i to Φ_i' at time t_0, and the first event matching Φ_i' but not Φ_i is delivered at time t_1, then the delay at subscriber c_i is defined as t_1-t_0.

Throughput: To gauge the load imposed on the system to achieve STRICTNESS by update propagation, we first evaluate *throughput* in the presence of an increasing amount of updates. More precisely we consider is the average number of events *delivered* by a subscriber per second. This throughput depends on the number of publishers, event production rates at each publisher, the selectivity of the subscriptions of the subscribers, and the rate at which each subscriber updates its subscriptions. Selectivity of a subscription is the probability that an event matches a subscription. A selectivity of 1.0 implies that a subscription is satisfied by every published event of the respective type and a selectivity of 0.0 implies that none do.

Spurious events: The effect of inefficient updates might be offset if brokers are powerful dedicated servers or individual clients are only interested in few events to start with. Increased stress might otherwise manifest, especially on resource-constrained clients. To gauge this stress, we measure the amount of spurious events delivered by clients. If a subscriber c_i changes its subscription

Φ_i to Φ'_i at time t_0, then spurious events are those matching Φ_i but not Φ'_i and received by the client *after* t_0 and filtered out locally to it (see line 10 in Figure 5). These capture the overhead imposed on clients.

5.3 Infrastructure

All brokers were executed on dual core Intel Xeon 3.2Ghz machines with 4GB RAM running Linux, with each machine executing exactly one broker. Subscribers were deployed on a 16-node 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 in many cases across LANs. 10msec delays were added to each network link to simulate wide area network characteristics as is done in EmuLab [3].

5.4 Highway Traffic Management (HTM)

Publish/subscribe systems have been used in several traffic management systems, the best example being the Tokyo highway system [23,24].

Scenario. Such a system consists of a CPSN with several sensors and cameras located at various points along the highway, monitoring road conditions, traffic density, speeds, temperature, rainfall, snow etc. So publishers are the various sensors and the subscribers are vehicles, and traffic monitoring stations. Consider a vehicle equipped with a GPS-based navigation system driving through the highway – many contemporary vehicles have touchscreen navigation systems with nearly real-time traffic information. Typically, the navigation system is interested in traffic density in the geographic area around it – an example being red, yellow and green colored highways in Google Maps [4]

The navigation system uses this information to plot alternate routes — with minimum traveling time — to the destination. Each sensor connects to one broker, and publishes events to the CPSN. While traveling a portion of the road covered by a broker, a car navigation system connects to the broker and subscribes to events of interest, parameterized by current location (GPS coordinates). The location of a moving car changes constantly and thus the navigation system updates its subscriptions periodically, or as initiated by the driver. Brokers in an HTM system are usually interconnected by a wired network.

[3] `http://www.emulab.net`
[4] `http://maps.google.com`. Select a U.S. city and click on "Traffic".

Setup. To evaluate our algorithms, we used a traffic management CPSN based on [23] with 20 brokers, and 10 publishers per broker, resulting in a total of 200 publishers. The rate of subscription updates is dependent on the following parameters: (1) the length of highway controlled by a broker (*Highway-length*), (2) periodicity of subscription updates by the navigation system (*Periodicity*), and (3) average number of appropriately equipped vehicles on the stretch of highway controlled by a broker (*Vehicles*). In an urban setting, if *Highway-length* = 10 miles, *Periodicity* = 1 update/minute, and *Vehicles* = 1000, then the number of subscribers attached to one broker is 1000. During any 1 minute interval, each of the 1000 cars updates its subscription, hence the number of updates/second is 1000/60 = 16.67 updates/second. One thousand subscribers means that the traffic density is 100 cars/mile of highway, which is sparse traffic. Assuming a six lane highway (3 lanes in each direction), traffic densities can easily reach 500 cars/mile (250 cars in either direction) during heavier traffic periods. Thus, frequency of subscription updates easily reaches 500x10/60 = 83.33 updates/second/edge broker. Hence, on this benchmark, we evaluate our algorithms with update frequencies ranging from 10 updates/second to 100 updates/second/edge broker. CPSNs in traffic management are not hierarchical, because highways around major urban cities are not hierarchical. Hence the only assumption on the CPSN used for this benchmark is that it is a connected undirected graph. The distribution of operators in subscriptions was 40% $\geq$, 40% $\leq$, and 20% =.

