ABSTRACT
We identify a class of content-based pub/sub applications with highly dynamic subscriptions. This includes location-based notification systems, predictive stock trading, and multiplayer games. The traditional method of handling subscription changes by engaging in expensive re-subscription protocols is inadequate when the workload is sufficiently large.

We propose evolving subscriptions as a technique to alleviate the overhead of subscription churn when handling regular patterns of subscription changes. Subscribers express as a function of time the evolution of its subscriptions. The pub/sub brokers can then independently modify the evolving subscriptions without requiring further communication with the subscribers.

In this demo, we present our design and implementation for supporting evolving subscriptions in the context of a multiplayer online game. We first show how evolving subscriptions are expressed to the pub/sub system. We then demonstrate its usage with a game client, which displays real-time performance measurements between the regular system and our enhanced version. We show the relative impact of using evolving subscriptions by running different experiments with our game engine.

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 [12], workflow management [5], stock-market monitoring [13], RSS filtering [11], complex event processing for algorithmic trading [10], and network monitoring and management [6].

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 [4] and availability purposes [8]. 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.

Even so, one weakness remains when dealing with high subscription churn. 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 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.

To this end, 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 avoids the expensive and frequent re-subscription process.

The goal of this demo is present our design of evolving subscriptions for content-based publish/subscribe systems. We show the capabilities of our implementation using the PADRES publish/subscribe engine [7]. We then demonstrate some practical usage of the work within the context of a multiplayer game called Mammoth. The performance of the solution across a range of scenarios is demonstrated through a monitor client.

2. SUBSCRIPTION MODEL

In a content-based pub/sub system, publications are sets of attribute/value pairs of the form:

Pub: \((a_1, \text{val}_1), (a_2, \text{val}_2), \ldots\)

where \(a_i\) is an attribute name and \(\text{val}_i\) is its corresponding value.

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

Sub: \({(a_1 \text{ op}_1 \text{ val}_1), (a_2 \text{ op}_2 \text{ val}_2), \ldots}\)

where \(\text{op}_i\) is the operator for the \(i\)-th predicate on attribute \(a_i\) with value \(\text{val}_i\).

A simple example predicate for attribute \(x\) could be \((x < 3)\). The operators are the typical comparisons such as \(<, >, \) etc. 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. 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 define evolving subscriptions by replacing each \(\text{val}_i\) with a function which returns a value depending on the time entered. Thus, an evolving subscription is of the form:

SubEv:

\(\{(a_1 \text{ op}_1 \text{ fun}_1(t)), (a_2 \text{ op}_2 \text{ fun}_2(t)), \ldots\}\)

where each function \(\text{fun}_i(t)\) has to return a value which is in the domain of the attribute \(a_i\) which it constrains.

For instance, a subscription \(S_{rev} = x < 1 + 0.5t\) increases steadily over time the range for \(x\) it is interested in. This expression could represent a subscription which is progressively moving along an x-axis at a constant speed.

Note that \(t\) represents the time units (e.g., in seconds) that have passed since the subscription was submitted.

3. STUDY CASE: MULTIPLAYER GAMES

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, in a conventional system, the area of interest is described by the subscription:

\(\text{Sub} = \{(x \geq -3), (x \leq 3), (y \geq -2), (y \leq 2)\}\).

Events in the game world (such as other players’ movements, pickup actions etc.), would then be sent as publications containing the coordinates of the location where the event takes place. For instance, the pick-up of an apple at coordinates \((4,3)\) results in a publication with attribute/value pairs \(\{(x, 4), (y, 3)\}\), and it would not match the above subscription.

A player moves by first selecting a destination. The player then progressively moves towards that destination. During the movement, he/she has to adjust its subscription because the center of his/her interest rectangle changes. In this setting, this means, whenever the player moves, he/she removes its current subscription and subscribes with his/her new location as the center of the rectangle. Doing this in a continuous fashion during the movement period is expensive. Thus, it is likely that one has to define a time interval, so that each time interval the player unsubscribes from his/her past rectangle and subscribes to the new one.

In contrast, with evolving subscriptions these continuous re-subscriptions are not needed. For example, when the player starts moving with a speed and a direction towards a destination so that he/she advances every second one x-coordinate to the right and one y-coordinate towards the top, then his/her interest can be expressed as an evolving subscription:

\(\text{SubEv} = \{(x \geq -3+t), (x \leq 3+t), (y \geq -2+t), (y \leq 2+t)\}\)

where \(t\) is initialized to 0 at the time of subscription. Now assume 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 matches. 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 true.
4. DESIGN AND IMPLEMENTATION

We propose three different designs for support evolving subscriptions: Periodic Evolving Subscriptions (PES), Lazy-Evaluation Evolving Subscriptions (LEES), and Cached Lazy-Evaluation Evolving Subscriptions (CLEES). Our implementation is built on top of the Padres pub/sub framework [7] by modifying its content-based engine to handle evolving subscriptions, and providing clients with an API to generate evolving subscriptions.

