12-2008

Group-aware Stream Filtering for Bandwidth-efficient Data Dissemination

Ming Li
Dartmouth College

David Kotz
Dartmouth College

Group-aware Stream Filtering for Bandwidth-efficient Data Dissemination

Ming Li and David Kotz
Institute for Security Technology Studies
Department of Computer Science, Dartmouth College
Hanover, NH 03755
{ mingli, dfk} at cs.dartmouth.edu

Abstract

In this paper [1] we are concerned with disseminating high-volume data streams to many simultaneous applications over a low-bandwidth wireless mesh network. For bandwidth efficiency, we propose a *group-aware stream filtering* approach, used in conjunction with multicasting, that exploits two overlooked, yet important, properties of these applications: 1) many applications can tolerate some degree of "slack" in their data quality requirements, and 2) there may exist multiple subsets of the source data satisfying the quality needs of an application. We can thus choose the "best alternative" subset for each application to maximize the data overlap within the group to best benefit from multicasting. An evaluation of our prototype implementation shows that group-aware data filtering can save bandwidth with low CPU overhead. We also analyze the key factors that affect its performance, based on testing with heterogeneous filtering requirements.

keywords: data filtering, data dissemination, overlay multicasting, bandwidth reduction

1 Introduction

Recent years have seen a new class of applications that need to monitor and adapt to continuously changing "context". Context is any information that can be used to characterize the situation of entities and is commonly expressed as location, time, environmental status, and other resources nearby. Monitoring applications for scientific study or emergency response purposes are context-aware applications and need to continuously stream fine-grained data from sensor-empowered context sources. For example, marine biologists may need to get high-resolution chlorophyll readings of a contaminated river for scientific analysis; alternately, predicting a fire's spread in an emergency requires fast-rate temperature, wind direction and wind intensity data feeds from the sensor networks deployed on the scene.

To disseminate high-volume data for monitoring applications may not be a big challenge for bandwidth-abundant wired networks. Since the monitoring for many scientific studies or for emergency response happens in an outdoor environment, deploying a wireless network is often more cost-effective than deploying a wired one for data dissemination. On the other hand, it is well recognized that the effective bandwidth of a congested wireless network is usually much lower than its link capacity. Hence, the challenge is to satisfy the high-volume data acquisition needs of the applications in bandwidth-constrained wireless networks.

Two main approaches have been proposed to tackle the problem. One is to eliminate redundant communications with multicast protocols when disseminating common data to multiple subscribers. The other

[1] Acknowledgement: This paper is accepted by the International Journal of Parallel, Emergent and Distributed Systems (Taylor & Francis), 2008.

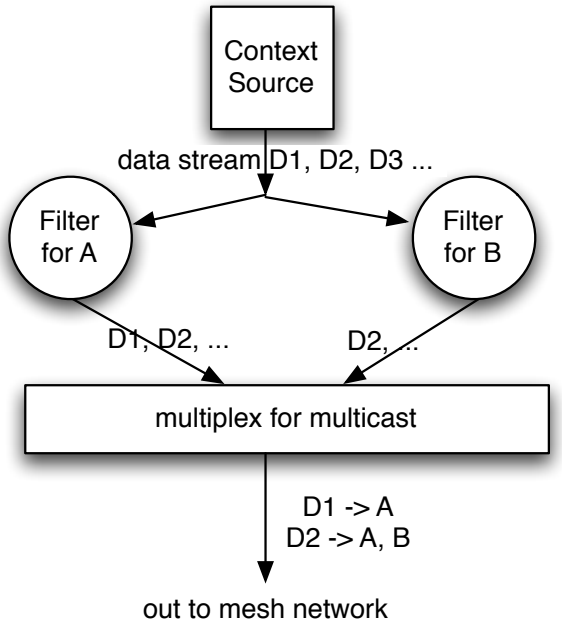

Figure 1: Multiplexing filtered streams for multicasting.

is to reduce the data at a context source, by applying application-specific filters at the source node to select only those tuples "important" for meeting the applications' data-quality requirements. Since source-sharing applications may use context source in different ways, filters deployed at the same source may select different portions of the source data. If there is sufficient overlap of the data selected by the filters, we can still multicast the data to further reduce bandwidth demands. Thus, at the source node, we multiplex the filtered streams to form a multicast stream. Figure 1 shows this process: two applications, A and B, share the same context source $< D1, D2, D3, ... >$, but each application's filter selects a different subset. The multicast protocol allows us to label each tuple with the list of the applications that should receive that tuple; thus each tuple is transmitted at most once on any link.

We here propose a solution that combines multicasting and filtering for context streams. In contrast to self-interested filtering, which only considers each individual application's needs, we propose *group-aware stream filtering* that considers the needs of individual applications, as well as those of other subscribers. The result of this "group-aware stream filtering" satisfies all subscribers' data requirements, and simultaneously ensures maximum data sharing among the subscribers to make the best use of a multicast protocol in saving bandwidth. Our work makes use of two overlooked, yet important, properties of context-aware applications: 1) many applications can tolerate some degree of "slack" in their data quality requirements, and 2) there may exist multiple subsets of the source data satisfying the quality needs of an application. We can thus choose the "best alternative" subset for each application, maximizing the data overlap within the group to best benefit from multicasting.

In the paper, we describe the following contributions of this work.

- Our approach uses multicast protocols in concert with data filtering to reduce unnecessary data traffic, which is crucial for a wireless dissemination system to support large-scale context sharing. The core of our approach is to exploit semantics of applications to reduce data communication. We treat saving bandwidth a goal as important as providing data to satisfy applications' quality needs.

- We developed a framework that encapsulates the general idea of group-aware filtering.
- We built a prototype system for evaluation. Our experiments with filters having diverse quality requirements show that this approach can effectively save bandwidth with low CPU overhead. We also analyzed the key factors that affect the performance of our approach.

In the rest of the paper, we first describe in Section 2 the basis for group filtering, and introduce our framework for the group-aware stream filtering in Section 3. In Section **??**, we discuss our evaluation of this concept based on a prototype implementation. We discuss related work in Section 5 and conclude in Section 6 with a description of future work.

2 Two key observations

In this section, we make two key observations about stream filtering for context-aware applications. The observations motivate our "group-aware stream filtering" approach detailed in the next section.

