Cooperative Data Dissemination in a Serverless Environment

Abheek Anand Sudarshan S. Chawathe
Department of Computer Science, University of Maryland, College Park, USA
{abheek, chaw}@cs.umd.edu

Abstract

We describe the design and implementation of CoDD, a system for cooperative data dissemination in a serverless environment. CoDD allows a set of loosely-coupled, distributed peer nodes to receive subsets of a data stream, by describing interests using subscription queries. CoDD maintains statistical information on the characteristics of data and queries, and uses it to organize nodes in a hierarchical, data-aware topology. Data is disseminated using overlays which are created to try to minimize the excess data flowing through the system, while maintaining low latency and satisfying fanout constraints. CoDD is designed to allow nodes to determine individual degrees of contribution to the system, and to adapt gracefully to temporal changes in the data distribution using an adaptive reorganization component. We present the results of our experimental evaluation of CoDD.

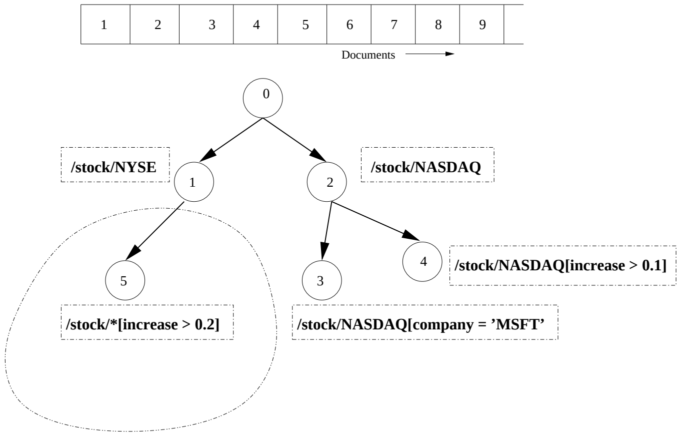
1 Introduction

In data dissemination systems, data is delivered from a set of producers to a large set of subscribers. Domains for such systems include stock tickers, news feeds and multimedia delivery. Most existing systems implement such functionality by maintaining a centralized server node. This node maintains state information and implements protocols to allow subscribers to express their interests and receive data that corresponds to this interest.

These systems achieve the necessary scalability by adding more centralized infrastructure, a process that is often expensive and time-consuming. Moreover, this incremental provisioning of server machines makes this architecture ill-suited for *flash* data streams, which are transient and unpredictable. Examples of such data streams could include data from sensors near an environmental disaster that needs to be disseminated for relief efforts and disaster management to nearby nodes. Such systems make it difficult to pre-provision server nodes, and requires the dissemination to scale organically with the set of subscribers.

This paper describes CoDD, a system for cooperatively disseminating data in a completely decentralized environment. We do not assume an existing infrastructure of server nodes, or any centralized repository of state. We also assume autonomy of participants. Thus each node can enter and leave the system independently, arbitrarily and often frequently. CoDD allows a producer node to generate a stream of data that needs to be disseminated to the set of participating consumers. Each consumer node describes an interest in a subset of the data stream, using a *subscription query* that describes the structure and content of the subset of data this node is interested in. Nodes cooperate to forward data to peer nodes, by contributing resources for receiving, filtering and transmitting data. Each node has varying computational and network resources, and we wish to incorporate these constraints into the design to make it feasible for low-power nodes to participate in CoDD.

There are several challenges to enabling such systems. A naive solution to this problem would be to multicast the data stream to each of the consumer nodes, with end nodes filtering the data that satisfies their requirements. However, this approach would entail a significant communication overhead, especially for participants with low-selectivity queries. The lack of centralized control and knowledge about temporal behavior of the participant sets and document distributions makes it difficult to pre-compute *globally-optimal* configurations. A further challenge is to effectively maintain node-specific constraints, such as limited bandwidth and network resources. The system should also perform well independently of varying data types and query languages, and scale gracefully with high rates of node joins and leaves.



(a) System State

```
<stock>
    <NASDAQ>
        <company>MSFT</company>
        <increase>-0.07</increase>
    </NASDAQ>
</stock>

<stock>
    <NYSE>
        <company>IBM</company>
        <increase>0.13</increase>
    </NYSE>
</stock>
```

(b) Example Data

Figure 1: An Example System

Example 1 Consider the stock-ticker system suggested by Figure 1(a), with documents coming in as a stream to the root node, labeled 0. We assume the dissemination protocol dynamically creates a tree overlay structure, with the source at the root and data flowing down the tree. The data consists of a stream of XML documents, each of which contains some stock quote information. For example, Figure 1(b) depicts two documents describing stock information about Microsoft and IBM on the NASDAQ and NYSE stock exchanges respectively. Each node subscribes to a subset of the data using an XPath subscription query [24]. For example, Node 2 has a subscription query `/stock/NASDAQ`, which matches all quotes for the NASDAQ exchange. Similarly, Node 1 requires all documents containing NYSE quotes. The root node, which is the parent node of 1 and 2, will filter documents needed by each of its children before sending them downstream. Node 4 requires all quotes for Microsoft on the NASDAQ, and Node 5 requires all quotes that have increased by more than 10% on the NASDAQ. Note that each of these queries is *contained* in the query of node 2. Therefore, adding 4 and 5 as children of 2 entails no overhead in terms of the documents sent down that subtree.

A natural constraint in building such a network is a restriction on the number of children each node has. We refer to this restriction as the `Fanout` constraint. It bounds the amount of network and computational overhead incurred at each node. (Since nodes in a P2P network are typically not dedicated servers but also support a number of other functions, such constraints are quite practical and are commonly found in deployed P2P systems.) Let us assume the fanout constraint in this example to be 2. (We pick a low value for illustration purposes; typical values would lie in the range 4-10.)

Consider the addition of Node 3 later on to the system. Node 3 requires all documents with quotes that have shown an increase of 20% or more, irrespective of the stock exchange. Since there are no containment relations for Node 3 with any of the existing nodes, its addition entails some spurious data being added on the network. Furthermore, lack of knowledge of future document distributions and node joins and leaves makes it very difficult to predict the optimal place to add the new node. For example, adding it as a child of node 1 would entail node 1 receiving some documents not required by it, but which need to be sent down to node 3 (eg. documents from the NASDAQ with high increases). Further complexities arise in the system if some node is unwilling to accept new children because of the fanout constraint, in which case the system needs to construct the best-possible configuration. Similarly, if the document distribution becomes very skewed at some point later, the amount of spurious data may exceed previous estimates, and the system would benefit from a reorganization of the topology. ■

CoDD dynamically organizes participating nodes into a hierarchical tree structure topology for the purposes of data dissemination. Data is generated as a stream of documents at the producer node, and flows down the tree. Each node filters incoming data, and transmits only the subset required by its descendants in the tree. The primary metric we aim to optimize on is the *normalized spurious data*, which is the fraction of documents received by a node that does not satisfy its subscription query. The topology construction protocols are the *proactive* component of CoDD, because they optimize on the metrics by trying to predict the future distribution of data.

CoDD also contains a reactive *reorganization* protocol. This component periodically reacts to changes in the distribution of the data stream by trying to reorganize the overlay topology. It allows the system to perform online optimizations, with no a-priori temporal information about data distributions.

There is a large body of work on data dissemination services. Several selective data dissemination services for distributed systems have been proposed [25]. More recently, systems such as Siena [4] and Rebeca [15] have been proposed for enabling fast publish-subscribe systems. These systems assume a pre-existing centralized infrastructure of server nodes, and thus maintain centralized repositories of state. Other related work deals with aggregating subscription queries [7], maintaining index structures for efficient filtering [8], combining common-subexpressions to process multiple queries [22] and developing continuous query systems [9] to enable dissemination. CoDD differs from these systems in that it is designed to be dynamically created using a set of loosely-coupled nodes, while maintaining the desirable properties of these systems such as bounded state, efficient filtering and a form of multiple query optimization to aggregate similar queries. We describe related work in more detail in Section 6.