Padres allows to deploy a broker network over multiple machines. Clients can connect to any broker in the network to subscribe and unsubscribe, as well as publish messages.

4.1 Periodic Evolving Subscriptions (PES)

Figure 2 shows the periodic evolving subscription architecture. When a client submits an evolving subscription to a broker, the broker creates an initial version of the subscription (with \( t = 0 \)) and evaluating the predicate functions. This copy of the subscription is added to the matching engine employed by Padres, called 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, with the closest time at the top. Note that while 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 \( T \). The subscription is tagged with the current global time as its submission time \( SUBT(S) = T \).

At the scheduled time of a PES, a new copy of that subscription is created (with the time \( t \) set to the current time) and replaces the previous version of the subscription in the RETE engine. Publications are matched to the subscriptions stored in the original Padres matching engine, without any changes. Overall, the PES engine is a lightweight modification to the original Pub/Sub design and is thus considered an unoptimized baseline for support evolving subscriptions.

4.2 Lazy-Evaluation Evolving Subs (LEES)

A basic diagram of the lazy-evaluation broker architecture can be seen in Figure 3. An evolving subscription that arrives at a broker is first analyzed to isolate its evolving predicates from the non-evolving ones. Then two new subscriptions, sharing the same subscription identifier, are created. The first one, containing the non-evolving, time-independent predicates, is stored as usual in the internal matching engine. The second one, with the evolving predicates, is stored in a new hash-table structure, the lazy-evo storage.

When a publication arrives, it is again sent to Padres’ standard internal matching engine as usual to find the set \( M_1 \) of subscription identifiers of time-independent matches.

Additionally, we look for time-dependent matches. For that, the publication is first labeled with the current global time \( T \). Then, for each sub-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 \( \text{true} \), the sub-subscription’s 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 subscribers.

The advantage of LEES is that the overhead of evolution is only incurred at the time of publication matching. When the rate of evolution maintenance exceeds the volume of incoming subscriptions, it is beneficial to only check evolution with each publication, rather than periodically replace subscriptions in the RETE engine, as is the case in PES.

4.3 Cached Lazy-Evaluation ES (CLEES)

A basic diagram of the CLEES architecture can be seen in Figure 4. 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 again sent to Padres’ standard matching engine to match the non-evolving predicates of our subscriptions, in the same way as we described 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
satisfied by the publication. If this is the case, we look whether we have a time-independent transformation of $S$ in our Lazy-Evo cache that has not yet expired. Each cached subscription $S$ is tagged with an $Exp(S)$ (the expiration time of the current evolution of the subscription). 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 time-independent cached version. Otherwise (there is no cached version or the version has expired), we take the original subscription and replace $t$ in the subscription predicates by $T - SUBT(S)$ to create a time-independent version and evaluate the publication on this version. Additionally, we put the newly created version 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.

Just like with LEES, if a publication matches both the non-evolving and the evolving predicates of a subscription, it is forwarded to the subscriber. CLEES is an extension of LEES which employs a cache to temporally store the current status of evolving subscriptions. This is useful when several publications in succession match an evolving subscription during a short window of time. Rather than computing the evolving predicates for each incoming publication, we can retrieve the subscription from the cache.

5. DEMONSTRATION

We demonstrate our designs by leveraging Padres as the network engine for Mammoth, an online multiplayer game research framework [9]. We previously designed three different network engines using the features provided by Padres for Mammoth [3]. For the purpose of this demo, we employed the Area-based engine design found in the previous work modified to leverage evolving subscriptions. Whenever a player moves to a target destination, the client translates the movement as an evolving subscription which transitions between the current (start) position to the destination (end) position at the provided speed. This differs from the original design, where subscriptions are continuously removed and added from the system to reflect the updated position of the player.

We show the relative performance of our solution compared to each other and to previous non-evolving designs using our monitor client. This client was previously used in [2] to compare the aforementioned network engines. The monitor client allows the user to control a player character, move it around the map, and use the command console to give orders to other characters (to make them follow a player’s character, for instance). Performance metrics such as the number of subscriptions or the network load, are displayed in real-time as the player executes commands.

In order to properly compare the network engines, the demo system will simultaneously run the game on all of them at the same time. This allows the user to notice the impact of using evolving subscriptions based on the behavior of the players. For instance, a scenario where all the characters in the game are actively moving is better suited for the evolving engine which can handle the subscription churn.

We will finish the demo with our observations about the advantages and disadvantages of using evolving subscriptions, and of each proposed design.

6. REFERENCES