2.1 Quality requirements of stream filtering

The goal of stream filtering is to select an "important" portion from a streaming data source according to the specific needs of an application. The result of this filtering reflects an applications' desirable data quality, which is normally measured as the accuracy, granularity, timeliness, or completeness of the data. For example, an application would like to get a temperature reading of a place whenever the reading has changed by n degrees. This n-degree data granularity requirement can be enforced by a "delta-compression" filter that compresses the streaming data at "delta", in this case n unit, intervals.

2.2 First observation

The first key observation we made about the context-aware applications is that they may tolerate some degree of "slack" in their data quality.

Consider a temperature source and delta-compression filtering for example. Given a time-ordered nine-tuple sequence from the source, $< 0, 35, 29, 45, 50, 59, 80, 97, 100 >$[2], the output that satisfies compression at 50-unit granularity will be $< 0, 50, 100 >$. We recognize that applications may find it harmless to tolerate a small deviation from the ideal compression granularity in the output. For instance, the application may be able to tolerate a maximum of 10-degree "slack" with regard to its ideal 50-degree granularity requirement.

2.3 Second observation

Our second observation is that more than one sequence from a data source can potentially satisfy an application's approximate quality requirements.

In the previous example, if the application tolerates a maximum of 10-degree slack in the 50-degree compression granularity, it is easy to validate that the following sequences satisfy the approximate granularity requirements as well: $< 0, 45, 100 >$, $< 0, 59, 100 >$, $< 0, 50, 97 >$, $< 0, 45, 97 >$, $< 0, 59, 97 >$.

[2]Here we represent each tuple as a single integer; in reality, each tuple may have several fields, but for simplicity we represent each by the value of its "temperature" field since it is that field that is used for filtering.

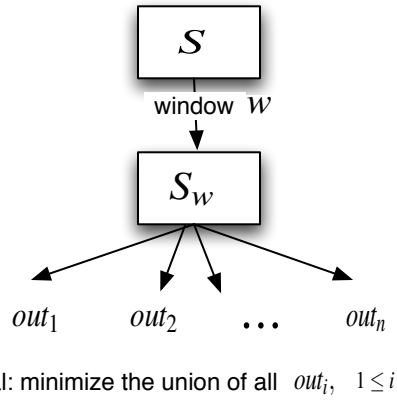

Figure 2: Group-aware stream filtering problem.

2.4 Group-awareness

Let us call the above delta-compression application A. Suppose application B shares the same source as A and tolerates a maximum of 5-degree slack in a 40-degree compression granularity. By the above definitions, it is easy to validate that the following sequences satisfy B's requirements: $< 0, 45, 97 >$, $< 0, 50, 97 >$, $< 0, 50, 100 >$, $< 0, 45, 100 >$.

Individually, A may choose $< 0, 50, 100 >$ as its output; B may choose $< 0, 45, 97 >$ as its output. This makes a total of 5 tuples to output when multiplexing the output streams for multicasting. If A and B are aware of each other's filtering needs, and both decide on, say $< 0, 50, 97 >$, as their individual output, then only three tuples need to be multicast to A and B to satisfy both filtering requirements. In effect, the "group-awareness" reduces the bandwidth demand by two tuples.

3 Framework for group-aware stream filtering

In this section, we formally define the problem the group-aware stream filtering tries to solve and show a general group-aware filtering algorithm whose basic idea we have briefly shown in the previous example.

3.1 Definition of the problem

We formally define the group-aware stream filtering problem as the following optimization problem.

Consider a source stream S and n filters $Op_1, Op_2, ...Op_n$ deployed at the source node for n subscribing applications Application 1, Application 2, ... Application n, respectively. Filters process source data in time-progressive batches or *data windows*. An output of a filter is a time-ordered sequence. For simplicity, we assume the timestamp of each tuple in the sequence is unique and thus the output can be represented as a set of tuples. We define the set that contains all satisfying outputs of Application i based on a data window $S_w \subset S$ as $PotentialOut_i = \{S' | S' \subseteq S_w, S'$ satisfies application i's quality requirements$\}$. The goal of grouped filtering is for each Op_i to pick an element out_i from $PotentialOut_i (i = 1..n)$ such that $|out_1 \bigcup out_2 ... \bigcup out_n|$ is minimized (see Figure 2).

3.2 Framework for group-aware stream filtering

Saving bandwidth is important to satisfy long-running applications' quality needs and for the system to scale well to a large number of co-existing applications that may cooperate for a common mission. Thus, applications deployed in a wireless bandwidth-conscious network are motivated to expose their approximate data needs for the system to reduce the overall bandwidth demand. Each application may reveal "quality-equivalent" candidates for each of its outputs. One way to specify quality equivalence is via a "reference point-based" approach. We define *reference points* as the output that a self-interested filter would normally produce. Then, applications can define a "slack" of a reference point to include all adjacent data points that are "slack" units away from the reference point as its candidate set. For instance, in the 50-degree delta-compression example mentioned in Section 2, the reference points of the 9-tuple sequence are $0, 50, 100$. If the application has a 10-degree "slack", we can identify the candidate replacement for each reference points by computing the contiguous range of tuples before or after the desired reference points, as long as each value is no more than 10 degrees below or above that of the reference point. For instance, 50 now has a candidate set consisting of $45, 50$.

In our framework, applications can declare quality "slack" with distance or membership functions. In the delta-compression example, a numeric temperature difference defines the slack. In a location trace, for example, the distance function may be the Euclidean function involving two or three attributes that describe a location. In general, a distance function can be applied to any state information computed based on multiple attributes of tuples. If a data stream represents observations made by many sensing devices, an application may declare a membership function that defines equivalence of the observations made by different devices if they are close in sensing location, time, or capacity.

We abstract the group-aware filtering process into the following continuous two-stage process at each filter.

1. **First stage: compute a candidate set**: select a set of candidates, i.e. tuples that can potentially satisfy the data quality requirements of the application. Communicate the candidate set to other source-sharing applications via global state.

2. **Second stage: deciding the output**: With reference to the global state, pick a subset of tuples from the candidate set for output. Communicate the choices via global state.

