Local Building Blocks for a Scalable Pub/Sub Infrastructure

Alexander Shraer* Sivan Bercovici † Gregory Chockler‡ Idit Keidar* Roie Melamed‡
Yoav Tock‡ Roman Vitenberg§

1 Introduction

A publish-subscribe (pub/sub) infrastructure [7] supports the dissemination of varied information to a large population of users with individual preferences. In this paper we focus on topic-based pub/sub systems, where users can subscribe to topics of interest, and receive all updates related to these topics. One notable application of such systems is real-time dissemination of trading data to stock brokers. Another example is a modern data center offering a large variety of application services that are accessed through the Internet. In such environments, the individual applications are typically replicated for performance and availability thereby creating overlapping multicast domains each of which is mapped into a separate pub/sub topic.

Modern pub/sub applications have stringent scalability requirement. Information is often published by multiple sources, and disseminated to potentially tens or even hundreds of thousands of users, who may be geographically dispersed. Moreover, the information is categorized according to a rich collection of topics, possibly tens of thousands, and users typically subscribe to many topics. In order to deal with the huge number of users as well as with the system's geographical scale, nodes are organized in a logical structure, e.g., a hierarchical [14] or other overlay organization [9, 6], where each node has a small number of neighbors.

The main challenge in adapting existing overlay network technology for pub/sub applications is coping with the immense number of topics. It is possible to create a single overlay for all topics, where nodes subscribed to each topic form a connected sub-graph, as we did in SpiderCast [6, 5]. Although this significantly reduces the number of links per user, it does not solve all the difficulties stemming from the vast number of topics, for example, the difficulty of ensuring reliable message delivery in each group of users subscribed to a topic. The memory and processing requirements of managing reliability for each topic separately would be prohibitive. It is therefore common to employ aggregation, either dividing topics into clusters, as done (with a centralized offline algorithm) in [15], or, as in QSM [14], by organizing subscribers into regions, in which their interests are shared with other subscribers; this approach is akin to the use of lightweight groups in early group communication systems [8]. Of course, the use of a single cluster for all topics is not feasible, as clients have limited resources (e.g., bandwidth), and cannot filter out thousands of irrelevant topics. Aggregating topics into a set of clusters can balance the two needs, and achieve scalability in the number of managed topics without excessive filtering [14, 15]. This paper focuses on such aggregation. The output of the aggregation algorithm can then be used to either construct a separate overlay spanning the nodes subscribed to each cluster, or serve as the subscription input to SpiderCast.

In order to be efficient, aggregation must take the overlap of interests into account, so as to minimize the amount of filtering at clients (or at brokers or proxies, if these are employed). Moreover, topic aggregation should dynamically evolve in response to changes in subscribers' interests.

This paper presents *local* algorithms for distributed topic clustering, i.e., their performance does not depend on the size of the system, but rather corresponds to the difficulty of the problem at hand. These algorithms are also *anytime*, in the sense that they always output a result which improves over time. In the context of topic aggregation, we allow for adjusting the assignment of topics to clusters dynamically according to changing needs, while preserving some level of optimality. Local subscription changes that have little impact on the desired aggregation do not incur global communication. A change in subscription to a handful of topics does not necessitate computing anew all the topic aggregations, i.e., the solution is incremental. Moreover, the solution is distributed and does not rely on global knowledge of subscriptions.

As the number of topics might be very large, the idea of our solution is to focus on the popular topics, i.e., those that interest more than some threshold of users. It is important

*Department of Electrical Engineering, Technion, Haifa, Israel. {shralex@tx, idish@ee}.technion.ac.il
†Department of Computer Science, Technion, Haifa, Israel. sberco@cs.technion.ac.il
‡IBM Haifa Research Labs, Haifa, Israel. {chockler, roiem, tock}@il.ibm.com
§Department of Informatics, University of Oslo, Norway. romanvi@ifi.uio.no

to optimize their assignment to clusters since they impact more users, and moreover we expect the rate of messages concerning popular topics to be high, especially when subscribers can also publish. Thus, filtering such topics might be expensive. To identify these topics we introduce a novel anytime local algorithm, Majority Filters, in Section 2.

In Section 3 we present a distributed version of the K-Means algorithm [10], which allows for assignment of the popular topics to clusters such that a certain cost function is minimized. Specifically, we are interested in minimizing the overall filtering cost induced by the assignment, e.g., as defined in [15]. To find a the best suitable cluster for a single topic, we introduce the local and anytime Multiple Choice Voting algorithm in Section 4: each node constructs a vector consisting of the *local filtering cost* induced by assigning the given topic to each one of the clusters; the Multiple Choice Voting algorithm then finds the cluster which induces the minimal *overall filtering cost*.

Finally, we note that the techniques introduced in this paper, although together comprise a solution for pub/sub, may be independently applicable in other domains: Majority Filters for collaborative detection of global abnormalities, such as spam or worms, K-Means for data mining applications, and Multiple Choice Voting for applications such as facility location [11].

2 Majority Filters

Given multiple sets of elements, we wish to filter those elements that appear in more than λ percent of the sets (i.e., have a global support above a certain λ-threshold). In order to find this subset of popular elements among the distributed sets, one can explicitly vote on each element, filtering out those with insufficient support. However, the large number of elements may make this solution infeasible. Instead, we use Bloom-filters [3] as the basis of our data-structure, which results in a compact representation of the set of elements, which in turn, makes the filtering task lighter. We present a decentralized local solution for filtering, based on Bloom-filters and distributed majority-voting protocols [17]. We extend this work to support multi-level filtering.

