Self-Evolving Subscriptions for Content-Based Publish/Subscribe Systems

César Canas[*], Kaiwen Zhang[†], Bettina Kemme[*], Jörg Kienzle[*], Hans-Arno Jacobsen,
[*]School of Computer Science. McGill University, Montreal, Canada
[†]Technische Universität München, Munich, Germany
[‡]University of Toronto, Toronto, Canada

***Abstract*—Traditional pub/sub systems cannot adequately handle the workloads of applications with dynamic, short-lived subscriptions such as location-based social networks, predictive stock trading, and online games. Subscribers must continuously interact with the pub/sub system to remove and insert subscriptions, thereby inefficiently consuming network and computing resources, and sacrificing consistency. In the aforementioned applications, we recognize that the changes in the subscriptions can follow a predictable pattern over some variable (e.g., time). In this paper, we present a new type of subscription, called *evolving subscription*, which encapsulates these patterns and allows the pub/sub system to autonomously adapt to the dynamic interests of the subscribers without incurring an expensive resubscription overhead. We propose a general model for expressing evolving subscriptions and a framework for supporting them in a pub/sub system. To this end, we propose three different designs for support evolving subscriptions, which are evaluated and compared to the traditional resubscription approach in the context of two use cases: online games and high-frequency trading. Our evaluation shows that our solutions can reduce subscription traffic by 96.8% and improve delivery accuracy when compared to the baseline resubscription mechanism.**

I. Introduction

Publish/subscribe is a communication paradigm widely employed to provide event dissemination between loosely coupled producers (publishers) and consumers (subscribers). Publishers submit publications which are matched and delivered by pub/sub agents (called brokers) to subscribers based on their registered subscriptions. Examples of popular systems include Facebook Wormhole [21], Google Cloud Pub/Sub, Redis, Apache Kafka [13], and Yahoo Pulsar.

In order to enable the operation of large-scale systems, publish/subscribe systems are commonly distributed over an overlay network of brokers for load balancing [6] and availability [14]. In this model, subscriptions are circulated among brokers to form delivery paths from publishers to subscribers [5]. Depending on the routing model employed, subscriptions are either flooded to all brokers or forwarded towards publishers with matching advertisements. In any case, there is no delivery guarantee provided for a subscription while it is being installed at the brokers. A subscriber who wishes to remove an existing subscription must send an unsubscribe message, which is again routed throughout the overlay. In addition, a typical publish/subscribe system provides no mechanism for modifying an existing subscription: the process of *resubscription* involves unsubscribing the existing subscription to be removed, and then sending a new subscription to be inserted. Thus, resubscription is a communication and computation intensive process which does not provide any delivery guarantee during its course of action.

In traditional pub/sub applications, such as stock-market monitoring [22] and RSS filtering [17], resubscriptions are not considered an issue since the subscriptions are considered static (i.e., the workload is *publication-heavy*). The focus is then put on matching and delivering a high throughput of events to these stable subscriptions. We have however identified certain modern publish/subscribe applications which produce *subscription-heavy* workloads. For instance, consider massively multiplayer online games, where players control avatars in a virtual world, subscribe to areas of interest, and receive updates about the game state. Content-based subscriptions are used to indicate interest in a specific location, which are then matched against publications occurring across the entire world. As the players move, their subscriptions must be updated to reflect their new position, thus incurring a resubscription process. Our previous paper has found that the resubscription overhead generated by moving players to be the main bottleneck for our content-based pub/sub engine in online games [4]. Other examples of subscription-heavy workloads include those of location-based social networks and predictive stock trading.

Even so, these highly dynamic workloads often have patterns in the way subscriptions are formed. In the case of games, a player repeatedly subscribes and unsubscribes when moving to reflect interest in the current space around him/her. From the point of view of the publish/subscribe system, these subscriptions appear to be independent and must be routed and processed separately. From the point of view of the game, it is clear that each successive subscription represents an incremental change from the previous one, which reflects the progressive nature of the movement of the player.

In this paper, we introduce the concept of *evolving subscriptions*. By exposing the pattern of subscription change found in the application semantics explicitly to the publish/subscribe system, it can autonomously update a subscription according to the prescribed pattern without requiring additional input from the subscriber. By doing so, evolving subscriptions avoid the communication and processing overhead otherwise incurred by frequent and expensive resubscriptions. In addition, our solution improves the delivery accuracy of publications while subscriptions are changing.

The purpose of this paper is to employ evolving subscriptions as a method to raise the scalability of content-based publish/-subscribe systems for applications with highly dynamic and patterned subscription workloads, such as online games. Our contributions are as follows:

1) We propose the model of evolving subscriptions, which extends content-based semantics to allow update patterns to be specified over well-defined variables such as time (Section III).
2) We provide three designs for supporting evolving subscriptions: Versioned Evolving Subscriptions (VES), Lazy Evaluation Evolving Subscriptions (LEES), and its cached variant (CLEES) (Section IV and V).
3) We provide reference implementations for all three designs by extending a distributed content-based publish/subscribe system (Section VI).
4) We demonstrate the utility of our approach in the context of online games and high-frequency trading and compare it to alternative approaches, such as parametric subscriptions. Our solution reduces subscription traffic by up to 96.8% and improves delivery accuracy by at least 10% compared to the baseline resubscription mechanism.

II. Related Work

Research on pub/sub subscriptions broadly falls into two categories: matching and routing. Work focusing on matching subscriptions describe specialized data structures maintained by brokers to store the subscriptions for the purpose of computing interest matches with incoming publications. Early work put the emphasis on scaling the matching time for a large number of subscriptions at a high publication rate [2], [1]. More recent work turn their attention to high-dimensional content-based filtering [23], [18]. As our work focuses primarily on optimizing subscription dissemination, it is orthogonal to these approaches and may be used in conjunction with any publish/subscribe matching engine. However, note that we position our work within the context of applications with high subscription churn; therefore, it is best paired with a matching engine optimized for a high rate of subscriptions and unsubscriptions, such as [10].

Research on routing shares a goal common to our work since the focus is to reduce the communication load for disseminating and maintaining subscriptions. One work on context-aware publish/subscribe allows subscriptions to be expressed containing variables which are modified separately as a change of context [8]. For instance, this would allow a player to subscribe to the area around its current position, and simply send a context update message whenever its moving. Our evolving subscriptions can fully realize context-aware publish/subscribe since it allows for subscriptions to be expressed over dynamic variables. In addition, our system does not require frequent context update messages if the changes can be expressed ahead of time, thus saving additional communication costs.

