Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud

Julien Gascon-Samson, Franz-Philippe Garcia, Bettina Kemme, Jörg Kienzle
School of Computer Science, McGill University
Montreal, QC H3A 0E9, Canada
Email: {Julien.Gascon-Samson,Franz-Philippe.Garcia}@mail.mcgill.ca,{Bettina.Kemme,Joerg.Kienzle}@mcgill.ca

***Abstract*—This paper presents Dynamoth, a dynamic, scalable, channel-based pub/sub middleware targeted at large scale, distributed and latency constrained systems. Our approach provides a software layer that balances the load generated by a high number of publishers, subscribers and messages across multiple, standard pub/sub servers that can be deployed in the Cloud. In order to optimize Cloud infrastructure usage, pub/sub servers can be added or removed as needed. Balancing takes into account the live characteristics of each channel and is done in an hierarchical manner across channels (macro) as well as within individual channels (micro) to maintain acceptable performance and low latencies despite highly varying conditions. Load monitoring is performed in an unintrusive way, and rebalancing employs a lazy approach in order to minimize its temporal impact on performance while ensuring successful and timely delivery of all messages. Extensive real-world experiments that illustrate the practicality of the approach within a massively multiplayer game setting are presented. Results indicate that with a given number of servers, Dynamoth was able to handle 60% more simultaneous clients than the consistent hashing approach, and that it was properly able to deal with highly varying conditions in the context of large workloads.**

I. Introduction

The publish-subscribe (pub/sub) concept is an extremely popular communication paradigm that is used across a wide range of application domains because it provides an efficient and elegant way to decouple content producers (publishers) from content consumers (subscribers). Facebook and Twitter are two popular world-wide systems that are built upon the pub/sub paradigm: users produce content (publications) that is consumed by other users that specify interest in it.

The literature distinguishes between two types of subscription languages. In *content-based* pub/sub, publications are tagged with a set of attribute/value pairs, and subscriptions are expressed as predicates over attributes [21]. A publication matches a subscription if its attribute values satisfy the subscription predicate. There also exist content-based systems where subscriptions can be made using elaborate predicates that are computed directly on the data itself and not only on attributes of the data [22]. Content-based pub/sub systems have received plenty of attention in the research community over the last 15 years, as matching is a challenging task that is computationally intensive, and difficult to distribute. Recent efforts have looked at how to provide a scalable cloud-infrastructure for content-based pub/sub [18], [4], [26].

In contrast, *topic*-based pub/sub, also referred to as *channel*-based communiction, is widely used in industry, and many open-source or industrial products exist. Conceptually it is much easier than content-based pub/sub. Users declare interest in content by submitting subscriptions to specific topics, also called *channels* [11][1]. Publishers tag their publications with a channel name, and all users that have subscribed to that channel receive the message. Implementing a basic channel-based pub/sub server is simple. Despite its simple data model, channel-based pub-sub is widely used across various application domains, such as traffic alert systems, mobile device notification frameworks (such as Google Cloud Messaging (GCM) used for sending push notifications to Android devices), chat/instant messaging systems, extreme weather alert systems, Twitter, and many more. In this paper, we use multi-player games as an application example. In these virtual environments, players declare interest in other players or in regions of the game, that is, players and regions become channels; then game events, such as player movement and player/object interactions, are published on these player or region channels.

Just as content-based systems, channel-based pub-sub systems need to be scalable in order to handle large-scale applications. Furthermore, in order to be cost-effective, cloud-based solutions need to offer elasticity, making it possible to adapt the number of pub/sub servers as the workload changes in terms of number of clients and/or number of messages that need to be transmitted. Finally, many pub/sub applications including virtual environments have stringent response time requirements; thus publications need to be disseminated to the interested clients as fast as possible, which requires minimizing the processing time within the pub-sub system as well as minimizing the number of hops a publication has to take from the publisher to the subscriber [25].

A common way for load-distribution in channel-based pub/sub is to distribute the channels across several pub/sub servers using consistent hashing [5]. With this, each server is responsible for a set of channels. To determine the distribution, each server has a set of virtual identifiers, channels are hashed to the same domain as the identifier space and assigned to the server with the closest virtual identifier. When a new server is added, existing servers can give an equal share of their virtual identifiers and the channels assigned to this identifier to the new server. Correspondingly, when a server leaves, it distributes its identifiers/channels to the remaining servers.

[1] We use the terms *channel* and *channel*-based pub/sub throughout the paper.

Consequently, each server is always responsible for an equal number of channels. Consistent hashing prevents the need to re-map all channels when a new server is introduced. However, consistent hashing has many drawbacks. More specifically, it can only work under the assumptions that channels are equally distributed across all virtual identifiers and that the load on each channel is equal. In real-world applications, these assumptions do not necessarily hold.

In this paper we present Dynamoth, a scalable, elastic cloud-based channel-based middleware that supports any number of publishers, subscribers and publications. We aim at supporting any application making use of channel-based pub/sub with a specific emphasis on applications that require strict latency bounds. In order to reach our scalability goals, instead of relying on consistent hashing, Dynamoth proposes a hierarchical load balancer that operates (1) at the system-level (macro load balancing) and (2) at the channel-level (micro load balancing).

At the *system-level,* Dynamoth proposes a dynamic mechanism to distribute channels across multiple pub/sub servers. Whenever the popularity of some channels changes, new channels are introduced or channels are removed, Dynamoth dynamically adjusts the load on individual servers. Furthermore, when the total load increases or drops significantly, Dynamoth automatically adds or removes pub/sub servers by spawning/despawing nodes in the Cloud to optimize Cloud infrastructure-associated costs.

At the *channel-level*, Dynamoth is capable of handling cases where specific channels have extremely high load, possibly orders of magnitude larger than other channels. Such situations can happen if a channel has a very large number of publishers, subscribers and/or publications.

In summary, this paper provides the following contributions:

- We provide a scalable channel-based pub/sub infrastructure where channels are distributed across many pub/sub servers. Clients are made aware of the channel assignments so that they can send their publications and subscriptions to the correct pub/sub servers, leading to low latency as no indirections occur.
- Our approach provides load-balancing and elasticity at the system-level. Channel assignments can change, and servers can be added or removed from the configuration on the fly as the workload patterns change. Reconfigurations do not interrupt message processing, and messages are guaranteed to be received by all subscribers despite the reconfiguration.
- Our approach provides load-balancing and elasticity at the channel-level. Highly-loaded channels can be replicated across several pub/sub servers in order to avoid overload or response time violations.
- We have implemented Dynamoth on top of an existing open-source pub/sub system, namely Redis, without any changes to Redis itself. Thus, we can take advantage of an already existing, highly-optimized pub/sub system. We believe that the concepts presented in this paper could be implemented on top of other pub/sub systems, as long as they offer the standard pub/sub interface.

We have evaluated the performance of Dynamoth by conducting extensive experiments over a massively multiplayer game application, showing that it performs significantly better than consistent hashing, and that it is able to adapt quickly to complex workloads that continuously change. Notably:

- Our experiments on a large-scale game application revealed that Dynamoth was able to handle 60% more simultaneously active players with the same set of pub/sub servers than consistent hashing.
- Our experiments revealed that Dynamoth was properly able to handle large-scale workloads that were subject to high variation over time, while minimizing the number of required pub/sub servers, and keeping average latency low.

