MothPad: Monitoring Pub/Sub Activity in Cyber-Physical Systems

César Canas  
School of Computer Science,  
McGill University  
Montreal, Canada  
ccanas2@mail.mcgill.ca

Kaiwen Zhang  
Technische Universität München  
Munich, Germany  
zhangk@cs.tum.edu

Bettina Kemme  
School of Computer Science,  
McGill University  
Montreal, Canada  
kemme@cs.mcgill.ca

Jörg Kienzle  
School of Computer Science,  
McGill University  
Montreal, Canada  
joerg.kienzle@mcgill.ca

Hans-Arno Jacobsen  
Middleware Systems Research Group  
arnojacobsen@msrg.org

ABSTRACT
Content-based publish/subscribe is an attractive option for disseminating event data in cyber-physical systems. To this end, we propose MothPad: a monitoring and visualization tool to demonstrate the performance of various pub/sub solutions within the context of location-based applications. MothPad consists of Mammoth, an online game research framework used as a cyber-physical system simulator, and PADRES, the publish/subscribe dissemination substrate. Both are instrumented and the performance is displayed in real-time using a monitoring client. We show the applicability of our approach through two case studies: network engines for online games and self-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. Brokers act as intermediaries, matching publications against subscriptions and forwarding matching publications to the subscribers. Publish/subscribe has been employed in the context of business process execution [14], workflow management [6], stock-market monitoring [15], RSS filtering [13], complex event processing for algorithmic trading [11], and network monitoring and management [8].

There are a growing number of modern applications in the context of cyber-physical systems and Internet of Things (IoT) systems which require real-time dissemination of event data. For instance, consider a traffic monitoring application, which receives streams of data from heterogeneous sources, such as loop detectors, cameras, and from on-board devices installed in cars (OBDS). This data must be aggregated by area and processed in real-time to derive meaningful actions in order to control congestion.

Cyber-physical systems (CPS) require a combination of accurate filtering in a system with dynamic queries while maintaining scalability and low latency. As pub/sub represents a good candidate for addressing these challenges, we propose MothPad: a framework for benchmarking pub/sub systems in the context of cyber-physical systems. MothPad consists of Mammoth, a virtual reality simulator, connected to PADRES, an extensible pub/sub middleware, both of which are instrumented and connected to a monitor client. This setup allows us to assess the performance of new pub/sub technologies designed for cyber-physical systems. For example, MothPad can be used to model, monitor, and visualize a traffic microsimulation application.

In this paper, we describe the architecture of MothPad, and present two use cases. First, we demonstrate the use of content-based pub/sub for location-based applications [2]. Second, we introduce the concept of self-evolving subscriptions for online games [3].

2. ARCHITECTURE AND DESCRIPTION
The purpose of MothPad is to provide users with a complete set of tools to quickly implement, simulate, visualize, and monitor a cyber-physical system (CPS). Therefore, MothPad is designed to be simple to use while providing a variety of flexible tools for the developer.

MothPad is created using Mammoth [10] and PADRES [9]. Mammoth is a Massively Multiuser Virtual Environment (MMVE) research framework used to study location-based applications, such as online games. PADRES is a distributed pub/sub middleware that can be used both to communicate between components and to express interest in a cyber-physical system location.
Figure 1: MothPad architecture

Figure 1 shows a general overview of our monitoring infrastructure. It consists of the following components:

1. Mammoth servers: One or more servers are used to manage the state of the virtual environment and to provide services to clients. They are core in enforcing semantics proper to the studied CPS.

2. Mammoth client: This component provides a graphical interface to visualize and interact with the virtual environment. The client provides monitoring tools such as overlays rendered on top of the world view. For example, overlays can be used to display pub/sub information such as subscriptions sent by the client. Remote-controlled clients can be created and assigned a set of AI behaviors (e.g., moving along a path). The AI component is extensible such that CPS-related scenarios (e.g., car traffic modeling) can be simulated.

3. PADRES brokers: Supporting both content and topic-based pub/sub, these brokers are organized in a specific topology and are responsible for distributing messages across subscribers and publishers. Subscribers express their interest using subscriptions while publishers submit publications to the brokers, where they are matched against known subscriptions and forwarded to corresponding subscribers. PADRES is extensible to support additional features, such as evolving subscriptions [4], aggregation [12], or data compression [7], which can then be monitored by MothPad.

4. Database: Every component in MothPad is instrumented to log relevant performance metrics to be stored in this database. The type of recorded data can be adjusted to tailor a specific application. By default, we deploy a MySQL database which is contacted using ODBC.