To the best of our knowledge, the work on parametric subscriptions is the most similar to our approach [12]. To address the same issue of dynamic subscriptions, the work introduces a new type of operation called subscription updates which can adjust the properties of subscriptions. An extension is also proposed to predict fluctuations in subscription dynamics and approximate the changes to avoid thrashing at a broker's routing table. While this approach does reduce the traffic overhead by avoiding the need to perform costly unsubscription and subscription operations, the subscription updates must still be propagated throughout the system. In our work, we avoid propagating updates altogether when the rate of update can be expressed as a locally computable function. Still, it is possible to use our evolving framework in conjunction with parametric subscriptions to address subscriptions which vary irregularly.

Outside the scope of publish/subscribe systems, but within the domain of spatial applications, the concept of dead reckoning is relevant to our interests. In networked games, dead reckoning techniques allow players to predict the movement of other participants and anticipate the future game state in order to hide the network latency [16]. The position estimations are then corrected as the messages arrive. In essence, our evolving subscriptions solution allow brokers to perform dead reckoning techniques to locally update subscriptions. The difference is that update messages are only required when the subscription evolution functions need to be changed. Note that our approach is compatible with more sophisticated dead reckoning techniques: In AntReckoning [24], player movements are recorded as trails of *pheromones*, which allow the system to predict future movement. Such techniques could be integrated in our pub/sub system by allowing subscribers to express the evolution of their subscriptions through such semantics.

More broadly, there exists work in the database community around adaptive filtering [9] or handling moving range queries [20]. Most of this research focuses on building efficient indexes and data structures for storing moving objects and queries. However, it does not consider the networking cost of sending and receiving position updates. Our current work is focused primarily on the distributed nature of pub/sub subscription matching and routing. Efficient indexes are orthogonal to our approach, and can be used in conjunction to raise the performance of update processing at the local broker level.

III. Model and use cases

This section first introduces as background the publish/subscribe model used in this work. It then extends this model with evolving subscriptions, and finally outlines different applications which benefit from these type of subscriptions.

A. Background: Publish/Subscribe Model

In this paper, we employ a content-based pub/sub model operating on a distributed overlay network of brokers. Each client (publisher or subscriber) is connected to a single broker. Brokers are connected to each other to form a topology. A publication is sent from a publisher to its directly connected broker, where it is then matched against known subscriptions and forwarded to the next hops (either downstream brokers or subscribers) towards matching subscriptions. Publishers can either publish directly or optionally first advertise their publication space. Subscribers can subscribe and unsubscribe by sending the appropriate message to the pub/sub network. Depending on the routing model used, (un)subscriptions are either flooded to all brokers or forwarded towards publishers with matching advertisements. Our solution is compatible with either routing approach (with or without advertisements).

Publications are sets of attribute-value pairs of the form:

$$Pub : \{(a_1, val_1), (a_2, val_2), ...\}$$

where a_i is an attribute name and val_i is its value.

In a content-based model, subscriptions are sets of predicates:

$$Sub : \{(a_1\ op_1\ val_1), (a_2\ op_2\ val_2), ...\}$$

where op_i is the operator for the i-th predicate on attribute a_i with value val_i.

Predicates are conjunctive: a publication P matches a subscription S if all predicates are satisfied. For example, a publication $P = (x, 2)$ matches $S_1 = \{(x < 3)\}$ but does not match $S_2 = \{(x < 1)\}$.

B. Evolving Subscription Model

Our evolving subscription language is an extension of the content-based pub/sub model, augmented to allow predicates to change based on the values of *evolution variables*, i.e., variables with values that can change while a subscription remains active. Evolution variables can have *continuous* changes (e.g., time) or *discrete* changes (e.g., stock price).

The evolving subscription language replaces the val component of a predicate with a function that considers evolution variables and returns a value. The attribute values in a publication are compared against the values returned by the functions with input the current values of the evolution variables.

Thus, an evolving subscription is of the form:

$$SubEv : \{(a_1\ op_1\ fun_1(v_a, v_b, ...)), ...\}$$

where each function $fun_i(v_x, v_y, ...)$ takes into account variables to return a value in the domain of the attribute a_i.

For instance, a subscription $S_{ev} = \{(x < fun1(t))\}$ where $fun1(t) = 2 * t$, and t represents time, increases steadily the range for x it is interested in.

C. Application Domains

In this section we outline a set of application domains that can benefit from evolving subscriptions.

1) Location-based Application — Online Game: As a first example, consider the case of a game where player characters are interested in all events happening in a 6-by-4 rectangular area centered around their current location. Assume a character is currently located at the [0,0] coordinates of the game world as depicted in Figure 1(a). In this case, the area of interest is represented in a conventional system by the subscription:

$$\{(x >= -3), (x <= 3), (y >= -2), (y <= 2)\}$$

Events in the game world (such as other players' movements, pickup actions etc.), are sent as publications containing the coordinates of the location where the event

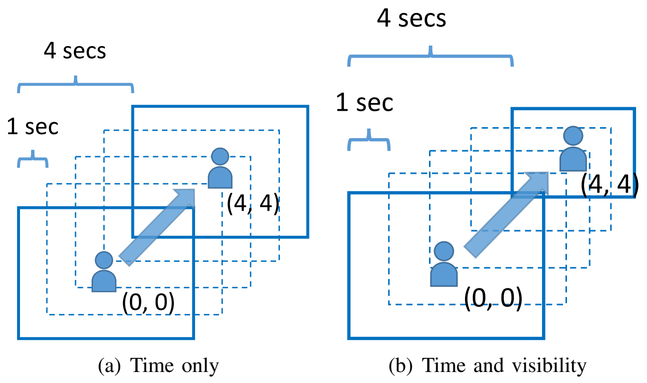


Figure 1. Game Example: Versioned Evolving Subscriptions

takes place. For instance, the pickup of an apple at coordinate (4,3) results in a publication with attribute/value pairs $\{(x, 4), (y, 3), (action, pickup), (object, apple)\}$, which would not match the above subscription.

However, whenever the player moves, he/she has to adjust his/her subscription because the center of his/her interest rectangle changes. Also, other factors like simulated rain or fog can reduce the in-game visibility, which would reduce the area of interest of the players depending on the intensity of the rain. In a traditional setting, whenever the player moves or the weather changes, he/she has to remove his/her current subscription and subscribe again with the updated location and area. Resubscribing continuously whenever the player moves is unpractical and should be avoided. Resubscribing whenever the visibility changes requires the players to be notified about the weather, which is an additional communication cost.