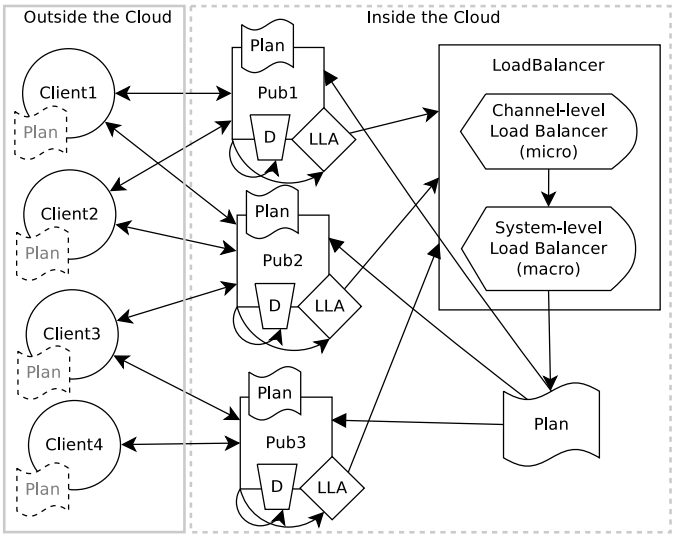


Figure 1: Overview of the Dynamoth Architecture

The remainder of the paper is structured as follows: section II describes the architectural design of Dynamoth; section III describes how Dynamoth is able to perform load balancing in order to scale; section IV describes the mechanism by which Dynamoth is able to reconfigure itself without impacting performance; section V describes the implementation and the experiments that have been run and section VI discusses other approaches in the literature.

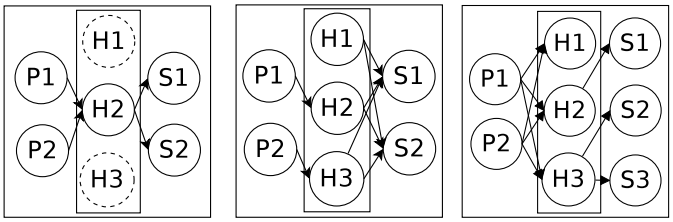
II. Dynamoth Middleware

A. Architecture

The Dynamoth architecture is depicted in Figure 1. The core of the system is a set of standard, independent pub/sub servers (`Pub1` to `Pub3` in the figure) that handle message dissemination between all clients. In our implementation, we use Redis[2], but it should be simple to replace Redis by any other pub/sub middleware with a standard API.

In our approach, we deploy on each node in the Cloud a standard pub/sub server and two further components, a *local load analyzer* (LLA) coupled with a *dispatcher* (D). The local load analyzer performs real-time monitoring of the load on the pub/sub server. The dispatcher module is needed during system reconfiguration to guarantee that messages are forwarded to all subscribers.

[2]http://redis.io/



(a) No Replication (b) All-Subscribers Repl. (c) All-Publishers Repl.

Figure 2: Channel Replication Strategies

There exists one load balancer node in the cluster that gathers input from all the local load analyzers and aggregates all the metrics. It determines if a configuration change is needed (e.g. whether some pub/sub server is overloaded). If this is the case, it determines how to balance the load in the system. To this aim, Dynamoth proposes the concept of a *plan,* which is used to resolve to which pub/sub server a given publication or subscription should be sent to. The plan is a more elaborate version of a lookup table where the keys are the channels (topics) and the values are the list of servers that should be used for each channel. Whenever a new plan is generated, it is propagated to the dispatchers located on the pub/sub server nodes. The dispatchers need this plan to ensure that all messages are forwarded to all subscribers during reconfiguration.

Clients interact with the system through the Dynamoth client library. The client library exposes a standard pub/sub API. In our implementation, it is identical to the original Redis API. The Dynamoth client library uses a client-specific plan to determine to which of the pub/sub servers to send publications and subscriptions to. The actual sending of messages is done using the standard Redis client library.

B. Mapping Channels to Pub/Sub Servers

We support three approaches of how to assign channels to pub/sub servers (see Figure 2). In the figure, we consider a channel c, a set of publishers (*P*-nodes) that will be publishing to c, a set of subscribers (*S*-nodes) that will be receiving publications flowing through c (subscribers of c) and a set of pub/sub servers (*H*-nodes) that will be used to route publications from the publishers to the subscribers.

In most cases, a channel c is assigned to one pub/sub server *H*, and clients send subscription requests and publications for channel c to H (figure 2a). This single-server mapping will work for most channels.

However, in some scenarios, the number of subscribers, publishers and/or publications on a given channel c might be too large for only one pub/sub server. For instance, if a given channel c has a very large number of subscribers, then this might lead to too many simultaneous connections on the pub/sub servers. This can also lead to important increases in the message processing delay since the pub/sub server has to send the same messages to all subscribers at the same time. If c has too many publications, then this could cause too many messages to flow on c, thus potentially overloading the pub/sub server (even if the server is in charge of that sole channel), or overflowing individual connections between the pub/sub server and a subscriber. To address the requirements of such heavy-load channels, Dynamoth performs load-balancing at the channel-level by allowing for more than one pub/sub server to map to any given channel. We refer to this as *channel replication*. Dynamoth proposes two channel replication approaches depending on the overload situation. In both approaches, several pub/sub servers are responsible for the channel. But they differ in the way subscribers subscribe to the channel and how publishers publish their messages. In both cases, it is important that all subscribers receive all publications regardless of which pub/sub server has been used to process the publication.

1) All-Subscribers Replication: With all-subscribers replication, all subscribers send their *subscription requests* for c *to all* the pub/sub servers responsible for c while *publishers* send their message for *c to only one* of the servers responsible for *c*. In Figure 2b, the three servers H_1, H_2 and H_3 can be used to process publications flowing through c; each publisher sends its publications through a random server (in this case, P_1 publishes to H_2 and P_2 publishes to H_3) and all subscribers have subscriptions to c on all servers H_1, H_2 and H_3 thus making sure that whichever server is used, all subscribers receive all publications. This replication scheme is relevant if there is a very high number of publishers and/or messages to be transmitted on c but the number of subscribers is within the limit of what a single pub/sub server can handle in terms of connections. An example from gaming could be a channel which is used by clients to send position updates within a region of a virtual world. The publishers are the players that control avatars located in the region, publishing position and state updates at a high frequency. The subscribers are the game servers responsible, for example, to perform interest management for the region. With all-subscribers replication, a player chooses a random pub/sub server for its publications, thus distributing the high message load over several servers.

2) All-Publishers Replication: With all-publishers Replication, a subscriber *subscribes to only one* of the pub/sub servers in charge of *c*, while *publishers* send their publications *to all* servers. In Figure 2c, once again all three servers H_1, H_2 and H_3 can be used to process publications flowing through c. Each publisher sends its publications to all servers, which requires sending n messages (*n* being the number of pub/sub servers that handle *c*), while each subscriber subscribes to c on one specific pub/sub server, and thus, receives the message once. This replication scheme is relevant if there is a very high number of subscribers for channel c, but a relatively low number of messages, because each publication message is sent n times. An example from gaming could be some form of broadcast channel through which a game server communicates some world-wide events that are of interest to all players. Without replication, a single pub/sub server might take a long time to disseminate such a publication to all subscribers, violating response time requirements. In contrast, when the message is sent to many pub/sub servers, the pub/sub servers can forward the message in parallel to the many subscribers.