Background A Bloom-filter is a compact data-structure used for the approximated representation of a set S, supporting membership queries. It is constructed from an m-bit array that is first initialized to zero. To represent the elements in S using a Bloom-filter, each element in the set is hashed using h hash-functions. The h resulting values are used as indices to the m-bit array, and the corresponding bits are set to 1. To query whether or not an input element is in the set, the element is hashed, and the h resulting indices are checked. If all corresponding bits are set, we say that the element is contained in the set. Thus, false-positives may occur due to the mapping of a non-contained element to a set of bits that map either one or more other elements. Nonetheless, this drawback is outweighed by the Bloom-filter's simplicity, space-efficiency, and lack of false-negatives [4].

We use a decentralized majority-voting protocol (e.g., [17]). In such protocols, each node has an input-bit, and votes on an output bit. The algorithm guarantees that once the system is stable, each node eventually converges to the same output bit — 1 if a λ support was detected, and 0 otherwise. Such a protocol requires no synchronization, and is local in the sense that in most of the cases, each node can compute the majority based on information from its near surroundings.

Related work Some previous work [13, 16] suggested to use agglomeration of Bloom-filters as a means for distributively collecting top-scored items (the distributed *top-k* problem). These solutions are not local, they rely on a coordinator and are single-shot protocols, as apposed to the fully decentralized, anytime local solution we propose.

Majority filter A *majority-filter* consists of an input Bloom-filter that represents the local set , an output Bloom-filter that represents the globally-popular set, and a λ threshold. Each node holds a majority-filter, initializing the first Bloom-filter using its local set. Once the first Bloom-filter is ready, each bit in that Bloom-filter is voted on through the use of the decentralized majority-voting protocol and the λ value. The results of the parallel-voting are stored in the second Bloom-filter. Upon stabilization, every node holds the same global Bloom-filter, where only the bits that have a λ-majority are set. Querying the majority-filter for membership is done in the same manner as with Bloom-filters, by querying the global Bloom-filter: the global Bloom-filter yields a true-membership answer for those elements that have a λ-support.

As we base our solution on local protocols such as [17], the convergence of *majority-filters* does not depend on the network size, but rather on the level of agreement on what elements are popular. Moreover, as the local *majority-voting* protocol adapts to dynamic input changes, the solution we propose also retains the property of anytime — each node's *majority-filter* represents the global set of popular elements, while adapting to changes in that set.

Analysis Much like the original Bloom-filters, our solution suffers from some false-positives, yet no false-negatives. In addition to the false-positive evident in the original Bloom-filters, *Majority-filters* also experience a false-positive popular element due to the aggregation of less-than-λ supported elements. We prove the following theorem in Appendix A:

Theorem 1. *The probability of a false-positive membership respond due to aggregation is bounded by*

$$e^{-n \cdot h \cdot (1-e^{\frac{-h \cdot l}{b}}) \cdot \frac{\epsilon^2}{4}}$$

where n is the number of nodes, h is the number of hash functions, l is the average number of elements in a set, b is the size of the majority-filter, and $\epsilon = \frac{\lambda}{1-e^{\frac{-h \cdot l}{b}}} - 1 \leq 2 \cdot e - 1$.

Assuming a configuration with $n = 1000, l = 20, h = 3, b = 50$ and $\lambda = 0.8$, we compute an upper bound of 0.0256. For $\lambda \geq 0.8$ the upper bound on the probability of a false positive response is shown in Figure 1. Combining Figure 1 with the analysis in [4], we conclude that the probability of false-positives due to aggregation is negligible in comparison to the one originating from mapping-overlap of an element with an in-set element (i.e., false-positives that appear in the original Bloom-filters).

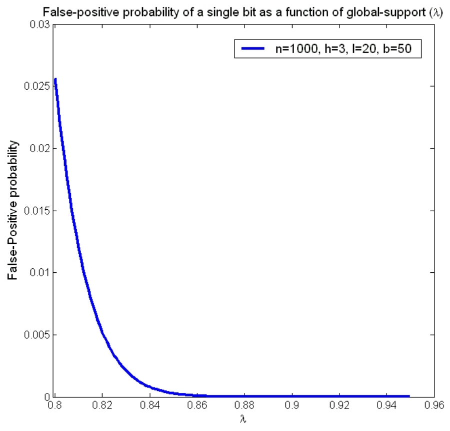


Figure 1. False positive rate due to aggregation

Under the previous configuration, changing $b = 100$, will reduce the false-positive due to aggregation probability to $5.26 \cdot 10^{-30}$.

Extension This majority-filter presented so far can categorize the global set of elements into λ-supported elements and less-than-λ supported elements. One might wish to categorize the global set elements into more than two categories. This may be easily archived by extending the presented data-structure to a majority-filter cascade. In a majority-filter cascade (abbreviated MFC), each node holds a set of majority-filters that are initialized with ascending values of λ. The first majority-filter, with its corresponding lowest λ_1 threshold, is used to filter the local set of elements, marking those elements which have at least λ_1 global support. These elements are used as input for the second majority-filter, which marks the elements with at least λ_2 global support, and so on. The analysis and optimization of the MFC are out of the scope of this paper. Nonetheless, one can easily notice that the data-structure remains compact, scalable, and anytime, much like to majority-filter building block from which it is constructed.

3 Distributed K-Means

We first use Majority Filters to identify popular topics, i.e., those that interest more than some threshold λ of the nodes. In the remainder of this section, when topics are mentioned, we mean the chosen popular topics.

