Incremental Maintenance of Global Aggregates

Nabeel Ahmed, David Hadaller and Srinivasan Keshav

School of Computer Science

University of Waterloo

200 University Avenue West

Waterloo, Ontario N2L 3G1

{n3ahmed, dthadaller, keshav}@cs.uwaterloo.ca

Technical Report CS-2006-19

June 21, 2006

Incremental Maintenance of Global Aggregates

Abstract

Providing local access to global information can improve the efficiency of many distributed applications. Examples include applications that aggregate sensor values, search in Peer-to-Peer systems, or perform Top-K queries in stream-oriented databases. Efficient computation of such aggregates is difficult due to the massive scale and dynamics of such systems and has led to the proposal of several approximate techniques based on randomized gossip algorithms [1], [2], [3]. However, *maintenance* of such aggregates has not been adequately addressed in the literature. Changes in node state, therefore, require a full and expensive re-computation of the global aggregate. This paper makes three contributions to this field. First, we propose a variant of the well-known FM aggregation scheme [4] that allows us to support incremental maintenance of aggregates. Second, we propose the concept of significance thresholds and illustrate their benefits. Finally, we present a detailed performance evaluation of our techniques and find that we can reduce computation time by 60% compared to recomputing the aggregate, as is traditionally done.

Index Terms

Distributed Systems, Peer-to-Peer Systems, Aggregate Computation, Gossip Protocols, Incremental Algorithms, Probabilistic Counting Algorithms

I. Introduction

Large, decentralized, and self-organizing networks will soon be commonplace. In such systems, there is a need to have local access to global information by means of computing a global aggregate over the entire system [5]. For example:

- In a sensor network, one may want to compute the average temperature over all sensors; the minimum or maximum temperature; or quantile values, such as the median temperature.
- In Peer-to-Peer (P2P) systems, choosing how to search for a document in a file-sharing system [6], choosing when to replicate a document in a replicated file system, and performing ranking in a full-text search system are examples of operations that can make use of global aggregates. For instance, PlanetP [7] is an example of a P2P text search system which uses a gossip-based protocol to collect ranking statistics such as document word frequencies.
- In a distributed file system such as the Google File System [8], having local access to the average system load can be useful for load balancing purposes. We elaborate on this idea further in Section IV.

Maintaining aggregates can be done in one of three ways.

1) *One-shot*: The global aggregate is computed once and is never updated. This approach is acceptable in systems where the aggregate doesn't change (e.g. the count of the number of nodes in a stable system).
2) *Drop-and-recompute*: The global aggregate is computed, and it is periodically erased and re-computed. This *refresh* procedure maintains a more accurate aggregate as values in the system change. For example, if

average temperature has been computed, and the temperature value changes at some nodes, the aggregate will be dropped and a new average temperature computed.

3) *Incremental*: Rather than performing a *refresh* as is done in drop-and-recompute schemes, the aggregate is incrementally maintained by sending update messages when a change occurs. This is what we propose in our work.

In contrast, if the rate of change of the system is extreme, the value of the global aggregate will be constantly fluctuating. Attempting to maintain an accurate aggregate in such a system is infeasible, regardless of the technique used, as the amount of message overhead required to accommodate such fluctuations would overwhelm the network. Therefore, our work only considers systems which experience a moderate amount of change.

In order to use incremental techniques to maintain a global aggregate, two issues must be addressed.

1) *Representing a Change*: An update must be represented in a compact and updateable way. We address this by extending the compact FM aggregation scheme [4] to support updates.
2) *How to Propagate a Change*: When a change occurs, it must be efficiently disseminated throughout the system. To address this, we use incremental routing protocols.

Two key ideas form the basis of our approach to efficiently maintaining aggregates. First, suppose some global aggregate has been computed and a change occurs at a node. It is necessary to determine whether the change will have a significant impact on the global aggregate. Significance is a system-specific attribute, and as such we assume the system designer can judge how accurately she wishes to maintain the global aggregate, specified using a threshold α. This threshold represents the maximum allowable error in the global aggregate. Therefore, if a local change will not change the global aggregate by at least α, no maintenance is required.

Second, we address what should be done in the event of a significant change at some node. A naive approach is to drop and re-compute the aggregate from scratch, which is inefficient for a system with only a few changes. We devise an update-specific incremental algorithm to efficiently maintain the global aggregate.

The three main contributions of our work are, (a) the use of a significance threshold to eliminate unnecessary message overhead in the event of small fluctuations in a node's value, (b) the design of an incremental aggregation scheme, allowing efficient update propagation, and (c) a comprehensive evaluation of the benefits of our approach.

We first present background and related work in Section II. We then describe our incremental techniques in Section III, followed by an example case scenario for our methods in Section IV. Finally, we present a detailed evaluation of the performance gains of using our approach in Sections V and VI. We conclude the paper with a discussion and some directions for future work.

II. Background and Related Work

In Section II-A, we present a formal problem statement. In Section II-B, we present previous work that has motivated the need for maintaining global aggregates. Section II-C presents routing techniques that we use for our incremental algorithm. Section II-D describes different aggregation techniques, followed by a formal presentation

of FM aggregation in Section II-E. Finally, Section II-F, briefly describes other example aggregates that we can also compute using FM.

A. Model

Consider a distributed system with N nodes where, at time t, the i^{th} node has local information s_i^t. In many problems of interest, N is very large and nodes arrive and depart over time. Moreover, communications between nodes may be lost, and nodes' values may change over time.

Our goal is to have the nodes self-organize to compute a function $f(S^t)$ where S^t is a *global multi-set* defined as $S^t = \{s_1^t, s_2^t, ..., s_i^t\}$. The function f is any aggregation query (such as sum or $average$), and the value of the resulting function $f(S^t)$ is the desired value to be known at every node.

The value of $f(S^t)$ may change over time as the local set at one or more nodes changes. Suppose the local set at node k changes. Then we wish to compute the function $f(S^{t'})$, at time t', where $S^{t'} = \{s_1^t, s_2^t, ..., s_k^{t'}, ..., s_i^t\}$. Since the values $s_1^t, s_2^t, ..., s_i^t$ have not changed in the interval t to t', the new aggregate $f(S^{t'})$ need not obtain new values from these unchanged nodes at time t'. Our work focuses on how to incrementally determine $f(S^{t'})$ based on $f(S^t)$, for systems that only experience data changes. We later discuss more elaborate change models as well.

B. Global Aggregate Maintenance

Computing global aggregates is very similar to maintaining global consistency in a traditional distributed system where protocols such as the Chandy-Lamport distributed snapshot protocol [9] have been used. However, such protocols are limited to small-scale distributed environments and do not scale well to the types of systems found today (e.g. peer-to-peer systems, sensor databases). As a result, gossip-based protocols have emerged as a lightweight and robust mechanism for computing such aggregates.

Gossip-based (or epidemic) protocols perform exceptionally well for computing global aggregates [1], [2], [3], [10]. They work as follows: At every time step of protocol execution, each node that has something to send selects one or more nodes to send its data. The dynamics of how information spreads through the network resembles the spread of an epidemic [11], which provides increased fault-tolerance [12] and resilience to failures.

