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

César Canas
School of Computer Science,
McGill University
Montreal, Canada

Kaiwen Zhang
Technische Universität
München
Munich, Germany

Bettina Kemme
School of Computer Science,
McGill University
Montreal, Canada

Jörg Kienzle
School of Computer Science,
McGill University
Montreal, Canada

Hans-Arno Jacobsen
Middleware Systems
Research Group

ABSTRACT

Traditional pub/sub techniques such as subscription covering are inadequate to fully handle workloads of applications with high subscription churn where new subscriptions follow a predictable pattern, such as some social location-based notification systems, predictive stock trading, and online games. In the context of those applications, the main unaddressed factor of traditional pub/sub techniques is that they have to wait for subscriber input before replacing older subscriptions with updated ones, creating undesirable overhead and timing issues.

In this paper we present a new type of subscription, called *evolving subscription*, that is able to autonomously adapt to the predicted needs of a subscriber. We propose a general framework for evolving subscriptions and discuss their advantages, limitations, and the type of applications they are fit to support. For this end, we develop and implement three different variations of evolving subscriptions, which are then evaluated and compared to the traditional re-subscription approach in the context of two use cases: online game and high-frequency trading. Our evaluation shows that our solutions perform differently depending on a number of workload factors, such as publication rate or the proportion of evolving to non-evolving subscriptions.

1. INTRODUCTION

Publish/subscribe is a communication paradigm commonly employed to enable asynchronous communication of event-based information between loosely coupled producers (publishers) and consumers (subscribers). Publishers submit data in the form of publications, subscribers express their interests in the form of subscriptions. Publish/subscribe has been employed in the context of business process execution [22], workflow management [6], stock-market monitoring [23], RSS filtering [18], complex event processing for algorithmic trading [15], and network monitoring and management [10].

In order to enable the operation of large-scale systems, publish/subscribe systems employ a variety of techniques to achieve scalability. At its core, the typical publish/subscribe architecture relies on a distributed overlay network of brokers for load balancing [5] and availability purposes [14]. Brokers are in charge of disseminating publications to matching subscriptions. This matching computation can be performed efficiently using simple topic-based semantics [1]. Topics represent channels which can be subscribed to. Publications addressed to a given topic are then delivered to all matching subscribers.

However, modern large-scale applications require a level of expressiveness beyond topic-based matching. In particular, content-based matching allows subscribers to express additional predicates which can filter publications within a topic. For instance, massively multiplayer online games have been shown to benefit from using content-based event dissemination [3]. Player clients, which control avatars in a virtual world, receive updates about the game state in the form of publications. Content-based publish/subscribe allows a client to form a fine-grained subscription based on the location of the player, thus limiting the amount of data received.

Nevertheless, there exists a tradeoff between scalability and expressiveness. Finding subscription matches in a content-based setting is an expensive process. While there exist works on efficiently computing content-based matches [8], the most successful techniques focus on controlling the subscription load. In particular, subscription covering substantially reduces the overhead of routing and maintaining large volumes of subscriptions [4]. The system can employ covering to eschew routing of subscriptions which are subsumed by a subscription of a larger size on the same interest space.

For certain applications, we argue that the established technique of covering alone is not sufficient to address the scalability issue imposed by large content-based workloads. Consider for instance the aforementioned online game scenario: the subscriptions issued by each player are similar in size as players have similar abilities in-game. However, covering is only effective is there exist a large variance in subscription size, which allows smaller subscriptions to be contained in the larger ones. Furthermore, in a game setting, subscriptions change frequently as players move around the

MIDDLEWARE '16 Trento, Italy

DOI: 10.475/123_4

game and update their interest. This high churn of subscriptions does not produce a stable structure to maintain covering relationships.

Even so, these highly dynamic workloads often have patterns in the way subscriptions are formed. In the case of online 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 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, the publish/subscribe system can autonomously update a subscription according to the prescribed pattern without requiring additional input from the subscriber. This essentially avoids the expensive and frequent unsubscription and resubscription process.

