Boosting Topic-Based Publish-Subscribe Systems with Dynamic Clustering *

Tova Milo Tal Zur Elad Verbin
School of Computer Science, Tel Aviv University
{milo,talzur,eladv}@post.tau.ac.il

ABSTRACT

We consider in this paper a class of Publish-Subscribe (pub-sub) systems called *topic-based* systems, where users subscribe to topics and are notified on events that belong to those subscribed topics. With the recent flourishing of RSS news syndication, these systems are regaining popularity and are raising new challenging problems.

In most of the modern topics-based systems, the events in each topic are delivered to the subscribers via a supporting, distributed, data structure (typically a multicast tree). Since peers in the network may come and go frequently, this supporting structure must be continuously maintained so that "holes" do not disrupt the events delivery. The dissemination of events in each topic thus incurs two main costs: (1) the actual transmission cost for the topic events, and (2) the maintenance cost for its supporting structure. This maintenance overhead becomes particularly dominating when a pub-sub system supports a large number of topics with moderate event frequency; a typical scenario in nowadays news syndication scene.

The goal of this paper is to devise a method for reducing this maintenance overhead to the minimum. Our aim is not to invent yet another topic-based pub-sub system, but rather to develop a generic technique for better utilization of existing platforms. Our solution is based on a novel distributed clustering algorithm that utilizes correlations between user subscriptions to dynamically group topics together, into virtual topics (called *topic-clusters*), and thereby unifies their supporting structures and reduces costs. Our technique continuously adapts the topic-clusters and the user subscriptions to the system state, and incurs only very minimal overhead. We have implemented our solution in the *Tamara* pub-sub system. Our experimental study shows this approach to be extremely effective, improving the performance by an order of magnitude.

Categories and Subject Descriptors. C.2.4 [Distributed Systems]: Distributed applications; C.2.1 [Network Architecture and Design]: Distributed networks.
General Terms. Algorithms, Performance, Experimentation.
Keywords. Publish-subscribe, Peer-to-Peer, Dynamic clustering.

*The research has been partially supported by the European Project EDOS and the Israel Science Foundation.

SIGMOD'07, June 12–14, 2007, Beijing, China.

1. INTRODUCTION

The amount of information available to Internet users is increasing rapidly. The need of users to be constantly updated with an up-to-date, accurate, and relevant data, out of this ocean of information, makes the publish-subscribe interaction scheme (pub-sub, for short) particularly appealing. In a pub-sub system, subscribers express their interest in certain events (e.g. the appearance of new relevant data item), and are subsequently notified of any event, generated by a publisher (information provider), that belongs to their registered interest. Pub-sub systems have raised considerable interest in the research community over the years. In this paper we focus on a relatively simple class of such systems, called *topic-based* systems, where users subscribe to *topics* and are notified on events that belong to those subscribed topics. With the recent flourishing of RSS news syndication, these systems are regaining popularity and are raising new challenging problems.

In most of the modern platforms for topic-based event notification, the events in each topic are delivered to the subscribers via a supporting, distributed, data structure (typically a multicast tree). Since peers in the network may come and go frequently, this supporting structure must be continuously maintained so that holes do not disrupt the delivery of events. The dissemination of events in each topic thus incurs two main costs: (1) the actual transmission cost for the topic events, and (2) the maintenance cost for its supporting structure. This maintenance overhead becomes particularly dominating when a pub-sub system supports a large number of topics with moderate event frequency; a typical scenario in nowadays news syndication scene.

The goal of this paper is to devise a method for reducing this maintenance overhead to the minimum. Our aim is not to invent yet another topic-based pub-sub system, but rather to develop a generic novel technique for better utilization of existing platforms. Our solution takes advantage of correlations between user subscriptions to group topics together, into virtual topics (called *topic-clusters*), and thereby unifies their supporting structures and reduces costs.

Before presenting our results let us briefly highlight some of the main properties of pub-sub systems in general, and topic-based ones in particular, and explain how a good solution to the problem addressed in this paper can contribute to better performance of such systems.

Background. Pub-sub is a distributed computing paradigm that consists of three principal components: subscribers, publishers, and an infrastructure for event delivery. Subscribers express their interest in an event or a pattern of events (say, the publication of new sports news). Publishers (e.g. CNN's sport department) generate events (e.g. post news). The infrastructure is responsible for matching events with the interests and sending them to the subscribers. Based on the way the subscribers specify their inter-

est, pub-sub systems can be classified into two main categories: *topic-based* and *content-based*. In *topic-based* pub-sub systems, subscribers specify their interest by subscribing to a *topic*, also known as feed, channel, subject, or group. Each event produced by the publisher is labeled with a topic and sent to all the topic subscribers. In other words, publishers and subscribers are connected together by a predefined topic. In *content-based* systems, on the other hand, subscribers specify their interest through event filters, which are boolean queries on the events content. Published events are matched against the filters and sent to the subscribers if they satisfy their specified filters.

Topic-based pub-sub is rather static and basic compared to a content-based one. Its simplicity however has the advantage of allowing for a very efficient implementation and a simple, intuitive, user interface. Content-based pub-sub typically requires more sophisticated protocols with higher runtime overhead, as well as more sophisticated user interaction. Because of this additional complexity, one generally prefers to use a topic-based pub-sub in contexts where events divide naturally into groups that correspond to users interest. A typical example is the increasingly popular RSS news syndication[16]. An RSS system is a simple topic-based pub-sub system. Publishers publish their news by putting them into an RSS feed and providing the URL for the feed on their website. RSS users subscribe to an RSS feed by specifying its URL to their RSS readers. It is interesting to note that most existing RSS applications rely on a rather primitive implementation where RSS readers *poll* the feeds periodically. But with the continuous dramatic increase in the number of RSS users, it is anticipated that, for scalability, future implementation will move to push-based platforms.

To allow for scalability, most modern topic-based platforms [4, 30, 18, 22] are based on a Peer-to-Peer architecture. More specifically, they often run over Distributed Hash Table (DHT) systems [21, 24, 29] and manage their operative layer using the DHT exported functions (API). In that sense, the DHT acts as a mediator layer between the underneath network and the system core. For each topic the system allocates a dedicated distributed structure (essentially a multicast tree) over the DHT, which serves as medium for delivering the events. Since peers may leave the network unexpectedly, the structure must be continuously maintained so that the delivery of events is not disrupted.

In all the topic-based pub-sub systems that we are aware of, the structures are built and kept separately for each topic, and the maintenance cost of each structure is proportional to the number of subscribers. A large number of topics each having a large number of subscribers entails a high overall maintenance cost. So the question motivating the present work is can this overhead be avoided or at least be significantly reduced?

It turns out that the answer is often positive. The key observation is that, in practice, one can typically detect correlations between users subscriptions, which can be used to group topics and reduce the overall maintenance cost. As a simple example, consider two popular news websites, CNN.com and FoxNews.com, each providing, among others, a Headlines feed. In a typical pub-sub system, each of the feeds would be modeled as a distinct topic, and the system would keep (and maintain) two separate structures for disseminating their corresponding events. Now, assume that we have a large number of "news addicts" that are subscribed to both topics. If we create a topic that is the union of the two and subscribe those users directly to the merged topic, instead of the individual ones, the redundancy in the underlying structures would be eliminated. The overall structures size would be reduced and hence also the maintenance cost. Furthermore, if it happens that a majority of CNN Headlines subscribers are also subscribed to FoxNews headlines, and the events frequency in FoxNews is not too high, it may be cost effective to eliminate the individual CNN topic all together, subscribing all its users to the merged CNN-FoxNews topic and simply provide those few users not interested in FoxNews a local filter that filters out the redundant events. In the above example the correlation between the user subscriptions is due to the fact that the topics provide semantically similar information. In general, there may be many other reasons for such correlations. For instance, users that are subscribed to updates for a given piece of software (say the free bitmap image editor GIMP [25]) are likely to be also subscribed to updates of other software pieces on which the given software depends (e.g. GTK+, libart and Pango [9]).

Challenges. To improve performance, we would like to form the "best" topic-clusters, (and correspondingly determine to which topics/topic-clusters each users should subscribe), so that the overall cost of events delivery and structures maintenance in the system is minimized. While this may appear to be a traditional clustering problem, there are three requirements, derived from the specific context, which together make the problem particularly challenging.
Adaptivity. A P2P pub-sub environment has a dynamic nature: users, topics and publishers may come and go; users may change interests; the events frequency in the various topics may change over time. A good solution thus must have a dynamic nature, continuously adapting the topic-clusters and the user subscriptions to the current system state.
Distribution. The decentralized P2P nature of pub-sub systems, where no central coordinator has full knowledge about the system's state and the users subscriptions, calls for a corresponding distributed clustering algorithm.
Low overhead. Finally, the continuous clustering efforts, as well as the adjustment of users subscription, should incur only very minimal overhead for the saving it brings to be meaningful.

While each of these aspects have been addressed, often separately, by previous research in the area of clustering (see Section 6 for an overview), to our knowledge none of these works provides a comprehensive solution that can be employed in the given context. In particular, many works aim at forming disjoint clusters, which is not a requirement here (and may indeed lead to inferior performance, as shown in our experiments.)