To the best of our knowledge, CoDD is the first selective data dissemination service for a *serverless* environment. By avoiding the need to setup expensive centralized servers and dedicated infrastructure, we ensure the CoDD is easily deployable, and incrementally upgradeable. The system works by assuming cooperating peer nodes, and is designed to accommodate per-node contribution constraints. The CoDD protocols define policies rather than mechanisms, and are therefore independent of the actual data and query semantics.

The main contributions of this paper may be summarized as follows:

- We motivate and develop the problem of selective dissemination of streaming data in a dynamic, resource-constrained, server-less environment.

- We present protocols for such dissemination. The low overhead of our protocols permits them to be used in scenarios in which the set of hosts, the network characteristics, the subscriptions, and the data characteristics change frequently.

- We describe the results of our performance evaluation of CoDD on synthetic and real data sets, with policies for efficient deployment in a real world system

The remainder of the paper is organized as follows. Section 2 describes some design choices and assumptions made by CoDD. Section 3 describes the protocols for adding and removing nodes from the system. Section 4 describes the overlay reorganization protocols, which allows CoDD to react to changes in data distribution. Section 5 provides a detailed experimental evaluation of the system to demonstrate its effectiveness. We describe related work in Section 6 and conclude in Section 7.

2 CoDD System Design

CoDD contains protocols to dynamically build and maintain an overlay of nodes for data dissemination. The system maintains a hierarchical tree structure of the nodes, with the producer node at the root. The producer produces a stream of documents, and these flow down the tree to the subscriber nodes. Each node maintains an aggregate filter query for each its children, which describes the set of documents needed by all its descendants along that subtree. Aggregating downstream filter queries reduces the amount of routing state at each node, and bounds the computational overhead incurred while filtering. On receiving a document, a node propagates the document to a child only if the document satisfies that child's aggregate filter query.

The **fanout** of a node is the maximum number of children it is willing to accept in the CoDD overlay. This limit allows each node to restrict the computational and network overhead it incurs by bounding the number of filter queries that it needs to evaluate and the number of network connections it needs to maintain.

When a new node joins the network, it is desirable to attach it at a point that produces a small impact on the amount of excess data that must flow through the network. Adding a new node to any point in the network potentially results in excess data flowing along the path from the root to the node. For each ancestor of the new node, this data corresponds to the documents required by the new node that do not satisfy the subscription query of the ancestor. Computing this value requires knowledge of the documents

Term	Definition
`n.children`	The list of children for node `n`
`n.parent`	The parent of `n` in the overlay
`n.ancestors`	The set of ancestors of `n` in the overlay (including itself)
`SubTree(n)`	The topology subtree rooted at `n`
`n.query`	The subscription query for `n`
`n.query(D)`	The subset $d \in D$ that satisfies `n.query`
`n.filter`	The aggregate filter query for `n`
`n.level`	The depth of the node `n` in the overlay tree

Figure 2: Terminology Used

published in the system in the future, and CoDD attempts to estimate this overlap using structures describing information about previously available documents.

We note that the desired maximization of overlap refers to an extensional overlap of data items seen in the past, not to an intensional overlap of queries. Although the latter certainly influences the former (e.g., query containment would ensure high overlap), the relationship may be weak in practice. For example, queries that are not very closely related logically (e.g., `//computer[model = "iMac"]` and `//computer[color = "tangerine"]` may have 100% overlap extensionally because of unknown dependencies (hypothetically, the only tangerine computers are iMacs)). Consequently, our protocols favor extensional techniques over intensional ones (although they use both). Figure 2 describes some of the terminology used in the description.

3 Overlay Construction

3.1 The System Join Protocol

The System Join Protocol defines a distributed procedure for a new node to attach itself to the CoDD system. It tries to produce topologies that minimize both excess data while maintaining low network latency. The protocol is outlined in Figure 3. In our descriptions, use use the notation "`send_sync_msg`" to denote synchronous message passing, and "`send_async_msg`" to denote asynchronous communication. The protocol uses a distributed greedy strategy to determine the insertion point for new nodes. An incoming node, `m`, obtains a seed set `S` of existing nodes using an out-of-band mechanism. Our implementation uses a special `system_agent` node for this purpose. Typically, the seed set `S` would consist of the `ROOT` node, but can consist of any arbitrary set to avoid a single point of congestion. For example, in a wireless network, it could be a random node in communication range of the new node. We define the function `max_overlap_node(Query q, Node n)` as the node $p \in n.children$ that the protocol determines will incur the least overhead upon accepting a node with query q as a child. Further, we denote this node p as the maximal-overlap child of n with respect to the query q.

The algorithm proceeds with the node `m` sending a `TYPE_ADD` message to each node $s \in S$. Each such node s then propagates this message recursively to its maximal-overlap children with respect to $m.query$. The algorithm terminates when the message reaches a node whose fanout constraint is not satisfied, and the new node is attached as a child of this node. Ties in the overlap are resolved by the node `m`. This allows the new node to choose from a set of possible points of entrance to the system, allowing it to make decisions based on network proximity or load balance.

By terminating the protocol when the first node that can accept new children is found, the protocol favors the construction of short trees, resulting in low latency. The early termination also results in low overheads: the number of messages exchanged is bounded by $h \times s$, where h is the maximum depth of the tree, and s is the (constant) size of the seed set. This design may, of course, lead to suboptimal decisions as far as query overlap is concerned. The alternative, of examining potential attachment points more exhaustively, would incur additional overheads. We quantify the effects of this design in Section 5.

3.1.1 Maintaining query overlaps

CoDD approaches the problem of determining maximal query overlaps using two types of schemes, a *statistics-based* approach and a *structural-match* approach.

The statistics-based approach tries to maintain statistics at each node, based on the set of documents seen so far, to allow it to estimate the selectivity of a given query. We currently use bitmaps and simple summary structures at each node to maintain complete information of the documents that have passed through it. Note that this component can be directly replaced with any existing selectivity estimation scheme to improve recall or space efficiency.

We describe the implementation of such a scheme for a data model consisting of XML documents and XPath subscription queries. Each node maintains a node-labeled *data summary* tree with bitmaps at each

```
// protocol for joining
proc join_codd() {
  seed_set s = send_sync_msg(system_agent,
                             TYPE_GETSEEDSET);
  list candidates = {};
  forall (node n in s) {
    send_async_msg(n, TYPE_ADD, this);
  }

  // wait for timeout, or all nodes to respond
  wait(candidates.length = s.size, TIMEOUT);
  if (candidates.length == 0) return FAIL;

  node p = min_cost_candidate(candidates);
  if (send_async_msg(n,TYPE_ADDCHILD,
                     this) < 0) {
    return FAIL;
  }
  // wait to receive data
}
```

Figure 3: CoDD Join Protocol

node. The bitmap at each node represents the set of documents having a node at that particular position in their XML data tree, and with the same label. This structure can be efficiently updated using new documents by a single pass on the XML tree of the new document. Lookups to check for the set of documents that match a given query can also be easily performed by simulating the twig pattern representing the given query on this structure, and checking to see which documents contain an instance of this twig query. For each node in the XML tree that simulates a query node, we consider the bitmap at that node. The logical 'AND' of each of these bitmaps yields exactly the set of documents that would have matched a given XPath tree query.