In a pub/sub system employing topic clustering, each node subscribes to all clusters including topics of interest. However, each cluster potentially includes many other topics, which requires the node to filter out information published for these topics. The goal of the clustering algorithm is to minimize the overall amount of excessive filtering incurred as a result of the assignment of topics to clusters. We use a distributed version of the K-Means algorithm:

```
1: while cost improves
2:     foreach topic $t$
3:         remove $t$ from from the cluster to which it is assigned
4:         use Multiple Choice Voting to choose new assignment for $t$
```

We iterate over the topics, in turn placing each one in the best possible cluster, until no improvement (or no significant improvement) is made. In order to decide which cluster is most suitable for a given topic, the local Multiple Choice Voting of Section 4 is employed: assuming that the topic is initially not assigned to any cluster, each node estimates the amount of filtering that will be performed locally if the topic is placed in each one of the clusters. Details about this distance metric are found in [15]. The resulting vector serves as an input for the algorithm of Section 4 which finds the cluster that minimizes the overall amount of filtering.

To speed up convergence, we allow multiple instances of Multiple Choice Voting to proceed in parallel, each instance finding a cluster closest to a particular topic according to our excess filtering metric, so that multiple topics can be potentially reassigned at each step. It is easy to see that this optimization does not break the safety of distributed K-Means as its halting condition is identical to that of the centralized K-Means algorithms so that the algorithm is guaranteed to have reached a local minimum when they halt. However, termination is no longer guaranteed since fluctuations in the assignment of topics to clusters (in situations when many nodes change their interests at once) are possible.

To deal with this problem, we combine distributed K-Means with a leader-based version as follows: we start by using a leader-based synchronous implementation of K-Means where a leader node iterates over the topics to compute the initial topic-to-cluster mapping. Although the leader coordinates the clustering, it does not have to know the subscription interest of other nodes, unlike in centralized K-Means solutions. We note that the coordinator needs to know the cost of the current assignment so that it can stop iterations as soon as the cost does not sufficiently improve.

After the initial clustering is made and the system has stabilized, the concurrent version is used to dynamically adapt to changes in popularity and subscriptions: as long as no such changes occur, no messages are sent; a node which detects a local change for topic t, initiates the Multiple Choice Voting protocol for t. Whenever a significant change in the node subscription that could potentially lead to instability is detected, the nodes fall-back to the leader-based solution to reset the mapping. Simulations show that the concurrent protocol converges even if there is a system-wide change in as much as 20 topics out of 100.

A system using our algorithm should provide continued service while the topology (based on current aggregation) is reconfigured. For this purpose, a make-before-break policy can be employed, i.e., keeping both old and new topologies alive for a while, and adding an out-of-band mechanism like system-wide gossip for detecting message losses and inconsistencies, and requesting explicit recoveries.

4 Multiple Choice Voting

In this section, we present an anytime local algorithm for distributed multiple choice voting. The algorithm decides which one out of k possibilities has minimal cost, when the cost is distributed among the system nodes. The algorithm works in conjunction with a spanning tree construction algorithm, which does not have to ensure its output to be a perfect tree all the time, but eventually must construct one. Our algorithm operates on the graph $G(V, E)$, which is the (current) output of the spanning tree algorithm. The algorithm tolerates dynamic changes both in the votes (input vectors) and in G, as long as G eventually becomes a tree after the change (a similar assumption was made in [17, 11]). If the inputs and topology are static for enough time, all nodes eventually output the same correct result. The algorithm uses the following notation:

Definition 1. (Neighborhood [2])
The r-neighborhood ($r \in R^+$) of a node v, $\Gamma_r(v)$, is the set of nodes $\{v | dist(v, v') \leq r\}$. $\widehat{\Gamma}(v) = \Gamma_1(v) - \{v\}$ denotes the set of v's neighbors.

Algorithm. Each node $u \in V$ has an input vector $S_u[1..k]$, where the i'th entry contains the cost of the i'th option for u, and eventually outputs an index m_u between 1 and k, such that the sum of $S_u[m_u]$ entries of all nodes $u \in V$ is minimized. For our application, where voting is used to assign a given topic to one of the clusters, the i'th entry of u's input vector holds the local filtering cost

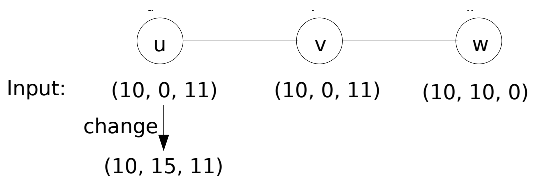
Algorithm 1 Minimizing the Global Cost, algorithm for node u

```
1: Input: a vector $S_u[1..k]$
2: Output: an index $m_u$
3: State: $\forall w \in \widehat{\Gamma}(u)$, vectors $S_u^{wu}[1..k]$ and $S_u^{uw}[1..k]$, initially zero
4: Definitions:
5:     $\forall i, O_u[i] \triangleq S_u[i] + \sum_{w \in \widehat{\Gamma}(u)} S_u^{wu}[i]$
6:     $\mathrm{argmin}(\text{vector } v) \triangleq min\{i \mid \forall j : (j \neq i \Rightarrow v[j] \geq v[i])\}$
7:     $m_u \triangleq \mathrm{argmin}(O_u)$
8:     $\forall w \in \widehat{\Gamma}(u) \forall i, O_u^{uw}[i] \triangleq S_u^{uw}[i] + S_u^{wu}[i]$
9: upon change in $\widehat{\Gamma}(u)$
10:     Create $S_u^{wu}[1..k]$ and $S_u^{uw}[1..k]$ for every newly added $w \in \widehat{\Gamma}(u)$
          and initialize to zero vectors
11:     Delete $S_u^{wu}$ and $S_u^{uw}$ for every $w$ no longer in $\widehat{\Gamma}(u)$
12:     send-neighbors()
13: upon receiving a vector $S$ from $w$
14:     $S_u^{wu} \leftarrow S$
15:     send-neighbors()
16: procedure send-neighbors()
17:     for each $w \in \widehat{\Gamma}(u)$
18:         if (($S_u^{uw}$ is the zero vector) or ($\mathrm{argmin}(O_u^{uw}) \neq m_u$)
19:             or ($\exists j$ s.t. ($O_u[j] - O_u[m_u] < O_u^{uw}[j] - O_u^{uw}[m_u]$))
20:             $\forall i, S_u^{uw}[i] \leftarrow S_u[i] + \sum_{v \in \widehat{\Gamma}(u) \wedge v \neq w} S_u^{vu}[i]$
21:             send $S_u^{uw}$ to $w$
22: Initially and upon change in $S_u$: send-neighbors()
```