Results. The contributions of this paper are the following.

- We introduce a simple generic model for describing the cost of events delivery and structures maintenance in a pub-sub system with topic-clusters.
- Based on this model we present a dynamic distributed clustering algorithm that continuously adapts the topic-clusters and, resp., the user subscriptions, to the changing system state. The algorithm employs local cluster updates to change the overall system configuration. Each local update is performed only when it is estimated to be (globally) cost effective. Furthermore, to minimize the overhead involved in gain estimations, a probabilistic component is employed to guarantee that (with high probability) gain estimation are computed only for updates that are likely to be beneficial.
- We have implemented the above algorithm in the *Tamara* pub-sub system. *Tamara* uses a standard popular topic-based pub-sub platform (*Scribe* [4]) to manage topics, topic-clusters, and user subscriptions. Our new dynamic topic-clustering algorithm is used to automatically group topics together, into virtual topics, and redirect the publishers' event notifications, as well as the users subscriptions, to these virtual topic. Our experiments shows that, compared to the standard use of Scribe, *Tamara* improves the performance by an order of magnitude.

It should be stressed that while our implementation uses Scribe to manage topics and user subscriptions, the technique that we propose is generic and can similarly be used to boost the performance of other existing topic-based pub-sub platforms.

The grouping of topics into sets has been previously proposed in the literature in a different context: To provide users with varying subscription granularity it was suggested to group topics into sets forming a sub-set hierarchy [26]. A main difference with the present work is the static nature of that grouping. In contrast our solution adapts continuously the topic-clusters to the user needs, guaranteeing, as we shall see, stable good performance even when users interests shift significantly.

The paper is organized as follows. Section 2 provides the necessary background for our study and presents the cost model used in the rest of the paper. Section 3 presents our dynamic distributed clustering algorithm and Section 4 describes the users subscription process. Section 5 describes the system implementation and experiments. Finally, Section 6 concludes with related and future work.

2. PRELIMINARIES

We start by describing the main features of a typical topic-based pub-sub system and next identify aspects that need to be added to support topic-clusters. Based on this we develop a simple generic cost model that will serve as the basis of the topics clustering algorithm presented in the next section. To give a concrete example, we use Scribe [4], a popular topic-based pub-sub system, as a running example throughout the paper, and explain how our ideas can be used to boost its performance.

Topic-based Pub-Sub. The interfaces of typical P2P topic-based pub-sub systems share four common operations: CREATE, PUBLISH, SUBSCRIBE and UNSUBSCRIBE. The usage scenario is simple: To generate events, one must first CREATE topics. Authorized users (e.g. using proper credentials for access control [4, 21]), also known as the *Publishers*, declare and create topics based on their suggested service. Each topic is virtually represented by an individual peer (often called *a channel*), which is recognized by a unique ID (called *a topic-ID*), and serves as a mediator between the publishers' side and the subscribers' side. To publish an event (send a message) for a given topic, the publisher calls the PUBLISH operation with a specific topic-ID. The message is passed to the appropriate channel and propagated from it to the topic subscribers. To become subscribers of a given topic, interested users call the SUBSCRIBE operation, with the appropriate topic-ID. The corresponding UNSUBSCRIBE operation removes the subscription.

Scribe. Many modern topic-based pub-sub platforms [4, 30, 18, 22] run over Distributed Hash Table (DHT) systems [21, 24, 29] and manage their operative layer using the DHT exported functions (API). Scribe [4] is an example for one such popular system. It is built on top of a DHT overlay called Pastry [21]. Upon a topic declaration, Scribe uses Pastry to locate a participating peer, using the topic-Id as a search key. This node serves as the topic channel. Following that, the subscribers to each topic form a distributed multicast tree which consists of the union of Pastry's routing paths from all subscribers to the channel peer, which now acts as the *root* of the topic multicast tree. The peers in this multicast tree include the subscribed peers, and often also some additional "helper" peers, that happen to reside those Pastry's routing paths from the subscribers to the channel, and are "recruited" to assist in disseminating the topic events. (See [4] for details.) The system's operational cost, in terms of communication, consists of two main ingredients:

Event dissemination To notify the subscribers on a topic update, the publisher sends a *Publish* message to the topic's channel (root of the multicast tree). The message is then disseminated through the multicast tree. The dissemination cost is proportional to the tree size.

Structure maintenance Scribe uses a "keep alive" mechanism to maintain the completeness of the multicast tree and to detect broken edges. It is based on a simple principle where each participating node must notify its status to at least one other node that it is *alive*, or else the node assumed to have failed and is eventually ignored. We omit the algorithmic details and only note that this maintenance is typically performed periodically, with the cost of each maintenance round being proportional to the size of the tree. Some optimization techniques can be applied, for topics with very frequent events, encapsulating the "keep alive" messages with the disseminated events. Such improvements are however rarely applicable to real life environments, since the frequencies of events are relatively low compared to the minimum time interval, between maintenance activities, that is needed for keeping the tree intact.

Topic-clusters. To reduce the systems' maintenance cost, we propose to group topics with similar sets of subscribers into a virtual topic, which we call a *topic-cluster*, thereby unifying their underlying supporting structures. Let us explain how pub-sub system with topic-clusters operates. Just like individual topics, each topic-cluster is given a unique id and is represented by a channel. When a cluster is created, the channels of its topics are informed. Now, when a publisher wishes to publish an event for a given topic, it notifies the topic's channel, which propagates the notification to the channels of all those clusters to which the topic belongs. The channels (of the individual topic and its clusters) then disseminate the event to their subscribers. Users may be subscribed to individual topics as well as to topic-clusters. In the latter case they get notified of the events of all the topics associated with the cluster. If a user that is subscribed to a cluster happens to be interested only in a subset of the cluster topics, a local filter is activated to eliminate the irrelevant events.

In a typical usage scenario, a user declares her interest in a set of topics. The system then determines a subscription policy for the user, namely a set of topics and topic-clusters that covers the user's interests, and subscribes her to them (possibly installing an appropriate filter to eliminate redundant events). When a user changes her interests she informs the system and it adjusts her subscriptions accordingly. The clusters shape is continuously optimized by the system to best fit users interests, (with the users subscriptions being adjusted, when needed, accordingly).

Observe that when a subscriber is interested in all the topics offered by a given cluster, subscribing her to the cluster, rather than to the individual topics, is likely to be cost effective w.r.t. structures maintenance as it eliminates the redundancy in the structures of the individual topics, thereby reducing the overall structures size. On the other hand, when a subscriber is interested in some of the cluster's topics, but not all, the reduction in maintenance comes at the expense of an increased cost for event dissemination - some redundant events are now being sent to the user and filtered upon arrival. The tradeoff between the saving in structure maintenance and the possible increase in event dissemination is a byproduct of the clusters granularity. These need to be determined carefully.

Cost computation. To make this more precise we use the following notation. Let S be a set of subscribers, let T be a set of topics, and let $C \subseteq 2^T$ be a set of (not necessarily disjoint) subsets of topics from T, which we call *topic-clusters*. For a subscriber $s \in S$, we denote the subset of topics from T that interest s by

$interests(s)$. At any given moment, s may be subscribed to some individual topics as well as topic-clusters. We denote these subscriptions by $topics(s)$ and $clusters(s)$, respectively. We assume that the subscriptions of each subscriber cover her interests, namely $interests(s) \subseteq topics(s) \cup \bigcup_{c \in clusters(s)} c$

For a topic $t \in T$ (resp. cluster $c \in C$), the total number of its subscribers is denoted by $size(t)$ (resp. $size(c)$). Namely $size(t) =\mid \{s \mid s \in S, t \in topics(s)\} \mid$, and $size(c) =\mid \{s \mid s \in S, c \in clusters(s)\} \mid$.

Our measurement of cost, for events dissemination and structure maintenance, is in terms of the overall number of messages exchanged between peers. To simplify we assume that all messages are of approximately the same size. Similar development can be done for the case where messages of different types have different sizes. We assume that for all topics maintenance is performed periodically, say, every M seconds. Here too a similar development can be done when topics have distinct maintenance intervals. The average number of events, for a topic t, sent by the publisher within a maintenance interval M is denoted by $freq(t, M)$. For brevity, when M is clear from the context we omit it and use $freq(t)$.

Maintenance Cost The efforts spent on structure maintenance within time interval M consist of one maintenance activity for each topic $t \in T$ and each topic-cluster $c \in C$, as described by the formula MC (for Maintenance Cost) below.

$$MC = \sum_{t \in T} cost_m(size(t)) + \sum_{c \in C} (cost_m(size(c)) + cost_a(|c|))$$

The formula uses two functions, $cost_m$ and $cost_a$. The function $cost_m$ gets as input the number of the topic(-cluster) subscribers and models the average maintenance cost for a channel with that many subscribers. In practice, the exact value of the function at any given moment depends also on the current network topology and the location of the channel and the subscriber peers in this topology. (For instance, in Scribe, $cost_m$ depends on the exact size of the multicast tree, i.e. the number of subscribers plus the number of "helper" peers participating in the tree.) For simplicity we use here instead an average cost estimation. The function $cost_a$ models, for each cluster, the cost of maintaining the association between the cluster and each of its topics. Given the number of topics in the cluster, $cost_a$ computes the average cost of maintaining association with that many topics.