In contrast, evolving subscriptions can be updated without requiring an expensive resubscription process during movement or sharing weather information with clients. For example, we can express player movement on a x-y plane at a constant speed with the following evolving subscription:

$$\{(x >= -3+t), (x <= 3+t), (y >= -2+t), (y <= 2+t)\}$$

where t is initialized to 0 at the time of subscription. Now consider the above pick-up publication containing coordinates $\{(x, 4), (y, 3)\}$. If this publication is sent at the same time as the subscription, it does not match it. But if it is sent one or two seconds after the subscription is submitted, it will match. For instance, when matching the publication at time $t = 1$, all predicates of the subscription $\{(4 \geq -3 + 1), (4 \leq 3 + 1), (3 \geq -2 + 1), (3 \leq 2 + 1)\}$ evaluate to `true`. This evolving subscription is depicted in Figure 1(a).

Now consider the following evolving subscription, which also takes into account visibility:

$$\{(x >= (-3 + t) \times v), (x <= (3 + t) \times v),$$
$$(y >= (-2 + t) \times v), (y <= (2 + t) \times v)\}$$

where v is a value between 0 and 1 that represents the visibility percentage in-game. When matching this publication at time $t = 1$, with a visibility value of 0.5 all predicates of the subscription $\{(2 \geq (-3+1) \times 0.5), (2 \leq (3+1) \times 0.5), (1.5 \geq$

$(-2+1) \times 0.5), (1.5 \leq (2+1) \times 0.5)\}$ evaluate to `true`. This evolving subscription is depicted in Figure 1(b).

2) Virtual Marketplace: As a second example, we consider a virtual marketplace application. In this system, consumers announce through publications how much they are willing to pay for certain goods. Producers of goods want to receive purchase offers if they are above a certain minimum price. This minimum price can be adjusted dynamically to reflect supply and demand.

For instance, a producer that has a steady flow of products from a factory might determine the minimum sale price using a time-based formula which considers the amount of warehouse space he has to store the produced goods that have not been sold yet. When the warehouse is close to empty, the minimum sale price at which purchase offers are interesting to the producer is high, since any products that are not sold can be stored in the warehouse. On the other hand, as the warehouse reaches its full capacity, the producer needs to liquidate products in order to make space for the new stock coming from the factory, and as a result will accept lower purchase offers.

3) High Frequency Trading: In an HFT scenario, an algorithmic trader subscribes to stock indices and performs operations at a very fast rate (10000 orders per second) [19]. Due to the ephemeral nature of the operations, which last milliseconds, it is imperative to update subscriptions quickly. By using evolving subscriptions, one can embed some of the extrapolation logic used by the tool directly into the subscriptions while retaining very fine-grained subscriptions which limits the volume of data received, thereby improving processing speed.

4) Smart Grid Monitoring: Consider the situation of monitoring a smart power grid where multiple power plants produce and distribute electricity over the grid, which are affected by the weather conditions (e.g., too much wind can overload certain power lines). In this context, a pub/sub system notifies grid management when the load on power lines exceeds a threshold. Evolving subscriptions are used to compute the threshold dynamically based on factors such as the wind.

D. Evolution Variables

While time is a commonly used example, our system can support any combination of discrete and continuous variables, as long as a pub/sub broker can locally evaluate the evolving functions given the values of the variables.

The pub/sub broker might have access to *external information*, such as the current weather conditions. As an example, we could formulate an evolving subscription matching events on the beach, provided that the sun in shining. This requires that the broker either pulls the data or registers for push notifications from the information providers.

Brokers could also have access to global system information, such as *the current mode* of the pub/sub system. For instance, there might be three different operating modes defined for the system, such as *standard*, *diagnosis*, and *critical*. In such a context, monitoring nodes issue evolving subscriptions which match important publications when in critical mode, no publications when in standard mode, and a sample of publications when in diagnosis mode, to collect system statistics.

Finally, evolving subscriptions can be used to prevent system overload at each broker. For instance, the *available outgoing bandwidth* at the broker can be used to reduce subscription selectivity when the communication load is high. For instance, an evolving subscription of the form $\{(distance < maxDist \times (maxBw - outgoingBw))\}$ matches all publications up to $maxDist$ when there is no load, and no publications at all when the system is fully loaded.

IV. Proposed engine designs

We now present three designs for handling evolving subscriptions in a content-based pub/sub system: Versioned Evolving Subscriptions (VES), Lazy Evaluation Evolving Subscriptions (LEES), and Cached Lazy Evaluation Evolving Subscriptions (CLEES). A concrete implementation for each engine is presented in Section V.

A. Versioned Evolving Subscriptions (VES)

VES maintains a non-evolving version of each evolving subscription by evaluating their predicate functions using the current value of the evolution variables. The non-evolving copies are then inserted into the existing matching engine of the pub/sub system. When the value of a monitored evolution variable changes, the VES engine re-evaluates associated evolving subs, removes the previous copy from the matching engine, and inserts the new version. This is similar to an unsubscription followed by a subscription, the key difference being that each broker does this autonomously without requiring further communication from the client.

In addition to the usual evolving predicates, each VES must indicate a minimum evaluation interval (MEI). This is the minimum amount of time each non-evolving copy of the subscription must stay active. This parameter is used to prevent an excessive amount of re-evaluations for continuous variables like time or discrete variables with frequent value changes. Figures 1(a) and 1(b) show the evolution of two subscriptions using VES where MEI is $1s$, using time as an evolution variable.

VES have one limitation in common with client-triggered resubscriptions: the MEI parameter limits the granularity of evolution updates (see the different overlapping rectangles in Figure 1(a)). Therefore, *false negatives* may occur: publications that should match but are not delivered to the subscriber. To compensate, one could subscribe to a larger space to avoid this issue. However, this introduces the opposite effect of *false positives*: publications that match the larger subscription, but are actually of no interest to the subscriber. Adjusting the MEI parameter controls the trade-off between accuracy and recalculation processing overhead.

To minimize false negatives and positives to the subscribers, brokers can analyze any incoming subscription and generate an overestimating version that avoids false negatives. Brokers can then use the original subscription to determine matches for directly attached subscribers, and the overestimating version when forwarding to other brokers.

B. *Lazy Evaluation Evolving Subscriptions (LEES)*

LEES are only evaluated when a publication arrives. Instead of maintaining a non-evolving version of each subscription, we evaluate them only *on demand*, whenever a publication containing the same attributes arrives.

Unlike VES, LEES does not use a minimum evolution interval. In our game example, Figure 2(a) shows the subscription area for a time period of 4 seconds when using LEES.

The drawback of LEES is that *for each publication*, the engine must evaluate all LEES that contain the publication's attributes. Hence, the cost associated with the evaluation increases linearly with the volume of incoming publications, which generates additional CPU load, especially for complex evolution functions.