However this approach fails early in the evolution of the system, if the statistics have not been seeded sufficiently. In that case, our system uses a structural match on the two queries and uses that to estimate the cardinality of the common documents matched by these. For example, if our query language is XPath, our structural match algorithm compares the tree structures and values described by two queries, in order to do an intensional estimate of the overlap between the two values. While this approach might yield results with higher error rates, CoDD compensates in the reorganization phase, as described in Section 4. For the sake of brevity, we omit a more detailed discussion of these techniques.

```
// message handler for join protocol
async proc message_handler(type t, args a) {
  switch (t) {
    case TYPE_ADD:
    // handle a request to add
      if (spare capacity available)
        send_async_msg(a.sender,
                       TYPE_ADDRESULT, this);
        return;
      node n = max_overlap_node(a.query, this);
      send_async_msg(n, TYPE_ADD, a);
    break;
    case TYPE_ADDRESULT:
    // received a response to add request
    // add the response to list of candidates
      candidates.add(a.node);
    break;
    case TYPE_ADDCHILD:
    // add this node as a child
      this.children.add(a.sender, a.query);
      if (this.parent != null)
        send_async_msg(this.parent,
                       TYPE_ADD_QUERY, a.query);
    break;
    case TYPE_ADD_QUERY:
    // update aggregate filter query
      this.query.add(a.query);
    // and update of parent as well
      if (this.parent != null)
        send_async_msg(this.parent,
                       TYPE_ADD_QUERY, a.query);
    break;
  }
}
```

Figure 4: Message Handlers for Join Protocol

3.1.2 Protocol Analysis

A CoDD topology can be modeled as a random split tree, as described in [13]. These trees have a capacity constraint on each node that corresponds exactly to our fanout constraint. We model the selectivities of the queries, and therefore the probability of the i^{th} child of a node N being chosen while propagating a `TYPE_ADD` message, by a probability p_i. Thus, each node has associated with it a probability vector $V = (p_1, p_2, \ldots p_f)$, where f is the fanout (or capacity) constraint.

With these assumptions, and using the universal limit laws for random trees [13], we have the following result.

Lemma 1 *Let $\mathcal{D}_n$ be the depth of the system with n participating nodes. Then, if we incrementally con-*

struct the topology according to the join protocol as described above, with no changes, the following is true in probability

$$\frac{\mathcal{D}_n}{log(n)} \rightarrow \frac{1}{\mathcal{H}} \qquad (1)$$

where $\mathcal{H}$ is the entropy of the probability distribution, defined by

$$\mathcal{H} = -\sum_{i=1}^{f} p_i log(p_i) \qquad (2)$$

■

This result indicates that the join protocol as described above, is expected to form trees where the depth of the tree is logarithmic in the number of nodes, with high probability. It is important to maintain this property, since it implies a low expected latency seen at consumer nodes.

Similarly we can bound the overhead of node additions, by counting the number of messages that need to be exchanged in the system during an execution of the distributed join protocol. Each such execution results in $h_i \times s$ messages being exchanged, where h_i is the depth in the CoDD tree of the seed node, and s is the (constant) size of the seed set. Thus, the control overhead of adding a new node to the system increases logarithmically with the size of the system.

We provide further experimental evidence of these analytical results in Section 5.

3.2 The System Leave Protocol

The system allows a node to leave the network after it has received the data it needs. The leave protocol for a node `n` reorganizes the subtree rooted at `n` locally. It proceeds by promoting the child node of `n` with the maximal overlap with `n` one level higher. We denote this node as `n`*. The protocol allows the restructuring to proceed with the minimal addition to the total data flowing on SubTree(n). The protocol then adds each of the existing child nodes of `n` except for `n`* according to the Join Protocol described above, with seed set {`n`*}. Finally, it adds each of the children of `n`* similarly to the subtree rooted at `n`*. Thus, the only set of nodes affected by `n` leaving are the descendant nodes of `n`. The algorithm is described in more detail in Figure 5.

```
proc leave_codd() {
  if (is_leaf(this)) {
    send_async_msg(this.parent, TYPE_LEAVE, this);
    return;
  }

  node n = max_overlap_node(this.query, this);
  send_async_msg(n, TYPE_NEWPARENT, this.parent);

  forall (node c in n.this.children) {
    send_async_msg(c, TYPE_NEWPARENT, n);
  }
}
```

Figure 5: CoDD Leave Protocol

4 Reorganization Protocols

The tree reorganization protocol is the reactive component of the system. It is initiated by nodes when they observe a rise in the number of spurious documents received by them, and results in the system possibly moving a *subtree* rooted at that node to another part of the network. The protocol associates a cost with the reorganization, and accepts one only when the observed gain to the system outweighs the cost of the move.

CoDD maintains a timeout parameter for each node, which defines a constraint on the amount of time that must pass between successive reorganization requests for any given node. A request for reorganization can be made periodically after a configurable time-limit has passed, or can be done on demand when the node detects an increase in the amount of spurious data it is receiving. Thus, the protocol allows the system to react to increases in the data overhead, while the timeout restriction prevents nodes from unfairly initiating these requests very often.

The protocol proceeds with the node, `n`, sending a `TYPE_REORGANIZE` message to the root node of the system. The root node checks to see that the requesting node is allowed to request for a reorganization, and that the time since its last such request exceeds the timeout parameter. The node then proceeds according to Figure 7. It maintains a current best node, and then traverses the tree in a breadth-first manner. The protocol prunes paths on the tree when it determines that they do not contain candidates that would provide a better solution node than the current best node. The algorithm terminates by returning a new parent for the node that needs to be reorganized.

We define the overhead associated with a node p with respect to the node n as the amount of spurious data that would be added to the system if we were to add n as a child of p. The overhead can be computed easily by each node, which maintains a set of the

```
// message handler for leave protocol
async proc message_handler(type t, args a) {
  switch (t) {
    case TYPE_LEAVE:
    // remove child
      this.children.remove(a.node);
      send_async_msg(this,
                     TYPE_REMOVE_QUERY, a);
    break;
    case TYPE_REMOVE_QUERY:
    // remove query from filter
      this.query.remove(a.query);
      if (this.parent != null)
        send_async_msg(this.parent,
                       TYPE_REMOVE_QUERY, a);
    break;
    case TYPE_NEWPARENT:
    // add node to new parent
      send_async_msg(a, TYPE_ADDCHILD, this);
    break;

}
```

Figure 6: Message Handlers for Leave Protocol

document ID's it has received up to this point. By comparing this set with the set of documents received by the reorganizing node, the benefit associated with this move can be estimated. The protocol prunes search paths on the tree by estimating the overhead of examining the children of a current node, and using this estimate to stop the search along paths that are known to entail higher overheads than the current topology.

The protocol maintains a queue of *prospective* candidate nodes that might contain possible solutions. If the node currently examined has a free child slot, it is processed by comparing its overhead with that of the current best node. If the fanout constraint for the node is already satisfied, then the protocol tries to estimate if a child of this node could contain a better solution than the current one. If so, the protocol adds each child of this node to the *prospective* candidate queue. Otherwise, this subtree is pruned from the search space.

In practice, this protocol is initiated by a node n when it discovers it is receiving spurious data beyond a particular threshold. To minimize the data it is receiving, the node n determines which of its children has the least overlap with itself. This can easily be determined by using the set of documents received at both the nodes. The node n then requests that this child be reorganized in the network tree, which lowers the overhead on n and potentially its ancestors.

```
proc reorganize() {
  best_parent_overhead = MAX_VALUE;
  queue prospective = {ROOT};
  best_parent = NULL;
  repeat {
    node c = prospective.dequeue();
    (curr_overhead, c.children) =
      send_sync_msg(c, TYPE_GETOVERHEAD, this);
    if (c has spare capacity) {
      if (curr_overhead < best_parent_overhead) {
        best_parent = c;
        best_parent_overhead = curr_overhead;
      }
    }
    else if (2*curr_overhead <
             best_parent_overhead) {
      forall (node p in c.children)
        prospective.enqueue(p);
    }
  } until (prospective.length > 0);

  if (best_parent == NULL) {
    // unable to get better parent
    return FAIL;
  }
  send_async_msg(best_parent,
                 TYPE_ADDCHILD, this);
}

// message handler for reorganize protocol
sync proc message_handler(type t, args a) {
  switch (t) {
    case TYPE_GETOVERHEAD:
      // return overhead for reorganize
      overhead = compute_overhead(this, a.node);
      return (overhead, this.children);
   }
}
```

Figure 7: CoDD Reorganization Protocol

Finally, note that the reorganization protocol works in a distributed manner. Therefore, the actual system works by sending appropriate control messages between nodes and recursively sending the processing down the tree. We examine the cost and benefits of reorganization, and explore a number of optimizations to improve these, in Section 5.