A *group-aware filtering manager* maintains and coordinates the filtering of all the filters in a group. First, it instantiates each filter according to its specification from each application. A filter specification specifies the type and parameters of the filter, and how its internal state should be initiated. Then the manager uses a global object $globalState$ to coordinate the filtering of data stream S with the procedure GROUPAWARE-STREAM-FILTERING shown in Figure 3. The global state mainly consists of 1) the *group utility* of each tuple, which counts how many applications have the tuple in their candidate set, and 2) *data-for-multicast* which records each application's chosen outputs that have not yet been multicast. Each filter uses its $isAdmissible$ (line 4) method to decide whether a tuple is admissable to its candidate set. If so, the tuple is added to the candidate set (line 6), and the tuple's group utility is updated in the $globalState$ (line 8). Next, the manager checks if the filter's candidate set is closed (line 9). If so, referring to the group utility, the filter decides its output (line 11), which is later reported to the $globalState$ (line 13).

3.3 Diverse filters

This two-stage group-aware filtering process supports a variety of filtering needs of monitoring applications.

The delta-compression filters capture a variety of data-granularity needs. In the motivating example, the state used for filtering is the value of the attribute "temperature". If the application is interested in the

GROUPAWARE-STREAM-FILTERING(S)

```
1  while ( (currentTuple ← S.getNextTuple())! = null);
2      do
3          for each filter f in the group
4              do if f.isAdmissible(currentTuple)
5                  then
                       ▷ first stage: get candidates
6                      f.candidateSet.add(currentTuple)
7                      f.state.update(currentTuple)
                       ▷ increment group utility of currentTuple
8                      globalState.groupUtility.increment(currentTuple)
9                  if f.candidateSet.closed(currentTuple)
10                 then
                       ▷ second stage: decide output based on group utility
11                     output ← f.candidateSet.decideOutput(globalState.groupUtility)
12                     f.state.update(output)
                       ▷ add f's decided output to globalState for multicasting
13                     globalState.dataForMulticast.add(output)
```

Figure 3: Group-aware stream filtering algorithm.

changing rate or "trend" of the temperature values, the filter computes the ratio of the change of temperature to the change of the timestamp. If a data stream consists of readings from multiple sensors of similar sensing capacities deployed in close vicinity, an application may be interested in the change of the "averaged" readings from those sensors.

Our framework supports filters beyond the delta-compression filtering theme. Many exploratory data-analysis applications may deploy *sampling* methods to choose a small set of data to derive properties about the whole population. The notion of candidate sets is inherent in many commonly-used sampling methods, such as reservoir sampling, subset-sum sampling and stratified sampling [9]. For example, reservoir sampling chooses a fixed number of samples from a given population. Each tuple in the result can be replaced randomly by another tuple in the population. In this case, the candidate set of each output tuple is the complete batch. Different from delta-compression filters, it chooses multiple (a fixed number) tuples, rather than just a single tuple, from a candidate set. Reservoir sampling can be useful to bound the output bandwidth demands for some applications. For detection-oriented analysis, predicates that recognize interesting patterns can first be applied to the time series to distinguish important data sequences from less important ones, and then a higher sample rate can be applied to the more important data segments. This sampling theme belongs to *stratified sampling* in statistics, as it first decides strata of data with different characteristics and then samples each stratum with a different sample rate. Our group-aware filtering framework supports sampling filters too: sampling filters can increment the group utility of each tuple while building the candidate sets (here, candidate sets are data strata). When sampling the candidate sets in the second stage, sampling methods account for the data already chosen by other applications in the group by seeking tuples with top-n (n is the desired sample size) group utilities.

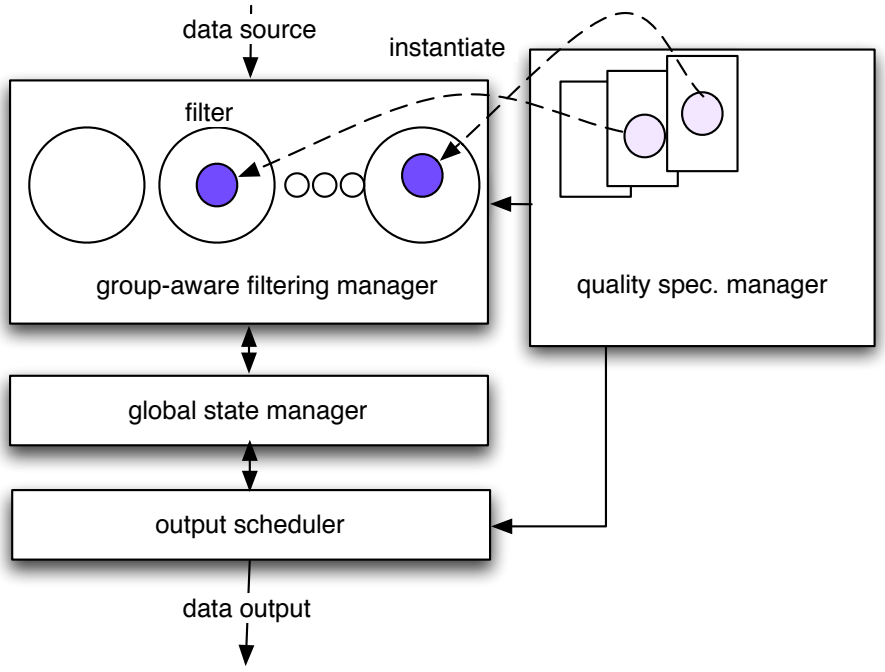

Figure 4: Framework for group-aware stream filtering.

3.4 Software architecture

The software architecture of the framework shown in Figure 4 consists of the following modules: 1) the quality specification manager, which facilitates applications to specify their data quality needs with a library of common predicates, distance functions and member functions, 2) the group-aware stream filtering manager, which coordinates a pool of filters for group-aware filtering, 3) a global state manager that maintains the state information shared by the filters, and 4) an output scheduler that merges the output chosen by each filter into the multicasting format before invoking the overlay multicasting protocol. The scheduler decides when to merge and multicast the filter outputs. It monitors the data filtering progress via the global state manager, and may periodically schedule outputs for multicasting. For instance, the output scheduler may wait for a fixed number of tuples before multicasting. For filters with latency constraints, the scheduler also needs to trigger merging of the output before violating any of the constraints. At the time of the scheduling, it may be the case that not every filter's candidate set is completed. In those cases, the scheduler signals via the global state manager the filters to decide outputs based on their current candidate sets. For simplicity, in our current implementation, the output schedule schedules a group output for every batch of input tuples.