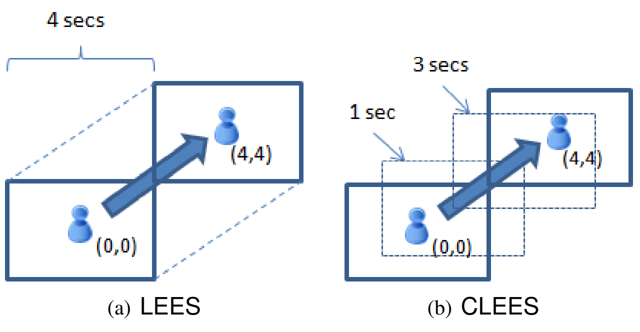


Figure 2. Game Example: (Cached) Lazy Evaluation Evolving Subscriptions

C. *Cached Lazy Evaluation Subscriptions (CLEES)*

Cached Lazy Evaluation Subscriptions (CLEES) is a combination of VES and LEES. Lazy evaluation is triggered the first time an incoming publication arrives, at which point a concrete version of the subscription is built using the current values of the evolution variables. This version is then cached and used for matching subsequent versions during a specified time threshold (TT). Once the cached version expires, the lazy evaluation is triggered again on the next incoming publication. Thus, the TT serves a similar function as the MEI in VES.

The advantage of CLEES over LEES is that it limits the evaluation overhead when the incoming publication rate is high. However, note that no matter what value of TT is employed, CLEES still depends on the publication rate. Thus, it is fundamentally different from VES which depends on the rate of change of evolution variables. This means the two techniques are suitable for different workloads. A truly hybrid solution which can adaptively switch between the two represents an interesting avenue for future work.

Using TT set to $1s$, Figure 2(b) shows a CLEES in a scenario where 3 publications arrive $1s$, $1.5s$, and $3s$ after subscribing. In this case, lazy evaluation is triggered at $1s$ and $3s$, while the first cached version is used at $1.5s$.

V. Implementation Details

In this section, we provide some implementation details for our proposed evolving subscriptions designs. Our prototypes are built on top of the PADRES pub/sub framework [11] by modifying its content-based engine to handle evolving subscriptions, and providing the appropriate client API.

A. *Versioned Evolving Subscriptions*

When a VES is received by a broker, an initial version of the subscription is created using the current values of the evolution variables (see Figure 3). This initial version is inserted into the regular matcher. In addition, the original VES is added to a new data structure, the evolving subscription queue (ESQ).

Subscriptions entering the ESQ are automatically ordered by the time remaining until they are scheduled to evolve again, as indicated by their minimal evolution interval (MEI). A group of handlers, each running on its own thread, monitor the ESQ and are in charge of updating the subscriptions. Whenever a thread becomes idle, it removes the first subscription S in the queue and adds it to the list of subscriptions that are ready to evolve. When the system detects that an evolution variable has changed (note that continuous variables are in an always changing state), it checks the list for subscriptions ready to evolve which depend on the changed variable. For each subscription which requires an update, the engine creates a new version with the current timestamp, and replaces the old version stored in the matcher. These replacement operations are synchronized by a lock to prevent concurrency problems. Additionally, the original VES is reinserted into the ESQ with a new scheduled time set to the current time plus its MEI.

When processing a publication, the VES engine solely employs the regular matcher with the currently stored versions of each subscription.

B. *Lazy Evaluation Evolving Subscriptions*

As shown in Figure 4, a LEES is divided into two subscription components with the same id. The non-evolving predicates are stored as usual in the matcher. The evolving predicates are stored in a new hash-table structure called the Lazy Evolution Matching Engine (LEME). An incoming publication is forwarded in parallel to both the standard matcher, which returns a set M_1 of subscriptions, and the LEME, which returns a set M_2. For the latter, the attributes in the publication are compared against the current state of the evolving predicates. Subscriptions that have their identifiers in both sets M_1 and M_2 are considered matched. The publication is then put in the output queue to be sent to these subscribers. Our system also handles subscriptions that contain only evolving or only non-evolving predicates. These subscriptions are only maintained in one of the matching engines, and flagged accordingly.

C. *Cached Lazy Evaluation Subscriptions*

As with LEES, a CLEES subscription is divided into two parts, one with the non-evolving predicates, and another with the evolving predicates which is stored in an ordered list known as the Lazy Evolution Storage (see Figure 5).

A publication matches a subscription if it is contained in both M_1, generated from the matcher, and M_2, generated by the Lazy Evolution Cache Matching Engine. The latter operates

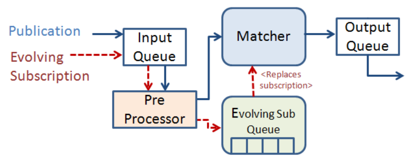


Figure 3. VES Engine Architecture

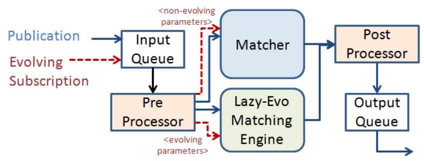


Figure 4. LEES Engine Architecture

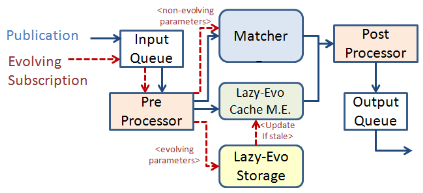


Figure 5. CLEES Engine Architecture

by looking up for a cached version of each subscription which can potentially match the publication (i.e., containing attributes found in the publication). If a non-expired version cannot be found, the engine generates a new version by retrieving the subscription from the storage and evaluating it with the current values of the evolution variables. This new version has an expiry time set to the current time plus the time threshold TT indicated in the CLEES.

In our implementation, we keep a separate cache for the evolving subscriptions. In principle, we could add the subscription versions into the existing matcher, like in VES. This would leverage any optimizations provided by the matcher, but also would raise the amount of contention on this shared indexing structure.

D. Consistency Model

Due to the distributed nature of our pub/sub system, it is possible that brokers are not completely synchronized with regards to the state of the evolution variables. Furthermore, there is certain latency required to fully transport a publication from its source to all intended subscribers. During that time, the evolution variables may have already changed.

To be completely consistent, an evolving engine should appear as if it was centralized on one machine. A publication is evaluated at the exact moment it enters the pub/sub system. In a distributed setting, the entry point of a publication is the broker directly connected to its publisher.