Therefore, 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 evolving subscriptions as a framework, which extends content-based semantics to allow update patterns to be specified.

2. We provide three designs for supporting evolving subscriptions: the periodic evaluation solution, the lazy evaluation solution, and its cached variant.

3. We provide reference implementations for both using the PADRES publish/subscribe system.

4. We demonstrate the utility of our approach in the context of online games and high-frequency trading and compare it to alternative approaches. We discuss the advantages of each approach with respect to a number of workload parameters.

2. RELATED WORKS

Research on pub/sub subscriptions can be broadly defined into two categories: matching and routing. Works 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. Established works put the emphasis on scaling the matching time for a large number of subscriptions at a high publication rate [2, 1]. More recent works turn their attention to high-dimensional content-based filtering [24, 19]. 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 [9].

Works focused on routing reduce the communication load for disseminating and maintaining subscriptions, and thus present a goal common to our work. The most well-known form of routing optimization is covering, where subscriptions detected to be subsumed by other subscriptions are no longer forwarded in the broker topology [12, 4, 16]. Covering is a general technique and can be used in addition to our evolving subscriptions to further reduce the subscription traffic. However, we argue that under the type of workloads employed in our studied applications, covering alone is not sufficient to significantly improve our performance. Section 6 compares the effectiveness of covering to our solution under a variety of workloads.

To the best of our knowledge, the work on parametric subscriptions is the most similar to our approach [13]. 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 resubscription 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 [17]. 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 [25] for instance, 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 works in the database community around adaptive filtering [7] or handling moving range queries [21]. These works focus on building efficient indexes and data structures for storing moving objects and queries. However, they do 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 therefore orthogonal to our approach, and can be used in conjunction in order to raise the performance of update processing at a local broker level.

3. MODEL AND USE CASES

This section first introduces the idea of evolving subscriptions conceptually, and then outlines three different applications which benefit from evolving subscriptions. We also briefly discuss other applications possible with variables other than time.

3.1 Conceptual Model

Evolving subscriptions are written in a subscription language that allows the expression of interest over time. At an abstract level, an evolving subscription is a function that takes as input parameters some of the information provided in publications, and internally uses this information along with a time variable to perform calculations over the subscription predicates and return a Boolean value (e.g. whether a publication matches the subscription or not). The crucial difference to non-evolving subscriptions is the inclusion of the time variable, which makes the matching of publications time-dependent; that is, the subscription *evolves* over time.

More concretely, the evolving subscription language is an extension of the content-based pub/sub model based on predicates over attribute sets as described in [6, 9], augmented with the evolution function over time.

In this paper, 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 corresponding value.

Subscriptions are a set of predicates over attributes that publications can possess:

```
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.

A publication P matches a subscription S if for each predicate in S on attribute a_i, P contains a value for a_i which evaluates the predicate to `true`[1]. As an example, a publication $P = (x, 2)$ matches a single-predicate subscription $S_1 = \{(x < 3)\}$ but does not match subscription $S_2 = \{x < 1\}$.

We now extend this language by replacing in an evolving subscription the *value* component of the predicate with a function that considers a time variable and returns a value. At any given time, this time variable has a concrete value, and thus, influences the outcome of the function calculation. That is, in an evolving subscription, not only the attribute values contained in a publication but also the current value of the time variable determine whether the publication matches the subscription.

Thus, an evolving subscription is of the form:

```
SubEv:
{(a_1 op_1 fun_1(t)), (a_2 op_2 fun_2(t)), ...}
```

where each function $fun_i(t)$ has to return a value which is in the domain of the attribute $attr_i$ which it constrains.

For instance, a subscription $S_{ev} = x < 2 * t$ increases steadily over time the range for x it is interested in. The domain for t is application-dependent. For instance, time as integer could represent elapsed time ticks, or the number of seconds that have passed since the subscription was submitted to the pub/sub system. In the remainder of the paper we assume exactly this: t represents the time units (e.g., in seconds) that have passed since the subscription was submitted.

[1]Our model requires a publication to fulfill all predicates of a subscription (conjunction). More general languages allow for both conjunctions and disjunctions as well as negations. In principle, our approach can be used with all of them with the appropriate implementation.

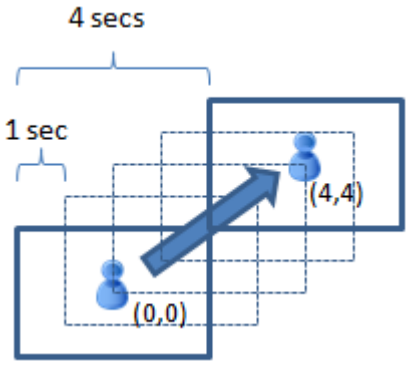

Figure 1: Evolving Subscription

3.2 Application Domains

We provide the following application domains which can benefit from evolving subscriptions.

3.2.1 *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. In this case, the area of interest is represented in a conventional system by the subscription:

```
Sub = {(x >= -3),(x <= 3),(y >= -2),(y <= 2)}
```

Events in the game world (such as other players' movements, pickup actions etc.), are then be sent as publications containing the coordinates of the location where the event takes place. For instance, the pickup of an apple at coordinate (4,3) results in a publication with attribute/value pairs $\{(x, 4), (y, 3), (action, pickup), (object, apple)\}$, and it 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. In a traditional setting, whenever the player moves, he/she has to remove his/her current subscription and resubscribe with his/her new location as the center of the rectangle. Doings this in a continuous fashion during the movement period is unpractical. Thus, one has approximate continuous movement periodically unsubscribing and resubscribing at fixed intervals.

In contrast, evolving subscriptions can support continuous movement with requiring an expensive resubscription process. For example, when the player starts moving with a speed and a direction so that he/she advances every second one x-coordinate to the right and one y-coordinate towards the top, then his/her evolving subscription can be expressed as:

```
SubEv =
{(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 $(4 \geq -3+1, 4 \leq 3+1, 3 \geq -2+1, 3 \leq 2+1)$ return to `true`.

3.2.2 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 low of products from a factory might determine the minimum sale price using a time-based formula that takes into account 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.

3.2.3 High Frequency Trading

In an HFT scenario, an algorithmic trader subscribes to stock indices and perform operations at a very fast rate (10000 orders per second) [20]. Due to the highly ephemeral nature of the operations, which last milliseconds, it is imperative for a trader 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.

3.2.4 Tidal Fluctuation

Finally, as a last time-based example, consider the situation of boat owners that anchor their boats in areas with high tidal fluctuations. When the tide is very low, these boats are docked in the sand and are very vulnerable to wind gusts. On the other hand, when the boats are floating in the water, they can easily withstand bigger storms.

In this context, one can imagine a system in which weather data is disseminated to interested parties using a pub/sub system. Boat owners that leave their boats anchored in bays with high tidal fluctuations could issue an evolving subscription that matches wind gust warning publications exceeding a certain threshold. The threshold is determined based on the estimated water height, which in turn can be calculated using the tidal coefficients corresponding to the current time and location of the boat modulated by a sinus function.

3.3 Other Evolution Variables

While *time* is our primary evolution variable, *any variable that can be evaluated locally at a pub/sub engine* could in fact be used to evolve subscriptions.

For instance, the pub/sub engine might have access to *external information*, such as the current weather conditions. In that case, an evolving subscription could use that information to subscribe to publications about events on the beach, provided that the sun in shining. This requires that the broker either pulls the external information from the information provider whenever the variable needs to be evaluated, or registers for push notifications, if possible.

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, *standard*, *diagnosis*, and *critical*. In such a context, one can imagine monitoring nodes that issue evolving subscriptions to the system to match *important publications when the system is in critical mode*, *no publications when in standard mode*, and *a random sample of publications when in diagnosis mode* in order to collect system statistics.

