Caching in content-based publish/subscribe systems

Vasilis Sourlas*, Georgios S. Paschos†, Paris Flegkas* and Leandros Tassiulas*
*Department of Computer & Communication Engineering
University of Thessaly, Greece, †CERTH-ITI
Email: vsourlas, gpaschos, pflegkas, leandros @uth.gr

Abstract—In a publish/subscribe network, message delivery is guaranteed for all active subscribers at publish time. However, in a dynamic scenario where users join and leave the network, a user may be interested in content published before the subscription time. In this paper, we introduce mechanisms that enable caching in such networks, while maintaining the main principle of loose-coupled and asynchronous communication. Furthermore we investigate two caching policies; caching in all candidate brokers (basic caching) which yields high survivability and low delay and caching in leaf brokers (leaf caching) which maintains low overhead and querying complexity. The comparison is performed via simulations and testbed measurements and insights are given for future work.

I. INTRODUCTION

The Internet has considerably changed the scale of distributed systems. Distributed systems now involve thousands of entities whose location and behavior may greatly vary throughout the lifetime of the system. These constraints make more dire than ever the need for flexible communication models. The publish/subscribe (pub/sub) paradigm has become an important architectural style for designing distributed systems and has recently been considered one of the most promising future network architectures that solves many challenges of the current Internet. Applications that exploit a pub/sub communication paradigm are organized as a collection of autonomous components (clients), which interact with publishing events (messages) by subscribing to the classes of events they are interested in. The event dispatcher (broker) is responsible for collecting subscriptions and forwarding events to subscribers. In content-based pub/sub systems the selection of a message is determined entirely by the client, which uses expressions (filters) that allow sophisticated matching on the event content.

There are several research efforts concerned with the development of an event notification service including IBM's Gryphon [1], Siena [2], Elvin [3], and REDS [4] which implement the pub/sub architecture. Most of them address scalability and ease of implementation by realizing the broker tree as an overlay network.

In a pub/sub system, any message is guaranteed to reach all interested destinations, (completeness property) [5]. The above holds for all clients that are active and therefore their subscriptions are known to the system at publish time. However, in a dynamic environment, clients join and leave the system during time, and it is possible that a client joins the network after the publishing of an interesting message. In pub/sub systems it is not possible for a new subscriber to retrieve previously published messages that match his/her subscription. Therefore, enabling the retrieval of previously published content by means of caching is one of the most challenging problems in pub/sub networks.

Despite the staggering volume, a large percentage of network traffic is redundant. Multiple users, at any given site, request much of the same content. Caching performs replication of content to serve identical requests locally, and to prevent them from overutilizing the network resources. A cache stores content on a storage device that is physically or logically closer to the user. In this paper, we put forward a different aspect of caching, focusing on preserving the information over time instead of making information available in nearer space. Potential applications that naturally require publish/subscribe messaging and could be enforced by our caching technique are software and antivirus updates, stock markets, web search engines, distributed sensor networks, etc.

Caching on the Web has been thoroughly investigated for cooperative and non-cooperative architectures. In [6] a cooperative hierarchical Web caching architecture was studied while in [7], the authors introduce the Internet cache protocol (ICP). All caching protocols presume the existence of a node (called data center) where all the information is stored. Any data request is forwarded to the data center which in turn responds with the requested data. Caching is used: 1) to save bandwidth and cost by placing the cache engines in strategically chosen points; 2) to improve productivity for end users by retrieving information much quicker. In other words, end users see dramatic improvements in response time and the whole procedure is completely transparent to them. In our work caching is used for making information available to future subscribers and therefore it cannot be directly compared with traditional caching methodologies.

Caching as a mechanism for storing data in pub/sub systems has not received attention in the literature. Nevertheless, there exists a limited number of efforts in the topic of retrieval of historic data. The work in [8] deals with content-based networks and particularly the authors propose two forms of subscriptions that allow a wireless ad-hoc network user to subscribe to past events and thus to improve the bootstrapping process. As bootstrapping process they refer to application layer mechanism that helps a mobile system to adapt to current contextual information which is only available locally. The first form uses logical mobility to harness possible client movements and proactive subscription in future locations to bootstrap virtual counterparts before the actual need for data forwarding. The second form is based on buffers and offers a
way to integrate data repositories distributed in the network. Their approach concentrates on the class of applications that commence normal operation after having seen a sequence of notifications. The essential idea is to provide the consumer/subscriber with a correct sequence of past notifications as if it had subscribed earlier. On the contrast our approach is not confined in any particular class of applications and in this sense, it is more general. A case not covered by [8] could be a stock market where a user is interested in stock prices published before the time of its subscription. In such case the user requests location independent information and he/she may be interested in any available old information and not only a small fraction of it.