of assigning the topic to cluster i, and the output of the algorithm is the index of the most appropriate cluster for this topic, i.e., the one that minimizes overall filtering.

Each node u maintains two vectors for each neighbor $w \in \widehat{\Gamma}(u)$: the last vector received from w, $S_u^{wu}[1..k]$, and the last vector sent to w, $S_u^{uw}[1..k]$. The pseudo-code uses several notations: $\mathrm{argmin}(v)$ gets a vector v as input and returns the minimal index corresponding to the minimal entry in v; the output of a node u at any given time is $m_u = \mathrm{argmin}(O_u)$, where O_u is the sum of S_u and the vectors S_u^{wu} received from all neighbors $w \in \widehat{\Gamma}(u)$, and O_u^{uw} denotes the sum $S_u^{wu} + S_u^{uw}$. Intuitively, $\mathrm{argmin}(O_u^{uw})$ is u's assessment of w's output. The output of every node u in a static system is eventually $m_u = \mathrm{argmin}(\sum_{v \in V}(S_v))$.

A node u initially sends its own vector to each neighbor w. It later communicates only if a change occurs that has the potential to alter the global decision. This can happen if (1) u's new local decision (m_u) is different than u's assessment of the decision of one of its neighbors (line 18), or (2) the cost advantage of option m_u, compared to some other option j, has decreased at u (line 19). The first rule is necessary since all the nodes have to reach the same decision. Thus, if the decision changes, a message must be sent. We will explain why the second rule is needed shortly. Consider the following scenario:



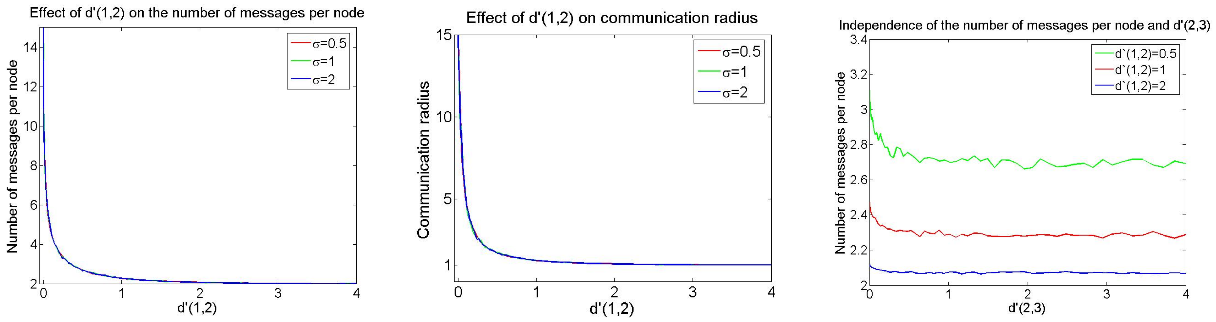


Figure 2. Performance of Algorithm 1 depends only on d'(1,2).

Initially, the inputs S_u, S_v and S_w are as shown in the figure above, $O_u = S_u$, $O_v = S_v$ and $O_w = S_w$. Thus, the initial output of u and v is 2 and the output of w is 3. Each node then sends its vector to its neighbors. When u receives v's vector, it calculates $O_u = S_u + S_u^{vu} = (20, 0, 22)$ and similarly $O_v = S_v + S_v^{uv} + S_v^{wv} = (30, 10, 22)$ and $O_w = S_w + S_w^{vw} = (20, 10, 11)$. Also, notice that $O_u^{uw} = S_u^{uw} + S_u^{wu} = (20, 0, 22)$. Thus, all output 2, which is indeed correct since $\sum_{v' \in V} S_{v'} = (30, 10, 22)$, i.e., option 2 has the lowest global cost. The system is now stable and no further messages are sent.