Finally, evolving subscriptions can be used to perform smart filtering to prevent system overload. For instance, the *available outgoing bandwidth* at the broker can be used to match incoming publications to less subscriptions when the outgoing bandwidth is high. A subscription with predicate $(distance < minDist * maxBw/outgoingBw)$ would be fulfilled by any publication independently of its distance value when the pub/sub engine is idle as $maxBw/outgoingBw$ will be a large enough value. On the other hand, when the engine is heavily loaded, only publications with a distance up to $minDist$ will be matched as $maxBw/outgoingBw$ will approach the value 1.

In other words, this technique allows a subscriber to express a criteria for publications he wants to receive no matter how loaded the system is, i.e., *at least* all publications for which $(distance < minDist)$. Conversely, an evolving subscription of the form $(distance < maxDist * (maxBw - outgoingBw))$ would allow to match *at most* all publications for which $(distance < maxDist)$, and no publications at all when the system is fully loaded.

4. PROPOSED ENGINE DESIGNS

We now present three designs for handling evolving subscriptions in a content-based pub/sub system: Periodic Evolving Subscriptions (PES), Lazy Evaluation Evolving Subscriptions (LEES), and Cached Lazy Evaluation Evolving Subscriptions (CLEES). For the purpose of this work, we employ time as the evolution variable. A concrete implementation for each engine is presented in the following Section 5.

4.1 Periodic Evolving Subscriptions

The idea behind Periodic Evolving Subscription (PES) is that the pub/sub engine, on a periodic basis, creates a concrete version of the evolving subscription by evaluating each predicate function using the current time value. This concrete subscription can then be integrated into the traditional matching engine of an existing pub/sub system. Thus, at each time period the subscription generated in the last time period is removed from the subscription set of the matching engine, and the current version is inserted.

Conceptually, it is similar to a periodic unsubscription and resubscription previously mentioned, with the difference that the pub/sub system does this autonomously and does not require the explicit unsubscribe and subscribe calls from the client to be propagated throughout the broker network.

A PES subscription, in addition to the usual predicates, has to indicate the evolution interval EI at which the subscription should be recalculated (e.g., every second). Figure 1 shows the periodic evolution of subscription S_{evolv} where the evolution interval is 1 second.

As is the case with resubscribing, Periodic Evolving Subscriptions has the same limitation: the interval between evolution updates limits the granularity of the subscriptions, as can be seen in the different overlapping rectangles in Figure 1. Thus, the subscription progressively loses precision as the length of the time period increases. One might therefore incur *false negatives:* publications that should match but are not delivered to the subscriber. To compensate, one could subscribe to larger ranges to avoid this issue. However, this introduces the opposite effect of *false positives:* publications that match the larger subscription, but at the current time are of no interest for the subscriber.

To minimize false negatives and positives to the subscribers, the broker that receives the evolving subscription can analyze the evolving function and determine an overestimating function that avoids false negatives. Brokers only use the original function to determine matches for subscribers that are directly connected to them. For matching subscriptions that have been forwarded from other brokers, the overestimating function is used. While this avoids delivering false positives to subscribers, additional traffic overhead is required within the broker network due to the use of the overestimating function.

Reducing the periodic time interval is another method to reduce false negatives and false positives, since it increases the accuracy of the subscriptions. On the other hand, this increases the CPU load on the pub/sub engine. Additionally, this continuous recalculation generates a fixed overhead regardless of the actual matching publication rate.

4.2 Lazy Evaluation Evolving Subscriptions

In a Lazy Evaluation Evolving Subscription (LEES), the subscription is only evaluated when a publication arrives. Thus, instead of updating the subscription on a regular basis to transform it to a concrete version, we evaluate the subscription only *on demand*, whenever a publication involving the same attributes arrives. This is similar to the parametric subscriptions detailed in [13], where the time at which a publication is received by the system is taken as an input parameter used to dynamically calculate the value that the subscription should have at that precise moment, before determining if the publication matches it.

Unlike PES, LEES do not require an evolution interval. In the context of our game example, Figure 2 shows the continuous 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 number of received publications on each broker, which generates additional CPU load, especially for complex evolution functions. Furthermore, subscription optimizations such as covering can not be applied in a straightforward way.