Dissemination Cost The dissemination of an event involves (1) notifying the responsible channel (of a topic or a cluster) and then (2) sending the event from the channel to the subscribers. In our cost computation we use two functions, $cost_n$ and $cost_d$, that model, resp., the cost of notifying the channel and the cost of disseminating the event to its subscribers. $cost_n$ has no parameters while $cost_d$ gets the number of channel subscribers as input. Here again the functions provide average cost estimation. We assume that the channel of each individual topic maintains the list of clusters to which the topic belongs and notifies their respective channels upon events arrival. The efforts spent, within the same time interval M on events dissemination, are thus described by the formula DC (for Dissemination Cost) below.

$$DC = \sum_{t \in T} freq(t) \times [(cost_n() + cost_d(size(t))) + \sum_{\{c \mid t \in c\}} (cost_n() + cost_d(size(c)))]$$

Overall Cost The overall efforts, spent in interval time M, on structure maintenance and events dissemination is termed Overall Cost (abbr. OC). Clearly $OC = MC + DC$.

The OC at a given moment naturally depends on the current topic-clusters and the user subscription. For good performance it is desirable that the clusters shape and the user subscriptions at each moment are such that they minimize the OC. Since the environment is dynamic (users, topics and publishers may come and go; users may change interests; the events frequency in the various topics may change over time), to keep the OC low, the clusters as well as the user subscriptions must be continuously adapted to the system state. As we shall see next, this adaptation also incurs some cost. A major challenge is thus to keep a low OC while not spending too much efforts on clusters and subscriptions update.

3. TOPIC CLUSTERS

To run a pub-sub system with topic-clusters, two main issues need to be addressed. The first is the cluster creation and the continuous adaptations of their shape to the system needs (considered in this section) and the second is the user subscriptions (considered in the next section).

The decentralized P2P nature of the pub-sub system calls for a corresponding distributed clustering algorithm. Our algorithm is based on a set of local "cluster update" operations, performed by individual channels (of topics and clusters) consulting only a relatively small neighborhood. These operations include: the grouping of two individual topics to form a new cluster; the addition of a topic to an existing cluster; the merge of two existing clusters into a single cluster; and conversely the removal of a topics from a cluster and the destruction of clusters.

Prior to each update, its contribution to the reduction of the Overall Cost, OC (as defined in Section 2), is estimated. Only updates that are determined to be beneficial are performed. It should stressed that it is impractical to try and evaluate the *exact benefit* that a given update may bring - this requires gathering information about the full network state and the possible affect of the update on all users; clearly an expensive task in our distributed P2P setting. Consequently, when estimating the effect of an update, we restrict our attention to *a limited part of the network* (to be detailed below) and execute the update only if it is proved beneficial w.r.t. that part. Our method is *safe* in the sense that it *underestimates the potential benefit to the whole network*. This means that all the updates that we decide to perform are guaranteed to improve the system's overall state (although we may miss some other beneficial updates). Our experimental study, described in Section 5, confirms that this safe approach provides a good tradeoff between the gained performance improvements and the efforts spent on them.

Since each such cost estimation nevertheless consumes some resources, we employ in our algorithm probabilistic components to guarantee that (with high probability) only useful updates (and their relevant measurements) are indeed being considered.

In the reminder of this section we consider each of the update operations. For each operation we explain (1) how its contribution to the OC is calculated, (2) what triggers the cost estimation (and consequently the update), and (3) how the update itself is performed.

3.1 Adding a topic to an existing cluster

The first update operation that we analyze is the addition of a topic to an existing cluster. This will prove useful later for the analysis of related operations like the merge of two topics to form a new cluster and the removal of topics from an existing cluster.

Benefit estimation. Assume that a set $\tilde{s} \subseteq S$ of subscribers is subscribed, among others, to a cluster c and to a topic $\bar{t}$ not belonging to c. Would it be more cost effective to add the topic $\bar{t}$ to the cluster c, and consequently remove the users subscription to $\bar{t}$?

The two alternatives are depicted in Figure 1. In the figure, T_c denotes the set of topics in the cluster c. S_c denotes the set of

subscribers subscribed to c. It includes two subsets: a group $\tilde{s} \subseteq S_c$ of subscribes that are also subscribed to $\bar{t}$, and the remaining set of subscribers $S_c \setminus \tilde{s}$. The figure details only partial information about the users: Users in both groups may also be subscribed to other topics and clusters (not shown in the figure). In particular, some of the users in $S_c \setminus \tilde{s}$ may also be interested in $\bar{t}$ but already get it from other clusters (consequently they are not directly subscribed to $\bar{t}$). Similarly, there may be additional subscribers to $\bar{t}$ besides those of c (namely $|\tilde{s}| \leqslant size(\bar{t})$). These are also not shown in the figure.

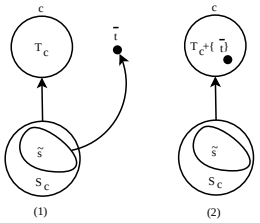


Figure 1: Add a topic to a cluster?

Let E_1 be an environment where the topic $\bar{t}$ is not a member of the cluster c (as in part 1 of Figure 1), and let E_2 be environment obtained from E_1 by adding $\bar{t}$ to c and removing, correspondingly, the subscription of the $\tilde{s}$ users to the topic (as in part 2 of the figure). If we consider the OC formula described in Section 2 for the two environments, we see that the only components that change are those regarding the portion of the network depicted in Figure 1. Let OC_1 and OC_2 denote OC restricted to these parts in the two environments E_1 and E_2, resp. More formally, let $size_1(\bar{t})$ denote the number of subscribers to $\bar{t}$ in E_1, and let $size_2(\bar{t}) = size_1(\bar{t}) - |\tilde{s}|$ be the number of its subscribers in E_2.

$$OC_1 = cost_m(size_1(\bar{t})) + cost_m(|S_c|) + cost_a(|T_c|) + \sum_{t \in T_c \cup \{\bar{t}\}} DC_1(t)$$

$$OC_2 = cost_m(size_2(\bar{t})) + cost_m(|S_c|) + cost_a(|T_c|+1) + \sum_{t \in T_c \cup \{\bar{t}\}} DC_2(t)$$

where DC_1 and DC_2 are obtained from the 2nd line of the DC formula in Section 2 by instantiating $size(\bar{t})$ with $size_1(\bar{t})$ and $size_2(\bar{t})$ resp. It is easy to see that if $OC_2 < OC_1$ then adding $\bar{t}$ to c and removing the subscription of the $\tilde{s}$ users to the topic is beneficial. Note that it is possible that further improvement may be obtained by changing subscriptions of other users according to the new configuration. But even if all other subscriptions stay the same, we are still guaranteed to have a better overall system performance.

To get some intuition about the behavior of the above formula, let us consider a concrete simple example. Assume that we are given a pub-sub system where the cost functions have the following behavior: The cost of event dissemination and structure maintenance for a given topic/cluster is roughly the same as the number of its subscribers, namely $cost_d(N) = cost_m(N) = N$; The cost of maintaining the associating between a cluster's channel and those of its associated topics is roughly the number of topics in the cluster, namely $cost_a(N) = N$; Finally, channels are notified of published events in a constant time, i.e. $cost_n() = 1$.

Instantiating the above formula we get that the update is beneficial if $OC_2 - OC_1 = freq(\bar{t})(|S_c| + 1 - |\tilde{s}|) + 1 - |\tilde{s}| < 0$. Or, put differently, when $|\tilde{s}| > \frac{1+freq(\bar{t})(|S_c|+1)}{1+freq(\bar{t})}$. The principle that the formula highlights is simple; the higher the frequency of events in $\bar{t}$ is, the more interested users among c's subscribers are required in order to make the addition of $\bar{t}$ to c profitable.

To see a numerical example, assume we have a cluster c with 10,000 subscribers (i.e. $|S_c|$=10,000), and $freq(\bar{t}) = 0.1$. Then we have $\frac{1+freq(\bar{t})(|S_c|+1)}{1+freq(\bar{t})} = 910$. Meaning, that as soon as the number of interested users $|\tilde{s}|$ passes 910, the update is profitable.

Triggering benefit estimations. Assume that an update of the above form was determined not to be profitable in a given environment. Looking at the formula above we can see that the update may become profitable if, for instance, the frequency of events in $\bar{t}$ decreases, the overall number of subscribers to c decreases, or the size of the subset that register to $\bar{t}$ increases.

The average frequency of events in a given topic can easily be maintained by the topic channels. Similarly, a rough estimation of the number of subscribers to a given channel can also be easily maintained as part of the maintenance of its underlying structure. A decrease in their value, below a certain threshold derived from the formula, can be used to trigger a new benefit estimation for the update. Tracking an increase in the number of cluster subscribers that are also subscribed to the given topic is more tricky. A naive solution would be to reevaluate benefits whenever a user declares her interest in a topic $\bar{t}$. Observe however that if the threshold is high, it may take many subscriptions (hence many useless cost reevaluation) before reaching a turnover point. For instance, in our previous example, it may take 910 subscriptions to $\bar{t}$ before the update becomes beneficial. To save redundant work we introduce a probabilistic component that assures, with a high probability, to eventually trigger the update when beneficial, while saving many unnecessary computations.