Gossip-based protocols have been shown to provide time complexities of the order of $O(\log N)$ [10]. Recent work [1], [2], [10] has also developed sophisticated techniques that reduce the state space of these protocols to $O(\log N)$. These protocols are, therefore, scalable as well as robust [13]. There has also been an interest in addressing the effect of network *churn* on the performance of these protocols. Although some theoretical work provides encouraging results, it does not consider the use of incremental updates. In particular, Jelasity et al. [3] employ an automatic restart mechanism where they restart the protocol periodically by dropping the current estimate of the global aggregate and re-running the protocol from scratch. This is not necessary if the changes in the global aggregate are minimal. This motivates the need to build more efficient mechanisms to support updates. Our work provides a first step in this direction.

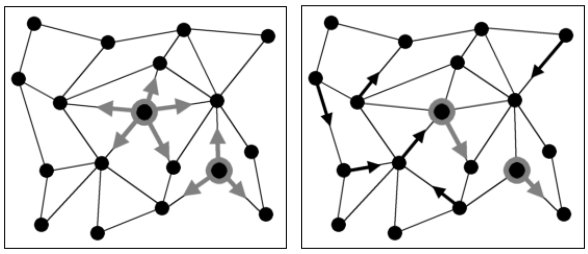

Fig. 1. The left diagram illustrates the IRWP protocol. The right diagram illustrates the standard random walk protocol. The gray arrows represent updates propagating through the network.

In order to use gossip protocols for global aggregate computation, it is important to understand the components which comprise the protocol. We present a logical decoupling of routing and aggregation for this purpose, similar to that proposed by Nath et al. [1].

C. Routing Mechanisms

We discuss three techniques for gossip-style information propagation that are used in the drop and re-compute schemes, followed by their incremental variations which we use in our work.

- **Flooding**: With flooding, each node begins by sending its local value s_i^t to all of its neighbours. The neighbours combine the received value with their local value and send the result (a subset of S^t) to all of their neighbours. This process repeats until each node has obtained the complete set S^t, which incorporates every node's value. Flooding provides optimal convergence but comes at the cost of excessive message overhead. Incidentally, this is also the mechanism used for maintaining routing tables in the widely used OSPF link state routing protocol [14].
- **Uniform Gossip**: In uniform gossip, during each timestep, each node selects one neighbour uniformly at random to send its information (a subset of S^t). This process repeats and the complete set S^t eventually propagates to the entire network. Uniform gossip protocols have been shown to provide exponentially fast convergence with low message transmission overhead [10]. The operation of this protocol is analogous to the spread of a *simple* epidemic, as discussed in [11].
- **Random Walk**: In random walk, the protocol is initiated by a node sending its local value (a *walk*) s_i^t to a random neighbour. The neighbour combines the received value with its local value and sends the result (a subset of S^t) to another random neighbour. This process repeats until every node has received the complete set S^t. k random walks can also be used simultaneously, where each random walk is initiated at a random node in the system. The value *k* acts as a tuning parameter that provides a tradeoff between convergence time and message overhead.
- **Incremental Flooding Protocol (IFP)**: Traditional flooding works by having every node flood its data to the

entire network. In incremental flooding, when an update occurs at a node, only that node floods an update message to the network.

- **Incremental Gossip Protocol (IGP)**: In incremental gossip, when an update occurs, in the first timestep, only the updated node is involved in the gossip procedure. Other nodes only begin gossiping if they receive the update. Therefore, nodes receiving the update become *active* and continue communicating with their neighbours until the update protocol terminates. The operation of this protocol is analogous to the spread of a *complex* epidemic, as discussed in [11].
- **Incremental Random Walk Protocol (IRWP)**: In incremental random walk, when an update (or updates) occur in the system, instead of starting random walks at random nodes in the network, all *k* random walks are initiated from the updated node(s), as is shown in Figure 1. The rest of the protocol then proceeds in the same fashion as the standard random walk protocol.

D. Aggregation Mechanisms

Aggregation refers to the process where nodes in a distributed system combine local node values to form a global aggregate. A problem encountered when computing aggregates is *double-counting*, where nodes may contribute to an aggregate more than once, causing inaccuracy in the final result. In order to avoid such problems, Order and Duplicate Insensitive (ODI) mechanisms can be used, as proposed by Nath et al. [1]. Alternatively, the aggregate could also carry a record of every contributing node. However, this does not scale well.

Nath et al. [1] achieve order and duplicate insensitivity through the use of the approximate FM counting algorithm, pioneered by Flajolet and Martin [4]. Other ODI techniques include a push synopsis approach proposed by Kempe et al. [10]. Due to the extremely compact nature of FM, we adopt this mechanism for maintaining and updating aggregates.

E. FM Aggregates

We now present a formal discussion of FM aggregates. We later discuss an extension to this framework to support updates to such aggregates in Section III-A.

Consider a multi-set $S^t = \{s_1^t, s_2^t, ..., s_N^t\}$ with N elements, as described in Section II-A. We would like to compute $f(S^t)$, where f is the *count* function. In FM, the multi-set S^t is represented as a bit vector of length k (typically $k = \frac{3}{2}\log_2(N)$). The value k is a system-dependent tuning parameter which should be chosen large enough to accommodate the effects of network churn. Initially the bit vector (denoted s_i^t) is set to zero. The aggregate FM bit vector (representing $f(S^t)$) is then generated as follows. Let $cointoss()$ denote a toss of a fair coin that returns either $heads$ or $tails$ with equal probability.

Algorithm 1 represents a single coin toss experiment that is performed at each node. Each node flips a coin until it either obtains a head or completes k coin tosses. The bit vector s_i^t maintained at the node is then updated by setting its *i*'th bit corresponding to the number of consecutive tails observed in the coin toss experiment. The

Algorithm 1 CTExperiment(s_i^t,k)

```
j = 0;
while cointoss() == 'tails' && j < k do
    j = j + 1;
end while
if j == k then
    j = j - 1;
end if
s_i^t[j] = 1;
```

Algorithm 2 FM Counting Algorithm

```
S^t = 0;
for i = 1 to N do
    CTExperiment(s_i^t,k)
    S^t = S^t ⋃ s_i^t;
end for
```

intuition is that as more nodes do coin toss experiments, the probability that a more significant bit at one of the nodes will be set is higher. The resulting bit vectors are then bitwise OR'ed to generate the aggregate bit vector S^t, as is shown in Algorithm 2. This occurs during the routing process, where nodes exchange subsets of S^t, and union the received subsets until every node has the complete set S^t. An algorithm for this process is shown for our update techniques in Section III. S^t approximates the count of the nodes, where the approximate count value is given as $2^{j-1}/0.77351$, where j represents the bit position of the least significant zero in S^t. Functions other than count can also be computed using variants of this basic counting algorithm and are discussed in Section II-F.2. The FM counting algorithm described above has some interesting properties that allows the algorithm to be deployed in a distributed fashion. In particular, it has:

- **Constant Insertion Time**: Each element (or contribution) of the multi-set can be inserted into the vector in $O(1)$ time.
- **Set Unions**: The union of FM bit vectors representing two multi-sets S_1 and S_2 is simply the bitwise OR of their respective bit vectors. Therefore, no ordering or other expensive operations are required.
- **Duplicate Insensitivity**: FM bit vectors are duplicate insensitive. Therefore, applying a local value more than once to the bit vectors does not affect the correctness of the resulting aggregate.