4.3 Cached Lazy Evaluation ES

Our Cached Lazy Evaluation Evolving Subscriptions (CLEES) is a cross between PES and LEES. A CLEES is neither evaluated upon each publication nor does it generate a concrete version every fixed time interval. Instead, at the first time a publication with the subscription's attributes arrives, the predicate expressions are evaluated with the current time value, building a concrete subscription on-the-fly. The publication is then matched against this version. Furthermore, this variation is cached for a given time period, and any following publications are matched against this cached version. After the time period is expired, the cached version becomes stale and is discarded. When the next publication arrives, a new version is created and cached. In short, the subscription is evaluated at the time of the first incoming publication, and only re-calculated when a new publication arrives at a time that is beyond the caching threshold.

CLEES, in addition to the subscription predicates, has to indicate the time threshold TT which determines how long a concrete version of the subscription remain in the cache. Conceptually, this is similar to the evolution interval EI in PES, but the cached versions are created only on demand.

Referring back to our game example and assuming a 1-second threshold, Figure 3 shows a CLEES in a scenario where 3 publications arrive 1, 1.5 and 3 seconds after submission of the subscription. In this case, both the publications at 1 and 3 seconds force a re-calculation of the subscription while the publication at time 1.5 can reuse the cached version created at time 1.

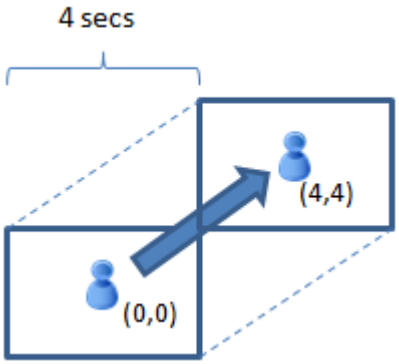

Figure 2: Lazy Evaluation Evolving Subscription

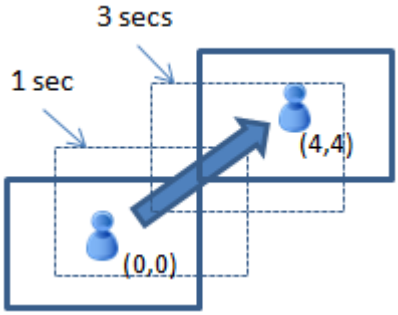

Figure 3: Cached Lazy Evaluation Evolving Subscription

5. IMPLEMENTATION

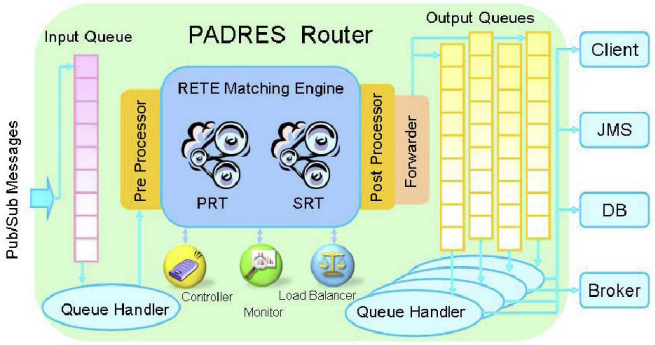

Figure 4: PADRES Router Architecture

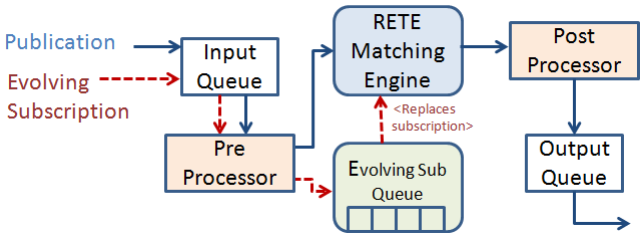

Figure 5: Periodic Evolving Subscription Architecture

In this section, we provide implementation details for our 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 clients with an API to generate evolving subscriptions.