Now suppose that the input vector of u becomes $S_u = (10, 15, 11)$. O_u is now $(20, 15, 22)$, thus m_u is still 2. Since $O_u^{uw} = S_u^{uw} + S_u^{wu}$ did not change, the rule on line 18 evaluates to *false*. However, 2 is not the correct result – the globally best option is now 3, since $\sum_{v' \in V} S_{v'} = (30, 25, 22)$. Using the rule on line 19, u can detect that the advantage of the second option over the third is 7 at u, whereas it is 22 according to u's assessment of v's state (O_u^{uw}) and sends a message to v. v then sees that the best option has changed to be 3 and (using the rule on line 18) sends an update to both u and w. Thus, a change occurring at u might seem insignificant to u and not trigger the rule on line 18, although combined with changes or data at other nodes, the global effect of this change could be significant. This is when the rule on line 19 is needed. We prove the following theorem in Appendix C:

Theorem 2. *Observe a suffix of any run of Algorithm 1 over a static communication tree with bidirectional links $G(V, E)$, where each node $u \in V$ has a static input vector S_u. Then (a) the algorithm eventually reaches quiescence, and (b) from this time onward, for every node $u \in V$, $m_u = \text{argmin}(\sum_{w \in V} S_w)$.*

Optimization. We insert the following line right after line 20:

$$\forall j \in \{1 \dots k\}, S_u^{uw}[j] \leftarrow S_u^{uw}[j] - S_u^{uw}[\text{argmin}(S_u^{uw})]$$

To avoid the numbers in messages growing to the sum of the whole spanning tree, we use this optimization in every step, and thus keep them smaller. Clearly, subtracting the same amount from each entry of the vector has no effect on the relative cost of the different possibilities, i.e., on the choice of the best option or the cost advantage of the best option relatively to any other option.

Experimental results. We simulated Algorithm 1 over a random binary tree consisting of 1000 nodes, with 3 possible choices ($k = 3$). Each node u (where $u \in [1, \dots, 1000]$), draws a 3-dimensional independent random Gaussian vector $\mathbf{S}_u \in \mathbb{R}^3$, i.e., $\mathbf{S}_u[i] \sim N(\mu_i, \sigma_i)$. For the sake of simplicity we chose an equal variance σ for these distributions, and set means $\mu_1 < \mu_2 < \mu_3$.

As Algorithm 1 is local, its performance does not depend on the size of G, but rather on the difficulty of the problem. In our case, the difficulty is to distinguish the choice having the minimal cost from other possibilities. Thus, we measured the performance of Algorithm 1 as a function of the distance between the distributions. In order to express this distance, we used the metric of *discriminability*, denoted d', defined as follows: the distance between options i and j is $d'(i, j) = \frac{(\mu_i - \mu_j)^2}{\sigma_1^2 + \sigma_2^2} = \frac{(\mu_i - \mu_j)^2}{2\sigma^2}$. Appendix B gives the rationale behind the choice of this metric. The difficulty of the problem is captured by $d'(1, 2)$, since this measures the distance between the two choices having the average minimal cost, whereas all other distances $d'(1, i)$ are larger than $d'(1, 2)$ as we set $\mu_2 < \mu_i$ for all $i > 2$ (in our case there are only three choices).

Figure 2 shows the results of the simulations. To illustrate how performance is affected by $d'(1, 2)$, in the left and middle graphs we plot the average number of messages per node and the average communication radius of a node in a run of our algorithm (we averaged over 30 runs), as a function of $d'(1, 2)$. The graphs show strong dependency – the farther apart are the distributions of options 1 and 2, the easier it is for the algorithm to locally converge. Even though the average radius of a node in our graph is approximately 17, we see that an average node can reach the correct result communicating only with neighbors at distance less than 5, unless $d'(1, 2)$ is very small, i.e., when it is difficult to (locally) distinguish options 1 and 2.

The rightmost graph of Figure 2 shows that algorithm performance does not depend on the distance between options 2 and 3, i.e., $d'(2,3)$. We see that for a given $d'(1,2)$ the performance of our algorithm is almost constant as $d'(2,3)$ changes. The graph of average communication radius shows very similar results. Note the behavior of the graph for small values of $d'(2,3)$: we can see a slight influence of $d'(2,3)$ in this range, especially for small values of $d'(1,2)$.The reason is that although $\mu_2 < \mu_3$ in all experiments, when μ_2 and μ_3 are very close (i.e., $d'(2,3)$ is close to 0), in some experiments option 3 had lower cost than option 2 because of the variance of both distributions. As $d'(1,2)$ decreases, the difficulty to distinguish between 1 and 3 in these cases increases, and we see a steeper curve for small values of $d'(2,3)$. As $d'(2,3)$ increases, the chances of option 3 to become second best get smaller, and we see that $d'(2,3)$ no longer influences performance.

Related Work. The problem of multiple choice voting was considered before in [17] and [11]. In [17], the purpose was to rank k possibilities and the algorithm separately compares each pair of choices, using the local majority-voting algorithm, implying bit complexity of $O(k^2)$. Consider a problem instance where the ranking, starting with the best choice, is $C_1, C_2, ..., C_k$ (this is the global ranking, not known to any single node in the system). Since all pairs of choices (C_i, C_j) are compared, the convergence time of their algorithm would depend on the maximum of $d'(i,j)$ over all i, j. Krivitski et al. [11] proposed a different solution as part of a distributed facility location algorithm. Their algorithm employs an average of $log(k)$ comparison phases, in each comparing one option to the rest using the local majority voting algorithm [17]. Since the result of one phase serves as input to the next, their algorithm employs a backtracking scheme. Only choices that win in a phase continue to the next phase. On average, half of the remaining choices are eliminated in each phase and therefore the average number of comparisons is $2k$. Thus, the convergence time of their algorithm depends on the maximum of $d'(i,j)$ over an average of $2k$ pairs of indices i and j. Our algorithm operates in one phase and has $\theta(k)$ bit complexity. As shown by our simulations, the performance of our algorithm depends only on $d'(1,2)$, i.e., on the distinguishability between C_1 and C_2. Thus our algorithm achieves much better locality for the multiple-choice voting problem.

Other work has considered aggregation in synchronous setting with a fixed topology [2, 1], and does not tolerate asynchronous communication or topology changes as our solution does.

5 Conclusions

We presented a solution for topic clustering in a dynamic pub/sub system with large number of topics. We cope with the challenges of scalability and dynamic behavior using local anytime algorithms.