5 Experimental Evaluation

5.1 Experimental Setup

The current implementation of CoDD consists of a set of Java classes, which can be deployed in a distributed setting. We have also developed a simulator for nodes, which allows us to test a large number of configurations locally. We conducted our experiments using the Sun Java SDK version 1.4.1_01 running on a PC-class machine with a 1.6 GHz Pentium IV processor and 256 MB of main memory, running the Redhat 9 distribution of GNU/Linux (kernel 2.4.22).

Each node is simulated by a Java object. We simulate network data transfers using standard interprocess communication mechanisms, such as local pipes for the simulator and sockets for the distributed deployment of CoDD. Each node runs an instance of a query engine corresponding to the data and query language in use. This engine evaluates each child's filter query on each data item, and enqueues the item to be sent to each of the children whose filter query it satisfies.

The primary metric we study is the **network communication overhead**. We define this overhead as the fraction of documents received by each node in the system that do not satisfy its subscription query. While CoDD tries to construct topologies that minimize this overhead, the protocols also try to ensure that the overlay topology gives **good network performance**. For example, we want the topology to have low depths to minimize the latency seen by individual clients. We also look at how reorganization effects the performance of the system, and the overhead seen at individual nodes across time for a variety of data and parameter sets. Further, we describe a set of optimizations to reduce the overhead of reorganization, while delivering close to original performance gains.

We run experiments using two kinds of datasets. The first kind of datasets represent an abstraction of the mapping between data and subscriptions. More precisely, we generate a stream of document ID's and a stream of node ID's. We do not make assumptions about the content of each document, or the subscription query language. Instead, we generate subscription queries by mapping document ID's to node ID's, where each such relationship indicates an instance of a document satisfying the query for the corresponding node. We capture correlations in subscription queries by grouping nodes into overlapping "interest-classes," which denote sets of nodes that are interested in a similar subset of documents. For the purposes of our experiments, we assign nodes to one or more of k category buckets, with each bucket representing an interest class. We use $k = 20$ by default. Subscription queries are modeled by mapping each data item to every member of a fixed number of these classes. We model changes in data distribution by performing a random shuffle operation on the buckets, at a rate described by the change-frequency parameter cf. Every shuffle operation consists of splitting c buckets into two sub-buckets, and then merging uniformly randomly chosen pairs to form new mappings. We use $c = k/5$ by default. Figure 8 describes these parameters in more detail. Unless described otherwise, the default synthetic dataset we used had a selectivity of 0.2, with 400 nodes, and 10000 documents being published in the system. The document distribution was made to change every 200 documents, and the default fanout was 6.

n	number of nodes
d	number of documents
s	selectivity of documents
cf	frequency of document distribution change
f	fanout limit
r	documents before reorganization
$rthresh$	The threshold of the reorganization

Figure 8: Experimental Parameters

In the second set of experiments, we generate XML documents corresponding to several well-known DTD's, and use synthetically generated XPath queries to capture the subscription interests of nodes. We describe this data set in more detail in subsection 5.4.

We denote the CoDD system with reorganization enabled as CoDD-R, and the system without the reorganization component as CoDD-NR.

5.2 Depth of nodes in the overlay

The depth of a node in the overlay network built by CoDD affects properties such as the latency of data delivery. We performed a set of experiments on the synthetic data sets, with and without reorganization enabled, and measured the depth of the nodes in the CoDD system. Figure 9 depicts the distribution of the weighted average depth of nodes. The weighted average depth of a node is its depth averaged over time, weighted by the number of documents it receives at each depth. Reorganization is seen to improve the depth of the nodes in the system significantly, by adapting the topology to take advantage of the dynamic document distribution. Moreover, we see in Figure 10 that the depth of the topology increases slowly with an increasing size of the node set, indicating that CoDD-R scales well with a large number of participants. This growth is seen to closely follow the $\log x$ curve, as expected from our analytical analysis earlier. However, without reorganization,

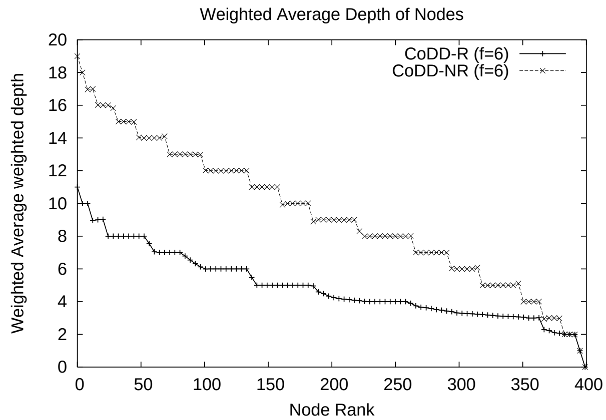


Figure 9: Distribution of weighted average depths

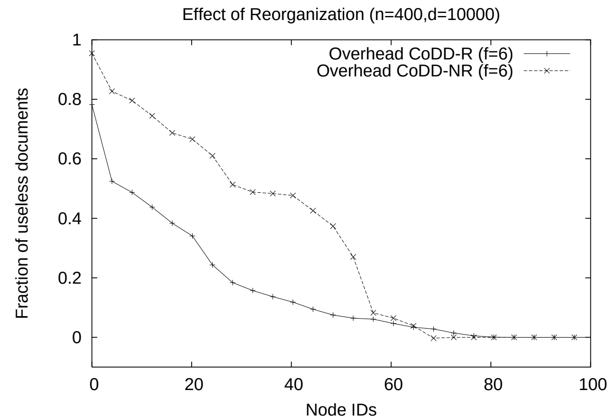


Figure 11: Overhead versus Reorganization Frequency

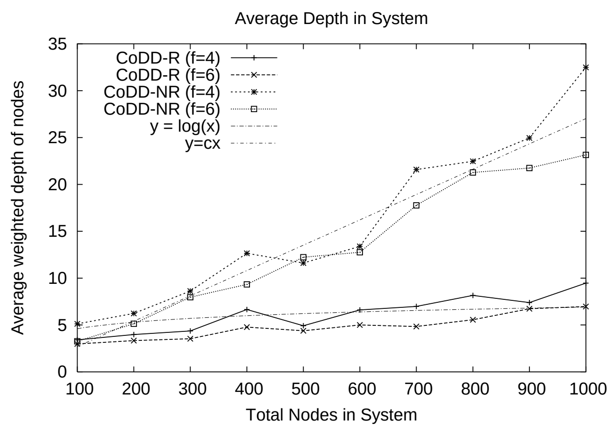


Figure 10: Average Depth versus Nodes

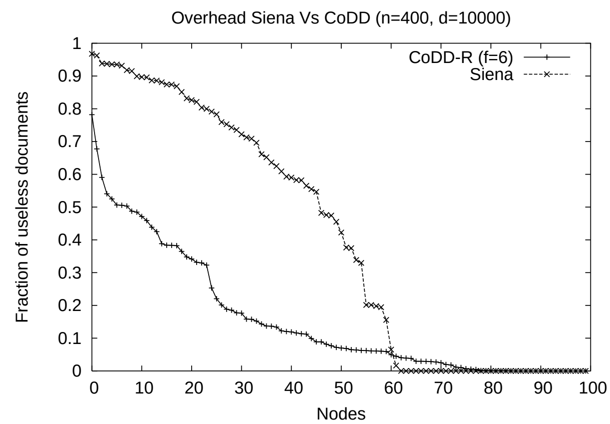


Figure 12: Overhead of Siena versus CoDD

this depth is seen to increase linearly. Thus reorganization is seen to dramatically increase the ability of CoDD to scale to large participant sets.