C. Bootstrapping and Initial Conditions

Initially, a Dynamoth system contains a set of one or more pub/sub servers and an initial global plan ("plan 0") which does not provide any specific channel mapping. When a plan does not contain mapping information for a channel (at startup and whenever new channels are dynamically created), it uses a consistent hashing algorithm to map the channel to a pub/sub server. Over time, the plan is updated when channels are assigned to pub/sub servers because of load balancing and channel replication. All dispatchers on the pub/sub server nodes have a copy of the complete current global plan.

Each client *C* maintains a client-specific partial plan $P(C)$, subsequently called *local plan.* At connection time it is empty and *C* uses consistent hashing to determine to which server to send subscriptions and publications for a given channel c. If the local plan is out of date and as a result an incorrect server is chosen, the server ensures that the subscription/publication reaches the correct server. Furthermore, it informs the client about the correct pub/sub server. Thus, over time, C updates its plan $P(C)$ with the correct channel/server assignments. Similarly, whenever the global plan changes, C is informed in this lazy manner. However, at any time $P(C)$ only contains information about channels that the client actually uses. Assuming that in large-scale settings, each client only interacts with a small subset of all channels, this approach keeps the plan information at the client side as small as possible. Minimizing the local plan size also enables the middleware to support multiple applications concurrently (in a gaming context, that could be many independent instances of a multiplayer game). Exactly how client plans are updated over time and during reconfiguration is described in section III-B.

III. Load Monitoring and Plan Generation

To perform load balancing and create new plans, Dynamoth must know the current load on all pub/sub servers. Moreover, it must estimate as precisely as possible the load distribution that will be obtained after the rebalancing occurs based on the current load. Our framework is able to accurately monitor and measure the load for every channel, on every pub/sub server, with minimal overhead, without the need to alter the pub/sub server software (Redis in our case).

A. Load Monitoring: Local Load Analyzers

To enable load monitoring, each node that runs a pub/sub server also runs a *local load analyzer (LLA)*. The role of the LLA is to continuously gather extensive load metrics for every channel managed by the pub/sub server. The recorded metrics for every time unit t (t is one second in our experiments) include the number and list of publishers, the number of publications, the number and list of subscribers, the number of sent messages, and the incoming and outgoing number of bytes transmitted.

The LLA is notified when the pub/sub server receives new subscriptions and unsubscriptions. This allows the LLA to discover new channels and keep track of the subscribers. In order to collect all metrics, the LLA registers as an "observer" to every channel hosted onto the local pub/sub server, and therefore receives a copy of every publication. The fact that the LLA runs locally on the same machine as the pub/sub server greatly reduces communication overhead and does not use any local bandwidth. Our empirical observations showed that: (1) running the LLA module had very limited CPU overhead and (2) the outgoing bandwidth of the pub/sub servers got saturated much more quickly than the CPU. This second observation can be explained by the fact that most publications will be sent to many subscribers. Therefore, our rebalancing algorithm doesn't take CPU load and incoming bandwidth into account since through our experiments, they were not a limiting factor, except for some specific cases where there is a huge amount of subscribers for a given channel and a significant amount of publications. This can lead to high CPU usage and is handled by using channel replication.

All LLAs send an aggregate update message at a predefined interval to the Load Balancer node. This message contains all metrics for all channels for all time units t_i since the transmission of the last update message, as well as additional information such as the theoretical maximum outgoing bandwidth supported by that server node, as well as the measured outgoing bandwidth on the network interface.

B. Generating a New Plan

Upon receiving the metrics from all Local Load Analyzers (LLAs) for every time unit *t*, the Dynamoth *load balancer* (LB) first computes the load ratio for all pub/sub servers. The load ratio LR_i for a given server i is defined as the measured outgoing bandwidth M_i divided by the maximum outgoing bandwidth supported by the server T_i (eq. 1).

$$LR_i = M_i / T_i \tag{1}$$

The LB then decides if a new plan should be generated or if the current plan should be kept. New plans are generated only after at least T_{wait} time units have elapsed since the last plan generation to make sure that most of the configuration overhead of the last plan change is completed before the next one is triggered. A new plan is generated using the Dynamoth rebalancer module in a two-step process: (1) channel-level rebalancing and (2) system-level rebalancing.

1) Channel-level Rebalancing: In this step, the LB checks the number of publishers, subscribers and publications on each channel and determines whether some channels could benefit from replication (all subscribers or all publishers). Algorithm 1 outlines how the LB determines whether any given channel should use any of the replication schemes. The first step involves computing the *publication-to-subscribers ratio* (P_{ratio}) and checking whether this ratio is above a given threshold. We also check whether we have a minimum amount of publications before triggering replication, since replication makes sense in cases where a given channel uses significant resources that cannot be managed by one pub/sub server. If both conditions are true, then the all-subscribers replication scheme will be used. The number of servers $N_{servers}$ that should be used is then computed (line 5), and all subscribers will connect to all $N_{servers}$ servers. A similar approach is used to determine if the all-publishers scheme should be enabled

instead (lines 3;8-10) and decide on the number of servers. At this step, we ensure that we have a minimum amount of subscribers to make sure that replication is relevant. If the conditions regarding all-subscribers and all-publishers are not met, then replication is not used for this channel (or is cancelled if it was active).

One corner case is the case where the amount of publications and subscribers are both very large (not shown in our algorithm due to space constrainsts); our system will then use the all-subscribers scheme since the all-publishers scheme causes publications to be sent to all pub/sub servers, which is more costly.

Upon enabling a given replication scheme for a given channel or if replication is already enabled for this channel but the LB determines that additional servers should be used (P_{ratio} or S_{ratio} increases), the load balancer selects the least-loaded servers first. Similarly, if replication servers need to be freed (the LB determines that replication is not needed anymore or that the number of servers can be reduced), then the busiest servers will be freed first.

```
1  begin
2      P_ratio = #publications/#subscribers;
3      S_ratio = #subscribers/#publications;
4      if P_ratio > AllSubs_threshold and
       #publications > Publication_threshold then
5          N_servers = P_ratio/AllSubs_threshold;
6          replicate(ALL_SUBSCRIBERS, N_servers);
7      end
8      else if S_ratio > AllPubs_threshold and
       #subscribers > Subscriber_threshold then
9          N_servers = S_ratio/AllPubs_threshold;
10         replicate(ALL_PUBLISHERS, N_servers);
11     end
12     else
13         replicate(NO_REPLICATION);
14     end
15 end
```

Algorithm 1: Determining if replication should be used

2) System-level Rebalancing: In this step, the LB analyzes the load on each pub/sub server. In general, Dynamoth can perform two types of load rebalancings: (1) a high-load rebalancing, which is needed when one or more pub/sub servers are overloaded in order to bring the load down, and (2) a low-load rebalancing, which takes place in the case where one or more pub/sub servers are underloaded in order to free servers that are not required anymore with the ultimate goal of shutting them down. Because pub/sub servers are most likely deployed in the Cloud, the LB aims at being efficient regarding the number of servers that need to be used in order to save costs, while maintaining adequate performance. The two next subsections explain our current LB algorithms for high-load and low-load rebalancing. In real commercial systems, more elaborate heuristics could be used.