In their current implementation, our engines rely solely on local broker knowledge to match publications, which could result in *false positives* and *false negatives*. However, the accuracy of our proposed designs is greater than that of the baseline resubscription process, where there is no reliability guarantee provided during the entire process. For instance, no publications is guaranteed to be forwarded after the unsubscription is processed and before the new subscription is fully installed. In Section VI, we assess the accuracy of our proposed solutions.

We suggest two solutions for preventing inconsistencies. One is to rely on evolution variables which can be locally kept consistent by each broker. For instance, elapsed time can be counted by each broker separately, assuming minimal clock drift, whereas absolute time requires precise synchronization (e.g., PTP [7]).

Another possibility is to record a snapshot of the values of read evolution variables at the entry point broker. This snapshot should be piggybacked into the publication, such that downstream brokers can use these values for evaluating the subscriptions. This approach works well with LEES and CLEES, but does render VES ineffective since it can invalidate the current version stored by the matcher, which would then trigger the generation of a new version based on the snapshot. This snapshot-based implementation is left as future work.

VI. Evaluation

In this section, we present two different sets of experiments. First, using a high frequency trading (HFT) use case, we compare the performance of our proposed evolving subscriptions engines (VES, LEES, CLEES) with two baseline approaches: resubscriptions (using unsubs and subs messages), and parametric subscriptions (using update messages), implemented as described in [12]. Second, we conduct a sensitivity analysis of our three proposed solutions over a wide variety of workload parameters using an online game application demo [3].

All experiments are run in a cluster of 100 computers. The machines run on Linux or Windows and have a Gigabit Ethernet connection. Brokers run on more powerful machines (four-core processor, 3 GHz, 4-6GB memory), while clients use less powerful ones (dual-core processors, 2 GHz, 2GB memory).

A. Metrics

1) Subscription message traffic: We measure the average number of subscription-related messages per minute received by any broker in the system. For the resubscriptions baseline, this consists of subscriptions and unsubscriptions. For the parametric subscriptions baseline, we additionally track subscription update messages. This is our main performance metric since our goal is to reduce communication load using evolving subs.
2) False positives/negatives: We compare the logs of publications received by subscribers in each system to the *ground truth*, which is the same respective logs produced using a centralized pub/sub system with the same deterministic workload. Any publication received by a subscriber which is not in its ground truth log is considered a false positive. Conversely, a publication in the ground truth but not received by a subscriber is a false negative. Due to the nature of our experiments the overall number of false positives and false negatives are very close to each other. Thus, we group them together as a single item in our result graphs.
3) Processing time: The time taken by each broker to process evolving subscriptions. For VES, this is the time taken to update the version of each subscription. For LEES

and CLEES, it is the processing overhead for performing on-demand evaluation of evolving subscriptions.

4) Throughput: This metric measures the impact of lazy evaluation for publishing using LEES and CLEES, which incur a processing overhead for every publication. The publication rate is measured in $pubs/s$.

B. Baseline Comparison: High-Frequency Trading

Description. High-frequency trading (HFT) is a type of algorithmic trading where computers are used to perform a large number of trades in a very short amount of time. Complex algorithms are used to analyze trends in the market before making split-second decisions. Thus, HFT algorithms require large amounts of data which are obtained by subscribing to different stock markets.

Setup. For this experiment, we create a small HFT environment where both subscribers and publishers connect to brokers in one of three simulated stock markets. Each individual stock market is modeled with four brokers: three edge brokers to which both publishers and subscribers can connect to, which are then connected to a core broker used as a gateway to other markets. A central broker connects all three stock markets.

Our workload consists of stock data from 500 different stocks, taken from the Standard & Poor's 500 stock market index. There is a total of 9 publishers, which represent brokerage firms, each connected to a different edge broker. These brokerage firms publish information about the current prices of stock as well as their availability, as determined by a small activity trace that we generated. Each publisher operates at a rate of 1000 $pubs/s$. Up to 90 different clients, representing HFT investing firms, are uniformly distributed across the three markets and are connected to edge brokers.

Every HFT client in the system is interested in ten different types of stock. In order to be notified about stock prices while minimizing incoming publication volume, clients send subscriptions for a narrow range of prices (e.g., \$15.27 - \$15.29) which are constantly updated. Every evolving subscription is replaced with a new one once per minute.

Results. Figures 6(a) to 6(c) compare the subscription traffic of our evolving solutions with the baselines of resubscriptions and parametric subscriptions. All three evolving solutions have almost the same performance with respect to this metric and are represented by a single line. On average, our solution reduces subscription traffic by 96.8% and 92.9% when the rate of change is 30 and 12 per second, respectively (see Figures 6(a) and 6(b)). Our solution is capable of leveraging high change rates to obtain even better traffic reduction performance, while parametric subscriptions provide a constant 50.6% reduction over resubscriptions, since they replace pairs of unsubscription and subscription messages with a single update message. On the other hand, the volume of traffic generated by our evolving subscriptions remains constant under a variable rate of change. It is however affected by the subscription validity period. In Figure 6(c), we observe the impact of tripling the evolving subscription replacement rate: the traffic reduction goes down to 90.5%, while the baselines remain unaffected.

Figure 7 shows the false positive and negative rates of all compared systems. Resubscriptions provide the worst performance, since the slow unsubscription and subscription process involves several rounds of messaging and is therefore not accurate. Similarly, the parametric subscriptions also involve network latency in propagating the subscription update messages, thereby introducing inconsistencies with the ground truth. Among our systems, the LEES system has very almost perfect accuracy since subscriptions are evaluated instantaneously. The VES and CLEES systems have poorer yet similar accuracy due to the coarse-grained nature of their interval-based maintenance (based on their MEI and TT values), which is especially prominent when dealing with a continuous variable like time.

Summary. Our evolving subscriptions reduce subscription-related traffic by 96.8% and improve delivery accuracy by at least 10% compared to a normal system using resubscriptions. Parametric subscriptions reduce traffic by 50.6%, but are still sensitive to high update rates. On the other hand, evolving subs are only affected by their validity period. In addition, evolving subscriptions greatly increase the publication matching accuracy over the two baselines, which are affected by the latency of communicating changes over the network.

C. Sensitivity Analysis: Location-Based Game

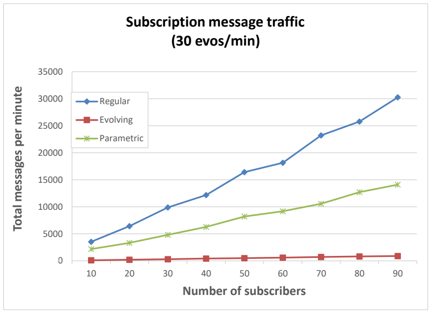
Description. The context of this set of experiments are location-based applications. We use a massively multiplayer online game (MMOG) where clients move around their in-game avatar and subscribe to events which are close to them. A single game server also acts as pub/sub broker and handles the players' subscriptions and the game event publications. The objective of these experiments is to highlight the differences between our proposed engines using various workloads.