These features allow us to compute an aggregate in a distributed fashion by having each node maintain an individual bit vector and propagate it to other nodes. Nodes union received bit vectors with their local bit vectors to obtain a combined aggregate. Using this approach, and given that bit vectors are duplicate insensitive, we can

use any underlying transport mechanism in order to disseminate this information. Note, although FM aggregates are compact, they are inaccurate because they can only approximate values to the closest power of 2, causing errors of up to 50%.

More accurate aggregates can be computed by maintaining multiple bit vectors for each of the items and taking the average of the results obtained from each [4]. This method decreases the standard error to within $O(1/\sqrt{m})$, where m is the number of bit vectors. This, however, can increase the state maintenance overhead for each node. Optimizations to reduce this state maintenance overhead have also been proposed, such as the Probabilistic Counting with Stochastic Averaging (PCSA) technique [4].

F. Computing Other Aggregates

The FM technique can be used to represent a variety of functions, as is described next.

1) Computing Sum: Let s_i^t in S^t have the value v. To compute *sum*, a node performs v coin toss experiments and unions the bits to produce the resulting bit vector for that value. The other nodes do the same for each of their items in S^t. We obtain the aggregate vector (called $sum(S^t)$) representing the global sum of the items in S^t by ORing these individual bit vectors. This approach is similar to that discussed by Considine et al. [2].

2) Computing Average and Standard Deviation: To compute $average$, we simply divide the *sum* of all the values in the system by the *count* of the number of nodes. *Standard deviation* can also be computed by maintaining both the sum and the sum of the squares of the items s_i^t locally at each node. We elaborate this idea further in Section III-A when we discuss how to incrementally maintain the standard deviation aggregate. Moreover, these aggregates can be extended to compute other functions such as non-centralized moments, as is discussed in [1], [2].

III. Incremental Algorithms for Global Aggregate Maintenance

We now present incremental techniques for maintaining up-to-date global aggregates. We first outline some goals that incremental algorithms should satisfy. We then present enhancements to the FM representation to allow updates to already computed aggregates. Finally, we present an incremental algorithm that we propose for supporting such updates.

Ideally, incremental algorithms should have:

- **Fast Convergence** The algorithm must provide convergence times comparable to or better than existing methods (e.g. $O(logN)$ for gossip-based protocols).
- **Cost-Effectiveness** The algorithm must provide significant cost advantages (e.g. in terms of computation time, communication cost) over existing methods.
- **Fault-Tolerance** The algorithm must exhibit fault-tolerance characteristics comparable to that of existing solutions.
- **Scalability** The algorithm must provide similar guarantees (in terms of performance and cost) on both small as well as large-scale systems.

A. Incremental FM Aggregates

We show how changes at a node can be reflected in a previously computed FM aggregate. We assume that our function f is the *sum* over the multi-set S^t, at time t. We later illustrate how this can be extended to incrementally maintain the standard deviation aggregate.

Suppose that the value of any given item s_i^t in the multi-set changes at time t', to $s_i^{t'}$. How do we incorporate the new value into the already computed $sum(S^t)$? Suppose that the initial value (s_i^t) sets the first 3 bits of the bit vector, which are then unioned with the bit vector representing the aggregate $sum(S^t)$. Later, when the value changes to $s_i^{t'}$, it may not be possible to determine which bits s_i^t originally contributed to the vector. Moreover, even if it were possible, since the first 3 bits may possibly have been set by one or more other items in S^t, these cannot be legitimately reset for $s_i^{t'}$. Therefore, intuitively, it seems that for this representation we may need to drop and recompute the entire sum again. However, as we describe now, this is not needed.

Two cases need to be considered for the new value $s_i^{t'}$:

- $s_i^{t'} > s_i^t$: In this case, we can simply perform $(s_i^{t'} - s_i^t)$ more coin toss experiments in order to incorporate the new larger value into the aggregate $sum(S^t)$.
- $s_i^{t'} < s_i^t$: In this case, we construct another multi-set $D^{t'}$ that is defined as follows:

$$D^{t'} = \{d | d = s_i^t - s_i^{t'}\}$$

  Using this multi-set $D^{t'}$ (referred to as a *delete vector*), we maintain the difference $s_i^t - s_i^{t'}$ for multiple updated items in S^t. Furthermore, we can apply the same sum function for summing up the values of all the items in $D^{t'}$ to generate the aggregate $sum(D^{t'})$. Once both $sum(S^t)$ and $sum(D^{t'})$ are computed, we can apply the FM evaluation function discussed in Section II-E in order to obtain the sum values for both sets S^t and $D^{t'}$. Using these two values, the new aggregate $sum(S^{t'})$ is the sum value of $D^{t'}$ subtracted from the sum value of S^t. In this way, delete vectors can be used to support updates to FM aggregates.

We now illustrate an example of how to incrementally maintain the second moment (i.e. standard deviation) of all the values. Recall, standard deviation can be computed as follows:

$$dev = \sqrt{\frac{N \sum_{i=1}^{N} (s_i^t)^2 - (\sum_{i=1}^{N} s_i^t)^2}{N(N-1)}}$$

The second component of the numerator represents the square of the sum of the values which can be updated using the methods described above. In addition, the first component of the numerator requires each node to maintain the sum of the squares of all the local values (easily done using techniques similar to sum). If a local change occurs, both the sum and the sum of the squares vectors can be updated and propagated in a manner similar to that described above. Therefore, in this way, the standard deviation aggregate may also be incrementally maintained using delete vectors.

Algorithm 3 Update Algorithm

1: /* *Phase: Susceptible* */
2: **if** *node updated* to $s_i^{t'}$ **then**
3: Update local aggregate S^t to $S^{t'}$
4: $\Delta S = S^{t'} - S^t$
5: **if** $\Delta S >= \alpha$ **then**
6: Send update $s_i^{t'}$ to random neighbour
7: Move to *infectious* phase in next round
8: **end if**
9: **end if**
10: **if** received update $s_i^{t'}$ **then**
11: $S^{t'} = S^t \bigcup s_i^{t'}$.
12: $rcvdcounter = 0$
13: Move to *infectious* phase in current round
14: **end if**
15: /* *Phase: Infectious* */
16: **if** (received update $s_i^{t'}$ || *Node can propagate*) && $rcvdcounter \leq \gamma$ **then**
17: Send update $s_i^{t'}$ to random neighbour
18: $rcvdcounter = rcvdcounter + 1$.
19: **else**
20: move to *recovered* phase
21: **end if**
22: /* *Phase: Recovered* */
23: Discard $rcvdcounter$

Although the update techniques described above allow additions/deletions to already computed aggregates, recall that FM suffers from accuracy problems. As a result, over time, as more and more contributions are made to the bit vectors, the vectors become *polluted*, i.e. they can no longer accurately represent the aggregate being maintained. In this situation, the vectors should be dropped and a fresh set recomputed. The decision of when to drop and recompute may be based on a system specific threshold (such as the number of update exchanges between neighbours) or through the use of other mechanisms to identify anomalies in the maintained aggregate. However, the general problem of when and how to initiate dropping and re-computation of the aggregate is an open area of research.