3) High-Load Rebalancing: If there ia a pub/sub server H_i with a load ratio LR_i that exceeds a given threshold LR^{high}, then a new *high-load* plan $P*$ *must* be generated so that $P*$ ensures that the load returns below a safe threshold for all servers. If this is not possible, then one or more additional servers have to be rented from the Cloud.

Algorithm 2 describes our heuristic for generating a plan to reduce the load on overloaded servers. The algorithm repeats as long as there is at least one pub/sub server with an estimated load ratio above LR^{high} . The pub/sub server with the highest load ratio (H_{max} with load ratio LR_{max}) is selected. Then, as long as the *estimated* load ratio $\overline{LR_{max}}$ remains above a certain threshold LR^{safe}, we do the following: (1) obtain the pub/sub server with the lowest load ratio (H_{min} with load ratio LR_{min}); (2) obtain the busiest channel c^{out}_{max} on H_{max}; (3) migrate this channel from H_{max} to H_{min} in the new plan $P*$, and (4) estimate the load ratio $\overline{LR_{max}}$ (on H_{max}) that we would get if $P*$ was applied. Of course, the estimated load on the server that receives the channel will be recalcuated as well to make sure that we do not overload that server.

```
1  begin
2      P* = P.copy();
3      while true do
4          (H_max,LR_max) = max(LR_i ∀ H_i);
5          if LR_max < LR^high then
6              return P*;
7          end
8          LR̄_max = LR_max;
9          while LR̄_max ≥ LR^safe do
10             (H_min,LR_min) = min(LR_i ∀ H_i);
11             c^out_max = getBusiestChannel(H_max);
12             P*.migrate(c^out_max, H_max → H_min);
13             LR̄_max = estimateLR(P*);
14         end
15     end
16 end
```

Algorithm 2: Generating a new high-load plan

4) Low-Load Rebalancing: If the global load ratio (averaged LR_i for all pub/sub servers i) is below a given threshold, then one or more servers can be freed. This operation is less critical for performance reasons, but nevertheless essential for cost saving purposes. Channels from the lowest loaded server are slowly migrated to the other servers as long as the load on the other servers stays below a given limit. When a server has no more channels, it is deallocated. If at any point the global load ratio increases such that it becomes higher than the low-load threshold, then the low-load rebalancing will be interrupted (and, if needed, a high-load rebalancing can be triggered). Due to space constraints, the detailed low-load rebalancing algorithm, similar in spirit to the high-load rebalancing algorithm, is not presented in the paper.

For all algorithms, the values of the various threshold parameters were determined empirically based on the capabilities of the machines at our disposal. Of course, with different hardware, those values would most likely need to be adjusted. In future work, one could explore the idea of having a mechanism to automatically set and update thresholds based on real-time conditions.

IV. Reconfiguration

Upon determining that a new global plan $P*$ should be applied, all stakeholders need to be informed. However, sending a new global plan to all clients at reconfiguration time would create a huge message overhead. Furthermore, global plans contain information about all channels, while individual clients are likely only interested in a few of these channels and therefore should only receive partial plan information on a need-to-know basis. Thus, we use a lazy scheme. At connection time, clients use consistent hashing to determine pub/sub servers and get to know the true location for a channel only when they actually send their first message for this channel. Similarly at reconfiguration time, their partial plans are only updated on a need-to-know basis using a lazy propagation technique.

For this to work and to not loose any subscriptions and publications that are sent to the wrong pub/sub servers, the servers must be able to handle wrongly addressed messages.

Initialization: Whenever a client does not have any server information about a channel, it sends the publication/subscription request to the server determined by consistent hashing. If this is not the correct server, the server sends a message back to the client informing it about the correct server. The client updates its local plan and then sends the message to the correct server.

Subscriber Change: Whenever a plan change moves a channel c from server H_0 to server H_1, all subscribers need to be informed and move their subscriptions from H_0 to H_1. We don't do this immediately for all channels, because this could lead to a spike of unsubscriptions and subscriptions at the time of reconfiguration, possibly causing performance bottlenecks. In order to stagger the reconfiguration of channels, we notify subscribers of the switch of channel c together with the first publication on c after the plan changes.

Publishing on old server: Furthermore, publishers are also not informed immediately. In fact, in our system, there is actually no central authority that would know the content of the local plans of the clients, as they all are maintained individually by the clients themselves. Instead, when a channel c has moved from server H_0 to H_1, and H_0 receives a publication message on c, it informs the publisher about the change, so it can update its plan and send its next message to the correct server. At that time, both H_0 and H_1 might have subscribers for c, as some but not all of the subscribers might have updated their subscriptions by then. Therefore, H_0 forwards the message to all subscribers of c still connected to it, and also sends the publication to H_1 so that it can deliver the message to all subscribers of c already connected to H_1.

Publishing on the new server: Finally, a publisher might already know the new location of c and send a publication for c to H_1 while there are still some subscribers connected to H_0. Therefore, H_1 forwards the publication not only to its local subscribers but also to H_0 so that it can disseminate it to the subscribers still connected to H_0. Such forwarding needs to occur until H_0 does not have any subscribers anymore.

For replicated channels, reconfiguration is more complicated as there are multiple pub/sub servers that are in charge of publications and subscriptions for a given channel. But in principle, it follows the same line of reasoning as described above. For space reasons, we cannot describe the details but refer to [13].

A. Reconfiguration Details

A challenge in our system is that we rely on ready-to-use pub/sub servers that cannot be modified. Thus, the forwarding functionality is implemented in the dispatchers that are collocated on the pub/sub server nodes as described in the following subsection and illustrated in figure 3.

1) Reconfiguration Setup: Each dispatcher has connections to all other pub/sub servers in order to be able to forward messages. Whenever a new global plan *P** is created, the LB sends it reliably to all dispatchers. Upon receiving a new plan P* such that server H_0 was responsible for channel c in the old plan and server H_1 in the new plan P*, the dispatchers of both H_0 and H_1 subscribe locally to channel c to receive all publications. Furthermore, the dispatchers intercept subscription and unsubscription requests submitted to their local pub/sub servers.

2) Incorrect Pub/Sub Server: Figure 3a illustrates by example what happens if a publication message m on channel c goes to an incorrect pub/sub server H_0 (step 1). The publication is first sent to the subscribers still using H_0 for c (C_2 in our example). As the dispatcher D_0 is also subscribed to c, it also receives the publication. If this was the first publication on c after the new plan $P*$ was received, D_0 publishes a <*switch to* H_1> message to c on H_0 in order to ask all subscribers to switch to H_1. H_0 then forwards this switch message to all subscribers, who then update their local plan and transfer their subscription to H_1 (steps 6, 7 and 8). D_0 also publishes the original publication to c on the new server (H_1) which delivers it to its own subscribers, if any (steps 3, 4 and 5). Note that some steps might execute concurrently (steps 2 and 3, or steps 4 and 6 for example).

3) Correct Pub/Sub Server: The case where a publication m on c is sent to the correct pub/sub server H_1 is much simpler (step 1 in figure 3b). H_1 delivers it to local subscriber C_2 (step 2). D_1 also receives m, and publishes m to c on the old server H_0 (steps 3 and 4). Finally, H_0 delivers *m* to C_1. Note that there might be a client who has already subscribed to c on H_1 but not yet unsubscribed from H_0. This client receives the message twice. Our client library ensures that each message is only delivered once. For that, we use the standard approach of globally unique message identifiers.