The process consists of three steps. When a user subscribes to $\bar{t}$ we first toss a coin (with probabilities as detailed below), to decide whether this new subscription should trigger a new benefit estimation or not. If yes, an estimate for $|\tilde{s}|$ is calculated by conducting a survey among c subscribers: We determine which fraction of the subscribers is also subscribed to $\bar{t}$ (using a simple sampling technique as in [1]) and multiply it by the overall cluster size (maintained, as described above, by the channel).

To conclude we only need to explain how the probability ω for winning the coin tossing is determined. We use a configurable parameter φ. ω is defined such that the coin tossing will succeed for at least one of the $|\tilde{s}|$ subscribers to $\bar{t}$, with at least φ probability. Namely, $1 - \varphi > (1 - \omega)^{|\tilde{s}|}$. Consequently,

$$\omega > 1 - (1 - \varphi)^{\frac{1}{|\tilde{s}|}}$$

To continue with our running example, recall that we needed at least 911 users to subscribe to $\bar{t}$, to make its addition to c cost effective. If we configure the argument φ to be 0.99, (namely we wish to perform a benefit estimation with at least 99 percent probability), we need to use coin tossing with winning $\omega > 0.005$. Note that it may be the case that the coin tossing happens to win early, before all the required $|\tilde{s}|$ users had been subscribed. However, since consecutive subscribers also toss the coin, there is a 99% probability that another successful toss will happen again between $|\tilde{s}|$ and $2 \cdot |\tilde{s}|$ subscriptions and the update then will indeed be performed.

Overall process. Whenever a user subscribes to a topic, one of the channels of the clusters to which the user is already subscribed to is notified. (We explain below how we chose the specific cluster). Upon receiving this notification, the channel tosses a coin, as described above, to decide if to treat the case or not. If yes, it conducts a survey among its subscribed users, again, as described above, and subsequently decides whether to add the topic to itself or not. Note that while in principle it may be useful to check *all* the clusters to which the user is subscribed, to reduce the overhead we use in our implementation a simple heuristic that chooses the cluster that has the maximum number of filtered out topics. Our experiment show this heuristic to be effective, but naturally other methods may be possible.

We conclude this subsection with two remarks regarding two closely related updated operations.

Removing a topic from a cluster. The process of removing a topic from an existing cluster is similar to the one described above. The removal of a topic becomes beneficial when $OC_2 - OC_1 > 0$, or, in other words, when the number of topic subscribers, $|\tilde{s}|$, drops below the threshold specified previously. The same algorithm as above is employed, except that now we track unsubscribe operations, rather than subscriptions.

Merging two topics to form a new cluster. We have considered above the addition/removal of a topic $\bar{t}$ to/from an *existing* cluster c. A very similar analysis can be applied to determine whether it is beneficial to merge $\bar{t}$ with another topic t to form a *new* cluster, (or, analogously to split the cluster into individual topics). We omit this here.

3.2 Merging two clusters

We next consider the merge of two existing clusters. While some aspects are similar to what we have seen above (e.g. the estimation of the benefit to be gained for the merge) some require particular care (e.g. the triggering of the cost estimation).

Benefit estimation. Assume that two clusters c_1 and c_2, share a set $\tilde{t}$ of common topics and a set $\tilde{t}$ of common subscribers. When is it beneficial to merge c_1 and c_2 into a single cluster $\tilde{c}$ and have their respective subscribers register to the new cluster instead of the two individual clusters?

The two alternatives, before and after the clusters merge, are depicted in Figure 2. The set of users subscribed to the cluster c_i, $i = 1, 2$ is denoted S_{c_i}. It consists of two subsets: a group $\tilde{s} = S_{c_1} \cap S_{c_2}$ that is subscribed to both clusters, and the remaining subscribers $S_{c_i} \setminus \tilde{s}$. Similarly, the set of topics in the cluster c_i is denoted T_{c_i} and is composed of two subsets: a group of topics $\tilde{t} = T_{c_1} \cap T_{c_2}$ that is common to both clusters, and the remaining cluster topics $T_{c_i} \setminus \tilde{t}$. As before, the figure details only partial information about the subscribers: Users may subscribed to additional topics and clusters not shown in the figure. In particular, some of the users in $S_{c_i} \setminus \tilde{s}$ may be interested in topics that are offered by the second cluster but get it from another source (e.g. are subscribed directly to these topics or to other clusters that provide them). This too is not shown in the figure.

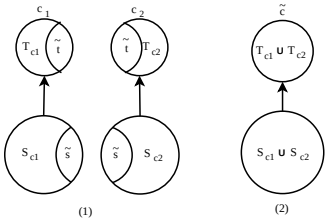


Figure 2: Merge clusters dilemma

Let E_1 be the environment where the clusters c_1 and c_2 are two distinct entities, and let E_2 be the environment obtained from E_1 by merging the two clusters into a new cluster $\tilde{c}$, eliminating the old clusters and subscribing the users of S_1 and S_2 to the new cluster. If we look at the OC formula given in Section 2, for the two environments, we see that the only components that change are those regarding the portion of the network depicted in Figure 2. Similar to section 3.1 let OC_1 and OC_2 denote OC restricted to these parts in the two environments, resp. Namely,

$$OC_1 = cost_m(|S_{c_1}|)+cost_a(|T_{c_1}|)+cost_m(|S_{c_2}|)+cost_a(|T_{c_2}|)+\sum_{t\in T_{c_1}\cup T_{c_2}} DC_1(t)$$

$$OC_2 = cost_m(|S_{c_1} \cup S_{c_2}|) + cost_a(|T_{c_1} \cup T_{c_2}|) + \sum_{t\in T_{c_1}\cup T_{c_2}} DC_2(t)$$

where DC_1 and DC_2 denote the 2nd line of the DC formula in Section 2, using, resp., the clusters in E_1 and E_2 (i.e. having two clusters in the first case and a single in the second). Here too it is easy to see that if $OC_2 < OC_1$ then merging the two clusters, and updating accordingly the subscription of the users, is beneficial.

To get some intuition, let us continue with our running example and assume that we are given a pub-sub system with the same cost functions as in Subsection 3.1. Namely, $cost_d(N) = cost_m(N) = N$, $cost_n() = 1$ and $cost_a(N) = N$. Instantiating the above formula we get that the update is beneficial if

$$OC_2 - OC_1 = \begin{array}{l} \sum_{t\in T_{c1}\setminus\tilde{t}} freq(t)\cdot|S_{c_2}\setminus\tilde{s}| \\ +\sum_{t\in T_{c2}\setminus\tilde{t}} freq(t)\cdot|S_{c1}\setminus\tilde{s}| \\ -\sum_{t\in\tilde{t}} freq(t)(|\tilde{s}|+1)-|\tilde{s}|-|\tilde{t}| \end{array} < 0$$

To see more closely how the formula behaves, let us further simplify it by assuming that the event frequencies in all topics are approximately the same, and denote the frequency by α. We get that the update is beneficial if

$$OC_2-OC_1 = \alpha(|T_{c1}\setminus\tilde{t}|\cdot|S_{c_2}\setminus\tilde{s}|+|T_{c2}\setminus\tilde{t}|\cdot|S_{c_1}\setminus\tilde{s}|-|\tilde{t}|\cdot(|\tilde{s}|+1))-|\tilde{s}|-|\tilde{t}| < 0$$

Again, the principle that the formula highlights is simple: the decision whether to merge the two clusters or not is influenced by two main factors, the number of common subscribers $\tilde{s}$ and the number of common topics $\tilde{t}$. The bigger these groups are, the more profitable is the merge.

To see a numerical example, assume that the two clusters c_1 and c_2 share 80 topics (i.e. $|\tilde{t}| = 80$) and have each additionally 20 more topics (i.e. $|T_{c_1} \setminus \tilde{t}| = |T_{c_2} \setminus \tilde{t}| = 20$). Also assume that the number of distinct subscribers to each of the clusters is 200 (i.e. $|S_{c1} \setminus \tilde{s}| = |S_{c2} \setminus \tilde{s}| = 200$). How many common subscribers should the clusters have for the merge to be beneficial? Using the above formula we get that the merge is profitable as soon as the set $\tilde{s}$ of common subscribers satisfies the inequality

$$|\tilde{s}| > \frac{\alpha|T_{c1} \setminus \tilde{t}| \cdot |S_{c2} \setminus \tilde{s}| + \alpha|T_{c2} \setminus \tilde{t}| \cdot |S_{c1} \setminus \tilde{s}| - |\tilde{t}|(\alpha + 1)}{\alpha|\tilde{t}| + 1}$$

Instantiated by the above numbers we get that $|\tilde{s}| > 79.11$. Namely, the merge is profitable if the number of common subscribers is greater than 79. Observe that this relatively low number of common users (i.e. $\sim 25\%$ of the overall number of cluster subscribers), is due the fact that the clusters share many common topics (80% of the clusters size). If the number of common topics would drop (say to 7 topics, i.e 25%), then the merge would become profitable only if the number of common subscriber was much higher ($|\tilde{s}| > 466$, i.e. $\sim 70\%$).