B. Incremental Update Algorithm

Our update algorithm combines the aggregation mechanism described in Section III-A with the incremental routing protocols discussed in Section II-C. The protocol proceeds in the form of *rounds* where, in each round, a

communication occurs between one or more nodes. Each node is assumed to know the length of a round. The time to converge to the correct global aggregate is referred to as an *epoch*.

The update algorithm proceeds in three phases, similar to the way in which epidemics spread in a population [11]. Each node passes through these phases independently for each update/infection introduced in the system. The three phases are:

Susceptible: In the susceptible phase, each node is prone to becoming infected by an update introduced at any of the nodes. A node becomes infectious if it receives an update, or *may* become infectious when a change to its local value occurs. Algorithm 3 illustrates this effect. At the beginning of an epoch (at time t'), when an update occurs at a node, it uses the incremental FM approach (discussed in Section III-A) to generate the new aggregate that incorporates the update. Using the newly generated aggregate $S^{t'}$, the node computes the change in the aggregate by computing $\Delta S = |S^{t'} - S^{t}|$. If ΔS is greater than or equal to a *significance threshold* α, the node propagates the change to its neighbours using an incremental routing protocol. Once this occurs, the updated node enters the infectious phase while other nodes in the system are still susceptible, with respect to the given update. If ΔS is strictly less than α, then the node does nothing as the change is not significant (i.e. it does not become infectious).

Infectious: If an update arrives at a susceptible node, the node unions it with its local aggregate and becomes *Infectious*. For all subsequent rounds before termination, depending on the underlying routing protocol, the node either does or does not continually send updates. For example, in incremental gossip, infectious nodes communicate at every round until termination. However, in incremental random walk, nodes only communicate when they receive an update (i.e. a random walk). Moreover, for aggregation, since FM can represent multiple unique updates in a single bit vector, nodes receiving many single updates will eventually propagate bit vectors that contain multiple updates. This allows faster dissemination of updates in the system and such updates are termed *co-operative*. Cooperative updates have implications on the termination of the algorithm, discussed in the next phase. As the algorithm proceeds, all nodes eventually enter the infectious phase.

Recovered: A node enters the recovered phase when it has ceased to propagate an update any further. Deciding when to terminate the propagation of updates in a decentralized system is a challenging and open problem. This problem is complicated as nodes may receive cooperative updates, making it hard to distinguish between individual update contributions. Therefore, the value of γ, which is the number of rounds that a node remains infectious (or active), is a tuning parameter that needs to be determined based on the characteristics of the system. Once γ rounds have passed, it is highly likely that the node will not receive any more updates that cause changes to the aggregate. Once all nodes enter the recovered phase, the system will have converged to the correct global aggregate.

C. Significance Threshold

The significance threshold (α) presented in the previous section functions as a tuning knob allowing the system designer to trade off the accuracy and cost of sending updates. Threshold values are chosen based on global change significance, which is defined as the significance of a local change to the global aggregate. Threshold values are also based on the query or aggregate that is being processed in the system.

We illustrate this by example. Suppose we are computing sum in a very large network. In such a network, updates that cause changes in the global sum above the significance threshold α should signal the propagation of the updates. In this case, α could be conservatively chosen to be equal to X, the percentage of error that can be tolerated for the aggregate (described in more detail in the next section). On the other hand, if we are computing max, then α may simply be the difference between a node's current value and the current maximum in the system. Thus, nodes can have different values of α and this value may also change every time an update occurs at a node. Determining the global significance of a local change as a function of the query type is difficult. For queries such as sum and max, the choice may be relatively straight forward, however, the general problem is hard. We provide intuition for selecting an appropriate value for the threshold by means of a use case that we describe next.

IV. Example

We present a practical scenario that makes use of incremental aggregate maintenance, and serves to provide intuition on how to select a significance threshold.

Consider a decentralized distributed file system similar to the Google File System [8]. In such a system, there exist areas of the file system that are accessed more often than others. In the case of Google, such *hotspots* would likely correspond to indices that are more popular than others at some point in time. Such access patterns (known as *flash crowds* [15]) result in a concentration of load on a small subset of servers in the file system. We would like to balance this load evenly across the system. One method to alleviate this problem would be to replicate the popular content to neighbouring servers, so that the load can be shared between them. However, in order to do this, the system must first identify the existence of imbalanced load. This can be achieved by computing the average load in the system and comparing that to the load experienced locally. If a node is experiencing greater than average load, then it ought to redistribute its load to a node with less than average load. Once this global average is available at each node, the load distribution algorithm can adopt any of the well-known load balancing techniques [14].

Obtaining such global information in a highly decentralized distributed file system is non-trivial. Gossip-based protocols are well-suited for computing such a global aggregate [16]. However, current gossip-based schemes drop and recompute the aggregate whenever a fresh value is needed. For the system we have described, this is clearly not a wise approach since significant load changes are not likely to occur simultaneously on all servers, precluding the need to drop and recompute the aggregate from scratch. We propose to use incremental techniques for this system, discussed next.

A. Computing Average Load

At system startup, the average load computation can be done using traditional gossip-based techniques. To compute the average, we compute two aggregates: the *sum* of all load values in the system and the *count* of the number of nodes in the system. Once these aggregates are computed, the average is simply *sum* divided by *count*. We now describe how to maintain these aggregates as the load in the system changes.

B. Maintaining Average Load

As the load distribution varies, the average load also changes but its variations are likely to be much smaller than the load variation at any given node. Therefore, because small variations of the average load in the system (such as an increase of 0.1%) won't have a large impact on load balancing decisions, we need only to focus on changes to the average load which are significant (i.e. changes that are greater than the significance threshold). This threshold α can be chosen to be the maximum amount of error in the average load that the system can tolerate. For example if α was 1%, then any changes in local load that cause a change of less than 1% in the average load, need not be communicated by a node, reducing message overhead. Determining global significance in a decentralized fashion is addressed next.

C. Determining Significance Locally

Determining the significance of a local change on the global aggregate is challenging in a decentralized environment. Clearly, if all nodes in the network change at approximately the same time, and none of the nodes communicate their change, the aggregate could become inaccurate. We address this problem by selecting conservative thresholds that assume the worst-case, that is, all other nodes also change simultaneously. This ensures that the actual error threshold will never be exceeded.

In order to add structure to the problem, we first consider the case where all nodes are experiencing the same load (i.e. the load is perfectly balanced). In this case, N nodes are equally loaded with L load, and we are only interested in load changes which cause the global average to change by $X\%$. The global average in this case is simply L. If the load at every node increases locally by some amount less than $X\%$, then the global average load is guaranteed not to have increased by more than $X\%$. Therefore, any local change that is within $\alpha = X\%$ of the current average load does not need to be communicated.