5.3 Evaluation of System Overhead

In Figure 11, we measure the overhead with a changing document distribution for the default synthetic dataset. We perform this experiment with a fanout of 6, and observe the overhead of the interior CoDD nodes with and without reorganization. Reorganization is seen to significantly improve the overhead observed by the nodes, and the average overhead of the system decreases by over 50% as a result of initiating reorganization. This trend indicates that the reorganization component allows CoDD adapts gracefully to changing document distributions.

5.3.1 Comparison with existing systems

Siena[4] is an event-based publish-subscribe system used to selectively disseminate data to a large set of subscribers. Siena assumes a centralized infrastructure of servers, and associates clients with individual servers. Servers communicate with each other depending on the requirements of the clients serviced by each server. For purposes of comparison, we try to emulate a serverless environment using Siena as follows. The network is built incrementally, with a fraction of randomly selected nodes designated as servers. These nodes are used to build a balanced tree of servers, and clients connect to a randomly selected server in this network. There are f clients per server, where f models the fanout constraint in the corresponding CoDD environment.

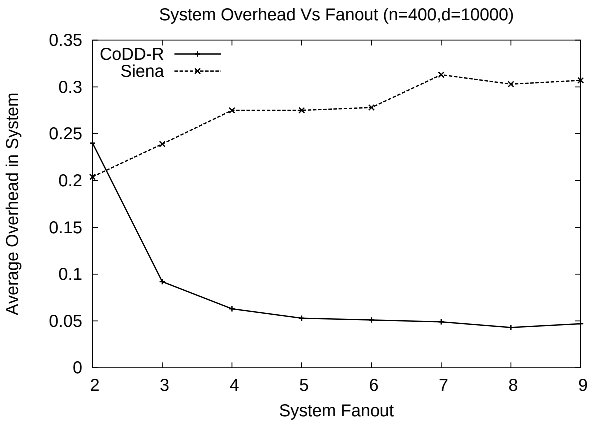


Figure 13: Average Overhead versus Fanout

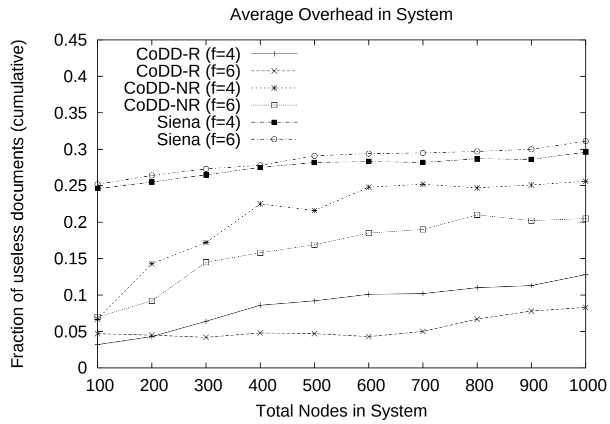


Figure 14: Average Overhead in System

In Figure 12, we show a comparison between Siena and CoDD using the default synthetic dataset. The CoDD system generates topologies that are data-aware and thus outperforms the Siena environment in terms of the metric of data overhead. This experiment is also biased in favor of the Siena environment, since the overhead of the Siena system is only due to the few designated server nodes. Thus, the tail of the distribution for Siena is much shorter. However, these interior server nodes incur a high overhead. CoDD is able perform significantly better than Siena without having to resort to any centralized computation or high communication overheads.

In Figure 13, we refer to the overhead in the system with a changing fanout. We compare the performance of CoDD with a centralized Siena topology, modified as described above to perform in a server-less environment. The CoDD system is seen to perform significantly better with an increasing fanout, with the tradeoff being a higher processing and network cost per node as we increase the fanout. The Siena system, on the other hand, has a performance that remains poor with increasing fanout, and the overhead seen in the system is also significantly worse. It is interesting to note that the Siena system performs worse at higher fanouts, because the interior nodes in the tree have a higher number of children, which would in general have very little overlap with the parent.

We also measure the overhead of the system, by varying the fanout and reorganization level in CoDD for different node set sizes. The results in Figure 14 indicate that the amount of overhead in the system is significantly lowered with the reorganization. Also, it can be seen that with reorganization, the overhead increases at a much lower rate with the number of nodes in the system, allowing the system to scale better.

5.4 CoDD on XML datasets

We performed another set of experiments with XML datasets, using XPath as a query language. Datasets were generated using real-life, publicly available DTD's. The first dataset is from the News Industry Text Format (NITF) DTD[1], which is supported by most of the world's major news agencies. The NITF DTD (version 2.5) contains 123 elements and 513 attributes. We also used two other DTD's to generate datasets to compare the performance of CoDD for different data workloads. They are the Protein Sequence Database (PSD) DTD[2], and the CIML DTD[3]. The characteristics of these datasets are further described in Figure 15.

DTD	Max Depth	Avg Depth	Elements
NITF	9.07	5.83	107.28
PSD	6.77	4.37	238.82
CIML	4.44	3.32	22.22

Figure 15: XML Document Data Used

We generated our test data sets of XML documents using IBM's XML Generator tool[4]. The tool generates random instances of valid XML documents, conforming to a given input DTD.

[1] Available at `http://www.nitf.org`

[2] Available at the Georgetown Protein Information Resource, `http://pir.georgetown.edu`

[3] The Customer Identity/Name and Address Markup Language, `http://xml.coverpages.com/ciml.html`

[4] `http://www.alphaworks.ibm.com/tech/xmlgenerator`

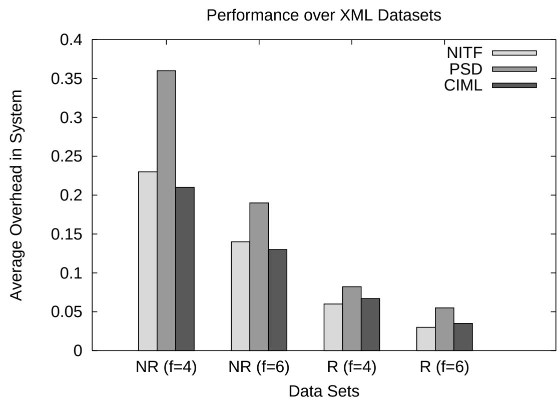


Figure 16: Performance for different datasets

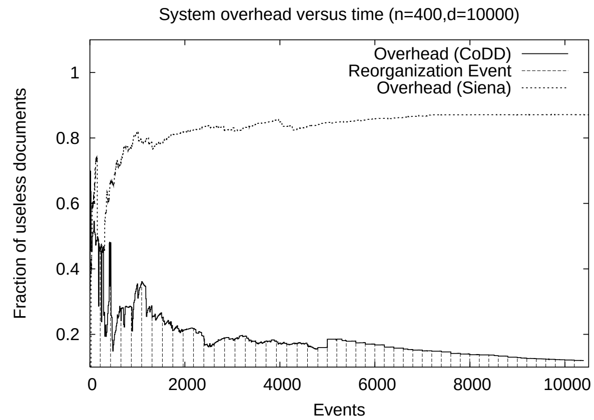


Figure 17: Overhead versus Time

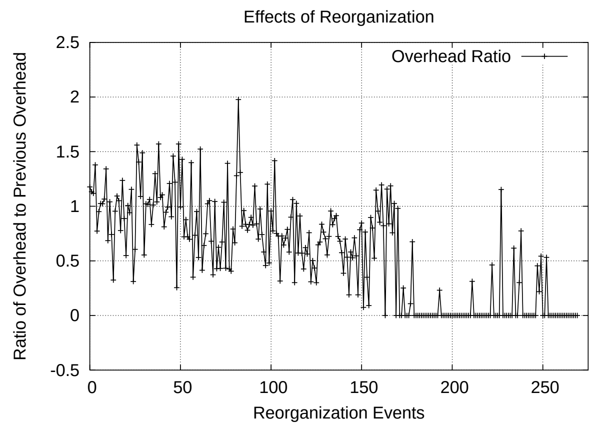


Figure 18: Reorganization Effects

To generate a workload of XPath queries to be used as subscription queries for each of the nodes, we developed an XPath query generator. The query generator takes as input a DTD file, describing the set of documents that we need to generate queries on. It also uses a set of parameters describing characteristics of the XPath queries needed. These parameters include p_*, which describes the probability of an element having a wildcard ($*$) tag, p_c, which is the probability of a node having more than one child (that is, a twig branch forms at that node), and p_λ, which is the probability of generating a closure axis. Our current workload uses values of 0.2, 0.3 and 0.2 for these parameters, respectively.