5. Graph client monitor: This monitor displays a series of modular charts generated using information from the database and displayed as auto-updating line graphs. This allows the user to observe and compare the effect of different actions in the environment as they occur.

Figure 2: MothPad for a Multiplayer Game

MothPad operates over a distributed architecture, with the option of scaling each type of component individually by deploying more nodes. The overhead incurred by MothPad over its components is the instrumentation needed for monitoring. This instrumentation consists of periodic writes to the database, and can be tuned to adjust its frequency.

Figure 2 shows an example of the visual components in MothPad. On the left side, a graph client monitor is displaying four different graphs with information about the total pub/sub traffic. The right side of the screenshot shows a Mammoth client rendering a game application and displaying rectangular overlays with information about that client’s pub/sub subscriptions.

3. NETWORK ENGINES FOR LOCATION-BASED APPLICATIONS

Content-based pub/sub systems are well-suited to distribute messages in location-based applications, where clients are interested in receiving updates related to a particular geographical area. The nature of pub/sub systems allows us to offload some of the typical responsibilities of these systems, like determining which clients are interested in which messages (a task normally performed by a server), into the pub/sub middleware.

Massively multiplayer online games (MMOGs), where thousands of players share the same virtual world, have become particularly popular. One of the biggest challenges of this type of game is that of scalability: to allow as many players as possible to join the game and interact among themselves. To address this challenge, we proposed in our Middleware 2014 paper "Publish/Subscribe Network Designs for Multiplayer Games" [2] a trio of pub/sub-based network engines that support the messaging requirements of a typical MMOG (or other location-based systems). Then, we were able to observe and compare the pub/sub activity of these engines in real time using MothPad, as realized in a demo presented at the same conference [1].

3.1 Proposed network engines

In this section, we describe the three different network engines developed in the context of a demo showcasing MothPad. The first two use a topic-based pub/sub system, while the last one uses a content-based approach. All network engines assign a topic to every client node and every in-game object in the system. Clients subscribe to their own topics
and can be contacted by publishing a message on that topic. Likewise, nodes can subscribe to an object to receive updates about it.

Our game worlds are divided into smaller sections known as tiles. These tiles are triangular in shape and obstacle-aware: they never cross impassible boundaries like in-game walls. Some of our network engines take advantage of the properties of these tiles to improve their performance [5].

**Object-Based Network Engine:** This engine, unlike the others, requires at least one node to run an interest management service. This service keeps track of player/object locations and determines when a client becomes interested in an object, at which point the service makes sure that a replica of the object is published under that client’s topic. After receiving a replica, the client subscribes to the corresponding object. In this way, to propagate a change on an object’s state, the update information is simply published under that object’s topic, to be received by all interested subscribers.

**Tile-Based Network Engine:** This engine assigns three special topic channels to every single tile in the game-world, as shown in Figure 3. These channels are used for most communications between nodes. The nodes that contain master copies of objects, known as master holders, subscribe to the replica request channel of the tiles where those objects are located. Client nodes subscribe to the notify update channel of all tiles they are interested in. When a player moves around the map, it determines by itself which tiles have become interesting and uses the request replica channel of those tiles to ask the master for any replicas located in the tiles. The master holders will then send the replicas to the clients using the replica reply channels of those same tiles. Propagating an object’s state change is done by publishing the update in the notify update channel of the tile that contains that object.

**Area-Based Network Engine:** This engine is similar to the tile-based one. It also uses three special topics but, instead of subscribing to the in-game tiles, nodes use a content-based approach to subscribe to rectangular areas of the map, as shown in Figure 4. The master holder of an object subscribes to an AoI rectangle on the replica request topic that is centered on the object’s position. A client, on the other hand, subscribes to the notify update channel of an area centered over the player they control. When the player moves around it publishes a petition for replicas in the request replica channel, centered over its player. The pertinent master holders will receive this request and send the appropriate replicas back, using the replica reply channel, making each publication on the position of the objects. Propagating an object’s state change is done by publishing the changes on the notify update topic, using the current position of the object.

### 3.2 Monitor usage

Using MothPad, we showcase and compare our three network engines, as shown in Figure 5. Users are able to control the actions of a few player characters, move them around the map, and use the command console to give orders to npc-controlled characters (to make them follow a player’s character, for instance). In this application, the performance metrics instrumented are the number of subscriptions, the volume of replica messages, and the volume of update notifications.