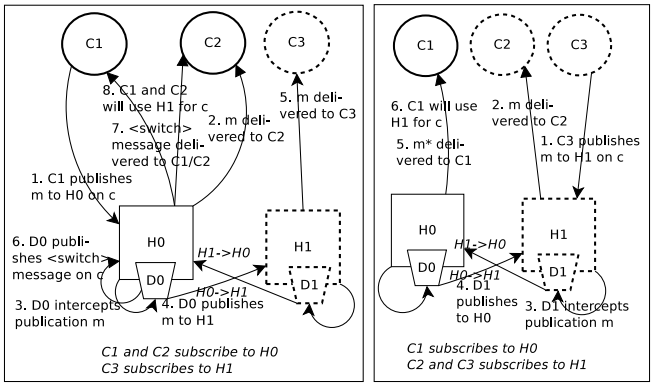
4) Client Subscribing and Moving a Subscription: If a client has outdated plan information for a channel c (using consistent hashing or having an outdated plan), then this client might send a subscription request for c to an incorrect server H_0. The dispatcher for H_0 then notifies the client that it subscribed to c on an incorrect server. Upon receiving such a message, the client immediately updates its local plan, subscribes to c on H_1 and unsubscribes from c on H_0. The same process is used when a client is asked to move an existing subscription: it subscribes to the channel on the new server and unsubscribes from the same channel on the old one.

5) Duration of Forwarding and Dispatcher Subscriptions: The question arises how long dispatchers should subscribe to channels and forward messages.

The dispatcher on H_1 must forward messages it receives for c to H_0 as long as there are still clients that are subscribed to H_0 instead of H_1. Thus, in order to avoid unnecessary forwarding, the dispatcher on H_0 notifies the dispatcher on H_1 as soon as there are no subscribers for c on H_0 anymore. The dispatcher on H_1 then stops forwarding messages.

The dispatcher on H_0 forwards to H_1 messages for c it receives from publishers that don't know yet about the switch. In principle, it could simply send to the publisher the information about the new server H_1, and the publisher could then republish the message on H_1. However, for performance and cost reasons, H_0 forwards the message directly to H_1, because this communication happens inside the Cloud. Over time, there will be less and less publishers that publish on the wrong pub/sub server (H_0). To avoid that the dispatcher on H_0 must be subscribed to c forever, we employ a timeout mechanism. Each client associates with each channel c in the local plan a timer. The timer is reset whenever the client sends or receives a publication on this channel, or when the server for this channel changes. When the timer expires and the client is not subscribed to the channel, then the client removes c's entry from the plan. Should it later try to subscribe to c or send a publication to c, it connects to the server that is determined through consistent hashing (just like during startup). The dispatcher on H_0 uses the same timer. It sets the timer for c when a new plan moves c from H_0 to another server H_1. It stops listening to c and forwarding messages to H_1 when the timer expires, because at this time no client has the outdated information for c anymore.

The dispatcher of the server H_c, i.e., the server for a channel c determined by consistent hashing, is always subscribed locally to c. Thus, it can determine when publications for c are sent erroneously to H_c and let the senders know the real server that is responsible for c. With this, whenever a client sends a message to the wrong server (be it a server based on an outdated plan or the server determined through consistent hashing), the dispatcher on that server receives the message and informs the client about the reconfiguration.



(a) Incorrect Server H_0 (b) Correct Server H_1

Figure 3: Handling Publications during Reconfiguration

V. Experiments

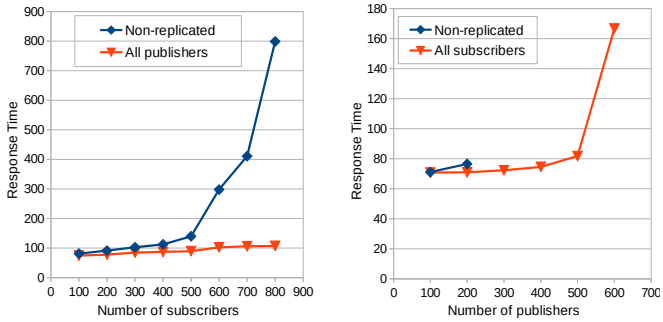
A. Implementation

Our Dynamoth implementation is highly modular and consists of around 110 Java classes and 10,000 lines of code. All Dynamoth components and algorithms are implemented as described in the paper. All inter-component communications are done using the pub/sub primitives offered by the Dynamoth API. The dispatcher and local load analyzer modules reuse the pub/sub interface, and standard Redis instances are used as the pub/sub servers. They are independent and do not communicate with each other. Because of this, any individual Redis instance can be replaced with any other pub/sub middleware as long as it support the basic pub/sub primitives.

As application we use a muliplayer online game (MOG) application. Mammoth [16] is a game engine for MOG that was developed at McGill University as a testing and research framework. Researchers are able to use the system to conduct all kinds of experiments related to MOG games in a realistic environment. Mammoth has a very modular architecture that allows easy replacement of components. One such component is the network infrastructure, which handles the transmission of all messages among all nodes. Mammoth requires the network engine to provide a channel-based pub/sub API as messages to updated players, messages to send replicas to clients and general broadcast messages by the server are sent using the pub/sub paradigm. Thus, we used Dynamoth as the network engine for Mammoth. For our experiments we used a specific sub-game within Mammoth, RGame, in which players are controlled by a simple AI that repeatedly chooses a random point on the map, moves the player towards that point and then takes a short break. The game world is split into a set of square tiles. Players subscribe to the tile in which they are located in, and publish their own state updates on the tile. Thus, all players receive update messages from all other players in the same tile. As players continuously move around, this application generates many subscriptions and unsubscriptions to tiles, and update position publications on these same tiles.

B. Experimental Setup

All experiments were conducted on 80 machines of the labs of the School of Computer Science of the McGill University over which we distributed client and server nodes. While each pub/sub server node received exclusive access to one of the lab machines, clients had to share machines in order to provide scalability results. All lab machines have dual or quad-core processors and at least 4 GB of RAM. Both client and server nodes were run in the same LAN. In order to emulate a typical Cloud setup, where the pub/sub server nodes would run in the Cloud connected by a LAN and the clients access the servers over a WAN, we adjusted our latency measurements using the King dataset [14] (a study that gathered millions of latency measurements between arbitrary DNS servers). We filtered the dataset to keep only measurements from North America. For each inbound publication message, we do the following: (1) if the publication comes from an infrastructure



(a) All Publishers (b) All Subscribers

Figure 4: Replication Experiments

node[3] and goes to a client node, then we sample one value from the dataset; (2) if the publication comes from a client node and is received by an infrastructure node, then we sample one value and (3) if the publication comes from a client node and is received by another client node, then we sample two values (round-trip). Each message received by the Redis middleware is put in a queue and is delivered to the application layer only after a timer corresponding to the sampled latency value(s) expires. Our experiments revealed that this delaying mechanism produced latency measurements comparable to what we could expect from running our infrastructure servers in the Cloud.