5.5 Algorithmic Trading (AT)

Algorithmic trading (AT) is the use of computer programs for entering trading orders, with the computer algorithm deciding on aspects of the order such as the timing, price, or quantity of the order, or in many cases initiating the order without human intervention.

Scenario. We consider the monitoring component of an algorithmic commodity trading system. By commodities we mean basic resources and agricultural products like iron ore, crude oil, coal, ethanol, salt, sugar, coffee beans, soybeans, aluminum, copper, rice, wheat, gold, silver, palladium, or platinum. We use a CPSN that disseminates commodity prices with 20 brokers, 5 publishers and 150 subscribers.

Metric	Incr. in throughput		Decr. in delay		Decr. in spurious events	
Benchmark	HTM	AT	HTM	AT	HTM	AT
UPDSiena vs. Siena	Fig. 8a up to ∼8×	Fig. 8g up to ∼4.4×	Fig. 8b up to ∼6×	Fig. 8h up to ∼3×	Fig. 8c up to ∼6×	Fig. 8i up to ∼6×
EJava (resub) vs. EJava	Fig. 8d up to ∼53%	Fig. 8j up to ∼25%	Fig. 8e up to ∼2×	Fig. 8k up to ∼4.4×	Fig. 8f up to ∼2.2×	Fig. 8l up to ∼3.4×

Table 1: Performance improvements for HTM and AT with parametric subscriptions

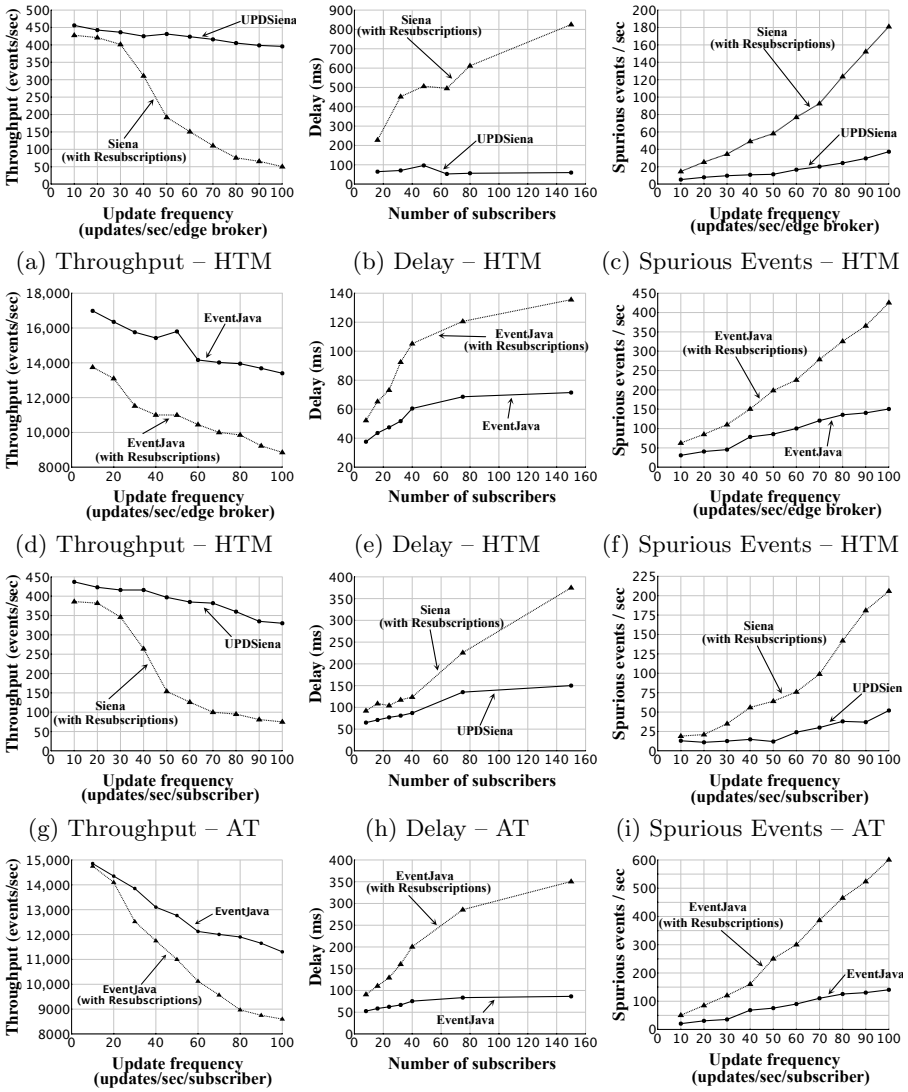

Fig. 8: Comparing re-subscriptions against parametric subscriptions in both algorithmic trading (AT) and highway traffic management (HTM) benchmarks for Siena and EventJava's Rete-based CPSN.