Setup. We use a small virtual game world controlled by a single server with an embedded broker. Up to 100 client machines, each controlling from 1 to 100 characters, connect to the broker when they join the game. Each character subscribes to a rectangular area in the game world centered around the current position of the player. The size of this area is affected by the $visibility$ in-game variable (which has a value between 0 and 100%), controlled by the game server.

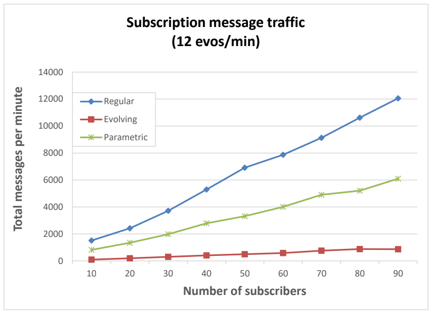
All 500 characters chose independently one direction to move towards for $10s$, before choosing a new direction and repeating the process. The evolving subscriptions are of the following form:

$$\{(x >= (-3 + (\Delta x \times t)) \times v), (x <= (3 + (\Delta x \times t)) \times v), \\ (y >= (-2 + (\Delta y \times t)) \times v), (y <= (2 + (\Delta y \times t)) \times v)\}$$

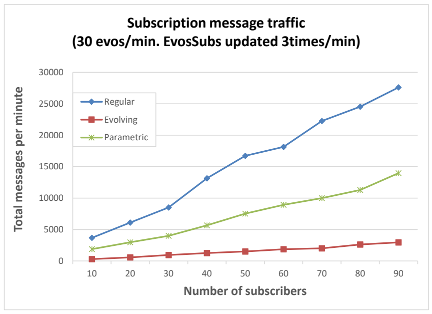
Where the subscription values depend on the player's position and speed. Each evolving subscription is replaced by the client with a new one every $10s$. For VES and CLEES, $1s$ is set to be the MEI and TT, respectively. These subscriptions are similar to those in the example of Figure 1(b), but with a longer validity length ($10s$ instead of $4s$). The Δx and Δy values indicate the movement speed, and determine, using time, the center position of subscription after each evolution. The v



(a) 30 changes/minute



(b) 12 changes/minute



(c) Evolving Subs Validity Period: $20s$

Figure 6. Comparison of Evolving Subscriptions with Baseline Approaches using the HFT scenario

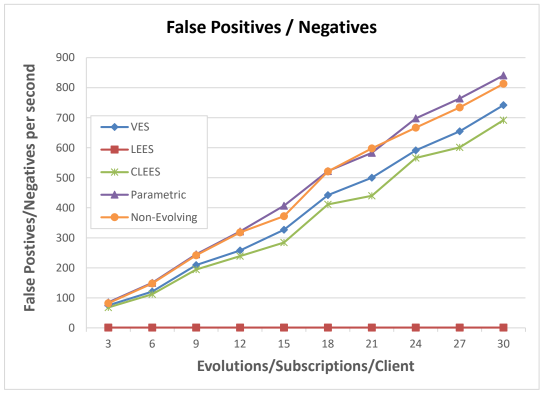


Figure 7. False Positives/Negatives

variable represents the visibility and determines the size of the area the players subscribe to after each evolution.

We used Mammoth, a massively multiplayer game research framework [15], to record a multiplayer game session with thousands of players in one virtual map. We then used the game trace to generate as many events as necessary for our needs.

Results. Figures 8(a) to 8(d) compare the processing times for handling evolutions across a variety of workload settings. We observe that CLEES outperforms the other two designs in most situations, mainly when the number of subscriptions is sufficiently high, and therefore represents our most scalable solution.

Compared to the other two, the VES design has the additional overhead of removing and inserting subscriptions into the regular pub/sub matcher, which grows linearly with the number of subscriptions. In contrast, CLEES maintains the subscriptions in a separate cache and thus reduces maintenance costs. However, the cost of matching publications with VES is lower, since the matcher employs an optimized indexing structure while CLEES matches all subscriptions with matching attributes. Therefore, we note that at low subscription counts, when the subscription maintenance overhead is still low, VES is better than either LEES or CLEES.

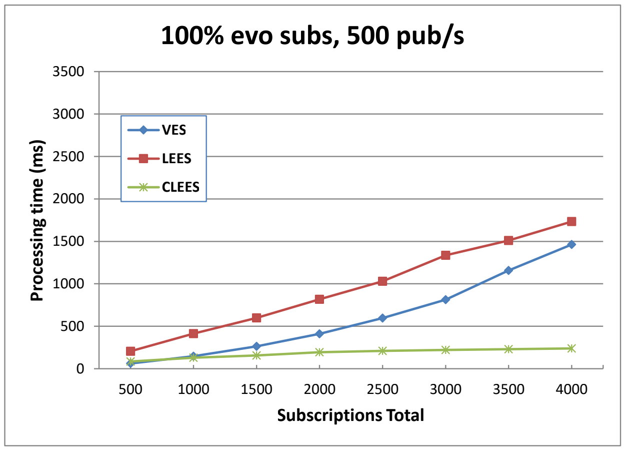
VES is also dependent on the evolution frequency, since Figure 8(d) displays increased overhead when the rate of evolution is doubled. On the other hand, both LEES and CLEES are unaffected by the rate of evolution since they are evaluated on-demand. However, they are affected by the rate of publications, as exhibited by Figure 8(b) where it is doubled.

Another interesting observation is that while VES is affected by the total number of subscriptions, it is unaffected by the proportion of evolution subscriptions. This is reflected in Figure 8(c) where we employ a 50/50 split of evolving and non-evolving subscriptions. The overhead of maintenance for VES is largely driven by the insertion and deletion costs of generated subscription versions into the matcher, which is dependent on the total number of subscriptions, no matter if evolving or not.