C. Experiment 1: Channel-level Scalability

We first assessed the scalability of the channel-level (micro) load-balancing capabilities of Dynamoth by running experiments with both replication schemes proposed by Dynamoth: "all publishers" and "all subscribers". For that, we run a set of micro-benchmarks that focus on specific overloaded channels.

1) All Publishers: In this experiment, we connected up to 800 subscribers to a given channel c. In our setup we ran 10 subscribers per machine. One publisher client sends 10 publication per second on channel c. This experiment was first attempted with replication disabled (only one pub/sub server was handling channel c) and then with replication enabled over 3 servers (3 pub/sub servers were in charge of server c). Under the "all publishers" model and under the replicated configuration, the publisher was sending its publications to all 3 servers and every subscriber was randomly subscribing to c on one of the 3 servers. Figure 4a details response time results.

We observe that with 100 subscribers, both the non-replicated and the replicated configurations yield similar response times. Then, as the number of subscribers grows, the response time for the non-replicated configuration continuously increases. This is explained by the fact that sending the message to a large volume of subscribers takes more time if it is done only by one server. Finally, above 500 subscribers, the CPU is not able to process the flow of publications anymore and the performance decreases exponentially. Using 3-server replication, the response times remain very low. This is due to the fact that each server only needs to process and forward publications to a 1/3 of subscribers. Thus, our all-publishers replication mechanism allows our system to scale properly in situations where there would be many subscribers on a given channel.

2) All Subscribers: We attempted to connect up to 800 publishers sending 10 publications per second each on a given channel c and only one subscriber. This experiment was ran with replication disabled and with replication enabled with 3 servers under the "all subscribers" model. Under this configuration, the subscriber was subscribing to c on all 3 servers and all publishers were publishing randomly to one of the 3 servers. Figure 4b details the response time results.

We observed that under the non-replicated configuration, we are able to support up to 200 publishers. After that, delivery of messages fails because the output buffer for the subscriber gets too full due to the high volume of publications. Under the replicated configuration, we are able to safely support nearly up to 600 publishers because each server processes only 1/3 of the publications. This demonstrates that our all-subscribers replication mechanism allows our middleware to scale to support scenarios where there are large amounts of publications.

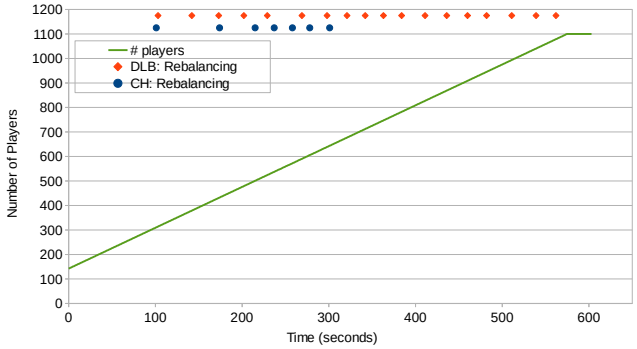
D. Experiment 2: Scalability

This experiments aims at evaluating the scalability of our Dynamoth architecture and the effectiveness of our load balancer in the context of a large-scale latency-constrained game application. At start of the experiment some 120 players are active in the game and over time more and more players join the game increasing the load in the system. Overall, we attempted to connect up to 1200 clients; once joined, each player sends 3 state updates (publications) per second. Up to 8 Redis pub/sub servers were available. We ran this experiment with our Dynamoth load balancer and we ran the same experiment again using only consistent hashing, the standard load balancing technique.

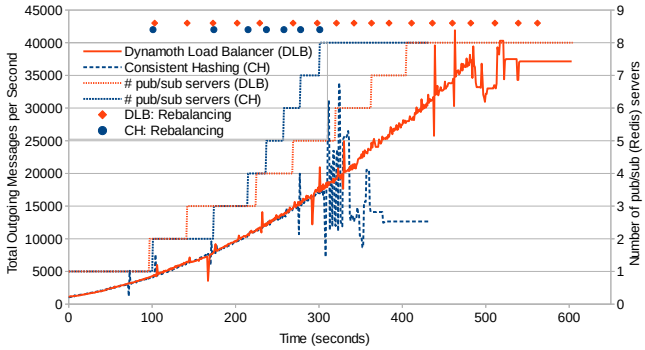
Figure 5 details our results for experiment 2. In all subfigures, the time is shown on the X-axis (in seconds). Figure 5a plots the number of players active in the system over time showing how the players slowly join the game. Figure 5b plots the total number of messages transmitted per second throughout the whole system over time, as well as the number of Redis pub/sub servers (between 1 and 8) that were currently active, for both the Dynamoth and consistent hashing experiments. Finally, Figure 5c plots the average response time experienced by clients over time (the time that elapses between the client publishing a state update and receiving the corresponding notification back from the pub/sub server). The diamonds and circles indicate the times where the load balancer triggered a reconfiguration respectively for the Dynamoth and consistent hashing approaches. In the context of a game, the playing quality will be optimal if the average response time remains below 150ms.

By using multiple Redis pub/sub servers, Dynamoth scales up to almost 1000 participants. We observe small spikes in

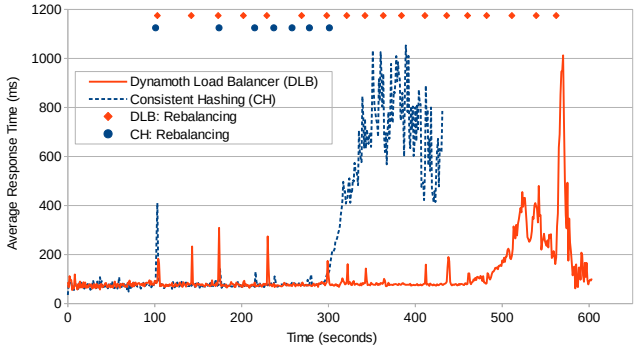
[3]A node that would be usually located in the Cloud: Local Load Analyzer, Dispatcher or Load Balancer



(a) Number of Players



(b) Total Outgoing Messages and Number of Pub/Sub Servers



(c) Average Response Time

Figure 5: Client Scalability Results

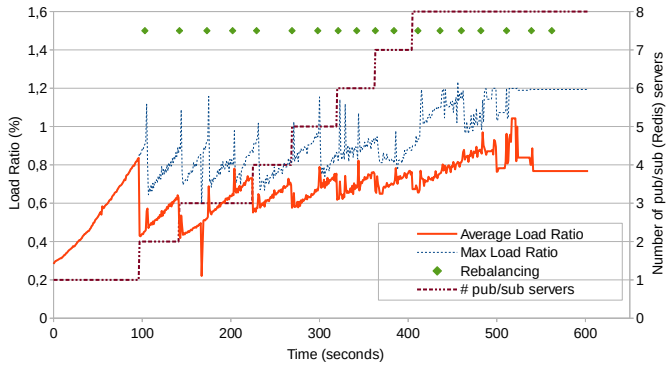


Figure 6: Dynamoth Load Balancer - Pub/Sub Server Load

the average response time around the time when new server is added and rebalancing takes place occurs, but those bursts are only of short duration and the average response time is otherwise always maintained at an acceptable threshold (around 75ms). The bursts happen because the application of the new plan occurs at a time the servers are already overloaded, further increasing the load for a short amount of time, leading to an additional delivery delay for some messages. However, our lazy plan propagation approach keeps this impact of plan changes very low, as explained in section IV. The results further show that our load balancer is conservative and first reuses the pool of active servers before deciding to spawn new servers. This keeps Cloud utilization costs low. Furthermore, even after the 8th and final server is deployed, the load balancer is still able to maintain an acceptable performance level for a while by applying incremental plan changes.