4 Evaluation

The main goal for our evaluation is to see how well group-aware filtering works in comparison to self-interested filtering, with stream data collected from real sensor deployments. We also analyze the key factors that affect the performance of the group-aware filtering.

4.1 Prototype system

We implement and integrate the group-aware filtering prototype with *Solar* [5], a general-purpose data dissemination system developed at Dartmouth College. The core of Solar is a p2p overlay infrastructure where each overlay node supports a suite of data-dissemination services, such as naming, data fusion, and multicasting. We package the group-aware filtering as a new service, working together with Solar's basic services on each overlay node.

Solar uses a content-based publish/subscribe model for flexible and scalable data dissemination. Publishers of context sources in Solar are called "sources" and applications can "subscribe" to sources in Solar to get the desired context information. Solar also allows an application to specify data operators, such as filters, to pre-process the source data.

For our testing, we replay real data traces as Solar sources and let a group of applications subscribe to the sources. Each subscribing application specifies a filter for its processing needs. The group-aware filtering service then deploys, according to a filter's type and quality requirements, a group-aware filter object on the source node. The union of the output of all source-sharing filters is published via Solar's overlay multicasting service to the remote applications.

4.2 Data sources

We chose data from real deployments of sensing devices for which the data stream has a sub-second data rate, so filtering is necessary and saving bandwidth for dissemination of the data is important.

The Networked Aquatic Microbial System (NAMOS) of the CENS project at UCLA [3] deployed embedded and networked sensors in Lake Fulmor for a marine scientific study during August 2006. The water was monitored by an array of thermistors and fluorometers, among others, installed on buoys of the lake. The data traces have data rates of about 10 ms per measurement and contain more than ten thousand measurements. These measurement traces make ideal data sources for our testing. Each NAMOS buoy trace tuple contains six temperature readings (we call them "thermo" readings), one reading from a fluorometer (we call it the "fluoro" reading), a timestamp, and some other weather-related readings. We create a source in Solar that replays the NAMOS buoy trace at about 10 ms per tuple, observing the original time intervals of the trace data.

4.3 Filters for testing

The goal of the NAMOS buoy deployment is to help marine biologists to collect multi-scale high-resolution information, such as the spatial and temporal distribution of the chlorophyll level in the lake, for scientific analysis. Using delta-compression filters or sampling filters is a valid way to enforce multi-scale granularities of the collected buoy data for these applications.

Table 1 lists the types of filters used for our testing. We notate a filter type with a type name followed by a set of parameters enclosed in a pair of parentheses. The first three types, $DC1$, $DC2$ and $DC3$, use the Delta-Compression (DC) theme and they vary from one another by how a candidate set is computed. $DC1$ filters monitor the change of a single attribute. It has three parameters: attribute name, delta and slack. $DC2$ filters monitor the change of the "trend" of an attribute. The trend reflects the change of an attribute over a unit of time. It has three parameters similar to those of $DC1$. $DC3$ filters monitor changes in the average of three attribute values. It has five parameters: three attributes used for averaging, and the delta and slack for delta compression. The last type SS (for Stratified Sampling) differs from the previous three types mainly in the second processing stage: the number of outputs (or sample rate) are determined by the candidate set's sample range. Sample range is the interval between the max and min value within a set of values. If the

[3] http://cens.ucla.edu

filter uses a fixed time interval to segment the time series, the sample range reflects the dynamics of the attribute within the interval. For applications sensitive to the dynamics of state change, it is reasonable to sample a time segment with high dynamics using a high sample rate. Thus, the parameters of SS filters are the attribute of interest, a threshold value that determines whether an interval is highly dynamic, a sample rate (percent of tuples) for more dynamic time segments, and a sample rate for less high dynamic intervals.

4.4 Metrics

The metric we use to measure the benefit of our group-aware filtering approach is the *output ratio*, defined as the total number of tuples output by the group-aware filtering over the total number of tuples output by self-interested filtering. It measures the extent to which the group-aware filtering can reduce the bandwidth demand compared with the self-interested filtering. The lower the output ratio, the more efficient group-aware filtering is in saving bandwidth. When the ratio is 1, that means group-aware filtering did not reduce bandwidth beyond self-interested filtering.

We measure the cost of filtering in terms of the average CPU cost for processing each batch of tuples. We compute the *overhead ratio*, which is defined as the cost of running a group of group-aware filters over the cost of running a group of corresponding self-interested filters. The computer used for this testing is an Apple PowerBook with 1.67 GHz PowerPC G4 and 1 GB memory. Our code is written in Java and ran with Java 1.4.2 on Mac OS 10.4.9.

To get some basic ideas on the performance of group-aware filtering with the NAMOS data traces, we made ten groups of filter specifications, as shown in Table 2.

Each of the first seven groups contains three filters of the same kind; each of the remaining three groups contains three heterogeneous filters. Recall that each filter selects data based on some feature or internal states computed upon the time series. The DC type filters select candidates based on state change between two tuples. The Average State Change (ASC) of a time series is the averaged state changes over the time series. We compute the ASC for the relevant attributes of each DC filter. We set the delta values no smaller than the ASC to prevent the filter from removing nearly no tuples, in which case filters may not be useful for compression. Specifically, we assigned ASC, 2·ASC and a value randomly generated between ASC and 2·ASC to be the delta values of the filters in the delta-compression groups. For example, we measured the ASC of thermo4 in the trace, which is 0.031 degree. Then we set the delta values of the filters in group 4 to be 0.031, 0.062 and 0.048 degree for each filter respectively. The slack value of each delta-compression filter was set to be 50% of the corresponding delta value. Setting the slack more than 50% might cause a tuple to be part of more than one candidate set for a filter.

Figure 5 shows the results. We compute the output ratio for each batch of 100 tuples and then compute the average and median of the output ratios across all batches. For eight out of the ten groups, the average output ratios were less than 80%, i.e. the group-aware filtering can reduce the bandwidth demand to 80% of the bandwidth demand of self-interested filters.