The XPath queries generated above had a selectivity of 0.28, 0.15 and 0.31 for the NITF, PSD and CIML datasets respectively.

We measure the normalized overhead for the system in Figure 16. The reorganization was triggered every 200 documents, and we measured the overhead in the system with and without reorganization. In Figure 16, we see that with reorganization, the system is able to run at overheads of less than 10% at low fanouts.

5.5 Effects of Reorganization

In this section, we describe several experiments, run on both the abstract and XML data sets, that describe the benefits and costs of reorganization in a CoDD system.

In Figure 17, we describe an experiment to show the temporal behavior of the overhead of the system, and how it behaves as and when we perform a reorganization. We ran this experiment on the default synthetic dataset, and show the average overhead in the system varying with time. Time is measured in discrete events, which increments every time a document is published, or a reorganization event occurs. We measure the average overhead of all the nodes in the system that have positive overhead (we only measure non-leaf nodes). The impulses indicate reorganization events. The average overhead of the system is seen to decay appreciably toward the end of the curve, indicating the system converging to a low overhead configuration. Further, when we compare this system against the overhead seen at interior nodes in a Siena-like configuration, the overhead of the CoDD system is seen to be significantly lower. Siena does not capture data-specific properties in the topology construction, and thus the overhead stays fairly constant over the duration of the experiment.

We also analyzed the data from the previous experiment to measure the effects of reorganization events on the overhead of the node that asks for a reorganization. In Figure 18, we quantify the effect of the reorganization by measuring the ratio of the overhead after the reorganization (and before the next one) with the overhead before the organization. The former is computed for the documents received *after* the node has moved to a new position. Thus, if the reorganization was to be successful this ratio should be as low as possible. The graph indicates an interesting property. Namely, this ratio fluctuates significantly initially. However, the latter 60% of the reorganizations are quite successful, because the statistics maintained by the system improve over time, allowing it to make more informed decisions. There are still some nodes that do not perform well after the reorganization. However, the effect of that is usually pretty low ($ratio \leq 1.5$).

The previous experiments have shown that frequent reorganization leads to several desirable properties. However, there is an inherent cost associated with reorganization. First, reorganization has a control communication overhead associated with it. Secondly, the node being relocated has to setup and break overlay connections, change filter queries for its ascendants, and possibly receive duplicate documents during the move to ensure no loss of information. Thirdly, statistics on document distributions for nodes that participate in reorganization change on addition of new children, which affects the components of the system that depend on statistics for good performance. We conducted a set of experiments to describe a number of optimizations and parameters that can be used to decrease this overhead, and increase the benefit of the reorganizations.

5.5.1 Reorganization Frequency

In the first experiment, we varied the frequency of reorganization. Recall that the frequency is determined by the timeout parameter described earlier, which describes the number of documents each node must receive before it can request a change in network topology. The results of running the experiment with 1000 nodes, on 10000 documents using the XML/NITF dataset, is shown in Figure 19. For the purposes of these experiments, we considered discrete time incremented every time a new node joins the system. We measure the *current overhead* of the system over windows of 10 time units, to determine how the system evolves over time. In Figure 19, we ran this experiment with a varying reorganization frequency. We can observe that the system performs comparably for reorganization timeouts ranging from 20–200. The overhead increases if we go to a much slower timeout

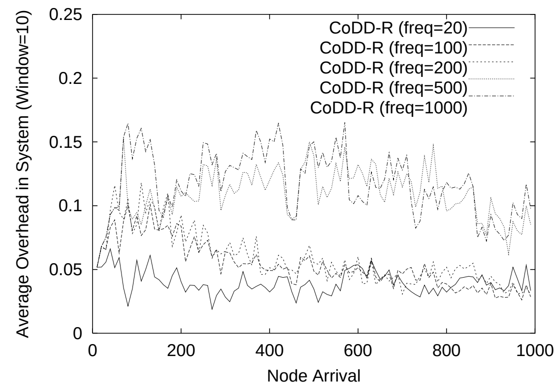


Figure 19: Effect of reorganization frequency

rate (500/1000 document timeouts). This plot indicates that the benefit achieved from running CoDD with faster reconfigurations diminishes rapidly. In particular, we can see that the performance of the system with reorganizations every 200 documents is close to that with much faster reconfigurations, but at a much reduced reorganization overhead.

5.5.2 Top-c Limited Reorganization

We conducted several experiments to measure the effect of limited reorganization, using the same dataset as the previous experiment. Limited reorganization associates a *cutoff* parameter, c, with the reorganization protocol. For every batch of reorganization requests, CoDD with limited reorganization only allows the top-c requests, as ordered by the overhead at the node requesting to be moved. This optimization allows the system to adapt to high reorganization request rates by answering requests from high-overhead nodes only. In Figure 20, we can see that running the reorganization with reasonably low cutoffs (10-20) can yield performance close to that of the system which grants all eligible requests. Limited reorganization thus gives the system designer a parameter to bound the amount of overhead associated with each reorganization.

5.5.3 Reorganization Thresholds

A reorganization threshold is the minimum overhead a node must incur for it to request a reorganization. The tradeoff associated with choosing an optimal threshold value is the gain of more aggressive reorganizations versus the cost associated with the topology reconfigurations. In figure 21, we look at the

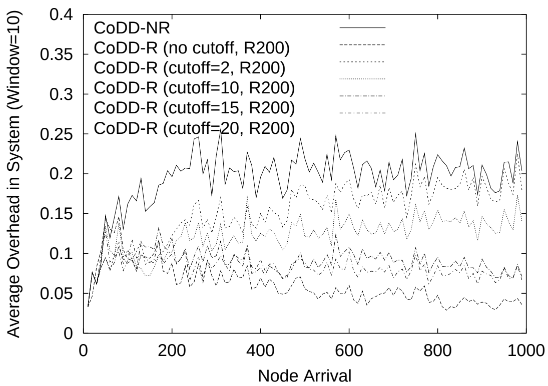


Figure 20: Limited Reorganization

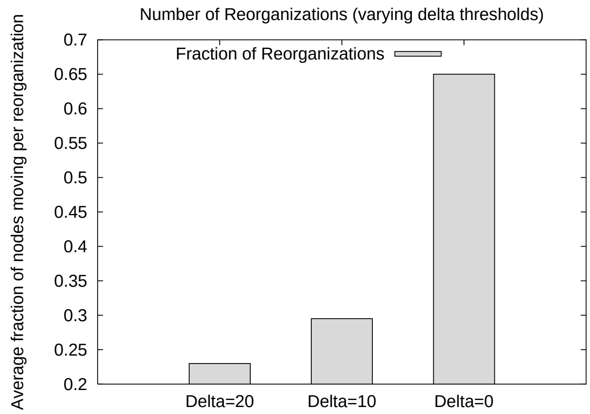


Figure 22: Average number of reorganizations with delta-thresholds

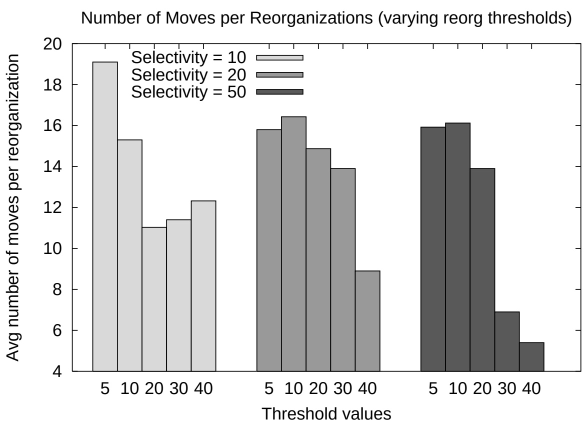


Figure 21: Average number of moves / Reorganization

average number of node moves that occur in the system with varying selectivities and thresholds, using the default synthetic data set. We see that in general a higher threshold value does result in fewer moves occurring. This decrease occurs more at a higher selectivity. This trend indicates that with high selectivities, we incur a higher overhead as a result of using lower thresholds: using higher threshold values is advantageous in these situations because it is seen to lower overheads appreciably. However at lower selectivities, we can afford to work with medium to low threshold values, since the performance gain from them is better and the overhead is not appreciably worse.