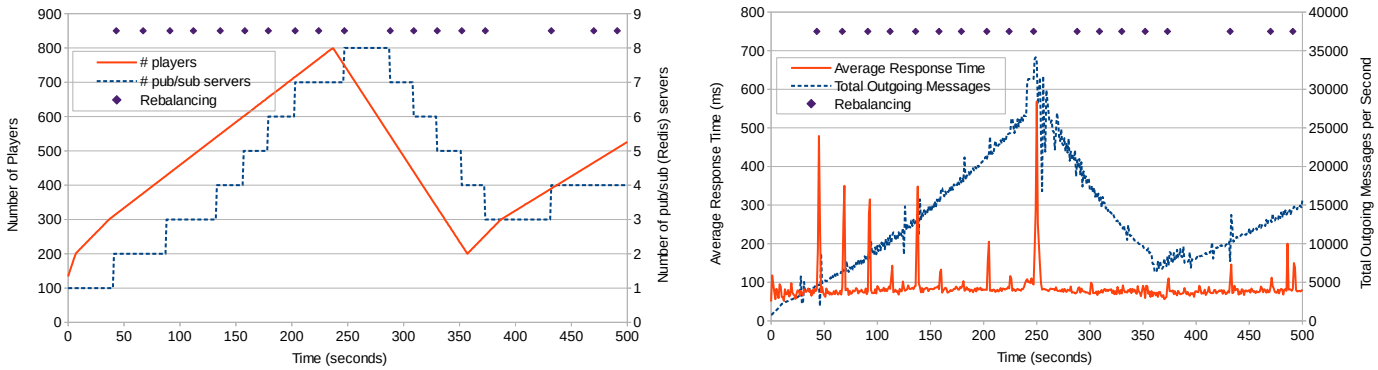
With consistent hashing, only up to 625 players can be supported before performance deteriorates. This is due to the fact that consistent hashing can not take individual server loads into account when a rebalancing occurs. Servers shed $1/N$ of their load to a newly deployed server, irrespective of their current load. As a result, highly loaded servers do not loose significant load and tend to overload again soon. Furthermore, this technique has to spawn a new server every time a rebalancing occurs, which is not cost efficient in a Cloud setting.

For that same experiment with Dynamoth, figure 6 plots the average load ratio (equation 1) of all active pub/sub servers, the load ratio of the busiest server as well as the number of Redis servers and the time points when a rebalancing occured. A load ratio of 1 or below is safe. According to our observations, a Redis pub/sub server will fail when the load ratio exceeds 1.15. We can see that our load balancer is able to maintain the average load below 1 until the system as a whole becomes overloaded. It is also able to maintain the load ratio of the busiest server below 1 for most of the experiment.

E. Experiment 3: Elasticity

In this experiment, we show how the Dynamoth load balancer handles fluctuating real-time conditions. We first inject step by step 800 clients into the virtual environment; then we remove 600 (to reach 200); then we connect a little less than 400 additional clients (to reach almost 600). Figure 7 shows the measurements gathered during this experiment. Figure 7a shows the number of players as well as the number of Redis pub/sub servers that were being used at a given time. Figure 7b shows the average response time and number of outgoing messages over time. In both figures, the points in time where the Dynamoth load balancer triggered a rebalancing are denoted by a diamond.

We observe that as the number of clients increases, rebalancings occur, which sometimes require the addition of new servers. When the number of clients decreases, rebalancings also occur and are able to release servers again to save Cloud infrastructure costs. Since those rebalancings have a lower priority, there is an observable delay between the time when the load decreases and the servers are removed. As in



(a) Number of Players & Number of pub/sub Servers

(b) Avergage Response Time & Outgoing Messages

Figure 7: Handling a Varying Number of Players

the previous experiment, when high-load rebalancings occur, we observe small spikes in average latencies as rebalancing adds additional load to already loaded servers. When scaling down, rebalancings do not cause spikes in average latencies because such rebalancings only occur when pub/sub servers are underloaded.

Overall, this experiment reveals that Dynamoth is correctly able to handle fluctuating workload patterns by providing adequate resources as needed.

VI. Related Work

There exists a vast literature on scalable pub/sub systems as well as a range of commercial pub/sub systems.

Scalable Channel-based Pub/Sub: Most commercial pub/sub system provide a channel-based interface. As an example, Amazon SNS [1] is a commercial channel-based pub/sub system for the Cloud designed to support push notifications. Google Cloud Messaging (GCM) [3] is Google's well known push notification framework. Both systems offer reliability through persistence and replication, but not much information exists in the literature that explains how these architectures scale.

On the research side, scalability in channel-based Pub/Sub has been explored in the area of peer-to-peer computing. Scribe [7] was one of the first to propose a decentralized multicast overlay architecture. It builds upon the P2P Pastry distributed hash table. Poldercast [24] is a dynamic peer-to-peer channel-based pub/sub system where all subscribers for a given channel are interconnected using a ring overlay (and additional random links); thus, any publication reaching a subscriber can then reach other subscribers in a linear fashion (worst case) or faster using the additional links. SpiderCast [10] and [9] are further P2P channel-based pub/sub systems that use distributed protocols to optimize the routing overlay. The Dynatops system [27] features a dynamic self-reconfigured channel-based pub/sub system with brokers that can handle scenarios where subscriptions are short-lived. In all these systems, there is an interconnected broker network which is usually considered to be across a wide-area network. While the various approaches aim at minimizing latencies, publications can take many hops across the broker network until they reach the subscribers. Our setup is very different as it follows a more classical client/server architecture where the servers are located in a Cloud center. Communication is always only two hops (from the client to the server and directly back to clients).

Regarding formal approaches, [23] proposes a formal modelling of Cloud-based pub/sub systems which includes a cost-model that can be used to predict the costs of deploying a given workload to a given set of pub/sub servers in the Cloud.

Scalable Attribute-based and Content-based Pub/sub: Content-based pub/sub systems have been extensively studied in the past. While such approaches allow for finer-grained publication-to-subscriber matching compared to channel-based, they require more CPU processing. In addition, they often involve an additional layer at the infrastructure (dispatching then matching servers), which might make them less suitable for applications which require strict latency bounds such as large-scale games. The BlueDove system [18] proposes a brokerless, two-layered scalable attribute-based pub/sub system which supports multi-dimensional attributes. The attribute space for each dimension is split over a set of Cloud matching servers. Subscriptions and publications are forwarded to matching servers using dispatching servers. The E-StreamHub middleware [6] aims at providing an elastic content-based pub/sub platform that adds/removes nodes based on the current load measured on the system (Dynamoth also uses the measured system load). The PAPaS system [4] aims at reusing the BitTorrent architecture to provide a hybrid peer-to-peer assisted content-based pub/sub system. Again, due to the P2P nature of this solution, we think that this can lead to higher latencies that are unsuitable for some applications, e.g., games. In [26], the authors conduct a performance evaluation of running two popular non-cloud-based pub/sub systems based on broker networks - PADRES [12] and OncePubSub - in the Cloud using different approaches. The middleware that was studied was not designed and optimized for the Cloud; therefore, we think that our system would be more tailored for the Cloud. In [19], the authors propose a Cloud broker-based scalable matching service for content-based pub/sub networks. [2] and [17] are two popular open source systems

which can scale by manually adding/removing nodes; however, they cannot be qualified as elastic because automatic addition/removal of nodes based on measured load is not done. [8] proposes a load-balancing approach for broker-based content-based pub/sub systems that is built over PADRES.