We now consider the unbalanced case. In order to compute the local significance of a change, the node computes the impact its change has on the global average. If a system with N nodes can tolerate no more than $X\%$ error, and assuming the worst case where all nodes change simultaneously, each node's local change will be significant if it impacts the global aggregate by more than $X\%/N$. The intuition is that if any node's change causes a change of less than $X\%/N$, and if all other nodes cause a change of $X\%/N$ (in the worst case), the total resulting change is guaranteed to be less than $X\%$. Although this conservative approach will require more communication overhead than necessary, it ensures that the actual error will never exceed $X\%$. Using this mechanism, global change significance can be approximated locally.

V. Simulation Model

A. Simulation Setup

We have implemented a custom simulator in C. As indicated in Section III-C, because the significance threshold is influenced by system-specific parameters such as system dynamics, available capacity, and the query type being computed, we do not model its effect in our simulations. We use the sum query for our simulations and only model

Component	Accuracy	Protocol Cost	Delay
Incremental Routing	*Does Not Affect*	*Affects*	*Affects*
Incremental Aggregation	*Affects*	*Affects*	*Does Not Affect*

TABLE I

EFFECT OF PROTOCOL COMPONENTS ON PERFORMANCE METRICS

data changes (not node/link failures) in the system. All nodes in the system maintain identical configurations and are equal in their capabilities. Unless otherwise indicated, the topology considered for our simulations is a clique. We present results of a sensitivity analysis on a more realistic topology in Section VI-E.

1) Assumptions: We make the following assumptions:

- **Uniform Data Distribution:** We assume data changes are uniformly distributed across the entire system, i.e. each node is equally likely to update its local state. This represents the worst case for our algorithm since it reduces the likelihood of cooperative updates (i.e. a single aggregate carrying multiple updates) from occurring in the system.
- **Synchronized Operation:** All nodes are assumed to have a synchronized time base, allowing them to identify the start and end of a round. We make this assumption for ease of simulation.
- **Centralized Control:** We assume centralized control for the initiation and termination of the update algorithm. Therefore, once the simulator starts the algorithm, it stops the algorithm when it detects that all nodes have converged to the same aggregate.

2) Simulation Metrics: We now briefly list the performance metrics used in our simulations. Each component of the incremental update algorithm affects multiple metrics, as shown in Table 1. These are briefly discussed further:

Accuracy. We use the mean error in the local estimate of the global aggregate to measure the accuracy of each of the aggregate maintenance schemes. The mean error is defined as $e_t = \frac{1}{N}\frac{1}{S^t}\sum_{i=1}^{N}|\hat{S}_i^t - S^t|$, where $\hat{S}_i^t$ is the local estimate of the aggregate at node i, S^t is the true value of the aggregate, and *N* is the total number of nodes in the network.

Protocol Cost. The cost of a protocol has three components: computation time (i.e. time to perform the coin toss experiments), communication time, and state maintenance overhead. In this study, we only consider the communication time and state maintenance overhead. Communication time consists of the message transmission overhead (or number of message transmissions) and message size. Since FM aggregates use constant size bit vectors, the state maintenance overhead and message size are constant. Therefore, we only analyze the message transmission overhead.

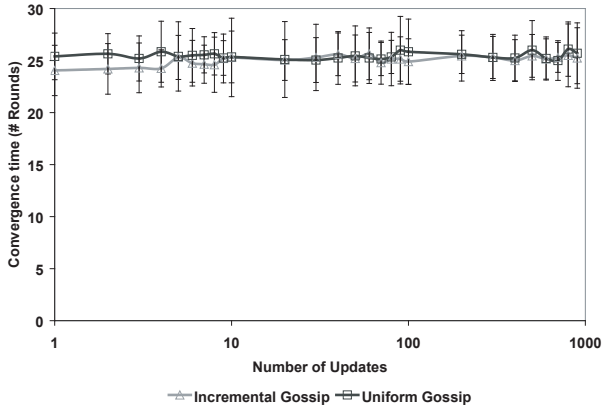

Fig. 2. Comparison of convergence times of IGP and Uniform Gossip protocols.

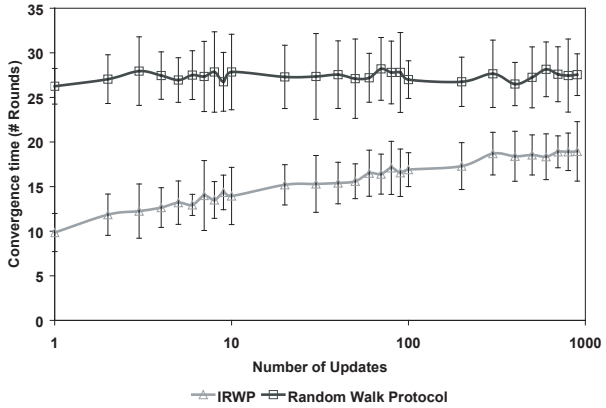

Fig. 3. Comparison of convergence times of IRWP and Random Walk protocols.

Delay. The delay metric is defined as the number of rounds taken to converge to the approximate global aggregate from the time the update(s) occurred. We also refer to this as the *convergence time* of the update algorithm.

In our simulations, we report the average measure over 500 epochs. We begin with a *stable* system, i.e. where the global aggregate has converged. During the update process, we introduce updates uniformly at random over the entire network. We experiment with 10,000 nodes in order to gauge the scalability of our techniques; for the random walk and incremental random walk protocols, we use as many random walks as there are nodes in the system. We show later in Section VI-D that this does not significantly increase the overhead of the protocol. Also, for IRWP, we uniformly divide the number of random walks across the updates to allow an equal allocation of resources to each update. Other allocation techniques are also possible and are left for future work. Finally, in order to allow for greater accuracy in computing the aggregate, we use fifty 32-bit vectors per node ($\approx 0.2k$ bytes) to store the aggregate.

VI. Simulation Results

We first present results comparing the performance of our incremental update techniques with existing drop and re-compute schemes. Second, we compare the different types of incremental update mechanisms against each other. Third, we present results quantifying the accuracy of incremental FM aggregation, followed by an analysis of the effect of increasing the number of random walks (RWs) on the incremental random walk protocol. Finally, we end

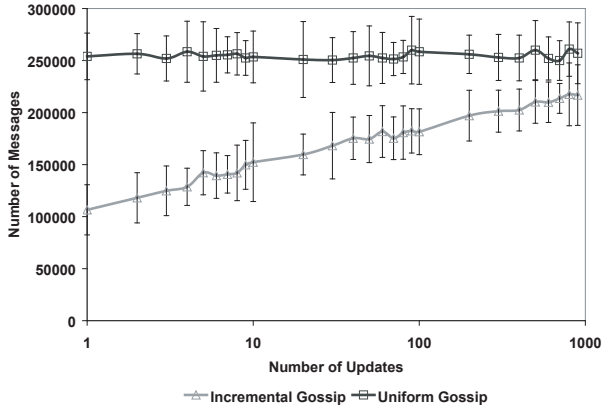

Fig. 4. Comparison of message transmission overhead of IGP and Uniform Gossip protocols.