The second form presented in [8] is more relative to our work. However, the focus is on using the reverse of advertisements paths to propagate the past subscriptions. This is restrictive since it assumes a relation between the publisher and the storing capability of the network which is not always true. In our case, we make this clear distinction of the two entities, and publishers are free to publish and roam or disappear from the network. Moreover, the buffering of notifications in every broker proposed in [8], clearly favors the most recent notifications but it is possible that the overall performance is actually deteriorated. For example, storing each message in every broker results in reduction of the network storing capacity. In our buffering/caching approach, we study several caching policies in order to identify the performance dependence on these policies and showcase optimality where possible.

In [9], the authors propose a historic data retrieval pub/sub system where databases are connected to various brokers, each associated with a filter to store particular information. The database holding the relevant information republishes historic events on receipt of a subscription query with a time-based parameter. This parameter categorizes the content in classes and therefore the work of [9] is related to topic based pub/sub systems. In the same work, every database advertises the class to the network so that subscribers interested in past information belonging to that class can request it from the specific database. This approach it totally different from ours since apart from using “heavy” database modules it also indexes the information (by assigning a class attribute) while we are interested in the caching problem for content-based delivery. Generally speaking, the content-based information delivery is much more demanding than the indexed counterpart. This property reflects upon routing efficiency, overhead, dissemination and query delays with the indexing being always helpful but resulting in a more restricted solution.

In [10] a mechanism for authorizing and controlling historic event republication is proposed as an extension to [9]. Specifically they describe how policy administrators can exploit context-aware access mechanisms to control data disclosure. Their approach unifies delivery for both live and historic events, providing a common interface for the administration of disclosure policy, while facilitating similar event-processing actions for both types of events. They also provide a database API (Application Interface) at each broker and use class attributes to advertise the stored information as in [9] but they also use policies to restrict and enrich information flows. For example in a healthcare scenario a request for old data might come from a user with limited access rights. In that case the databases transform the stored data before sending them back (removing for instance the name of a patient).

In this paper, we describe and evaluate through simulations and testbed experiments our design and implementation of a caching technique on networks that use the content delivery pub/sub communication paradigm. Particularly, we provide each broker with a limited cache and enhance the pub/sub paradigm with a request/response mechanism so that nodes can retrieve previously published events. For the purpose of this paper we call those events (messages) as old messages. Our proposed solution maintains the basic pub/sub principles i.e. the loose-coupled and asynchronous communications. This is because we do not assume predefined caching points and the requests for old content are handled transparently by the network. We study and compare different caching strategies based on the replication degree of the content with regards to network overhead, delay and content life-time in the caches.

The rest of the paper is organized as follows. In section II, a brief introduction of the pub/sub architecture is given, followed by the description of the proposed request/response mechanism in section III. The caching policies are described in section IV and section V is devoted to performance evaluation via simulations. In section VI we evaluate our system with testbed measurements and conclude the paper in the last section.

II. THE PUB/SUB ARCHITECTURE

We consider a pub/sub system which use the subscription forwarding routing strategy [2]. The routing paths for the published messages are set by the subscriptions, which are propagated throughout the overlay network so as to form a tree that connects the subscribers to all the brokers in the network.

Particularly, when a client \(s_i\) issues a subscription, a \(\text{Subscribe}(f_i, s_i)\) message containing the corresponding subscription filter \(f_i\) is sent to the broker \(n_i\), the client is attached to. There, the filter is inserted in a Subscription Table (ST), together with the identifier of the subscriber. Then, the subscription is propagated by \(n_i\), which now behaves as a subscriber with respect to the rest of the dispatching network, to all of its neighboring brokers on the overlay network, this time with the syntax \(\text{Subscribe}(f_i, n_i)\). In turn, the neighbors record the subscription and re-propagate it towards all further neighboring brokers, except for the one that sent it. Finally, each broker \(n_k\) has a ST, in which for every neighboring broker \(n_j\) there is an associated set of filters \(F(n_j)\) containing the subscriptions sent by \(n_j\) to \(n_k\). In other words, \(n_k\) may have in its ST the record \(f_j \rightarrow n_k\), which means that neighbor \(n_{k-1}\) is subscribed to \(f_j\).

This scheme is usually optimized by avoiding subscription forwarding of the same event pattern in the same direction.
exploiting “coverage” relations among filters. That is, a subscription is forwarded to a neighboring broker only if it is not being covered by a subscription already forwarded to the same neighbor. We say that a subscription $f_1$ covers another subscription $f_2$, denoted by $f_1 \geq f_2$, iff any event matching $f_2$ also matches $f_1$ [11].