Setup. In AT, the number of publishers is small – commodity prices are published by commodity exchanges and stock quotes by stock exchanges. For this benchmark, we assume that a subscriber is a computer at an AT firm. Our benchmark had 200 event types, which includes the price quotes of 100 commodities, analyst predictions, etc. In the experimental setup used for this benchmark, we employed a hierarchical broker overlay network, which is typical in stock and commodity price quote dissemination. Stock and commodity markets publish quotes and information into a market data system, like DOWJONES newswires, Reuters Market Data Systems (RMDS), which are at the top of the hierarchy. At the next level are large clearing houses (e.g., Goldman Sachs, Merrill Lynch, J.P Morgan). The next level contains large brokerages and trading firms, to which small trading firms connect. In the overlay network used for this benchmark, publishers and subscribers are separated by at least 3 brokers. The distribution of operators was 35% $\leq$, 33% $\geq$, and 32%=.

5.6 Results and Analysis

The performance improvements of UPDSiena over Siena, and EventJava (with parametric subscriptions) over EventJava respectively are summarized in Table 1 and detailed in Figure 8. Apart from speedups we observe that:

1. The drastic drop in Siena's throughput in both benchmarks (Figures 8a and 8g) is due to the increase in time spent processing un-subscriptions/re-subscriptions – recall that both operations involve computing the least upper bound and rearranging subscriptions in a poset, the complexity of which is linear in the size of the poset. The same poset is used for event forwarding.
2. The throughput of EventJava degrades more gracefully with an increasing update frequency (Figures 8j and 8d) as opposed to the Siena (Figures 8g, 8a) because Rete constructs a *separate* event flow graph to "remember" events that partially match subscriptions in a poset, representing each subscription as a chain of nodes. Hence, event filtering and forwarding at the brokers is independent of the updates to the poset.
3. The drastic increase in the number of spurious events received per second by a Siena or an EventJava (re-subscriptions) subscriber corresponds to the increase in delay between a variable update and the receipt of the first matching event for both benchmarks.
4. The increase in the number of spurious events received by an EventJava subscriber ($>$ 100 spurious events per second) as opposed to UPDSiena (Figures 8f, 8l vs. Figures 8c, 8i) is due to (1) the high event matching throughput of Rete compared to Siena's algorithm, and (2) the presence of a separate event flow graph. Since a broker using Rete processes more events per second, more spurious events are delivered to subscribers in CPSNs using Rete before an update propagates to the broker.

5.7 Throughput Scalability

Given that the empirical evaluation in Sections 5.5 and 5.4 consider update

frequencies between 10 and 100, one obvious issue is the scalability of throughput with increasing number of updates. The original Siena does not scale beyond 100 updates/second/subscriber. Figure 9 shows that EventJava with parametric subscriptions retains a throughput of well above 9000 events/second, even when the update frequency per receiver is 1000 updates/second. However, the throughput of EventJava with re-subscriptions degrades faster and drops to 4000 events/second. At 1000 updates/second/-subscriber, the throughput of EventJava with parametric subscriptions is $\sim$2.25$\times$ that of EventJava with re-subscriptions. This experiment is independent of the benchmarks described in Sections 5.5 and 5.4, and used a programmatically generated (artificial) workload with 200 event types, 20 brokers, 100 publishers and 150 subscribers and a broker overlay which was a connected graph and non-hierarchical.

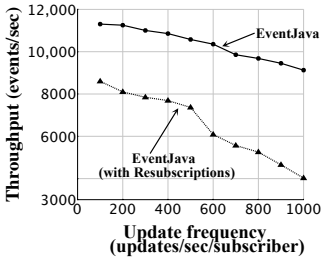

Fig. 9: Throughput scalability

6 Conclusions

The publish/subscribe paradigm supports dynamism by allowing new publishers as well as subscribers to be deployed dynamically. This ability allows applications to adapt online by issuing new subscriptions. The mechanisms used to that end are not geared towards important changes *within* subscriptions. We thus propose parametric subscriptions. Through the novel concept of broker variables our algorithms proposed in this paper and implemented in two CPSNs (and easily adapted to others) retain the scalability properties of common CPSNs.