Table 3 shows the average CPU costs of the test, and Figure 6 shows the ratio of CPU time for group-aware filtering to CPU time for self-interested filtering. In general, the group-aware filtering is more complex than self-interested filtering, as it involves group coordination. It is not surprising, therefore, that some of costs were more than double. For groups with simple filters, such as $DC1$ and SS filters, the absolute costs were all below 35 ms per batch. For groups with more complex filters, such as $DC2$ and $DC3$, the cost increased both for group-aware filters and self-interested filters, due to the more complex computations. It may also be due to our unoptimized Java implementation. In all those cases, the CPU cost for processing a batch of 100 tuples were below 700 ms, that is, the cost for processing each tuple was below 7 ms, which is less than the data arrival rate (10 ms) and hence group-aware filtering will not cause congestion in these cases.

We also compare the incurred latency due to filtering with the multicast latency. We set up 5 nodes on

Filter type	Select candidates based on	Decide output
DC1(attrib, delta, slack)	change of attrib between delta--slack and delta+slack	choose any 1 tuple
DC2(attrib, delta, slack)	change of trend(attrib) between delta--slack and delta+slack	choose any 1 tuple
DC3(attrib1, attrib2, attrib3, delta, slack)	change of average(attrib1, attrib2, attrib3) between delta--slack and delta+slack	choose any 1 tuple
SS(attrib, timeInterval, threshhold, highSmplRt, lowSmplRt)	change of timeStamp within timeInterval	choose any n% of the tuples, where n=highSmplRt, if sampleRange(attrib) is no less than threshhold; or n=lowSmplRt, if sampleRange (attrib) is less than threshhold

Table 1: Types of group-aware filters for evaluation.

Group	Filter 1	Filter2	Filter 3
1	DC1(fluoro, 3.012, 1.506)	DC1(fluoro, 7.024, 3.012)	DC1(fluoro,5.000, 2.500)
2	DC1(thermo2, 0.0230,0.0115)	DC1(thermo2, 0.0460, 0.0230)	DC1(thermo2, 0.0315,0.0107)
3	DC1(thermo4, 0.0310, 0.0155)	DC1(thermo4, 0.0620, 0.0310)	DC1(thermo4, 0.0480, 0.0240)
4	DC1(thermo6, 0.0250, 0.0125)	DC1(thermo6, 0.0500, 0.0250)	DC1(thermo6, 0.0345, 0.0172)
5	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0600, 0.0300)	DC3(thermo2, thermo4, thermo6, 0.0452, 0.0226)
6	DC2(fluoro, 11.59, 5.79)	DC2(fluoro, 5.79, 2.89)	DC2(fluoro, 7.50, 3.75)
7	SS(thermo4, 1000, 0.1500, 50, 20)	SS(thermo4, 1000, 0.3000, 50, 20)	SS(thermo4, 1000, 0.2300, 50, 20)
8	DC1(thermo4, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	DC1(thermo5, 0.0300, 0.0150)
9	DC1(thermo4, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	DC2(fluoro, 3.00, 1.50)
10	DC1(thermo4, 0.0300, 0.0150)	DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150)	SS(thermo4, 1000, 0.1000, 90, 50)

Table 2: Specifications for ten groups of filters.

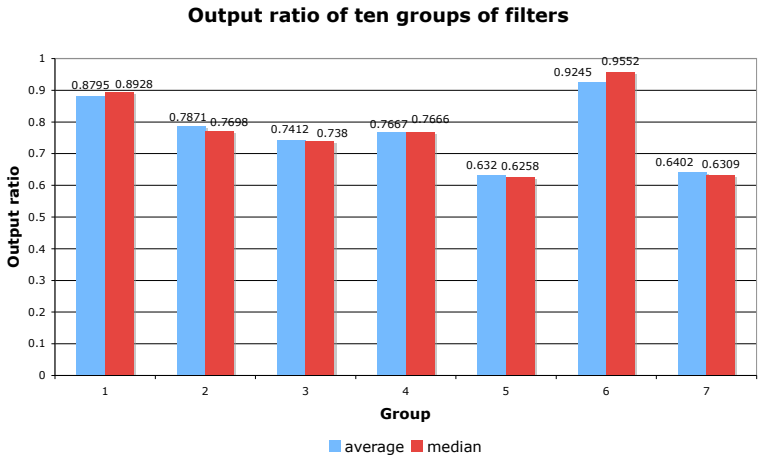

Figure 5: Benefit of group-aware filtering. The smaller an output ratio is, the better the performance is.

Group	Group-aware (ms)	Self-interested (ms)
1	28.03	10.46
2	28.86	10.4
3	22.20	13.36
4	26.45	14.95
5	386.00	260.44
6	684.75	334.50
7	31.00	14.90
8	21.92	11.82
9	32.29	14.35
10	41.76	16.30

Table 3: Average CPU cost per batch of 100 tuples.

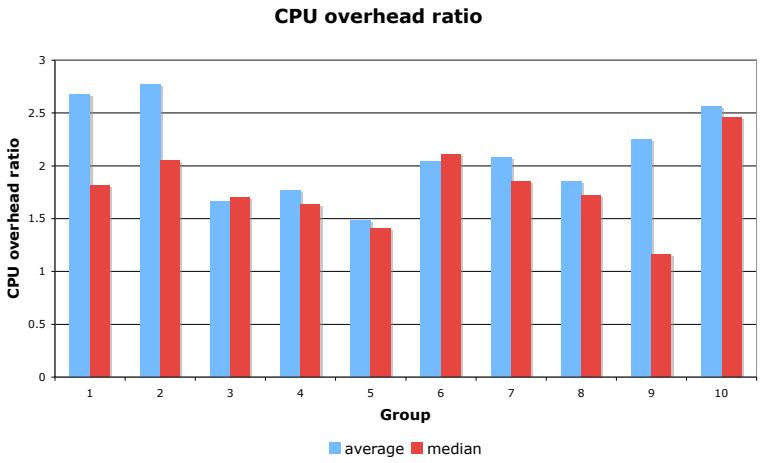

Figure 6: CPU Overhead ratios.

Emulab[4] in a Distributed Hash Table (DHT) ring for Solar's overlay multicast network. The links connecting the nodes were set to be 5 Mbps, a typical throughput for 802.11b networks. We measured the average latency of multicasting the batch of tuples between two nodes to be about 253.41 ms. Note that for wireless networks with effective bandwidth lower than 5 Mbps and with more nodes, the multicast latency for multicasting should be much longer. For simple filters, the incurred filtering latency per batch (< 35 ms for 100 tuples) was negligible compared with the multicasting latency (≥ 253 ms). For complex filters, such as $D2$ and $D3$ type filters, whose incurred latency was longer than the multicasting latency, we can reduce the significance of filtering latency by letting the output scheduler adjust the batch size for earlier outputs. We evaluate how the batch size affected the cost below.