5.5.4 Delta Thresholds

In Figures 23, 24 and 22, we investigate a new optimization to reduce the cost of reorganization. A delta-threshold is a system parameter used by CoDD to determine the utility of a reorganization. It defines the fraction by which a reorganization should decrease overhead for it to be worth it for the node to move. We investigate three parameters values, 20, 10 and 0, corresponding to the improvement resulting in a decrease in immediate overhead of 20%, 10% and 0%, using the default synthetic data set. Therefore, the protocol denies all reorganizations that have a perceived benefit of reorganization less than the delta-threshold. As we see in Figures 22, the number of reorganizations decreases significantly with a higher delta threshold. Further, Figure 23 indicates that the performance seen by individual nodes due to reorganization, described in terms of the overhead ratio, is not significantly worse with a higher delta-threshold. However, the performance of the overall system in terms of overhead does not deteriorate significantly, as can be seen in Figure 24. This optimization gives us a parameter for decreasing the cost of reorganization, and indicates that we can tune the reorganization to occur only with an appreciable gain associated with it. The delta-threshold parameter can be set by the system designer depending on the selectivity of the system and the cost of reorganization moves.

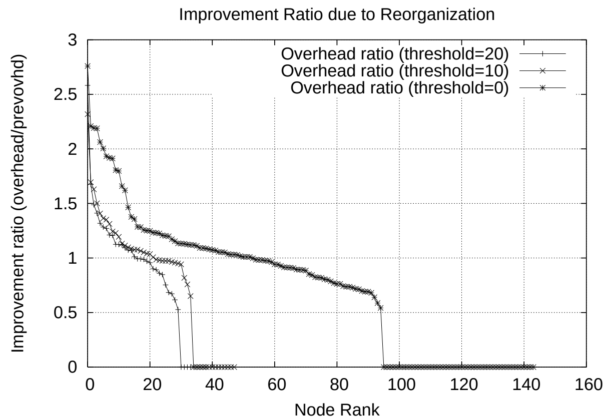


Figure 23: Adding delta-thresholds to the reorganization

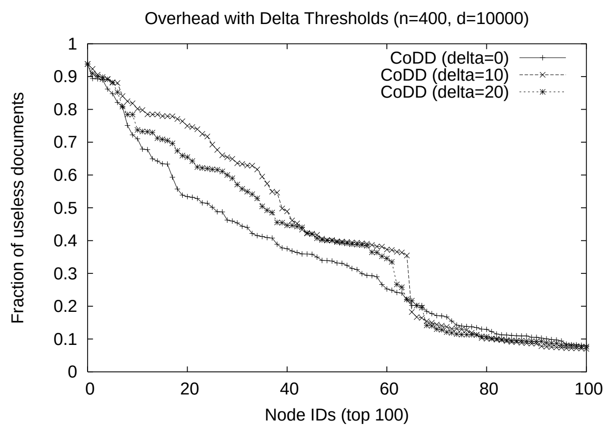


Figure 24: Overhead versus delta thresholds

6 Related Work

As noted in Section 1, work on data dissemination typically focuses on a well-controlled environment (centralized or distributed). The goal of the Query-Merging(QM) problem is to reduce the cost of answering a set of subscription queries by merging different sets of queries together. For a given cost model, the QM problem takes as input a set Q, and outputs a set M, where each element of M is a set of original queries to be merged, and cost(M) is minimized. Previous work has shown QM to be NP-Hard in the general case ($|Q| > 2$) [10]. This work also quantifies the effect of merging for various cost models, and presents a set of heuristics for approximating the QM problem. CoDD concentrates on constructing and maintaining the underlying data-distribution infrastructure, and on effectively distributing communication and computation overhead in a decentralized environment. CoDD does not try to optimally group queries or evaluate heuristics to compute globally optimal solutions. Instead, it relies on adaptive reorganization techniques to push the system toward a low-cost solution.

Siena [4] is a closely related event-based publish-subscribe system. Clients in Siena subscribe to messages by registering subscription queries, and these messages are routed using cooperating brokers that share subscription information. The primary data model used Siena represents events as a set of typed attributes, and queries which describe constraints over a subset of these attributes. Subscriptions are aggregated using internal data structures that describe a partial order on the covering relationships of these queries, and can be efficiently traversed to match incoming notifications. The brokers comprise a centralized infrastructure that the protocol maintains, and scalability is achieved by adding more brokers to the system. Siena also describes a number of topologies for connecting these servers: note however that individual clients are unaware of the details of the server cloud topology. Rebeca [15] and Gryphon [16] are other systems, developed in both academic and industrial environments, that implement various dissemination techniques using similar architectures. CoDD distinguishes itself from these systems by proposing protocols that are decentralized, and work with loosely-coupled subscriber nodes without centralized servers. Further, CoDD allows nodes to self-manage their level of cooperation using reorganization and capacity constraints, thereby enabling low-power nodes to participate in the system. Finally, the extensional overlap used by CoDD allows it to work with more complex data models and query languages, without changing any of internal details of the protocols.

The dynamic and self-organizing environment of sensor networks [18, 26, 12] is similar to the environment addressed by CoDD. However, the main emphasis in sensor networks is on *acquiring* relevant data from sensors using aggregation at upstream nodes. Minimizing power consumption is the primary metric while designing such systems. An aggregation service, such as TAG [20], provides low-level primitives to allow a system to compute aggregates over data flowing through the system, while discarding irrelevant data and compacting related data into more compact records where possible. TAG describes a SQL-like declarative query language for this purpose, and includes various optimizations, like synchronized communication epochs during data collection to reduce power consumptions. Acquisitional query processors [19] focus on locating and estimating the cost of acquiring data. Instead of a-priori assuming an existence of data, they provide language constructs for defining the rate of sampling data, and specifying event-based and lifetime-based queries over continuous data streams. These techniques are tailored towards minimizing power-expensive operations such as frequent communication and data sampling. In contrast, CoDD concentrates on *disseminating* data to interested subscribers. Thus, it is in a sense the dual of the acquisitional nature of sensor networks. CoDD is not restricted by strict power consumption requirements, and assumes a connected underlying network with no restriction on frequency of transmission, allowing it to concentrate on creating data and query-specific topologies.

An interesting common aspect is the process used to disseminate queries to sensors. Semantic Routing Trees [18] allow the dissemination tree to exploit knowledge of data semantics and send queries only to sensors with relevant data. Each node maintains unidimensional index intervals for each attribute A of the data for each of its children, which represents the range of values for A available at that child. This information is used to route queries that depend on attribute A only to children that are known to have relevant data. For example, when a query q arrives at a node with a constraint on attribute A, the SRT can be used to forward it only to those nodes whose index overlaps the range of A in q. CoDD uses a similar idea to construct overlay topologies, using instead extensional commonalities in data streams. Instead of using attributes of the data, we rely on statistics about the number of documents matched to obtain an estimate of the relevance of a query at a particular node. This technique allows CoDD to work well with arbitrary data and query sets, and in the absence of complete knowledge of data syntax and semantics.