Triggering benefit estimations and updates. The above formula reveals several factors that affect the usefulness of the merging of the clusters. These include (1) the event frequencies of the topics, (2) the number of topics in each individual cluster and the number of its subscribers, and (3) the number of the common topics and common subscribers. As explained in the previous subsection, it is easy to maintain an estimation (and detect changes in the values) of (1) and (2). It is slightly more tricky to maintain an estimate on the numbers $|\tilde{t}|$ of common topics and the number $|\tilde{s}|$ of common subscribers. We explain how this is done below.

$|\tilde{t}|$: The channel of every cluster c maintains, for each of its topics, a list of the ids of all other clusters to which this topic belongs. The channel can then easily determine the number of common topics with any other cluster, by counting in how many lists it appears. Observe that these lists can be maintained with a fairly little overhead: Recall that the channel

of each topic records the clusters to which the topic belongs. Whenever the topic is added/removed to/form a cluster, the topic's channel notifies the other topic clusters about the update. Since topics typically belong only to a small number of clusters, and clusters are not updated too frequently, the overhead involved is low.

$|\tilde{s}|$: Recall that the channel of each cluster c maintains an estimation about the overall number of its subscribers. Each channel also samples periodically its subscribers to know to which other clusters they are subscribed. The distribution of answers, together with the estimation about the overall number of subscribers, allows to estimate how many subscribers the cluster shares with all other clusters. To trigger a merge operation, the channel of c contacts the channel of the cluster c' having maximum number of common subscribers, passing to c' its size and the number of common users and lets c' determine (using its own gathered size estimations) whether the number of common subscribers justifies a merge. A simple locking mechanism is used to prevent two clusters from attempting to simultaneously perform the same merge, or to merge a given cluster into two distinct clusters.

Note that in principle it may be useful to contact other clusters, besides the one with maximal number of common subscribers. We have chosen this heuristic to reduce overhead (with our experiment proving it to be effective).

We conclude with two remarks.

Splitting two clusters. In principle the same formula can be used to determine when it is beneficial to split a cluster. Observe however that the number of possible candidate splits to be considered is exponential in the size of the cluster. To avoid the implied overhead we decided not to include an explicit split operation in our implementation and instead rely on a the removal of single topics from a cluster (as described in the previous subsection) to shrink clusters when needed.

Single topics. We have considered above the merge of two clusters. If one (or both) clusters are viewed as singleton sets, similar reasoning can be used to trigger the merge of a single topic to a cluster or the merge of two single topics, in a analogous way to what we have seen in the previous paragraph.

4. SUBSCRIPTION

Consider a system with a set T of topics, a set C of clusters, and as set S of subscribers, that are possibly already subscribed to some topics and clusters that cover their current interests. Assume that a subscriber $s \in S$ declares her interest in some additional topics, and/or is no longer interested in some other topics. How should we update s's subscriptions to achieve best performance?

It easy to see that, given a set of topics that the user is interested in, finding the optimal subscription, (namely one that covers her interests while reducing the OC cost function to minimum), is difficult. Indeed, a simple reduction from the Set Cover problem shows the problem to be NP complete (details omitted). Note that this also implies that even the incremental version of the subscription problem, where a user that is currently subscribed in an optimal manner changes just one of her interests, is NP complete, (or else one could apply it iteratively to get a PTime optimal solution to the general problem - a contradiction.) We thus employ a heuristic, using a greedy (un)subscription algorithm, that iterates over the requested changes in the user interests and treats each individual request for adding/removing a topic heuristically.

We first explain how a SUBSCRIBE request to a new topic is treated and next consider UNSUBSCRIBE.

SUBSCRIBE. First recall that for a user s, $topics(s)$ and $clusters(s)$ denote, resp., the set of topics and clusters to which the user is subscribed (prior to the new SUBSCRIBE request). For a topic t we use $associated(t)$ to denote the set of clusters to which the topic t belongs.

Assume that a user s wishes to subscribe to a new topic t. We first check if t belongs to any of the clusters to which s is already subscribed (and is being currently filtered out). If yes, then we simply update the filter, and we are done. Otherwise, we need to choose between subscribing s directly to t, or to some new cluster $c \in associated(t)$ that includes t. Note that if s is currently subscribed to some individual topics $t_1, \ldots, t_n$ that are also members of the new cluster c, then, if we subscribe s to c, we can remove these direct subscriptions - their events will be already received from c. (Events of the other topics in c, that do not interest the user, will be filtered out). The two alternatives are depicted in Figure 3. T_c denotes the set of topics in the new cluster c and S_c denotes its set of subscribers (before s expressed her interest in t.) The user s may be also subscribed to additional clusters and topics that are omitted from the figure for simplicity.

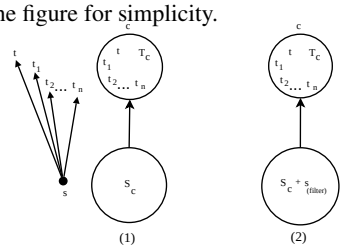


Figure 3: Subscribe instruction dilemma

To choose between the two alternative we use a cost analysis similar to that of the previous section. Let $\tilde{t} = (topics(s) \cup \{t\}) \cap c$ be the set topics to which s is subscribed directly, in the first alternative, and which is obtained from c, in the second alternative. Let E_1 denote the first environment and E_2 the second one. If we consider the OC formula from section 2, for the two optional environments, we see that the only components that are different are those regarding the portion of the network depicted in Figure 3. Let OC_1 and OC_2 denote OC restricted to these parts in the two environments E_1 and E_2, resp. More formally, let $size_1(t)$ be the number of subscribers for each $t \in \tilde{t}$ under E_1, and let $size_2(t) = size_1(t)$–1 be the number of its subscribers under E_2.

$$OC_1 = \sum_{t \in \tilde{t}} cost_m(size_1(t)) + cost_m(|S_c|) + cost_a(|T_c|) + \sum_{t \in T_c} DC_1(t)$$

$$OC_2 = \sum_{t \in \tilde{t}} cost_m(size_2(t)) + cost_m(|S_c|+1) + cost_a(|T_c|) + \sum_{t \in T_c} DC_2(t)$$

where DC_1 and DC_2 are obtained from the 2nd line of the DC formula in Section 2 by instantiating $size(t)$ with $size_1(t)$ and $size_2(t)$ resp. It is easy to see that if $OC_2 < OC_1$ then subscribing s to the cluster c is more beneficial, and vice versa.

To continue with our running example, for a pub-sub system with cost functions as in Section 3.1 (e.g. $cost_d(N) = cost_m(N) = N$, $cost_n() = 1$ and $cost_a(N) = N$) and where topics have an average events frequency α, subscription to the cluster is better if $OC_2 - OC_1 = 1 - |\tilde{t}| + \alpha \cdot |T_c \setminus \tilde{t}| < 0$. Or, put differently, when $|\tilde{t}| > \frac{1+\alpha|T_c|}{1+\alpha}$. This indeed captures the intuition that the lower the rate of events α is, the less common topics are required to make the subscription to the cluster beneficial.

To conclude note that one could check all the clusters in $c \in associated(t)$ and choose the one with highest benefit (if any).

To reduce the overhead, we employ a simple heuristic, considering only the cluster where the size of $\tilde{t}$ (i.e., $|topics(s) \cap c|$) is maximal.

UNSUBSCRIBE. Assume that a user s looses interest in a topic t. If s is directly subscribed to s, namely $t \in topics(s)$ then we simply remove the subscription. Otherwise, if s receives t from some cluster c, we have two alternatives. We can stay subscribed to the cluster c and simply filter out t's events upon arrival, or we can remove the subscription to c altogether and subscribe directly to those other topics in c that are still of interest. The analysis of the two options follows exactly the same lines as above, except that now the set of topics to be considered is $\tilde{t} = (topics(s) \setminus \{t\}) \cap c$.

When a subscriber declares (a change in) her interests, the system performs the (un)subscription, as described above, based on the current clusters' shape, and the user can immediately start getting interesting events. In parallel, these new (un)subscriptions may trigger, as explained in Section 3, some changes in the clusters shape, which may further improve the original subscription.

5. IMPLEMENTATION & EXPERIMENTS

In order to validate our approach and demonstrate the gain in performance it brings compared to traditional topics-based pub-sub, we implemented the algorithms and ran extensive experiments. We summarize now the main features of our implementation, and highlight our main experimental results.

5.1 Implementation

We have implemented the above ideas in the *Tamara* pub-sub system. *Tamara* was developed in Java on top of the popular topic-based pub-sub platform Scribe [4] (which is itself built on top of the DHT system Pastry[21]). *Tamara* exposes to users the same API as Scribe for defining topics, publishing events, and (un)subscribing. It uses our new dynamic clustering algorithm to automatically group topics into clusters and redirect the publishers' event notifications, as well as the users subscriptions, accordingly.