In general, scalability in most existing channel- and content-based pub/sub is achieved by a broker network where a client is connected to one of the brokers and brokers might forward messages among each other to reach all subscribers. In our system, any publication only goes through a single pub/sub server (except during reconfiguration). In order to handle individual high-loaded channels, we use a novel mechanism, namely channel replication to avoid a single bottleneck.

Spatial pub/sub (SPS) systems are a special class of pub/sub systems where subscribers subscribe to regions in a virtual space (e.g., [15], SPEX [20]). While we use a gaming appliction for our evaluation, Dynamoth is a general purpose pub/sub system that is not specifically designed for games but can support arbitrary applications.

VII. Conclusion

We presented Dynamoth, our channel-based pub/sub middleware platform optimized for latency-constrained environments. Dynamoth uses independent pub/sub servers deployed in the Cloud to handle the delivery of all publications in a broker-less way in order to minimize latencies. A major contribution is our hierarchical load-balancer which can perform rebalancings at the system-level (macro) and at the channel-level (micro). System-level load balancing enables our system to scale to arbitrary numbers of publishers, subscribers and publications in real time in order to adapt to the current load conditions. Additional pub/sub servers are dynamically allocated from the Cloud when needed and removed when they are not required anymore. Channel-level load balancing (replication) allows our platform to handle special channels which exhibit high load patterns such as channels with extremely large numbers of publishers, subscribers and/or publications: such channels can be mapped to more than one pub/sub server. Dynamoth also proposes an elaborate propagation mechanism to notify all relevant clients of changes to channel assignments with very minimal impact on performance, while ensuring uninterrupted delivery of all messages. We built an implementation of Dynamoth and ran extensive, large-scale experiments using a multiplayer game prototype as an application testbed. Our experiments reveal that Dynamoth is able to scale in an elastic manner as the number of subscribers, publishers and publications grow while maintaining low response time despite the very high variability in the workloads. When the load decreases, unnecessary resources are automatically released.

As future work, we are looking at how we could integrate CPU load into our load balancing algorithms for environments where CPU is a constrained resource (such as virtual CPUs in the Cloud). We will also look at integrating a cost model in our load balancing model in order to minimize Cloud-related costs. Reliability, achieved either through replication or persistence is another interesting aspect.

References

[1] Amazon SNS. http://aws.amazon.com/fr/sns/ (2014)
[2] Apache Hedwig Project. https://cwiki.apache.org/confluence/display/BOOKKEEPER/HedWig (2014)
[3] Google Cloud Messaging. https://developer.android.com/google/gcm/index.html (2014)
[4] Ahmed, N., Linderman, M., Bryant, J.: Papas: Peer assisted publish and subscribe. In: Workshop on Middleware for Next Generation Internet Computing (MW4NG). pp. 7:1–7:6 (2012)
[5] Aurenhammer, F.: Voronoi diagrams-a survey of a fundamental geometric data structure. ACM Comput. Surv. 23(3), 345–405 (1991)
[6] Barazzutti, R., Heinze, T., Martin, A., Onica, E., Felber, P., Fetzer, C., Jerzak, Z., Pasin, M., Riviere, E.: Elastic scaling of a high-throughput content-based publish/subscribe engine. In: ICDCS. pp. 567–576 (2014)
[7] Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: Scribe: a large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communications 20(8), 1489–1499 (2002)
[8] Cheung, A.K.Y., Jacobsen, H.A.: Load balancing content-based publish/subscribe systems. ACM Trans. Comput. Syst. 28(4), 9:1–9:55 (Dec 2010), http://doi.acm.org/10.1145/1880018.1880020
[9] Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Constructing scalable overlays for pub-sub with many topics. In: ACM Symposium on Principles of Distributed Computing (PODC). pp. 109–118 (2007)
[10] Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In: DEBS. pp. 14–25 (2007)
[11] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
[12] Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S.: The padres distributed publish/subscribe system. In: FIW. pp. 12–30. Citeseer (2005)
[13] Garcia, F.P.: Channel replication in dynamoth: Implementation and analysis. Tech. Rep. COMP400-F14-FPG, Department of Computer Science, McGill University (2014)
[14] Gummadi, K.P., Saroiu, S., Gribble, S.D.: King: Estimating latency between arbitrary internet end hosts. In: ACM SIGCOMM Workshop on Internet Measurment (IMW). pp. 5–18 (2002)
[15] Hu, S.Y., Chen, K.T.: Vso: Self-organizing spatial publish subscribe. In: Self-Adaptive and Self-Organizing Systems (SASO). pp. 21–30 (2011)
[16] Kienzle, J., Verbrugge, C., Kemme, B., Denault, A., Hawker, M.: Mammoth: a massively multiplayer game research framework. In: Foundations of Digital Games (FDG). pp. 308–315 (2009)
[17] Kreps, J; Narkhede, N.R.J.: Kafka, a distributed messaging system for log processing. In: NetDB'11 (2011)
[18] Li, M., Ye, F., Kim, M., Chen, H., Lei, H.: A scalable and elastic publish/subscribe service. In: IPDPS. pp. 1254–1265 (2011)
[19] Ma, X., Wang, Y., Pei, X.: A scalable and reliable matching service for content-based publish/subscribe systems. IEEE Transactions on Cloud Computing PP(99), 1–1 (2014)
[20] Najaran, M.T., Hu, S.Y., Hutchinson, N.C.: Spex: Scalable spatial publish/subscribe for distributed virtual worlds without borders. In: ACM Multimedia Systems Conference (MMSys). pp. 127–138 (2014)
[21] Pietzuch, P., Bacon, J.: Hermes: a distributed event-based middleware architecture. In: ICDCS Workshops. pp. 611–618 (2002)
[22] Rosenblum, D.S., Wolf, A.L.: A design framework for internet-scale event observation and notification. In: ESEC. pp. 344–360 (1997)
[23] Setty, V., Vitenberg, R., Kreitz, G., Urdaneta, G., van Steen, M.: Cost-effective resource allocation for deploying pub/sub on cloud. In: Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on. pp. 555–566 (June 2014)
[24] Setty, V., Van Steen, M., Vitenberg, R., Voulgaris, S.: Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub. In: International Middleware Conference (Middleware). pp. 271–291 (2012)
[25] Tarng, P.Y., Chen, K.T., Huang, P.: An analysis of wow players' game hours. In: NetGames 2008. pp. 47–52 (2008)
[26] Zhang, B., Jin, B., Chen, H., Qin, Z.: Empirical evaluation of content-based pub/sub systems over cloud infrastructure. In: Intl. Conference on Embedded and Ubiquitous Computing (EUC). pp. 81–88 (2010)
[27] Zhao, Y., Kim, K., Venkatasubramanian, N.: Dynatops: A dynamic topic-based publish/subscribe architecture. In: DEBS. pp. 75–86 (2013)