4.5 Factors affecting performance

Next, we look into the factors that affect the performance of group-aware filtering. Surely, source data characteristics directly affect the result of each filter and thus the performance of the group-aware filters. Here, given fixed source data, we focus on the other factors that affect the filtering performance. Those factors include 1) types of the filters in a homogeneous group, 2) filter parameters, such as delta and slack values for delta-compression filters, 3) batch size, 4) number of filters in a group (group size) and 5) the composition of filters in a heterogeneous group.

Filter type of a homogeneous group. In Figure 5, the first seven groups are homogeneous groups of seven different types, with different output ratios. Although the first four groups are all of type DC1, Group 1's output ratio was higher than the other three groups. The main reason is that Group 1 works with the fluoro feature of the trace and the next three groups work with temperature. Group 1 and Group 6 both worked with the fluoro feature of the source trace, yet they computed different internal states and thus the output

[4]http://www.emulab.net is a cluster for distributed-systems research

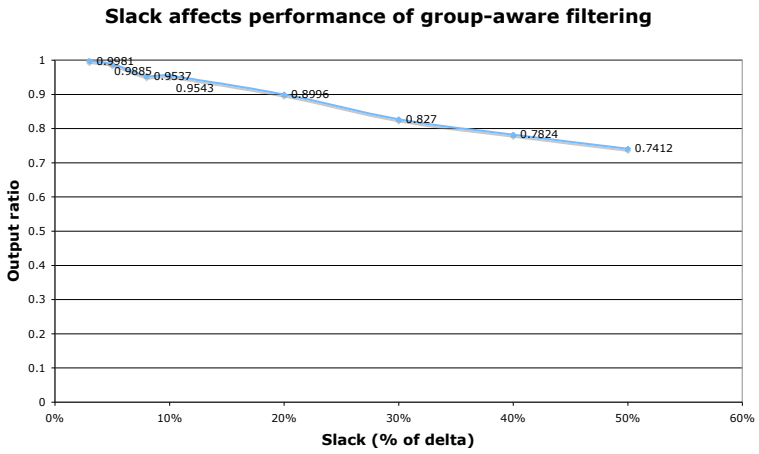

Figure 7: Slack's effect on the performance of DC type filters.

ratios were different: about 10% difference in the output ratio. Thus the attributes that the filters work with fundamentally affect how candidate sets are selected, and thus the output ratio of group-aware filtering.

Slack of a delta-compression filter. Intuitively, when a delta-compression filter has a larger slack, there are potentially more tuples in a candidate set and thus it is more likely to find candidate sets that overlap with other filters, and reduce the combined output set. We retested filters in Group 3 by varying their slacks from 3% to 50% of the corresponding delta values. The results in Figure 7 confirm our intuition. When the slacks were 20% of the corresponding delta values, the output ratio was more than 90%; when the slacks increased from 20% of delta to 50% of delta, the output ratio decreased from 90% to below 75%.

Delta of a delta-compression filter. The delta value of a delta-compression filter affects its filtering stride and thus the Number of Candidate Sets (NCS) it has for a fixed data sequence. In general, the bigger the delta value, the fewer candidate sets can be found. Yet, a change in the delta value means changing the potential overlap between candidate sets. It is equally likely for a decreased NCS of a filter to increase or to decrease the total number of outputs of the group, depending on how the new candidate sets relate to those of the other filters in the group.

We explored this situation with a group of three DC1 filters working with the thermo4 attribute. We fixed the slack values at 0.0155 (or $50\%\cdot$ASC) for all filters. We fixed the delta values for two filters at 0.0930 ($2\cdot$ASC) and 0.01340 ($3\cdot$ASC) and randomly generated a number between 0.0310 ($1\cdot$ASC) to 0.0930 ($2\cdot$ASC) as the delta value for the third filter.

Figure 8 shows the result of the test. We have a few observations. First, the curve is mostly level with a few outliers. For example, the difference of output ratios was less than 5% for the delta values between 0.050 to 0.075. This difference was probably caused by the big slack value of 0.0155; a large portion of the output tuples of the filter whose delta value was changing were covered by the candidate sets of the other two filters. The sudden increase of 20% in the output ratio when the delta changed from 0.045 to 0.049 can

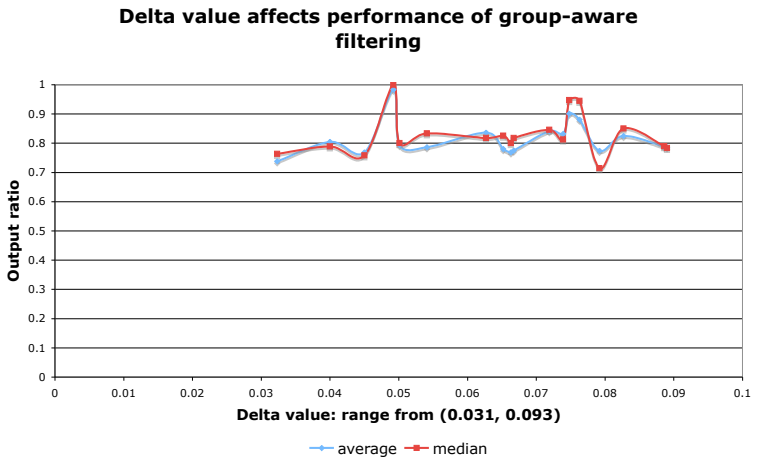

Figure 8: Delta's effect on the performance of DC type filters.

be explained by the fact that many of the output points of the changed filter moved out of coverage of the candidate sets of the other two filters. This caused the total number of output by the group (and the output ratio) to increase dramatically. Another effect of increasing delta value is that the total number of output tuples by the corresponding self-interested filter may decrease and if there is no change in the total output by the group-aware filters, the output ratio may go up.

Group size. The number of filters in a group may also affect the performance of group-aware filtering. We tested groups of DC1 filters working with the thermo4 attribute. For each group size we generated 10 groups of DC1 type filters on thermo4 and we fixed the slack value to be 0.015. We randomly chose delta values from the range of 0.031 (1·ASC) to 0.186 (6·ASC). We averaged the output ratios across the ten groups for each group size. We varied the group size from 3 to 20. Figure 9 shows the results in box-plot, which plots the statistic summary of the median, maximum, 25% quartile, 75% quartile and minimum output ratio of the ten tests of each group size. The circles represent outliers.