References

[1] Y. Birk, I. Keidar, L. Liss, and A. Schuster. Efficient dynamic aggregation. In *DISC*, pages 90–104, 2006.

[2] Y. Birk, I. Keidar, L. Liss, A. Schuster, and R. Wolff. Veracity radius: capturing the locality of distributed computations. In *PODC*, pages 102–111, 2006.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey, 2002.

[5] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub-sub with many topics: Problems, algorithms, and evaluation. In *PODC*, 2007.

[6] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: A scalable interest aware overlay for topic-based pub/sub communication. In *DEBS*, 2007.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[8] B. B. Glade, K. P. Birman, R. C. Cooper, and R. van Renesse. Light-weight process groups. In *OpenForum 92 Technical Conference*, Nov. 1992.

[9] R. W. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rassmussen. Corona: a communication service for scalable, reliable group collaboration systems. In *CSCW*, pages 140–149, 1996.

[10] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[11] D. Krivitski, R. Wolff, and A. Schuster. A local facility location algorithm for sensor networks. *DCOSS*, 2005.

[12] R. N. McDonough and A. D. Whalen. *Detection of Signals in Noise*. Academic Press, San Diego, CA, 1995.

[13] S. Michel, P. Triantafillou, and G. Weikum. KLEE: A framework for distributed top-k query algorithms. In *VLDB 2005*, pages 637–648, 2005.

[14] K. Ostrowski and K. Birman. Extensible web services architecture for notification in large-scale systems. In *ICWS*, 2006.

[15] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, pages 320–326, 2005.

[16] G. Weikum, H. Bast, G. Canright, D. Hales, C. Schindelhauer, and P. Triantafillou. Towards self-organizing query routing and processing for peer-to-peer web search. In *ECCS '05*, pages 7–24, 2005.

[17] R. Wolff and A. Schuster. Association rule mining in peer-to-peer systems. *ICDM*, 2003.

A Properties of Majority Filters

Theorem 1 (restated) *The probability of a false-positive membership respond is bounded by* $\sqrt[4]{e^{-n\cdot h\cdot(1-e^{\frac{-h\cdot l}{b}})\cdot\epsilon^2}}$ *where* n *is the number of nodes,* h *is the number of hash functions,* l *is the average number of elements in a set,* b *is the size of the majority-filter, and* $\epsilon = \frac{\lambda}{1-e^{\frac{-h\cdot l}{b}}} - 1 \leq 2\cdot e - 1$.

Proof. To analyze the probability of false-positives, we start by examining the probability of a regular Bloom-filter's bit to be zero:

$$p_0 = (\frac{b-1}{b})^{h\cdot l} = (1-\frac{1}{b})^{h\cdot l} \simeq e^{-\frac{h\cdot l}{b}} \tag{1}$$

Thus, the probability of a bit to be one is:

$$p_1 = 1 - p_0 \simeq 1 - e^{-\frac{h\cdot l}{b}} \tag{2}$$

We would like evaluate the amount of false-positive elements left after the λ-support filtering. In order to do that, we will need to compute the probability of a bit to be one, with a global support of at least λ. In the following analysis, we assume that only the none-globally-supported elements populate the filter:

$$p_1^\lambda = \sum_{i=\lceil\lambda\cdot n\rceil}^{n} \binom{n}{i} \cdot p_1^i \cdot p_0^{n-i} \tag{3}$$

Through the use of Chernoff bounds, we can now conclude an upper bound for the probability of a bit to be false-positively set. Let X be a random-variable representing the number of bits globally set:

$$p_1^\lambda = P(X > (1+\epsilon)\cdot\mu) = P(X > \lambda\cdot n) \leq e^{\frac{-\mu\epsilon^2}{4}}$$

$$\mu = n \cdot p_1$$

From the above two equations we can conclude ϵ:

$$\epsilon = \frac{\lambda}{p_1} - 1$$

under the assumption that $\epsilon \leq 2\cdot e - 1$, the following bound holds:

$$p_1^\lambda = P(X > \lambda\cdot n) \leq e^{\frac{-\mu\epsilon^2}{4}}$$
$$= \sqrt[4]{e^{-n\cdot(1-e^{\frac{-h\cdot l}{b}})\cdot\epsilon^2}}$$

$$p_{false-positive} = (p_1^\lambda)^h \tag{4}$$
$$\leq e^{-n\cdot h\cdot(1-e^{\frac{-h\cdot l}{b}})\cdot\frac{\epsilon^2}{4}} \tag{5}$$

□

B Discriminability - Rationale

In order to evaluate the *argmin* algorithm presented in the previous section, we use the following model. Assume that each node n (where $n \in [1,\ldots,N]$), draws a K-dimensional independent random Gaussian vector $\mathbf{v}_n \in \mathbb{R}^K$. For the sake of simplicity let us further assume that the elements of $\mathbf{v}_n$ have equal variance $\mathbf{v}_n[k] \sim N(\mu_k, \sigma)$.

Let $\bar{\mathbf{v}}_n = \frac{1}{N}\sum_n \mathbf{v}_n$ be the empirical mean of the node population. For large N, $k^* = argmin(\bar{\mathbf{v}}_n) = argmin(\mu)$. Our goal is, however, for each node to find k^* without having to communicate with all the other nodes.

In order to achieve that, each node n maintains the sample mean vector of a certain neighborhood NE around it,

$$\mathbf{u}_n = \frac{1}{M}\sum_{m\in NE}\mathbf{v}_n,$$

where $M = |NE|$. Note that $\mathbf{u}_n[k] \sim N(\mu_k, \sigma/\sqrt{M})$.