We are currently investigating several extensions. For instance, we are considering uniformly representing all predicates based on $<, \leq$, or $=$ internally as range queries where the upper and lower bounds are implicitly variables, assuming minimum and maximum values for the respective data-types in the case of wildcards, and over-approximating summaries to normalize subscriptions at all levles. This allows us to easily support *structural* subscription updates, i.e., the *addition* of predicates. Furthermore, we are investigating approximation techniques for oscillating variables and prediction algorithms for high frequency monotonic variable changes.

References

1. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus - An Architecture for Extensible Distributed Systems. SOSP '93. 58–68
2. Lati, R.: The Real Story of Trading Software Espionage. AdvancedTrading.com `http://advancedtrading.com/algorithms/showArticle.jhtml?articleID=21840150`

3. The Economist: Moving Markets: Shifts in Trading Patterns are Making Technology Ever More Important. http://www.economist.com/business-finance/displaystory.cfm?story_id=E1_VQSVPRT 2006.
4. Aite Group: Algorithmic Trading: Hype or Reality? http://www.aitegroup.com/reports/20050328.php 2005.
5. Cugola, G., Margara, M., Migliavacca, M.: Context-aware Publish-Subscribe: Model, Implementation, and Evaluation. ISCC 2009. 875–881.
6. Schwiderski-Grosche, S., Moody, K.: The SpaTeC Composite Event Language for Spatio-temporal Reasoning in Mobile Systems. DEBS 2009. 1–12.
7. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide Area Event Notification Service. ACM TOCS **19**(3) (August 2001) 332–383
8. Pietzuch, P., Bacon, J.: Hermes: A Distributed Event-Based Middleware Architecture. ICDCS 2002 Workshops (DEBS 2002). 611–618
9. Jafarpour, H., Hore, B., Mehrotra, S., Venkatasubramanian, N.: Subscription Subsumption Evaluation for Content-based Publish/subscribe Systems. Middleware 2008. 62-81
10. Jafarpour, H., Hore, B., Mehrotra, S., Venkatasubramanian, N.: CCD: Efficient Customized Content Dissemination in Distributed Publish/Subscribe. Middleware 2009. 62-82
11. Fiege, L., Gärtner, F., Kasten, O., Zeidler, A.: Supporting Mobility in Content-based Publish/Subscribe Middleware. Middleware 2003. 103–122
12. Aguilera, M., Strom, R., Sturman, D., Astley, M., Chandra, T.: Matching Events in a Content-Based Subscription System. PODC'98. 53–62
13. Li, G., Hou, S., Jacobsen, H.: A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems based on Modified Binary Decision Diagrams. ICDCS 2005. 447–457
14. Triantafillou, P., Economides, A.A.: Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems. ICDCS 2004. 562–571
15. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. ACM TOCS **21**(3) (2003)
16. Gupta, A., Sahin, O.D., Agrawal, D., Abbadi, A.: Meghdoot: Content-Based Publish/Subscribe over P2P Networks. Middleware 2004. 254–273
17. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A Large-Scale and Decentralized Application-level Multicast Infrastructure. IEEE JSAC **20**(8) (October 2002) 100–110
18. Zhuang, S.Q., Zhao, B.Y., Joseph, A.D., Katz, R.H., Kubiatowicz, J.D.: Bayeux: An Architecture for Scalable and Fault-tolerant Wide-are Data Dissemination. NOSSDAV 2001.
19. Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Spidercast: a Scalable Interest-aware Overlay for Topic-based Pub/Sub Communication. DEBS 2007. 14–25
20. Eugster, P., Jayaram, K.R.: EventJava: An Extension of Java for Event Correlation. ECOOP 2009. 570–594
21. Cormen, T. H., Rivest, R., Leiserson, C., Stein, C.H., Introduction to Algorithms. MIT Press. 2009.
22. Forgy, C. L.. On the Efficient Implementation of Production Systems. PhD Thesis, Carnegie-Mellon University, 1979.
23. Schneider, S.: DDS and Distributed Data-centric Embedded Systems. *Dr. Dobb's Journal*. http://www.drdobbs.com/embedded-systems/196601852
24. Barnett, D. Publish-Subscribe Model Connects Tokyo Highways. *Industrial Embedded Systems*. http://www.industrial-embedded.com/articles/barnett/