Network layers. Figure 4 illustrates the layered architecture that we adopted in our design, starting from the highest level - the application that utilizes *Tamara* through its API, and ending with the Network itself.

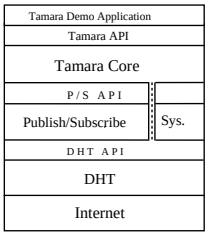


Figure 4: Tamara network hierarchy

Although our current implementation runs on top of Scribe and Pastry, our design (i.e. *Tamara core*, see next paragraph) is generic and can be applied to other pub-sub systems sharing similar standard features. *Tamara* leverages the basic event notification qualities of Scribe for creating and managing its topics and its clusters, and utilizes the Pastry's capabilities for locating nodes of interest. On initiation, a *Tamara* peer first tries to recognize the underneath Network, and to connect to an already established Pastry ring. Upon success, it further allocates a new Scribe instance using the created Pastry node ID. In that sense, both Scribe and Pastry act as mediator layers between *Tamara* and the underneath network.

Tamara core. Each network node in *Tamara* may play different roles, being simultaneously a publisher, a subscriber, and/or a channel for one or more topics/clusters. The different facets of a node are represented by distinct *entities* and managed using two main software components. (1) The **Entities Manager** serves as interface between the node's entities and the network. For any incoming message/request, the *Entities Manager* identifies the destination entity and checks the legitimacy of the request. For instance, a "publisher" entity should not receive a "merge" request as such requests are handled only by topic/cluster entities. The *Entities Manager* may also be requested to supply necessary information about its entities. For example, each *topic* entity holds information about the average frequency of events, the list of clusters (IDs) to which the topic belongs, and an estimation number of its subscribers. (2) Each entity is implemented as a set of state machines that run in parallel and are managed by a **State Machine Manager**. Each request (e.g. "Create a topic") received by the entity invokes a new machine that is in charge of its execution. The machine interacts with other machines (of the same or other entities) to achieve its goal, sending messages, performing local operations, receiving answers, etc. The design is modular and new state machines supporting additional operations can be easily integrated.

Tamara's software is available for download and we also plan to make it available under an open-source license.

Simulator. To estimate the gain in performance that *Tamara* brings relative to a standard use of Scribe, we had built a simulator that allows to emulate the two environments, with varying parameters, and to compare their performance. The simulator is written in Java and can be instantiated with different configurations as explained below. At any given moment the simulator can report the current value of OC for the running environment, as well as the value for its two main ingredients, namely the current costs for events dissemination and for structures maintenance. When emulating *Tamara* the simulator can also report the additional efforts spent on clusters formation, their adaptation to the changing user interests, and on users (re)subscriptions. The simulator can also output statistical data, like the current total number of clusters, the average number of their associated topics, the average number of topics with direct subscribers, the number of cluster updates, etc.

We ran two series of experiments to validate our approach. First, we used synthetic data to vary the main parameters that may affect the performance and quantify their effects on the relative gain. Second, we used real data, in the context of a real life application concerning the dissemination of updates of GNU/Linux software to users. These two series of experiments are presented next.

5.2 Experiments on synthetic data

We run two types of such experiments. The first set was aimed at quantify the quality of *Tamara* as a *clustering algorithm*. The second focused on the dynamic aspects of the system and evaluated the adaptability of the clustering to shifts in users interests. The parameters we varied in the experiments are the following.

Users and topics: In both sets of experiments we ranged the number of topics from ten to three thousand and the number of users from a few hundreds to ten thousands. As the results were practically independent of the number of topics/users we show here only a representative sample conducted with 120 topics and 3000 users.

User interests: We considered various distributions for the user interests, ranging uniform distribution to ones with increasingly stronger clustering characteristics (i.e. where user interests naturally divide the topics into several clusters). We show below two representative samples, one where users' interests form clear clusters and the second where they have a uniform distribution, and use them to explain the overall trend.

Maintenance interval: In our analysis of the Overall Cost (OC) we assumed that, for each topic, maintenance for its underlying structure is performed periodically, every M seconds. After a few experiments, it became evident that increasing (resp. decreasing) the value of M has essentially the same effect as decreasing (resp. increasing) the average number of events being disseminated within such an interval and the rate of changes in user interests. We thus show below experiments for a fixed value of M of six minutes[1], while varying the other parameters.

Events frequency: To get a feeling about events frequency in real pub-sub systems, we examined 30 RSS feed channels of five leading news sites: *abcnews.com*, *bbc.co.uk*, *CNN.com*, *foxnews.com* and *news.sky.com*, considering popular topics such as Headlines, World news, Business, Sport, Technology and Entertainment. We used this as a base reference, and built several models for the distribution of events frequencies (in the interval M of 6 minutes). We present here three representatives, one having the same events distribution as the base reference, the second having more frequent events, and the third having less frequent events. The models, referred below as the *Average*, *Frequent* and *Infrequent* models are depicted in Tables 1(a), (b) and (c), resp.

Frequency	% of topics
0.6-0.6	15
0.2-0.4	5
0.1-0.2	25
0.02 - 0.1	45
0.002 - 0.02	10

Frequency	% of topics
1-3	5
0.5-1	20
0.1-0.5	50
0.01-0.1	20
0.001-0.01	5

Frequency	% of topics
0.1-0.2	10
0.01-0.1	40
0.001-0.01	50

Table 1: (a) Average model (b) Frequent mode (c) Infrequent model

Cost functions: The Overall Cost (OC) formula uses four configurable cost functions, $cost_n$ and $cost_d$ that model the cost of notifying the channel and resp. disseminating the event to the subscribers, $cost_m$ that models the average maintenance cost for a channel, and $cost_a$ that models the cost of maintaining the association between a cluster and its associated topics. We have conducted experiments with various values for the functions, including in particular functions that model the average and worse case costs for the corresponding actions in Scribe and Pastry and similar pub-sub/DHT systems (we omit the details here for space constraints). The experimental results that we obtained were similar for all these functions. We present below a representative sample that uses the same cost functions as in our running example: $cost_n() = 1$, $cost_d(N) = cost_m(N) = cost_a(N) = N$.

Experiment time and Subscription rate: To test the effect of the rate of user subscriptions (or changes in user interests) on the system's performance, we have fixed the average number of requests per users and varied the time interval in which all requests are issued. In the experiments below each user subscribes/changes subscription to approximately 25 topics (i.e. there is a total of 75,000 (un)subscription operations) and the overall experiment time ranges from two month to a single day.

Quality of clustering. As mentioned above, our first set of experiments aims at quantify the quality of *Tamara* as a *clustering algorithm*. We started the system from an empty state with no subscribers, let users join the system and declare their interest, watched the clusters being formed and measured the corresponding performance gain. In the first experiment reported below we examine a case where the data holds strong clustering characteristics. Users subscribe on the average of 25 topics out of the 120 available topics; the subscriptions here were chosen so that the topics split naturally into 10 clusters, each containing approximately 12 topics.

[1]This is in fact close to Scribe's real default value for its maintenance interval

Events	Algo.	Structure maint.	Event diss.	OC
Average	Tamara	-78.75%	+45.45%	-63.73 %
	K-means	-53.75%	+9.1%	-46.15%
Frequent	Tamara	-73.75%	+15.62%	-48.21%
	K-means	-65%	+9.3%	-43.75%
Infrequent	Tamara	-88.75%	+150%	-77.38%
	K-means	-65%	0	-61.9%

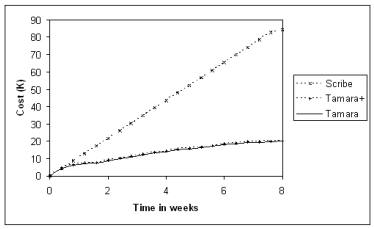
Table 2: Clusters quality

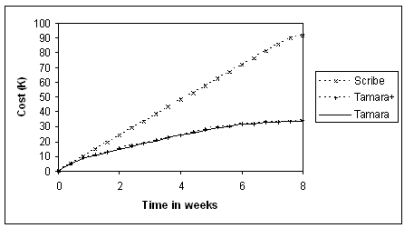
The three graphs in Figures 5(a),(b), and (c), depict the behavior of Scribe and *Tamara*, for the Infrequent, Average, and Frequent models of event frequencies, resp. The X-axis describes time (a period of two months in which more and more users subscribe to the system). The Y-axis describes bandwidth consumption. The figures show the values of OC for both systems (denoted in the figure by Scribe and Tamara, resp.). For *Tamara* it also shows the total operational cost (denoted in the figure by Tamara+), that includes, in addition to OC, the costs of clusters formation and adaptation, the subscription of users to the clusters, and the changes, when needed to these subscription. (We count here all additional messages exchanged due to these actions). It is interesting to note that this overhead turns out to be so low that the Tamara and Tamara+ curves practically merge. As time passes, and more and more users subscribe, the costs increase. But is in all cases it is significantly lower for Tamara+ than for native Scribe. We can see that the gain somewhat increases (resp. decreases) when the events frequency decrease (resp. increases).