Based on that information the node must decide which is the vector element k with the minimal mean (mu_k). Let us denote D_n the decision of node n.

For the simplicity of exposition let us also assume $\mu_k < \mu_{k+1}$ (Clearly, this ordering is not known to the individual nodes). How difficult it is to find the correct k^*?

The probability of making the wrong decision at node n is

$$P_e = P\{\bigcup_{k>1} \mathbf{u}_n[k] < \mathbf{u}_n[1]\}$$

which can be bounded

$$P_e \leq \sum_{k>1} P\{\mathbf{u}_n[k] < \mathbf{u}_n[1]\}$$

The probability of pair-wise confusion can be written exactly:

$$P_{e(1,k)} = \int_{\frac{\alpha_k\sqrt{M}}{2}}^{\infty} (2\pi)^{-0.5}\mathtt{exp}(-u^2/2)du$$

where $\alpha_k = \frac{|\mu_1-\mu_k|}{\sigma}$. This means that the probability of error decreases (1) when the size of the neighborhood increases, and (2) when α increases. The term $\frac{|\mu_1-\mu_k|}{\sigma}$ is known as *discriminability*, and is an inherent measure for the difficulty of the pair-wise decision problem [12]. Thus, in order to make a decision with a certain degree of certainty, for a given distribution of $\mathbf{v}_n$, one must communicate with a sufficiently large neighborhood. Note that when $\alpha_2 \ll \alpha_3$, the dominating factor in P_e is $P_{e(1,2)}$.

We therefore investigate the size of the neighborhood the *argmin* algorithm communicates with as a function of the different values of α_k.

C Correctness of Algorithm 1

We prove that if G does not change, then eventually no messages are sent, and every node outputs $\operatorname{argmin}(\sum_{w \in V} S_w)$. Denote by R_v^u the set of nodes reachable from a node v without using the edge between v and u, and $OR_v^u \triangleq \sum_{w \in R_v^u} S_w$. Additionally, notice that if messages are not currently being sent between v and u, then $S_u^{uv} = S_v^{uv}$, $S_u^{vu} = S_v^{vu}$ and $O_u^{uv} = O_v^{vu}$. First, we prove the following lemma, which shows that if quiescence is reached then all outputs are the same.

Lemma 3. *If no messages are sent, then every two nodes $u, v \in V$, output $m_u = m_v$.*

Proof. Suppose for the purpose of contradiction that there exist two nodes $u', v' \in V$ s.t. $m'_u \neq m'_v$. Since G is a bidirectional tree, there exists a single path between u' and v'. Somewhere along this path, there exist two neighboring nodes $u, v \in V$, s.t. $m_u \neq m_v$. Since no messages are being sent, the rule on line 18 is false at node u. Therefore $m_u = \operatorname{argmin} O_u^{uv}$. Since no messages are in transit between u and v, $O_u^{uv} = O_v^{vu}$ and we get $m_u = \operatorname{argmin} O_v^{vu}$. By our assumption that $m_u \neq m_v$ we get that $\operatorname{argmin} O_v^{vu} \neq m_v$, violating the rule on line 18 in v. Therefore, a message must be sent, contradicting our assumption that the system has reached quiescence. □

The following lemma proves that when messages are no longer sent, the best option m_u at node u, is at least as good (compared to any other option) in the subgraph reachable through u's neighbor v as it looks from the vector received from v.

Lemma 4. *Observe a suffix of any run of Algorithm 1 over a static communication tree with bidirectional links $G(V, E)$, where each node $u \in V$ has a static vector S_u, and the system has reached quiescence time. Observe any node $u \in V$. Then for each $v \in \widehat{\Gamma}(u)$, and for every index $j \in \{1 \dots k\}$, $S_u^{vu}[j] - S_u^{vu}[m_u] \leq OR_v^u[j] - OR_v^u[m_u]$.*

Proof. Observe any node $u \in V$ and $v \in \widehat{\Gamma}(u)$. The proof is by induction on $|R_v^u|$, the number of nodes in R_v^u. Note that from Lemma 3 we have $m_u = m_v$.

Base Case. $|R_v^u| = 1$, i.e., $R_v^u = \{v\}$. Then $OR_v^u = S_v$ and $O_v = S_v^{uv} + S_v$. Assume by contradiction that there exists an index j s.t. $S_u^{vu}[j] - S_u^{vu}[m_u] > OR_v^u[j] - OR_v^u[m_u] = S_v[j] - S_v[m_u]$. By adding $(S_v^{uv}[j] - S_v^{uv}[m_u])$ on both sides of this inequality, and the fact that $S_u^{uv} = S_v^{uv}$ (since no messages are currently sent) we get that $O_v^{vu}[j] - O_v^{vu}[m_u] > O_v[j] - O_v[m_u]$. Since $m_u = m_v$, this violates the rule on line 19 in v. This would cause v to send a message to u, which contradicts our assumption that no more messages are sent in the system.

Induction Hypothesis. for every node $u \in V$ and $v \in \widehat{\Gamma}(u)$, if $|R_v^u| \leq k$ then for every index $j \in \{1 \dots k\}$, $S_u^{vu}[j] - S_u^{vu}[m_u] \leq OR_v^u[j] - OR_v^u[m_u]$.