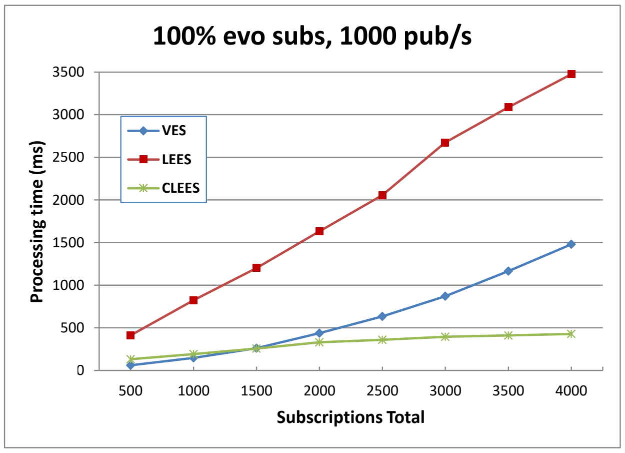
This observation is confirmed by Figure 9, where the processing time for a 50/50 split of evolving subscriptions and non-evolving subscriptions has the same processing time as a pure workload of evolving subscriptions (assuming the latter evolves at half the rate to keep the total number of evolution updates constant). Figure 9 also shows that increasing the update frequency, while decreasing the number of evolving subscriptions to maintain a constant volume of evolutions, results in faster processing times since the number of subscriptions in the matcher diminishes. For instance, 2000 subscriptions with a $2s$ evolution period generate 1000 $evolutions/s$, which is the same as 500 subscriptions with a $0.5s$ period. Yet, the 2000 subscriptions take $1000ms$ to process, while the 500 subscriptions case takes only $200ms$.

For LEES, the proportion of evolving subscriptions in the system is important. In Figure 8(c), this solution receives a 38.7% improvement when the workload contains only 50% evolving subscriptions. This is because lazy evolution is performed using a separate data structure storing evolving subs, unlike VES which depends on the matcher.

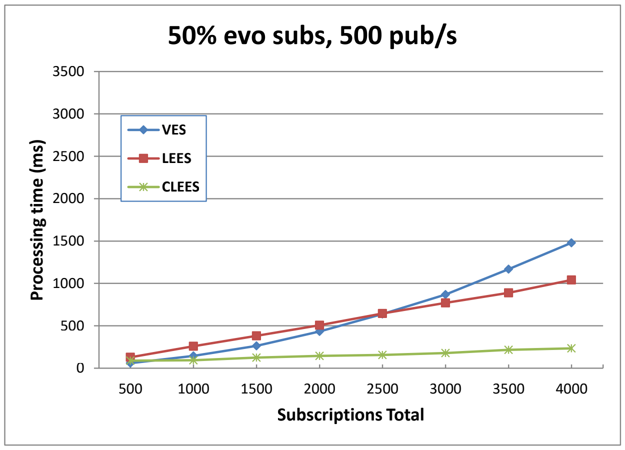
Figures 10(a) and 10(b) show two factors which affect LEES: number of evolving subscriptions, and the ratio of subscriptions to clients. Since the evolution overhead for lazy evolution is incurred at publication time, these solutions hit a bottleneck in terms of maximum sustained publication throughput, which is shown in those graphs. In Figure 10(b), we keep the number of evolving subscriptions constant, but distribute them to a varying number of subscribers, ranging from 1 to 1000. In other words, we test scenarios where subscribers can submit



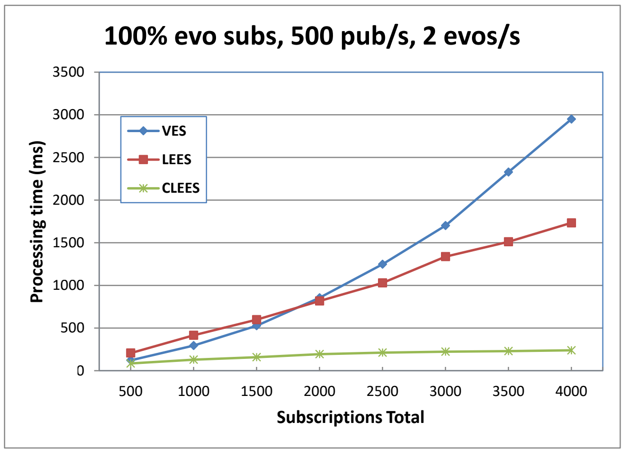
(a) Standard



(b) Double Publication Rate



(c) Halved Proportion of Evolution Subs



(d) Double Evolution Frequency

Figure 8. Sensitivity Analysis for VES, LEES, and CLEES using the Game Scenario

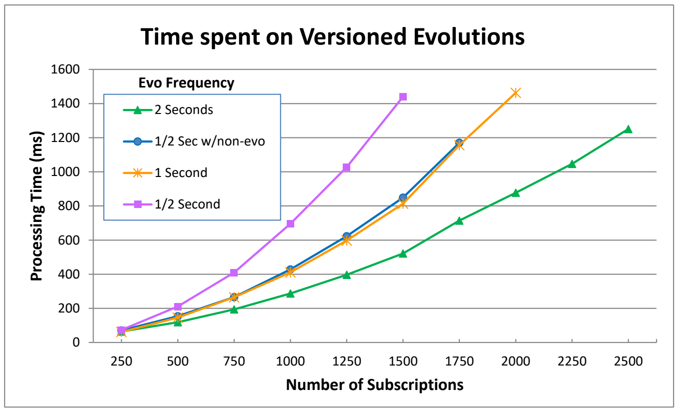
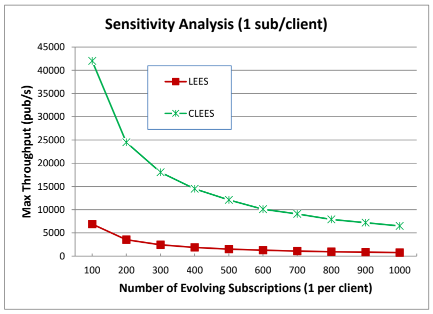


Figure 9. Number of Total Subs (VES)

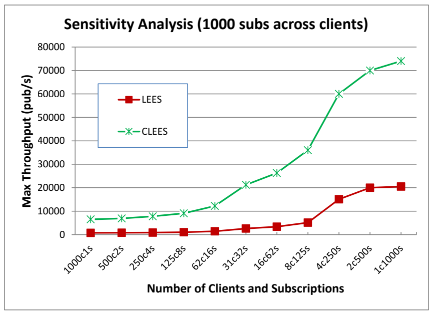
more than one subscription. We find that LEES performs better when the subscriptions are less dispersed, since there is a greater chance of a match being found for each subscriber. As soon as a match is found, the lazy evolution terminates since it is known that the publication must be forwarded to that client. When the subscriptions are distributed to a large amount of clients, the lazy evaluation process must exhaustively evaluate through all the evolving subscriptions of clients with no matching subscriptions, thus increasing the processing time. However, when the number of clients is large enough, the system is affected by a different bottleneck and the processing time no longer increases dramatically.

Note that CLEES is less sensitive to these factors, as shown in Figures 10(a) and 10(b), since the cache converts many of the expensive evolving evaluations into simple predicate matching using cache hit subscriptions.