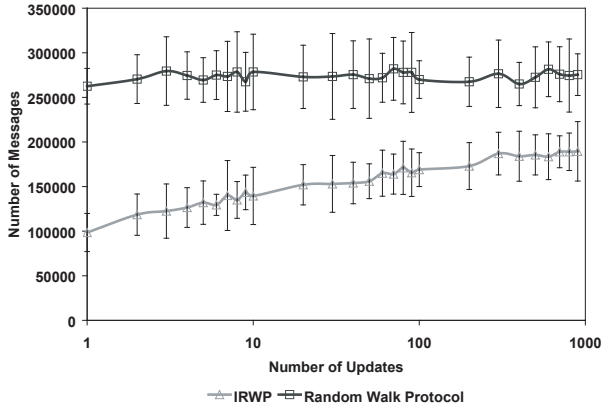

Fig. 5. Comparison of message transmission overhead of IRWP and Random Walk protocols.

with a sensitivity analysis of our methods.

A. Incremental Updates vs. Drop and Re-compute Schemes

We first analyze factors that affect the convergence time and cost of our update techniques.

- *Convergence Time:* Figure 2 compares the performance of the incremental gossip protocol (IGP) with uniform gossip. We observe that the convergence times of both protocols are nearly identical with varying numbers of updates. The update propagation technique in IGP models the spread of a complex epidemic that takes $O(logN)$ rounds to converge, which is the same convergence time as uniform gossip (i.e. a simple epidemic). Therefore, IGP does not provide any improvement in convergence time. Figure 3 compares the convergence times of incremental random walk (IRWP) and standard random walk. IRWP reduces mean convergence time by as much as 60% for a small number of updates. Although random walk and IRWP use the same number of random walks, IRWP concentrates all its walks only on updated nodes, providing more resources per update and thereby allowing faster convergence. For random walk, because the aggregate is dropped and re-computed, every node originates a random walk whether or not it has an update, which reduces the number of resources per update.
- *Protocol Cost:* Figure 4 compares the message transmission overhead of incremental gossip with uniform gossip. As the number of nodes with updates increases, the cost of the two protocols converge. For a small

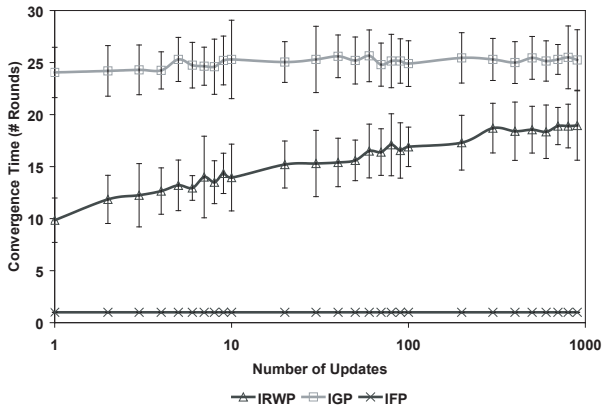

Fig. 6. Comparison of convergence times of incremental routing protocols.

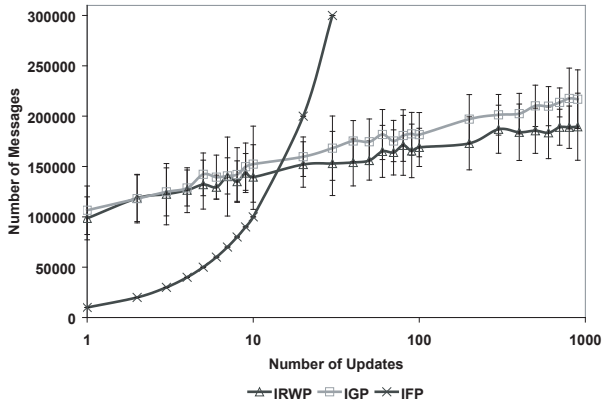

Fig. 7. Comparison of message transmission overhead of incremental routing protocols.

number of updates, we see an almost 60% decrease in the number of messages required for convergence. Because IGP requires only updated nodes to gossip in earlier rounds whereas all nodes communicate in uniform gossip (due to a drop and re-computation), the total message overhead for IGP is substantially lower. However, as the number of updates increases, more nodes communicate in earlier rounds, which causes an increase in the number of messages, resulting in a smaller performance gap between uniform gossip and IGP. When we compare incremental random walk with standard random walk in Figure 5, we see similar trends. The reason why IRWP uses fewer messages than random walk is because it takes fewer rounds to converge. Because IRWP and random walk use the same number of walks in each round, IRWP incurs a lower communication cost when compared to random walk.

B. Comparison of Incremental Routing Protocols

We now compare the relative performance of the incremental routing protocols (Figures 6 and 7). Note that the number of messages per round is different for each protocol. For illustrative purposes, we also present results for incremental flood (IFP).

- *Convergence Time:* Figure 6 compares the convergence times of the incremental routing protocols. We see that IFP converges the fastest because it simply floods the update in the first round to all nodes in the clique. IRWP converges faster than IGP because IRWP performs more transmissions in earlier rounds than IGP.

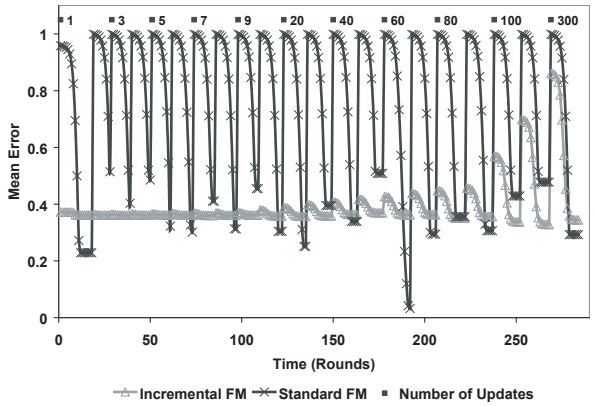

Fig. 8. Accuracy of standard FM and incremental FM as updates are made in the system; the points at the top of the graph indicate the number of updates introduced into the system at that point in time. Standard FM drops and recomputes the aggregate when updates are introduced.

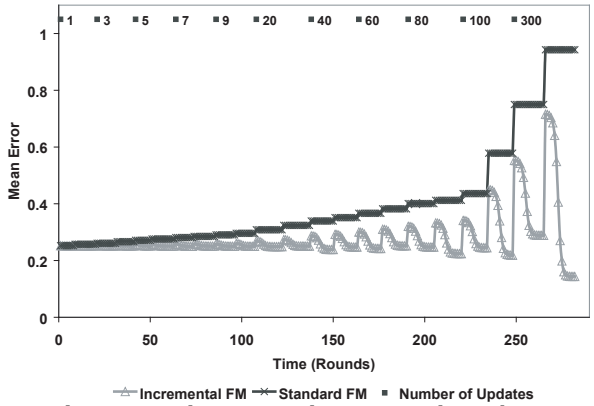

Fig. 9. Accuracy of standard FM and incremental FM as updates are made in the system; standard FM does not drop the aggregate in this case.