To assess the quality of the clusters generated by our distributed dynamic algorithm, we have compared them to what would have been generated by a standard centralized algorithm such a K-means[13][2]. Table 2 shows the delta (in percentages) relative to Scribe, obtained by Tamara and K-means, for the value of OC and its two components: events dissemination and structure maintenance. A negative delta (say, -65%) for a given algorithm means better performance (i.e. the considered algorithm consumed 65% less resources than Scribe), and vise versa. We can see that, in general, clustering increases the cost of events dissemination (since now users may get redundant events that are filtered upon arrival) but greatly reduces the cost of the structures maintenance, and that their relative weight in the overall value of OC defends on the events frequency. Surprisingly, although K-means operates in a centralized manner and can utilize full knowledge about the network state, it nevertheless achieves less good OC cost than Tamara. This is for two main reasons. First, like many common clustering algorithms, K-means computes disjoint clusters. This is not necessarily best for the problem at hand. Second, K-means is a general purpose clustering algorithm that is suitable for the optimization of a single target functions (e.g. events dissemination cost) while Tamara is tuned to optimize the particular OC cost function that we use here, which is the sum of two such target functions.

When the rate of user subscriptions increases, so does the cost of cluster formation and maintenance. This is illustrated in Figures 6(a) and (b) below, where the same number of subscription as above (75,000) is now issued in just two weeks, and in one day, resp. The event frequencies here are based on the Average model. We can see that at the beginning of an intense subscription period, (e.g. as the one depicted in Figure 6(b)), *Tamara* spends efforts on analyzing the subscriptions and forming the appropriate clusters. These efforts pay off soon after, and the performance becomes superior to that of native Scribe. It is important to note that at the end of such an intense period, when the user interests become more stable, the overhead will drop significantly and the performance will go back close to what is described by the Tamara solid line in the figure.

[2]We have also examined other clustering algorithms, but since K-means performed better we use it here as a comparison point.





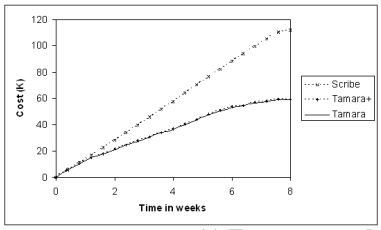
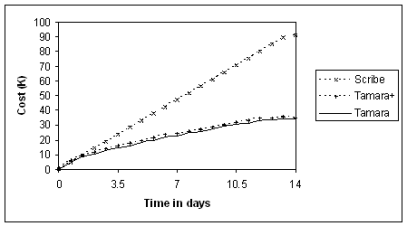


Figure 5: (a) Infrequent model for event freq. (b) Average model (c) Frequent model



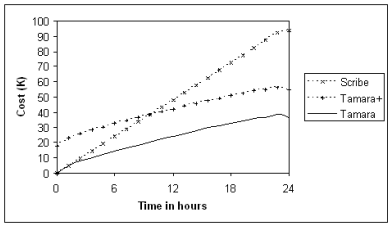


Figure 6: (a) Two weeks (b) One day

To conclude let us consider the case where users subscriptions have uniform distribution, or in other words no clear dependencies between topics exist. The behavior of Scribe and Tamara is depicted in Figure 7(a). As before we have 75,000 subscriptions issued during two months and events frequency based on the Average model. Interestingly, even when no natural clusters exist, Tamara still performs better. This is because it observes that topics with very low event frequencies can be clustered, even when their subscribers are fairly different, as the gain from less structures maintenance bypasses the loss from higher cost of events dissemination.

Adaptive clustering. Our second set of experiments focused on the dynamic aspects of *Tamara*. In other words, we evaluated the adaptability of the clustering to shifts in users' interest. We started the system from a given configuration, with clusters that match perfectly current user interests, let users change their interests, and watched the system as it adapts the clusters (and correspondingly the users subscriptions) to the new state. In the first experiment reported below we examine a case where the new set of user interests have strong clustering characteristics. We consider afterwards the case where the new subscriptions have uniform distribution.

The three graphs in Figures 8 (a), (b), and (c) depict the behavior of Scribe and *Tamara*, for the Infrequent, Average, and Frequent models of event frequencies, resp. The figure also shows what would have happened if we did not adapt the clusters' shape and kept them, as in the beginning, for the whole run. This is represented by the SC (for Static Clusters) line in the figure.

It is evident that Tamara's performance is superior and that the overhead of clusters maintenance is negligible. Observe that the cost for Scribe is stable (and high) throughout the experiments as the overall number of users and subscriptions remains similar. For *Tamara* the cost increases a bit at the beginning when users interests start shifting and the existing clusters are not suitable anymore, but decreases again as soon as the clusters get updated and reflect, more and more, the new discovered dependencies between topics.

To assess the quality of the new clusters generated at the end of the run we again compared them to what would have been generated by the K-means clustering algorithm, for the new set of subscriptions. Table 3 depicts the results. It also illustrates what could be the effect of a static approach, that ignores changes in users interests and sticks to the original clusters.

Observe that, in this case, the new clustering computed by Tamara is sometimes slightly inferior (w.r.t. OC) to that of K-means. This is because, when starting from an given system state (clustering),

Events	Algo.	Structure maint.	Event diss.	OC
Average	SC	-55%	+141.66%	-29.34%
	Tamara	-70%	+100%	-47.82%
	K-means	-63.75%	+8.33%	-54.34%
Frequent	SC	-25%	+39.4%	-6.2%
	Tamara	-67.5%	+30.3%	-38.9%
	K-means	-62.5%	+12.12%	-40.7%
Infrequent	SC	-63.75%	+200%	-51.2%
	Tamara	-96.25%	+275%	-78.57%
	K-means	-65%	0	-61.9%

Table 3: Adaptive clusters quality

Tamara's goal is not just to move to a new, more suitable, state, but also to do the change without inflicting too much overhead (i.e. it may not fully optimize the clustering to avoid consuming too much resources). This is critical for the saving that the new clustering brings to be meaningful.

Here again, when the rate of users subscription increases, so does the cost of clusters maintenance. The results are similar to what we have seen in the previous subsection and we omit them.

To conclude, consider the case where the new subscriptions have uniform distribution. The results are depicted in figure 7(b). We can see again that even when there are not salient dependencies between topics, clustering may still be useful for unifying some of the structures of topics with very low event frequencies. Note however that the difference between the old clustering and the new one is now less significant.

5.3 Experiments on real data

We tested our algorithms in the context of a real life application concerning the dissemination of updates of GNU/Linux software packages to users. A GNU/Linux distribution consists of thousands of software packages. The subset of packages that each user installs depends on her particular hardware/software configuration. To keep the software up to date, users would like to get notified about updates to the installed packages. Dependencies exist between software packages: to install an updated package, one often needs to have also installed updated versions for other packages that it uses. Viewing a software package as a topic, a user that subscribes to a given topic (package updates) is likely to subscribe also to topics (updates of other packages) on which it depends. Consequently there are topics that are likely to be subscribed by many users (e.g. kernel or gcc updates), while others are subscribed by particular user groups with common configurations.

The data that we used for our experiments was taken from FTP logs of a Debian GNU/Linux mirror site, recording the download requests of users during a full year. We tracked thousands of anonymous individual Linux users and interpreted each request for a package download as a sign for the user's interest in that package (over three thousand packages all together). The download history

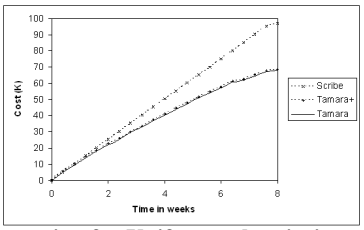
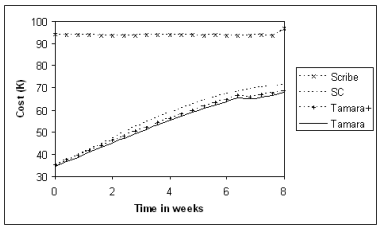


Figure 7: (a) Clustering for Uniform subscription



(b) Adaptive clustering for Uniform subscription

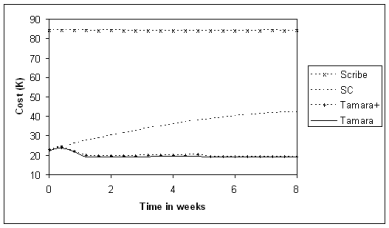
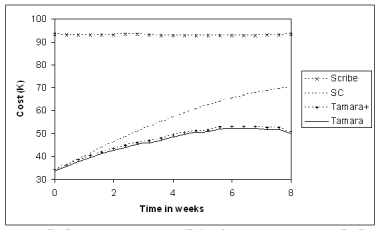
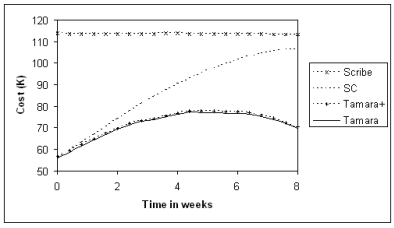


Figure 8: (a) Adaptive clust. for Infrequent model



(b) Average model



(c) Frequent model

of users over time was used to infer changes in interests. The frequencies of events (package updates) that we used is based on statistics gathered by Mandriva[14], a Linux distribution editor, over the last six years.