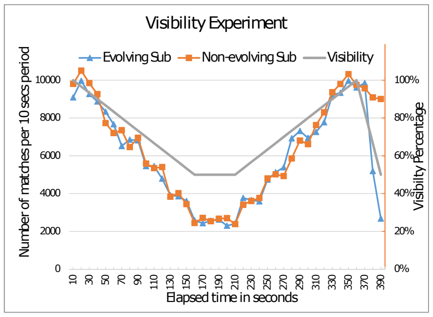
Conclusion. CLEES offers the best performance in terms of processing time which is 45.9% better than VES and 76.4% faster than LEES. It is also less sensitive to the volume of evolving subs and the ratio of subs to client, unlike LEES. On the other hand, VES is the least scalable solution as it is affected by the evolution period and the total number of subscriptions in the system, even by subscriptions which are not evolving. However, it has the advantage of not being affected by publications, and can thus achieve greater performance in low subscription rate scenarios with infrequent evolutions.



(a) Number of Evolving Subs



(b) Ratio of Subs/Clients



(c) Visibility Experiment

Figure 10. Sensitivity Analysis for LEES and CLEES

Pub/sub approach	Subscription message traffic	False positives/negatives	Processing time	Main advantages
VES	Low	Low to medium	Medium-high $\propto$ evo. freq.	Low matching cost Best with few subs
LEES	Low	Very low	Medium-high $\propto$ pub rate & sub/client ratio	Best with low evo. freq.
CLEES	Low	Low to medium	Low-medium $\propto$ pub rate	Overall best performance
Parametric	Medium	Medium	Low	More flexible than evo. subs
Non-evolving	High	Medium	Low	Simplicity and ease of use

Table I
RELATIVE COMPARISON BETWEEN PUB/SUB APPROACHES

Table I offers a relative comparison of the different pub/sub approaches tested on our experiments.

D. *Visibility-Based Game Experiment*

The purpose of this experiment is to show that our evolving subscription predicates are robust enough to handle many types of evolving variables simultaneously, reducing the amount of resubscription messages required by a traditional pub/sub approach.

This experiment shares the same setup and subscriptions as the ones in Section VI-C. 250 clients move around the map while the visibility variable is changed every $3s$, moving from 100% visibility at the start of the experiment to 50% at the middle, then back to 100% before finally dropping back to 50% in the last $20s$.

Figure 10(c) shows the number of matching publications across the broker network during the experiment. When the visibility is at 50%, each subscription covers $1/4$ of its original area, therefore the number of matches also drops by 75%. Both the evolving and non-evolving approaches generate the same publication volume, but the clients in the non-evolving case send 10 times more subscription messages and must be constantly updated with the latest visibility values. In the final $30s$ of the experiment, the visibility values are not sent to any client, which explains why the evolving subscriptions react properly to the quick change in visibility while the non-evolving ones do not.

VII. CONCLUSION

This paper proposes the concept of evolving subscriptions, which are content-based subscriptions where predicates can be expressed using functions over variables, such as time. Subscription evolution is managed automatically by the publish/subscribe engine. We propose three different mechanisms for efficient evolution, each suitable for different types of workload.

Our evaluation, based on a implementation in a real pub/sub system, shows the advantage of evolving subscriptions in terms of number of messages exchanged between clients and the pub/sub system (96.8% subscription traffic reduction) as well as the execution overhead at each pub/sub broker.

VIII. ACKNOWLEDGEMENT

This research was supported by the Alexander von Humboldt Foundation.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM.

[2] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 443–452, 2001.

[3] C. Canas, K. Zhang, B. Kemme, J. Kienzle, and H.-A. Jacobsen. Evolving pub/sub subscriptions for multiplayer online games: Demo. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 344–347.

[4] C. Canas, K. Zhang, B. Kemme, J. Kienzle, and H.-A. Jacobsen. Publish/Subscribe Network Designs for Multiplayer Games. In *ACM/IFIP/USENIX 15th International Conference on Middleware*, 2014.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, Aug. 2001.

[6] A. Cheung and H.-A. Jacobsen. Green Resource Allocation Algorithms for Publish/Subscribe Systems. In *31th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 812–823, June 2011.

[7] R. Cochran, C. Marinescu, and C. Riesch. Synchronizing the linux system time to a ptp hardware clock. In *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*.

[8] G. Cugola, A. Margara, and M. Migliavacca. Context-aware publish-subscribe: Model, implementation, and evaluation. In *ISCC'09*.

[9] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. Agile: Adaptive indexing for context-aware information filters. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 215–226, 2005.

[10] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, and K. Ross. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD Conference*, pages 115–126, May 2001.

[11] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The padres publish/subscribe system, 2010.

[12] K. R. Jayaram, P. Eugster, and C. Jayalath. Parametric content-based publish/subscribe. *ACM Trans. Comput. Syst.*, 31(2):4, 2013.

[13] A. Kafka. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of*, 5(1), 2014.

[14] R. S. Kazemzadeh and H.-A. Jacobsen. Reliable and highly available distributed publish/subscribe service. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 41–50, Washington, DC, USA, 2009. IEEE Computer Society.

[15] J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker. Mammoth: a massively multiplayer game research framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games, FDG 2009, Orlando, Florida, USA, April 26-30, 2009*, pages 308–315, 2009.

[16] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st Workshop on Network and System Support for Games*, NetGames '02, pages 79–84, 2002.

[17] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. COBRA: Content-based filtering and aggregation of blogs and RSS feeds. In *NSDI*, 2007.

[18] M. Sadoghi and H.-A. Jacobsen. Analysis and optimization for boolean expression indexing. *ACM Trans. Database Syst.*, 38(2):8:1–8:47, July 2013.

[19] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.*, 3(1-2):1525–1528, Sept. 2010.

[20] S. Saltenis, C. S. Jensen, S. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[21] Y. Sharma, P. Ajoux, P. Ang, D. Callies, A. Choudhary, L. Demailly, T. Fersch, L. A. Guz, A. Kotulski, S. Kulkarni, S. Kumar, H. C. Li, J. Li, E. Makeev, K. Prakasam, R. van Renesse, S. Roy, P. Seth, Y. J. Song, B. Wester, K. Veeraraghavan, and P. Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *NSDI'15*.

[22] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, 2005.

[23] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. Indexing boolean expressions. *Proc. VLDB Endow.*, 2(1):37–48, Aug. 2009.

[24] A. Yahyavi, K. Huguenin, and B. Kemme. Interest modeling in games: The case of dead reckoning. *Multimedia Syst.*, 19(3):255–270, June 2013.