In order to properly compare the network engines, MothPad runs all three network engines simultaneously. One of the network engines is fully operational and handles all network communication, as required by the game, while the other two work under a “dummy mode” where they perform their usual tasks and save their performance metrics in the database but do not actually publish any messages. Thus, it is possible to perform an action and observe in real time the number and type of messages that each network engine generates in response to it. An example of the type of comparisons produced is shown in Figure 5, where each graph contains three readings, one for each engine.

Using MothPad, we can observe the performance of each network engine under a variety of scenarios. For instance, in a high-density situation, where many NPCs are actively moving in a constrained area, the object-based engine generates more subscriptions than either the tile-based or area-based approach. In contrast, a low-density scenario where NPCs are instructed to travel long distances has the inverse effect.

### 4. SELF-EVOLVING SUBSCRIPTIONS

While optimized for scalability, pub/sub is nevertheless weak to high subscription churn. In cyber-physical systems, this is the case when a subscriber 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 application, 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 object.

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.

We use MothPad to observe the performance impact of using self-evolving subscriptions in the context of cyber-physical systems. We use the monitor to compare in real-time three proposed approaches to supporting evolving subscriptions, as described in [4]. MothPad is used in this context for our demo presented at DEBS 2016 [3].

4.1 Subscription model

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

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

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:

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

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 \(P \equiv (x, 2)\) matches a single-predicate subscription \(S_1 \equiv \{(x < 3)\}\) but does not match subscription \(S_2 \equiv \{x < 1\}\).

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:

\[
\text{SubEv: } \{(a_1 \text{ op}_1 \text{ fun}_1(t)), (a_2 \text{ op}_2 \text{ fun}_2(t)), \ldots\}
\]

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

For instance, a subscription \(S_{\text{ev}} = x < 1 + 0.5 \cdot t\) 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.

4.2 Example: 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 6. In this case, in a conventional system, the area of interest is described by the subscription:

\[
\text{Sub} = \{(x >= -3), (x <= 3), (y >= -2), (y <= 2)\}.
\]

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 a traditional setting this means that, 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:

\[
\text{SubEv} = \{(x >= -3+t), (x <= 3+t), (y >= -2+t), (y <= 2+t)\}
\]

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 \((4 \geq -3 + 1, 1.4 \leq 3 + 1, 3 \geq -2 + 1, 3 \leq 2 + 1)\) return true.

4.3 Design and implementation

We propose three different designs for supporting 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 [9] by modifying its content-based engine to handle evolving subscriptions, and providing clients with an API to generate evolving subscriptions.

Periodic Evolving Subscriptions (PES): Figure 7 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 evaluates 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 \( EI = 5 \) seconds is submitted at global time \( T \). \( S \) is added to ESQ with a scheduled time of \( SCHEDT(S) = T + EI \) and a submission time of \( SUBT(S) = T \). Any subscriptions with an earlier scheduled time will be sorted before \( S \) in the queue.

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 to support evolving subscriptions.

Lazy-Evaluation Evolving Subs (LEES): A basic diagram of the lazy-evaluation broker architecture can be seen in Figure 8. An evolving subscription that arrives at a broker is first analyzed to isolate its evolving predicates from the non-evolving ones. Then two new sub-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-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 forwarded 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, so the publication is put in the output queue to be sent to the interested 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’s beneficial to only check for evolutions with each publication, rather than periodically replace subscriptions in the RETE engine, as is the case in PES.

Cached Lazy-Evaluation ES (CLEES): A basic diagram of the CLEES architecture can be seen in Figure 9. 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.

4.4 Demonstration

We demonstrate our designs by using MothPad. For the purpose of this case study, we employed the Area-based engine design described in Section 3 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 continuous removed and added from the system to reflect the updated position of the player.

The client, shown in the right side of Figure 10 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 with the graph monitor (left side of the figure) as the player executes commands.

In order to properly compare the three evolving subs designs, the system simultaneously runs the game on all of them. 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.

The client also shows a subscription overlay (as shown in the right side of Figure 10), showing the radius of the current subscription used by the client. This overlay whenever a new subscription is submitted, or an evolving subscription is updated. Using this overlay, it is possible to display the subscriptions in effect for all three designs, as well as the non-evolving subscription, to highlight the differences between the behaviour of each engine.

5. CONCLUSION

Pub/sub constitutes a good candidate for enabling large-scale dissemination of real-time data for cyber-physical systems. We developed MothPad to demonstrate the effectiveness of pub/sub using Mammoth, a virtual reality simulator, and PADRES, an extensible pub/sub system. Both are instrumented and connected to a monitor client which displays results in real-time. We demonstrate the applicability of our approach through two use cases: content-based pub/sub for location-based applications and self-evolving subscriptions for online games.

6. REFERENCES