Induction Step. we prove the claim for $|R_v^u| = k + 1$. Since G is a tree, i.e., there is only one path between every two nodes in V, we know that $R_v^u = \{v\} \cup \bigcup_{w \in \widehat{\Gamma}(v) \wedge w \neq u} R_w^v$ and thus for each such w, $|R_v^u| > |R_w^v|$. By induction assumption, $S_v^{wv}[j] - S_v^{wv}[m_v] \leq OR_w^v[j] - OR_w^v[m_v]$. Assume by contradiction that there exists an index j s.t. $S_u^{vu}[j] - S_u^{vu}[m_u] > OR_v^u[j] - OR_v^u[m_u]$.

$$O_v[j] - O_v[m_u] =$$
$$= (S_v + \sum_{w \in \widehat{\Gamma}(v)} S_v^{wv})[j] - (S_v + \sum_{w \in \widehat{\Gamma}(v)} S_v^{wv})[m_u]$$
$$= (S_v[j] - S_v[m_u]) + \sum_{w \in \widehat{\Gamma}(v)} (S_v^{wv}[j] - S_v^{wv}[m_u])$$

By induction hypothesis

$$\leq (S_v[j] - S_v[m_u]) +$$
$$+ \sum_{w \in \widehat{\Gamma}(v) \wedge w \neq u} (OR_w^v[j] - OR_w^v[m_v]) + (S_v^{uv}[j] - S_v^{uv}[m_u])$$
$$= (OR_v^u[j] - OR_v^u[m_u]) + (S_v^{uv}[j] - S_v^{uv}[m_u])$$

By contradictive assumption

$$< (S_v^{uv}[j] - S_v^{uv}[m_u]) + (S_u^{vu}[j] - S_u^{vu}[m_u])$$
$$= O_v^{vu}[j] - O_v^{vu}[m_u]$$

Since $m_u = m_v$, this violates the rule on line 19 in v. This would cause v to send a message to u, which contradicts our assumption that no more messages are sent in the system. □

Theorem 2 (restated) *Observe a suffix of any run of Algorithm 1 over a static communication tree with bidirectional links $G(V, E)$, where each node $u \in V$ has a static input vector S_u. Then (a) the algorithm eventually reaches quiescence, and (b) from this time onward, for every node $u \in V$, $m_u = \operatorname{argmin}(\sum_{w \in V} S_w)$.*

Proof. For the proof of (a) we need to consider the worst possible scenario for the algorithm. Observe any node u. The first step of the algorithm will make u know about its 1-neighborhood, i.e., the input values of all nodes that are directly connected to u. During second step u will update those neighbors that have different choice than u. Thus, it takes at most 2 steps to inform the 1-neighborhood of u about the best decision in this neighborhood. During the next step, the 2-neighborhood (those nodes that are connected to u indirectly through one other node) will learn the decision of the 1-neighborhood. This might trigger the send-condition at those nodes of this neighborhood which disagree with the decision of the 1-neighborhood nodes. It will take 2 more steps until u gets any new information originated by the nodes that were now "triggered". It will take 2 more steps to convey the decision of the 2-neighborhood back to any node of this neighborhood of u. Overall 5 additional steps. It can be similarly shown, that if in the next step someone in the 3-neighborhood of u is triggered since it disagrees with the 2-neighborhood decision, it will take 7 more steps to update all nodes in the 3-neighborhood. If we sum up and see how long it will take to update the $r = \text{Radius}(G)$ neighborhood (the maximal ranked neighborhood for any node), we get that this takes the following number of steps: $2+5+7+9+...+(2(r-1)+1)+(2r+1) = r^2+2r-1$. This is the worst number of steps this algorithm may take.

For the proof of (b), let us add to V a new node f s.t. S_f is the zero vector, and connect f to some single node $u \in V$. Denote the resulting graph by $G(V', E')$. Notice that f does not affect O_u or m_u. After the system reaches quiescence, we get $O_f = S_f + S_f^{uf} = S_f^{uf} = S_u^{uf} = O_u$ (the second and the last equality are true because S_f is the zero vector). For every index j, by Lemma 4, we get that $S_f^{uf}[j] - S_f^{uf}[m_f] \leq OR_u^f[j] - OR_u^f[m_f]$. Since $m_f = \text{argmin}(O_f) = \text{argmin}(S_f^{uf})$, for any index j, $S_f^{uf}[j] - S_f^{uf}[m_f] \geq 0$. Since $R_u^f = V$ and thus $OR_u^f = \sum_{w \in V} S_w$, we get that $(\sum_{w \in V} S_w)[j] - (\sum_{w \in V} S_w)[m_f] \geq 0$.

We now show that for every entry $m'_f < m_f$, $(\sum_{w \in V} V_w)[m'_f] > (\sum_{w \in V} V_w)[m_f]$. Suppose for the purpose of contradiction that for some entry $m'_f < m_f$ it holds that $(\sum_{w \in V} V_w)[m'_f] \leq (\sum_{w \in V} V_w)[m_f]$, i.e., $(\sum_{w \in V} V_w)[m'_f] - (\sum_{w \in V} V_w)[m_f] \leq 0$. By Lemma 4 (as was applied above), this means $S_f^{uf}[m'_f] - S_f^{uf}[m_f] \leq 0$. Since $O_f = S_f^{uf}$, we get $O_f[m'_f] \leq O_f[m_f]$. Since we assumed that $m'_f < m_f$, this contradicts the definition of m_f as $\text{argmin}(O_f)$.

Therefore, m_f is an index corresponding to the minimal entry in $\sum_{w \in V} S_w$, and it is the minimal such index. This proves that $m_f = \text{argmin}(\sum_{w \in V} S_w)$. From Lemma 3 we get that for every node v, $m_v = m_f = \text{argmin}(\sum_{w \in V} S_w)$, which completes the proof of (b). □