Overall there is a downward trend in the median of the output ratios for the results: that is, adding more applications to the group seems to decrease the output ratio, because the increase of the total output tuples due to adding new filters was smaller than the increase in overlap among candidate sets. The large slack value of 0.015 made the candidate sets of newly added filters easy to overlap with those of other filters in the group.

Group size also affected the CPU cost of the filtering. Figure 10 shows a roughly linear increase of CPU cost per batch of 100 tuples when we increased the number of applications in the group from 3 to 20, both for group-aware filtering and for self-interested filtering. The group-aware filtering required about double the CPU cost as self-interested filtering, due to the complexity of group coordination. The CPU cost for any of the groups was less than 61 ms, however, which was low compared with the latency of multicasting.

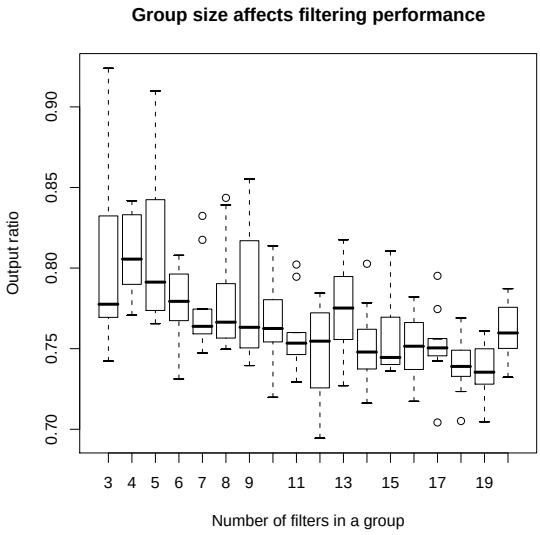

Figure 9: Group size's effect on the performance of DC filters.

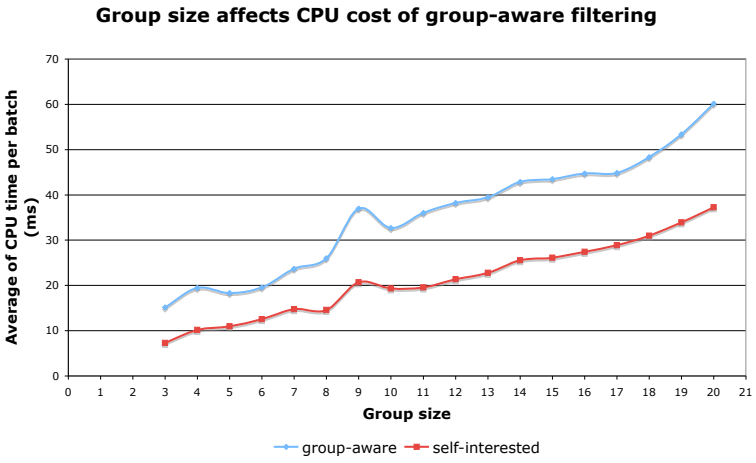

Figure 10: Group size's effect on the cost of DC filters.

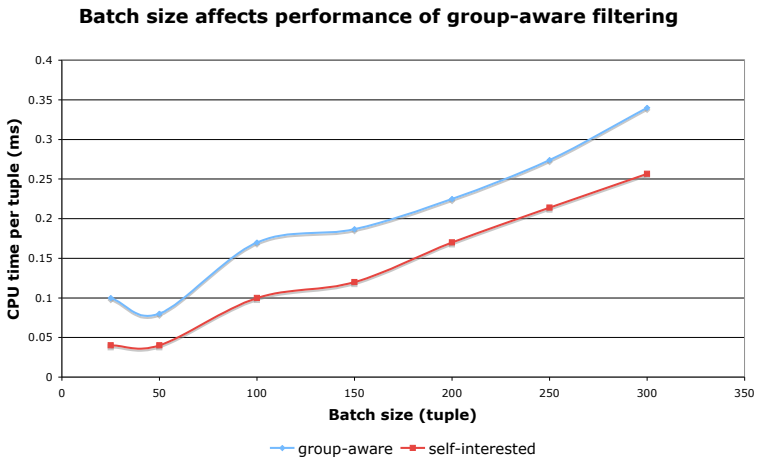

Figure 11: Batch size's effect on the CPU cost for Group 4.

Batch size. Batch size is the number of tuples to process before multiplexing the outputs for multicasting. For simplicity, in our current implementation we fixed the batch size for each test. Here we examine how the batch size affect the performance.

We studied a group of three DC1 filters using Group 4's specification 1, varying the batch size from 25 to 300. Figure 11 compares the CPU time per tuple of the group-aware filtering with the CPU time for self-interested filtering. The group-aware filtering took almost twice as much time as did self-interested filtering. The two curves are roughly linear and roughly parallel. The CPU cost of group-aware filtering increased from 0.04 ms to 0.25 ms when the batch size increased from 25 to 300.

Group composition. The group composition also affected the performance. The last three groups in Figure 5 are heterogeneous groups of filters. All three groups consisted of the same two filters, DC1(thermo4, 0.0300, 0.0150) and DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150), and with a third filter different enough to make the output ratios quite different.

When we created a new group with three filters, DC1(thermo4, 0.0300, 0.0150), DC3(thermo2, thermo4, thermo6, 0.0300, 0.0150) and SS(thermo4, 1000, 90,50), similar in composition as Group 10, we found that the output ratio jumped up to 0.9479 (not shown). The reason is that the different third filter accepted about 90% of all the tuples. One "bad" filter that needs almost all the tuples can immediately diminish the overall benefit of using group-aware filtering.

4.6 Discussion

We get a glimpse of the performance of group-aware filtering from the above experiments with a fairly diverse filters. In most cases, we found that the group-aware filtering could reduce the bandwidth demand below 85% (or lower percent) of the original bandwidth demand of the self-interested filtering. For simple filters, the CPU overhead was low and negligible compared with the latency of overlay multicasting in a

wireless network. We conclude that group-aware filtering is likely to be a valuable approach for saving bandwidth.