Since the observed events frequency were fairly low - packages rarely get updated more than once within a day - it is possible to perform maintenance less frequently, namely increases maintenance interval M, without risking much loss of events. Figures 9(a), (b), and (c), depict the behavior of Scribe and *Tamara*, for maintenance intervals of 2, 12, and 24 hours, resp. The frequencies of events are adjusted to the length of the maintenance interval. The system here starts from an empty state with the Linux users gradually joining the system and declaring their interests.

As the maintenance activities become more sparse, the savings offered by Tamara are smaller, but still provide in the long run significant performance improvement. In particular, even for sparse maintenance, *Tamara* is cost effective as soon as 250 users join the system, and when all the thousand users are subscribed the performance is more that 4 times better than that of native Scribe. Similar significant performance gains are observed when users change their interests. We omit this here for space reasons.

6. RELATED WORK AND CONCLUSION

We studied in this paper topic-based pub-sub systems and proposed a novel technique for minimizing the maintenance overhead for their topics. Our solution is based on a new distributed clustering algorithm that takes advantage of "correlations" between user subscriptions to dynamically group topics and thereby unifies their supporting structures and reduces costs. Our technique continuously adapts the topic clusters and the user subscriptions to the system state, and incurs only very minimal overhead. We have implemented our solution in the *Tamara* pub-sub system, and showed experimentally that it is extremely effective.

The publish-subscribe paradigm have received much attention in recent years. Research on topic-based pub-sub led to the development of several systems, such as Scribe[4], Bayteux[30], and CAN[19]. While different in their implementation, they share (as described in Section 2) similar API and structure, and may employ the clustering technique developed in this paper to improve performance. Complementary to our work are projects like Green [23] that introduce a middle-ware to support flexible pub-sub on top of diverse network types (e.g. Internet vs. mobile networks).

The flourishing of RSS news syndication led to the development of several systems (e.g. Corona[18] and FeedTree [22]) designed to alleviate the load on RSS feed providers and allow for better scalability. They optimize the *polling* strategy of news and/or replace it by *push* technology. The clustering technique that we propose here may help to improve performance in the push-based components of such applications.

The grouping of topics into sets has been previously proposed in the literature in the context of *type-based* pub-sub: To provide users with varying subscription granularity, these systems grouped topics into sets forming a sub-set hierarchy [26]. A main difference with the present work is the static nature of this predefined grouping. In contrast, our solution adapts, continuously, the topic clusters to the changes in user needs. It would be interesting to try to design a suitable user interface that exposes the current clusters to the users, to ease their subscription task (interpreting a subscription to a cluster as subscription for the set of topics it currently contains.)

Much research has been devoted in the database community over the last ten years to content delivery and data dissemination. See, e.g., [7, 8, 5] for a very small sample. In this paper we focus on a relatively simple class of such systems, that disseminate events of predefined topics. The relative simplicity of these systems is precisely what makes them attractive for simple application. As mentioned in Section 2, content-based pub-sub allows for more flexible subscription, based on the content of messages. Some example systems include JEDI[6], SEINA[3] and Kyra[2]. The added flexibility requires more sophisticated protocols, typically with higher runtime overhead. The use of clustering algorithms for enhancing the performance of content-based pub sub has been considered in [20]. However, all the algorithms considered there are centralized and the dynamic aspect of subscription is not addressed. Distributed clustering algorithm for content-based pub-sub has been proposed e.g. in [28, 27]. The algorithm in [28] assumes the use of a central coordinator. Such a coordinator may not exist in a fully distributed P2P environment as the one considered in this paper. Furthermore, changes to user subscriptions simply reactivate the central algorithm and the minimization of this overhead is not addressed. The adaptive algorithm in [27], on the other hand, is fully distributed, but allows only clusters where the subscribers are interested in *all* topics. It also assumes an unstructured overlay network, while the DHT overlay used by typical topic-based pub-sub systems is structured. It would be interesting to see whether the ideas developed in the present work are applicable to content-based systems as well.

Several distributed clustering algorithms appear in the literature, (e.g. [12, 11, 17]), but assume static input and do not account for changes in the data. Adaptive clustering is considered in [10, 15].

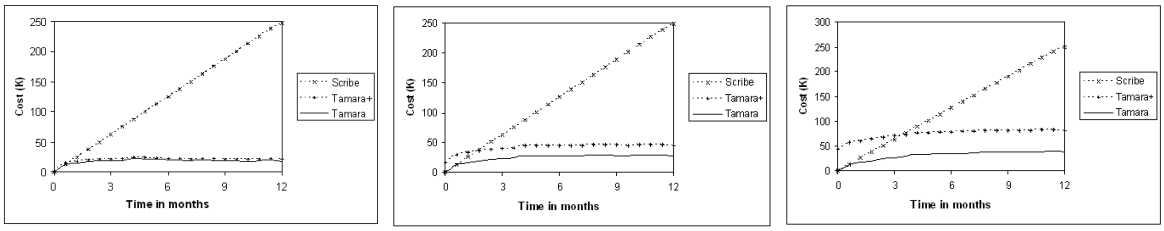


Figure 9: (a) Linux updates for $M = 2$ hours **(b) $M = 12$ hours** **(c) $M = 24$ hours**

But these works assume constant dimension for the clustered objects. This is not the case in our context: In a dynamic pub-sub system, users and topics may come and go. Viewing the topics (or users) as objects, this implies that the arity of the vectors describing the objects may changes in time. To our knowledge the present work is the first one to provide a distributed clustering algorithm that allows, with very minimal overhead, to continuously adapt the topic clusters and the user subscriptions to the changing state of the system.

7. REFERENCES

[1] J. E. Bartlett, J. W.Kotrlik, and C. C.Higgins. Organizational research: Determining appropriate sample size in survey research. *Information Technology, Learning, and Performance*, 19(1), 2001.

[2] F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. *IEEE INFOCOM*, 2004.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[5] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *Proc. ACM SIGMOD*, pages 587–598, 2006.

[6] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. on Software Engineering*, 27(9):827–850, 2001.

[7] Y. Diao, P. Fischer, M.J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 341–342, 2002.

[8] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale xml dissemination service. In *Proc. VLDB*, pages 612–623, 2004.

[9] Freshmeat. The largest index of unix and cross-platform softwares. *http://freshmeat.net/projects/gimp/.*

[10] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starovic, and B. Tangney. Dynamic clustering in an object-oriented distributed system. In *Proc. Objects in Large Distributed Applications OLDA-II*, Ottawa, Canada, 1992.

[11] E. Januzaj, H. Kriegel, and M. Pfeifle. Towards effective and efficient distributed clustering. In *Workshop on Clustering Large Data Sets (ICDM2003)*, Melbourne, FL, 2003.

[12] R. Jin, A. Goswami, and G. Agrawal. Fast and exact out-of-core and distributed k-means clustering. *Knowledge and Information Systems*, 10(1):17–40, 2006.

[13] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions*, 24(7):881–892, 2002.

[14] mandriva.com. Mandrive linux distribution. *http://www.mandriva.com/.*

[15] S. Moritz and B. Ernst W. DDC: A dynamic and distributed clustering algorithm for networked virtual environments based on p2p networks. In *Proceedings of CoNEXT'05*, France, 2005.

[16] nielsen-netratings.com. NIELSEN/NETRATING. *http://www.nielsen-netratings.com/pr/pr–050920.*

[17] E. Ogston, B. Overeinder, M. van Steen, and F. Brazier. A method for decentralized clustering in large multi-agent systems. In *Proc. Int. Conference on Autonomous agents and multiagent systems(AAMAS'03)*, pages 789–796, New York, NY, USA, 2003.

[18] V. Ramasubramanian, R. Peterson, and Emin Gun Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *Proc. of Networked System Design and Implementation, San Jose, California*, 2006.

[19] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proc. Int. COST264 Workshop on Networked Group Communication*, volume 2233, pages 14–29. LNCS, 2001.

[20] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proc. Int. Conference on Distributed Computing Systems (ICDCS'02)*, page 133, 2002.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, Germany, 2001.

[22] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing web micronews with peer-to-peer event notification. In *Proc. Int. Workshop on Peer-to-Peer Systems (IPTPS05)*, New York, 2005.

[23] T. Sivaharan, G. Blair, and G. Coulson. GREEN: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *Proc, Int. Symposium on Distributed Objects and Applications (DOA05)*, Agia Napa, Cyprus, 2005.

[24] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, pages 149–160, 2001.

[25] GIMP. GNU image manipulation program. *http://www.gimp.org/.*

[26] E. Patrick Th., G. Rachid, and S. Joe. Type-Based Publish/Subscribe. Technical report, 2000.

[27] S. Voulgaris, E. Riviere, A. Kermarrec, and M. van Steen. Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. In *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS),*, California, USA, February 2006.

[28] R. Zhang and Y. C. Hu. Hyper: A hybrid approach to efficient content-based publish/subscribe. In *Proc. Int. Conference on Distributed Computing Systems (ICDCS'05)*, pages 427–436. ICDCS, 2005.

[29] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[30] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. Int. workshop on Network and OS support for digital audio and video*, pages 11–20. ACM, 2001.