5.1 General Broker Architecture

PADRES deploys a distributed broker network. Clients can connect to any broker in the network and perform subscribe, unsubscribe, and publish operations. A detailed diagram of the architecture of a single Padres broker, taken from[11], is depicted in Figure 4.

Every PADRES broker is comprised of an input queue, several output queues, and a router component. When a server receives a message, it is first sent to the input queue before being passed along to the router module, which identifies the message type (publication, subscription, unsubscription, etc.) and handles it accordingly. Subscriptions are assigned a new unique identifier *subId* and stored in the broker's internal Rete-based matching engine (called RETE). Publications are matched against stored subscriptions in order to determine matches. If that is the case, the publications are then moved to the corresponding output queues for delivery.

In the following, we first describe how each of the three designs for evolving subscriptions are implemented within a single PADRES broker. Then, we discuss the extensions needed to support a distributed broker network. In our descriptions, the variable t used in subscription predicates represents the number of seconds passed since the subscription was submitted. Furthermore, each PADRES broker maintains a variable containing the current global time T expressed as the number of seconds passed since an arbitrary base time.

5.2 Periodic Evolving Subscriptions

When a client submits a periodic evolving subscription (PES) to a broker, the broker creates an initial version of the subscription by setting $t = 0$ and evaluating the predicate functions, which is added to the RETE. Additionally, the original PES is added to a new data structure, the evolving subscription queue (ESQ). Subscriptions entering the queue are automatically ordered by the time of their next scheduled evolution. Each subscription assumes its time variable to change individually in reference to the time the subscription was submitted. Internally, this has to be transformed according to the global time within the matching engine. For instance, assume a subscription S with an evolution interval EI of 5 seconds is submitted at global time T. S is added to the ESQ with a *scheduled time* of $SCHEDT(S) = T + EI$ and a submission time of $SUBT(S) = T$. Any subscription with an earlier scheduled time will be sorted before S in the queue.

A group of handlers, each running on its own thread, monitor the queue and are in charge of updating the subscriptions. Whenever a thread becomes idle, it removes the first subscription S in the queue, creates a copy where $t = SCHEDT(S) - SUBT(S)$, waits until the current global time is equal to the scheduled time of the subscription, and then replaces the old version stored in the PADRES matching engine with the newly created one. This replacement operation is synchronized by a lock to prevent concurrency problems. Additionally, the original PES is reinserted into the subscription queue (ESQ) with the new scheduled time $SCHEDT(S)$ set to the current global time plus the evolution interval EI indicated in the subscription itself.

Publications are matched against the subscriptions stored in the RETE. Figure 5 shows the periodic evolving subscription architecture.

5.3 Lazy Evaluation Evolving Subscriptions

An LEES that arrives at a broker is first, at the preprocessing step, divided into two subscription components, sharing the same subscription id. The first one contains the non-evolving predicates, and is stored as usual in the RETE. The second one, containing evolving predicates, is stored in a new hash-table structure, the lazy evolution matching engine LEME. The subscription is also tagged with the current global time as its submission time $SUBT(S) = T$.

An incoming publication is first forwarded to the standard internal matching engine to find the set M_1 of subscription matches.

Additionally, we look for matches in the evolving component. For that, the publication is first labeled with the current global time T. Then, for each subscription in our LEME component, we check whether the attributes in its predicates are covered by the publication. If this is the case, we set $t = T - SUBT(S)$ in each of the predicate functions and evaluate the predicates. When all predicates evaluate to `true`, the subscription identifier is added to the set M_2 of time-dependent matches

Subscriptions that have their identifiers in both sets M_1 and M_2 have all their predicates matched by the publication, and the publication is put in the output queue to be sent to these subscriber. Our system also handles subscriptions that