However, this effect is reduced as more updates are applied to the system because more nodes in IGP also begin communicating in earlier rounds.

- *Protocol Cost:* Figure 7 compares the message transmission overhead of the incremental routing protocols. Incremental flood performs the worst since the number of update messages increases linearly with an increasing number of updates. Surprisingly, we observe that IRWP and IGP have approximately the same message transmission cost despite their differing convergence times. This is because although IGP takes longer to converge, it uses fewer messages per round, resulting in roughly the same total message overhead as IRWP.

The main conclusion we draw from these results is that although IRWP and IGP behave differently, each protocol provides a tradeoff between the number of messages per round and convergence time; with both having the same overall message overhead. In Section VII, we discuss the ramifications of choosing one protocol over the other.

C. Accuracy of Incremental FM

Figure 8 compares the accuracy of incremental FM aggregation with standard FM aggregation, where the aggregate is dropped and recomputed each time an update occurs. Updates are injected into the system as soon as it converges

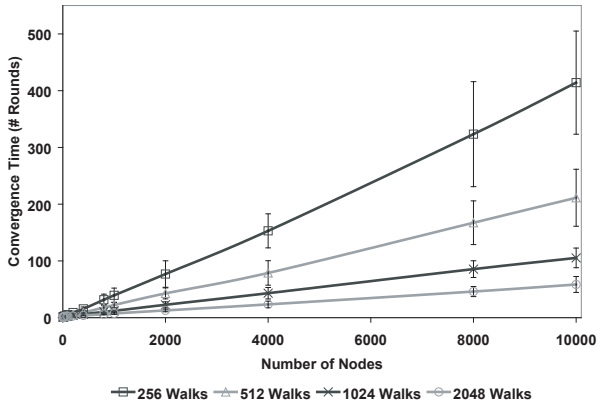

Fig. 10. Analysis of using different amounts of walks on IRWP convergence time.

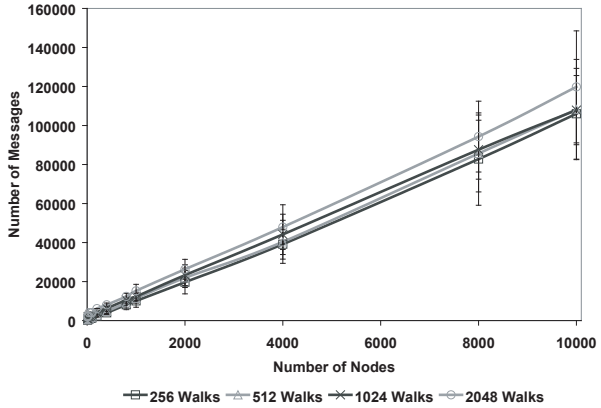

Fig. 11. Analysis of using different amounts of walks on IRWP message transmission overhead.

to the global aggregate. We also progressively increase the number of updates injected each time. The figure shows the mean error in the aggregate across all nodes in the system. The points at the top of the graph indicate the number of updates that were introduced into the system at specified points in time. Whenever an update occurs, standard FM immediately drops the aggregate at all nodes and performs a re-computation, thus resulting in a large average error in the aggregate (assuming the old aggregate is not maintained while the new aggregate is being computed). In other words, the naive drop and recompute approach is almost always in error. Because Incremental FM does not drop the aggregate, the mean error is significantly lower. Therefore, for a small number of updates, incremental FM maintains a high level of accuracy.

Instead of dropping the aggregate immediately, a more clever non-incremental algorithm could choose to drop and re-compute after a larger interval of time to avoid excessive oscillations. However, not reacting to updates reduces accuracy, as shown in Figure 9. Here we show the accuracy over time for standard FM that does not drop and re-compute the aggregate. We see the inaccuracy of standard FM increasing over time. Also note that the number of updates applied in the simulation constitutes a change to only approximately 11% of the nodes in the system. Even with such a small degree of change, the accumulated error reaches almost 100%. This demonstrates the advantages of using incremental techniques rather than standard drop-and-recompute schemes.

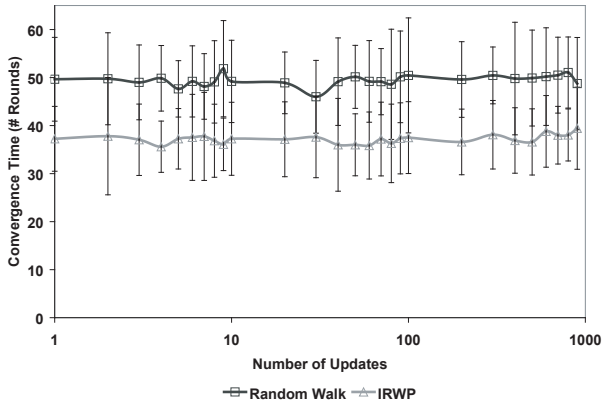

Fig. 12. Comparison of IRWP and Random Walk convergence times on a PLRG topology.

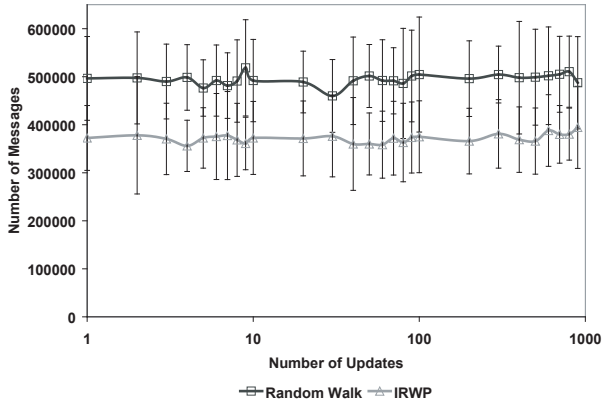

Fig. 13. Comparison of IRWP and Random Walk message transmission overhead on a PLRG topology.

D. Effect of Multiple Random Walks

- *Convergence Time:* Figure 10 illustrates the effect of increasing the number of random walks for IRWP on its convergence time. We observe that doubling the number of walks results in approximately a 50% decrease in the convergence time. This decrease eventually diminishes as the number of walks are increased, since the system eventually becomes saturated.
- *Protocol Cost:* Figure 11 illustrates the effect of increasing the number of random walks on message transmission overhead for IRWP. Unexpectedly, the message transmission overhead is nearly independent of the number of random walks. We attribute this to the *cooperative* nature of the updates. Because each walk carries multiple updates, increasing the number of random walks causes a decrease in convergence time. Therefore, the message transmission overhead does not change. This is the reason we chose to use a large number of walks for our simulations.

E. Effect of Complex Topologies

We now present a sensitivity analysis for our incremental routing protocols. For this purpose, we used a BRITE-based [17] power-law random graph (PLRG) with 10,000 nodes. An example PLRG for a 100 node system is shown in Figure 14. Due to space limitations, we only present results comparing the random walk and IRWP protocols. Results comparing uniform gossip and IGP follow similar trends.