Our experiments also reveal that it is, in general, hard to predict the performance of group-aware filtering. The benefit of group-aware filtering ultimately comes from how the candidate sets of the filters intersect with one another, which is subtly determined by the compounded effect of filter types, group composition, and the delta values for the delta-compression filters, as well as the source data. Our experiments confirmed that increasing the slack in a DC-type filter seems always to have a positive effect on the benefit of group-aware filtering.

Our evaluation and analysis point to some interesting directions for improving the system. Among all the factors we described, only the group size, group composition and batch size may be adjustable by the system to enhance the performance. First, we need to monitor the state changes in source data. This might help us to predict which filters have high I/O ratios (defined as the ratio of the filter's input and output). A group that includes filter with high I/O rate means that most of the tuples in the source are needed for the output and the group-aware filtering might not bring much benefit, yet its complexity might bring unnecessary latency to the data. In those cases, it is desirable to separate those "bad" filters from the rest to reduce CPU overhead. Second, we also need to monitor the performance of group-aware filtering to adjust the parameters of the filtering, such as group size. Third, we need to develop strategies to (re)group the filters. Grouping affects the the group size, which negatively affects CPU overhead and in some case may violate the latency constraints of some of the filters. Also, grouping applications according to their network topology may help to save multicasting overhead due to simplified groups. Also, the batch size affects the overhead of group-aware filtering. It is desirable for the output scheduler to dynamically adjust the batch size, such that the incurred latency due to group-aware filtering will not violate the time constraints of the filters. All of these optimizations represent future work.

5 Related work

Bandwidth-reduction mechanisms, such as sampling, summarising, and filtering, have been actively studied in recent years in the systems community [2, 3, 4, 9, 11]. Most of the mechanisms are discussed in the context of a single streaming application. Only a few research efforts have looked into group optimization for streaming applications, but these mechanisms are either based on traditional compiler rewriting techniques, or the simple grouping of stateless filters [1, 6, 7, 10, 12]. When data reduction is based on simple filters, grouping the filters for evaluation of common sub-expressions in the filters has been shown to save computation time [10, 12]. We have different objectives; the goal of our work is to trade CPU time for bandwidth savings. Dynamic code generation is used in Echo [8], a high-performance publish-subscribe system, to generate filters that run 100 times faster than those relying on virtual machines. It would be interesting to see how we can adopt this technique for group-aware filters to meet applications' time constraints.

Our work exploits the semantics of a stream processing application to improve resource management in a dissemination system. IBM's Gryphon [13] also uses the semantics of aggregation functions of the application to compress a sequence of data into a reduced sequence that will have the same effect on applications' states. Zhao et al. [14] propose a case-based language to specify an application's sophisticated processing needs, which identifies what sequences are semantically equivalent so that the system can reorder sequences and compensate for data lost in the network. We have a different goal than either project: rather than using a complicated language to describe the needs, we opt for a simple framework with libraries of distance functions and member functions to let applications describe the approximate nature of their data requirements.

Johnson et al. [9] summarized a general structure for sampling operators. The structure also contains stages, as we propose. If we see our group-aware filtering from a sampling point of view, our algorithm is a

special kind of sampler in that it picks an output from a candidate set of outputs. But our process involves coordination across a group of applications, which never occurs in Johnson's single-application oriented sampling.

6 Conclusion and future work

There is a well-recognized challenge faced by wireless data dissemination systems: how to satisfy the high-volume data acquisition needs of outdoor monitoring applications with bandwidth-limiting wireless transport. Resolving this challenge requires an aggressive approach beyond traditional bandwidth-saving methodologies, such as multicasting and self-interested filtering. At the application level, we exploit the the approximate nature of data acquisition requirements, and optimize the output for a group of context source-sharing applications to reap further bandwidth saving with multicasting. We derive a general two-stage process for this "group-aware filtering" approach and our framework supports a wide variety of data-filtering needs.

Our evaluation of a set of heterogeneous filters based on real sensor traces shows encouraging results and reveals some key factors that affect the performances of the group-aware filtering. The performance is affected by several factors, such as group composition, group size and batch size that may be adjustable by the system to enhance filtering performance. Our future work mainly includes 1) developing an online performance monitoring service for group-aware filtering, 2) developing strategies to dynamically group and regroup filters based on topology or CPU cost, and 3) developing dynamic policies in the output scheduler to adjust the CPU overhead of group-aware filtering to meet the time constraints of the filters.

7 Acknowledgments

The authors want to thank Guanling Chen, Kazuhiro Minami, members of the ARTEMIS project at the Institute for Security Technology Studies, and other members of Center for Mobile Computing at Dartmouth College, for their valuable suggestions and feedback. This research program is a part of the Institute for Security Technology Studies, supported under Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate, and by Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security or the US Department of Justice.

References

[1] Suresh Aryangat, Henrique Andrade, and Alan Sussman. Time and space optimization for processing groups of multi-dimensional scientific queries. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 95–105, New York, NY, USA, 2004. ACM Press.

[2] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *SODA '02: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 633–634, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[3] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Sampling algorithms: lower bounds and applications. In *STOC '01: Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, pages 266–275, New York, NY, USA, 2001. ACM Press.

[4] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 263–274, New York, NY, USA, 1999. ACM Press.

[5] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *MobiQuitous '04: Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems*, pages 246–255. ACM Press, 2004.

[6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, New York, NY, USA, 2000. ACM Press.

[7] Reynold Cheng, Ben Kao, Sunil Prabhakar, Alan Kwan, and Yicheng Tu. Adaptive stream filters for entity-based queries with non-value tolerance. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 37–48. VLDB, 2005.

[8] Greg Eisenhauer, Karsten Schwan, and Fabian Bustamante. Publish-subscribe for high-performance computing. *IEEE Internet Computing*, 10(1):40–47, 2006.

[9] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 1–12, New York, NY, USA, 2005. ACM Press.

[10] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 49–60, New York, NY, USA, 2002. ACM Press.

[11] Don P. Mitchell. Consequences of stratified sampling in graphics. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 277–280, New York, NY, USA, 1996. ACM Press.

[12] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 563–574, San Diego, California, June 2003.

[13] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998.

[14] Yuanyuan Zhao and Rob Strom. Exploiting event stream interpretation in publish-subscribe systems. In *PODC '01: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, New York, NY, USA, 2001. ACM Press.