Requests to unsubscribe from an event pattern are handled and propagated analogously to subscriptions, although at each hop entries in the subscription table are removed rather than inserted. For an in-depth discussion of the pub/sub architecture see [12].

III. ENABLING CACHING IN PUB/SUB SYSTEMS

In this section, we describe the key points of the proposed enhancement. By installing caches in brokers and introducing a request/response mechanism we aim to provide a pub/sub system with the ability to make the old information available for future clients. Our first consideration here, is not to tamper with the core functionality of the pub/sub paradigm. The relative question is: can a pure content-based pub/sub system preserve information in an efficient way?

A. Caching points

In our system, each broker is selected as a candidate caching point for a message as long as it has in its subscription table at least one client subscribed in this message. A published message is transferred to all brokers with client subscribers. Also, a broker with a client subscriber is easily reached by a request message (the same way it is reached by a publish message as described in the next section) through the paths set by the subscription tables.

In the literature, efficient techniques for the selection of the caching points have been proposed ([13]-[16]) but their application would require additional functionality not supported by a pure pub/sub system (i.e. mechanism for discovering the location and content of caches).

B. Request/Response mechanism

In order to retrieve old information, we add to the system two additional types of messages, Request() and Response().

When a client node $s_2$ interested in old content appears in the network, apart from subscribing, he/she also makes a request by sending a Request($f_2, s_2$) message, see figure 1 (step 3). The Request() message is propagated similarly to the Publish() message, but additionally it carries along all the broker identifiers of the path it has crossed. Broker $n_a$ upon receiving the Request($f_2, s_2$) message checks in its ST$_a$ for subscriptions matching filter $f_2$. The subscription can be either from another broker or from a client. For every existing subscribed broker ($n_b$ in this case) the broker forwards the Request($f_2, n_a \rightarrow s_2$) message with its identifier appended. If no further broker subscription exists the Request message is dropped. At a distant broker $n_i$, the syntax of the arriving request message would look like Request($f_2, n_{i-1} \rightarrow \cdots \rightarrow n_a \rightarrow s_2$). Each broker

-recipient of a request message- with at least one client subscription, searches in its cache for messages matching the initial filter $f_2$. If a matching is found, a Response() message is initiated.

The Response() message carries an old message as well as the sequence of nodes carried by the initiating Request(). When a broker receives a Response() message, it pops off its identifier from the sequence and forwards it to the first broker of the remaining sequence. In the end, client $s_2$ will receive the message. With the above procedure, every new subscriber and only that one will receive every old message matching its filter and is still cached in at least one cache in the system.

Figure 1 illustrates the whole procedure. Step 1 showcases a typical subscription procedure. Notice that apart from the Subscribe() message also a Request($f_1, s_1$) message is sent but it is dropped in broker $n_b$ since there are no other broker subscribers to be sent, and no client subscribers whatsoever. Step 2 illustrates a typical publish procedure. Candidate broker $n_b$ caches the message in its cache. In step 3 another client subscribes and requests. The requested filter is matched with filter $f_1$ sent by broker $n_b$ so the request is forwarded to $n_b$. Here we have set $f_1 \geq f_2$ and the subscription gets covered at $n_b$. In step 4, client $s_2$ receives the response message and broker $n_a$ caches it.

C. Handling multiple responses

Multiple caching at different brokers has as side effect the possible production of multiple identical responses on a single request. In order to deal with this effect, we supply our system with the following duplicate response dropping mechanism. Every broker with client subscriber, upon the arrival of each response message, checks whether the message appears already in its cache and in case this is true, drops the response message. Otherwise it forwards the message according to the technique described in section III-B.

In the example of figure 2, we suppose that brokers 1, 4, and 7 have in their cache the same “black” message. If now a client connects to broker 5 and subscribes with some filter
numerous policies that can be proposed, including random drop policies. One policy of interest in case of messages with variable popularity, is to order the cached messages with popularity order, or some weighted metric taking into account popularity and time spent in the cache. Alternative policies are left as future work.

IV. CACHING POLICIES

A caching policy provides a rule on whether to cache a message or not on each particular broker. Each caching policy in this paper obeys the basic rule cache only on brokers having at least one client subscriber who is interested in this publication. We call these brokers, candidate brokers. Further selection is possible using topological properties of the brokers.

A. caching at every candidate: basic caching

Upon receiving a publish message a candidate broker caches the message by putting it at the top of the cache. Upon receiving a response message the broker that has as a subscriber the client that issued the request message also caches the message. In the example of figure 2, brokers 1, 4, 7 were candidate brokers at publish time and they all cached the “black” message. At request time, candidate broker 5 will also cache the message under the basic caching policy.