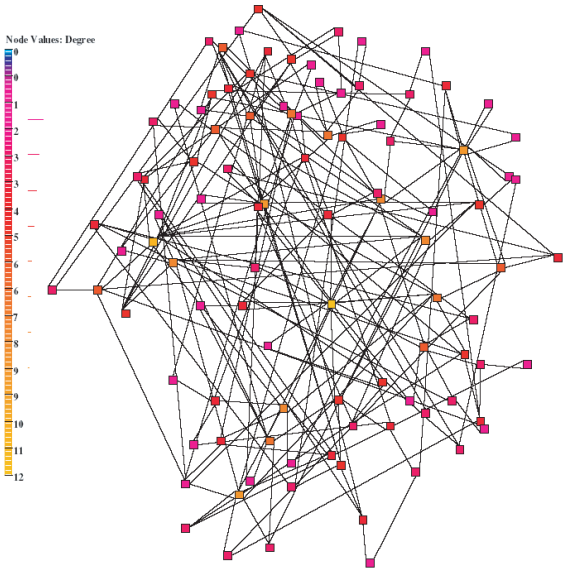

Fig. 14. Sample 100 node PLRG topology, generated using BRITE. The line graph on the left represents the frequency of node degrees.

Figures 12 and 13 present a comparison of the convergence times and message transmission overheads for IRWP and standard random walk. We see that the results closely match those of the clique topology. Although the differences between random walk and IRWP are more noticeable for the clique, even for the PLRG topology, IRWP performs significantly better both in terms of convergence time and message overhead than standard random walk. This indicates that the update mechanism is relatively insensitive to the underlying topology and can potentially be applied to arbitrary topologies.

VII. Discussion

Computing aggregates in a distributed setting is a challenging yet necessary requirement for many systems [7], [8], [16]. Recent literature on gossip protocols provides useful theoretical properties for such protocols, highlighting their suitability for computing aggregates in decentralized distributed systems. However, most of this literature lacks a comprehensive examination of the performance of these protocols in a simulation or experimental setting. Our work is a first step towards understanding the behaviour of gossip protocols by means of simulation. We also examine ways to improve gossip in such scenarios. In particular, we examine opportunities for performing incremental updates and show that it is possible to perform better than standard gossip-based techniques. We present an approach to updating FM aggregates that maintains the compactness of the aggregate. We also discuss the use of thresholds that allows the system designer to tradeoff the accuracy and cost of the update algorithm. Finally, we provide a comprehensive evaluation of the performance of gossip protocols and show that our techniques are able to perform better than existing methods, both in terms of convergence time and cost.

We briefly indicate some insights we gain from our evaluation:

- *Update performance*: Although a wealth of theoretical literature on gossip protocols presents tight upper bounds on the performance of these protocols, we find that from a practical standpoint, we can improve on these techniques further by making these protocols incrementally updateable. Surprisingly, we find that these extensions yield performance benefits of up to 60% when compared to existing techniques.
- *Routing Policy*: In comparing the performance of different routing protocols, we find that incremental versions of random walk and uniform gossip perform similarly in terms of performance and cost. However, we hasten to point out that although both techniques have similar performance, each protocol has its own characteristic set of advantages. Incremental gossip provides a solution that can be easily deployed in a distributed setting, without requiring any pre-allocation of resources (or walks) to updates. However, with random walk, the system designer is provided with the additional degree of flexibility of being able to tune the number of walks based on the number and type of updates (e.g. assigning more walks to more crucial updates). Therefore, we argue that choosing the most suitable routing policy is based on the application scenario being considered and the desired degree of flexibility.
- *Accuracy Improvement*: In terms of accuracy, we make interesting observations for the drop and re-compute schemes. If the system is too reactive to updates, oscillations may occur due to frequent drops. On the other hand, if a node retains the FM aggregate over time, even when a small fraction of the system experiences change ($\approx 11\%$), the aggregate's accuracy rapidly decreases, with the mean error increasing to almost 100%. However, in contrast, our update technique adequately addresses this problem by keeping the amount of error acceptably low.
- *Resource Independence*: Intuition suggests that using more walks in the incremental random walk protocol would increase the cost of the protocol. However, surprisingly, this is not true. This is because the protocol is able to converge faster with more walks, making the cost of the protocol effectively independent of the number of allocated resources. We believe this result will encourage an exploration of random walk to examine its suitability to other types of applications.

VIII. Conclusions and Future Work

Computing global aggregates is an important problem and has been studied extensively in the traditional distributed systems literature [18]. However, this work relies on the stability and reliability found in such systems, which precludes its use on today's large-scale decentralized systems; a domain where gossip protocols have been shown to be more applicable. Our work examines the benefits of using incremental gossip-based techniques for maintaining global aggregates. Our contributions are:

1) We propose a variant of a well-known FM aggregation scheme that allows efficient update propagation.
2) We establish the notion of local significance thresholds and illustrate their benefits.
3) We present a detailed performance evaluation of our techniques, and find that we can achieve a reduction of as much as 60% in computation time compared to existing methods.

There are many avenues to explore in future work. On the theoretical side, we plan to come up with a formal analysis of the accuracy and cost of our methods. We are also developing formal proofs to illustrate the correctness of our techniques. On the practical side, we plan to incorporate a more elaborate change model that also takes into account node and link failures. Additionally, we are exploring methods that can be used to decide when to drop and recompute incremental aggregates. Finally, another set of problems that we also hope to address pertain to the implementation issues of random walk. In particular, given a fixed number of walks, how to allocate resources to each update in a decentralized fashion. Using this framework, we can also come up with other allocations that may be more desirable in different scenarios (e.g. prioritized allocation of walks to updates).

REFERENCES

[1] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson, "Synopsis diffusion for robust aggregation in sensor networks," in *Proceedings of SenSys '04*, November 2004.

[2] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, p. 449.

[3] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Journal*, vol. 15, no. 5, November 2004.

[4] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[5] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Trans. Comput. Syst.*, vol. 21, no. 2, pp. 164–206, 2003.

[6] M. Zaharia and S. Keshav, "Adaptive peer-to-peer search," Submitted for Publication, January 2005.

[7] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen, "PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities," in *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Press, June 2003, pp. 236–246.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 29–43.

[9] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[10] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 2003, p. 482.

[11] N. Bailey, *The Mathematical Theory of Infectious Diseases and its Applications*. Hafner Press, 1975.

[12] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 1, pp. 8–32, 1988.

[13] C. Gkantsidis, M. Mihail, and A. Saberi, "Random walks in peer-to-peer networks," in *Proceedings of INFOCOMM 2004*, March 2004.

[14] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[15] T. Stading, P. Maniatis, and M. Baker, "Peer-to-peer caching schemes to address flash crowds," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 203–213.

[16] B. Wong, A. Slivkins, and E. G. Sirer, "Meridian: a lightweight network location service without virtual coordinates," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 85–96, 2005.

[17] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: an approach to universal toplogy generation," in *Proceedings of MASCOTS*, Aug. 2001.

[18] M. T. Ozsu and P. Valduriez, *Principles of distributed database systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.