There have been several optimization techniques proposed to enhance the performance of push-based dissemination systems. These include techniques for pushing popular data to servers close to the consumers [3], and data structures and caching policies for fast publish-subscribe [14]. In addition, adaptive approaches integrating push and pull-based systems have also been proposed. Pull-based algorithms have low overheads because they do not require state to be maintained at servers. However, they do not offer high fidelity in the presence of rapidly changing data or strict coherency requirements, because the clients request for data at a rate that is oblivious to the characteristics of the data stream. For example, knowledge about unpredictable data rates is available only at the server, and this information is useful in deciding the rate of requesting information. While push based algorithms offer a solution to some of these problems, they can incur significant computational and state-space overhead because of the extra state needed at the server end. Moreover, the centralized storage of such state makes them less resilient to failures. Adaptive push-pull systems [11] allow clients to indicate their requirements in terms of data fidelity and coherency, which can be used by the system to tune the delivery mechanism used on a per-client basis. CoDD concentrates on creating and maintaining an underlying infrastructure to efficiently manage this dissemination. It does not make assumptions about the data delivery mechanism used, and the topologies created by CoDD can be extended relatively easily to use the optimization techniques described above in the presence of specific client requirements.

The decentralized services provided by CoDD have several characteristics that are similar in nature to the resource discovery problem in pervasive computing environments. For example, the VIA system [5] allows domains to organize into clusters corresponding to the resources available at each domain. Resource descriptions are described using a list of attribute-value pairs called a metadata tag, and queries are specified using a set of constraints on a subset of these attributes. These clusters are constructed by a process called generalization, which aggregates queries by creating a set of upper-bound queries by replacing some attribute value constraints by "wildcard" constraints. These upper-bound queries are then used to discover commonalities in metadata information and build a topology corresponding to these commonalities in a bottom-up fashion. VIA* [6] extends these techniques to allow queries to be generalized based on an impedance parameter. The query impedance is the average number of times a data attribute does not match a query attribute, and describes the relevant importance of that attribute in contributing to query matches. The extensional method for grouping nodes used by CoDD may be viewed as a generalization of this idea. The emphasis in CoDD is low-overhead protocols for dynamic environments that emphasize node autonomy. In VIA, groups of nodes are managed by an administrative domain controller. It would be interesting to explore combinations of these methods.

INS/Twine [2] adopts a peer-to-peer approach to solving the problem of resource discovery. Each resource is published in an underlying distributed hash table (DHT) such as Chord [23] by transforming resource descriptions into a set of numeric keys. Queries to locate resources are routed to the node in the DHT responsible for the resource, using techniques that enable queries with partially specified attribute sets, and allow for an even data distribution throughout the network. The use of a structured distributed hash table restricts the system from building topologies that take the data distribution into account. Instead, topologies are governed completely by the hash function and the key structures used. Moreover, the maintenance of this structure in an environment with a high rate of change in the node sets can often be an expensive operation. For example, most DHT's have an $O(\log(n))$ bound on the cost of node leaves. This overhead is often compounded by graceless node leaves, where a node leaves the system without properly informing its ancestors and transferring relevant state. This overhead can be significant in a typical peer-to-peer like system, where the median up-time for nodes is as low as 60 minutes [21]. In contrast, CoDD creates data-aware tree structures that work towards minimizing the overhead of excess data. Unlike in a DHT, the structure of the topology is not governed by data-oblivious parameters like a universal hash function. This flexibility and lack of strict structure allows CoDD protocols to react well to high node turnover rates.

The *NiagaraCQ* system is designed to efficiently support a large number of subscription queries expressed in *XML-QL* over distributed XML datasets [9]. It groups queries based on their signatures. Essentially, queries that have similar query structure by different constants are grouped and share the results of the subqueries representing the overlap among the queries. Our system complements some of the ideas proposed for the NiagaraCQ system, by providing a mechanism for distributing the computation and dissemination overhead over a large set of peer-nodes. While NiagaraCQ works with change-based and timer-based events to indicate new data, our model assumes a source stream of data available at the root.

Recent work in the context of managing streaming data handles several issues that are closely related to the CoDD system. The *Fjords* architecture [17] has been developed for managing multiple queries over the numerous data streams generated from sensors. Fjords presents a set of operators that can be used to form query execution plans with data being pushed from sensors in conjunction with data pulled from traditional sources like disks. In contrast to CoDD, the emphasis in Fjords is maintaining a high throughput for queries even when the data rate is unpredictable. The *Aurora* system is another stream processing system that manages sensor data [1]. Aurora provides operators for supporting monitoring applications, which include primitives for operations such as sample, filter and aggregate streaming data, and a query model to evaluate queries on this data. The system provides various query optimization techniques that aim at adaptively modifying query execution plans, and a cost model for evaluating these plans. CoDD primarily aims at evaluating filter queries on data streams, and on effectively disseminating this data to interested consumers. It would be interesting to incorporate some of the ideas from the above systems into our protocols to perform more complex operations on these data streams, and we leave that for future work.

7 Conclusions

We presented CoDD, a collection of robust and efficient protocols for data dissemination in a serverless environment. Specifically, we presented a system for disseminating arbitrary documents when preferences are expressed using an appropriate query language. Our protocols do not assume a centralized information repository, and instead rely on minimizing the communication overhead of making distributed decisions. They are also designed to work well with autonomous nodes frequently joining and leaving the network. The reactive component of these protocols is designed to adapt to changing characteristics of the data stream. We also presented an empirical evaluation of CoDD based on our implemented testbed. Our experiments show that CoDD scales to 1000s of nodes, with data overheads of 5–10%.

In continuing work, we are using our testbed to perform a more detailed experimental study of the sensitivity of our protocols to parameters such as fan out, skew, and drift, and to test our implementation on a live network. We are also working on extending our methods to data queries (not just filters) and on exploiting commonalities between multiple data streams in a decentralized network.

References

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Pervasive 2002 - International Conference on Pervasive Computing*, Zurich, Switzerland, Aug. 2002.

[3] A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time for distributed information systems. In *Proceedings of the International Conference on Data Engineering*, New Orleans, Louisiana, 1996.

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

[5] P. Castro, B. Greenstein, R. Muntz, P. Kermani, C. Bisdikian, and M. Papadopouli. Locating application data across service discovery domains. In *International Conference on Mobile Computing and Networking (MOBICOM)*, pages 28–42, 2001.

[6] P. Castro and R. Muntz. An adaptive approach to indexing pervasive data. In *ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 2001.

[7] C. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2002.

[8] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering*, pages 235–244, Feb. 2002.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 379–390, May 2000.

[10] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Efficient query subscription processing in a multicast environment. In *Proceedings of the International Conference on Data Engineering*, pages 83–83, Mar. 2000.

[11] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *Proceedings of the International World-Wide Web Conference*, pages 265–274, 2001.

[12] A. Deshpande, S. K. Nath, P. B. Gibbons, and S. Seshan. Cache-and-Query for wide area sensor databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2003.

[13] L. Devroye. Universal limit laws for depths in random trees. *SIAM Journal on Computing*, 28(2):409–432, 1999.

[14] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2001.

[15] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.

[16] The Gryphon Messaging System. `http://www.research.ibm.com/gryphon/`.

[17] S. Madden and M. J. Franklin. Fjording the Stream: An architecture for queries over streaming sensor data. In *Proceedings of the International Conference on Data Engineering*, Feb. 2002.

[18] S. R. Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks*. PhD thesis, University of California, Berkeley, 2003.

[19] S. R. Madden, M. J. Franklin, J. M. Hellerstein, , and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2003.

[20] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for adhoc sensor networks. In *Symposium on Operating System Design and Implementation*, Dec. 2002.

[21] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN'02)*, Jan. 2002.

[22] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[23] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Diego, CA, Sept. 2001.

[24] The XPath Language Web Page. http://www.w3.org/TR/xpath.

[25] T. W. Yan and H. Garcia-Molina. Distributed selective dissemination of information. In *Proceedings of the IEEE International Conference on Parallel and Distributed Information Systems*, pages 89–98, 1994.

[26] Y. Yao and J. E. Gehrke. Query processing in sensor networks. In *Conference on Innovative Data Systems Research*, Jan. 2003.