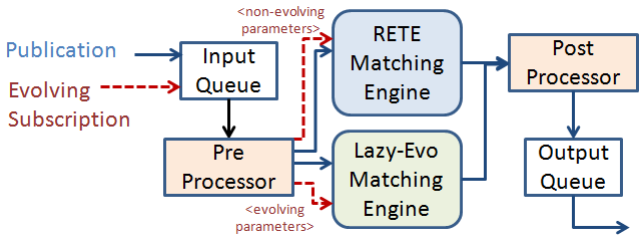

Figure 6: Lazy Evaluation Evolving Subscription Architecture

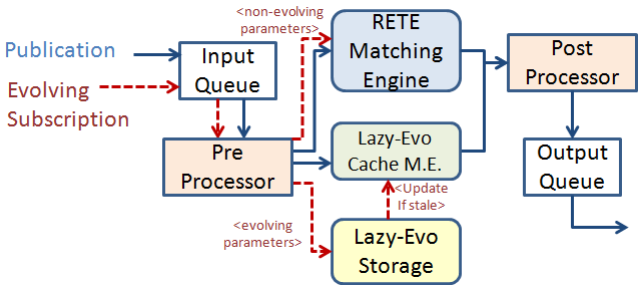

Figure 7: Cached Lazy Evaluation Evolving Subscription Architecture

contain only evolving or only non-evolving predicates. These subscriptions are only maintained in one of the matching engines, and flagged accordingly. A diagram of the lazy evaluation broker architecture can be seen in Figure 6.

5.4 Cached Lazy Evaluation ES

As with LEES, a CLEES subscription that arrives at a broker is separated into two subscriptions, one with the non-evolving predicates (which is fed to the internal matching engine), and another with the evolving predicates which is stored in a simple ordered list known as the lazy-evo storage.

When a publication arrives, it is sent to the standard matching engine to match the non-evolving predicates of our subscriptions, as for LEES. Furthermore, the publication is sent to the lazy-evo storage component. For each subscription S in the storage, we check whether the attributes in its evolving predicates are covered by the publication. If this is the case, we look whether we have a version of S in our Lazy-Evo cache that has not expired. Each cached subscription S is tagged with an $Exp(S)$. If $Exp(S)$ is equal or bigger than the current global time T, then the cached version is still valid. If this is the case, our Lazy-Evo Cache matching engine evaluates the publication on this cached version. Otherwise, we take the original subscription and replace t in the subscription predicates by $T - SUBT(S)$ to generate a new version and evaluate the publication on this version. Additionally, we put the newly created concrete subscription into the cache, discarding the old version if it exists. The expiry time of this newly created version is set to $T + TT$ where TT is the time threshold indicated in the CLEES. If a publication matches both the non-evolving and the evolving predicates of a subscription, it is forwarded to the subscriber.

A basic diagram of the CLEES architecture can be seen in Figure 7. In our implementation, we keep an extra cache with concrete versions of evolving subscriptions. In principle, we could integrate them into Padres' original engine, just as we did for periodic subscriptions. This would leverage Padres' innate optimizations for subscription matching. On the other hand, adding and removing subscriptions to the engine is more expensive and handling expiry adds additional overhead.

6. EVALUATION

6.1 Experimental Setup

6.2 Sensibility Experiments

7. CONCLUSION

8. 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. Publish/Subscribe Network Designs for Multiplayer Games. In *ACM/IFIP/USENIX 15th International Conference on Middleware*, 2014.

[4] 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.

[5] 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.

[6] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *TSE*, 2001.

[7] 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.

[8] 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.

[9] 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.

[10] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behavior. In *SIGKDD*, 1999.

[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] H. Jafarpour, B. Hore, S. Mehrotra, and N. Venkatasubramanian. Subscription subsumption evaluation for content-based publish/subscribe systems. In *Middleware 2008*, volume 5346 of *Lecture Notes in Computer Science*, pages 62–81. 2008.

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

[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] I. Koenig. Event processing as a core capability of your content distribution fabric. In *Gartner Event Processing Summit*, 2007.

[16] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, pages 447–457, 2005.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] C. Schuler, H. Schuldt, and H.-J. Schek. Supporting reliable transactional business processes by publish/subscribe techniques. In *TES*, 2001.

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

[24] 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.

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