B. caching at leaf brokers: leaf caching

Leaf brokers are the candidate brokers having only client subscribers for the message under question. In the above definition, the broker subscriber that sent the message is not counted. In the example of figure 2, at publish time of “black” message we have the candidate brokers 1, 4, 7 and only 1 and 7 are leaf brokers. Note that broker 4 has two broker subscribers. Under leaf caching, only 1 and 7 cache the “black message”. Similarly, for the “gray” message, only broker 11 qualifies as leaf broker. Also, in the case of request from 5 (step 3), broker 5 will not cache the message since there are two broker subscribers on this broker (4 and 6).

V. PERFORMANCE EVALUATION

In this section we evaluate the proposed mechanism using a discrete event simulator. N brokers are organized in a balanced binary tree (complete for $N = 7, 15, 31, 63, ...$) and clients are dynamically generated on each broker according to a birth and death process with birth rate $\lambda$ and death rate $\mu$. Each broker has a cache capable of storing $k$ messages. Initially all caches are empty until the publish time. On publish time, we generate $M$ different messages matching exclusively the $M$ possible subscriptions. We let the system operate under the dynamic client environment using the FIFO priority policy and compare the basic and leaf caching policies. We are looking at the following interesting metrics.

- The absorption time of message $m$ is the time passed from the publishing of $m$ until it gets disappeared from the network. This metric is indicative of the capability of the network to maintain messages in its memory.
• The *minimum hop distance* is measured for each successful response and corresponds to the minimum number of hops between a responding broker and the broker where the client making the request is attached to. This metric is indicative of the delay of responses as a function of hops in the network. It also makes a picture of the message replication.

• The *responses per request* is measured for each successfully responded request and corresponds to the number of total responses for the given request. This metric is representative of the replication and the overhead in the network.

• The *cache slots searched per request* is the total sum of occupied slots of each searched cache during a successfully requested request. In a content-based network the caches must be searched exhaustively for any possible match. This metric is representative of the cache querying complexity.

The above metrics are random variables and we estimate their mean by simulating thousands of observations. We set three experiments, one varying the dynamics of the clients ($\lambda/\mu$), one varying the cache size as a percentage of $M$ and one varying the number of brokers.

In the first experiment (figure 3) we can identify the exponential nature of absorption time and also verify that the basic caching offers the best survivability of information. The basic caching is also superior in terms of the minimum hop distance. The advantages of thin replication schemes is showcased for caching is also superior in terms of the minimum hop distance. The basic caching offers the best survivability of information. The basic caching is better in terms of the minimum hop distance.

In figure 4 the impact of cache size is demonstrated. In most cases, using a small percentage of cache (that is $k = 0.3M$) is enough to achieve order performance.

In figure 5 the scalability issues are demonstrated. Complexity and overhead are linear to the size of the network while delay grows very slowly for the basic caching policy. The survivability of the network is improved with the size of the network making the need for cache size less and as $N$ grows, the performance improves.

VI. SYSTEM DESIGN AND EXPERIMENTATION

We build our implementation on top of REDS [4], by making modifications on the routing layer. Particularly we add Request() and Response() as functions in Subscription-ForwardingRoutingStrategy of the Routing Strategy interface. In order to cache messages we introduce the GenericCache and the Cache interfaces, similar to the GenericTable and the SubscriptionTable already implemented in REDS. Transferring of the new type of messages requires also slight modifications of TCPTransport and UDPT Transport at the Transport interface.

A. Testbed Evaluation

We used 7 laptops equipped with a 1.6 GHz Intel Celeron M CPU, 512 MB of RAM. The 7 computers were connected via Ethernet switch and the pub/sub overlay network was organized in a complete binary tree. The choice of topology is made for symmetry reasons in order to avoid favoring some brokers. Clients are dynamically deployed using a birth and death process with birth rate $\lambda = 0.1$ which corresponds to the generation of one subscriber per broker per message each.
in content delivery pub/sub networks. The proposed concept equips the pub/sub with the ability to retrieve old information. Evaluation via simulations presents the performance of the system regarding information survivability, delay, overhead, search complexity and scalability. The mechanism is also implemented and measured in a testbed. This work can be extended in many ways, from deriving applications to exploring new caching and priority policies as well as extensions for wireless ad-hoc networks.

ACKNOWLEDGMENT

This work has been supported by the European Commission through the FP7-ICT-224263 ONELAB2 and FP7-ICT-216366 EURO-NF programs.

REFERENCES


