Event Dissemination via Group-aware Stream Filtering

Ming Li and David Kotz
Department of Computer Science, Dartmouth College
Hanover, NH 03755
USA

Abstract

We consider a distributed system that disseminates high-volume event streams to many simultaneous monitoring applications over a low-bandwidth network. For bandwidth efficiency, we propose a group-aware stream filtering approach, used together with multicasting, that exploits two overlooked, yet important, properties of monitoring applications: 1) many of them 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. We provide a general framework that treats the group-aware stream filtering problem completely; we prove the problem NP-hard and thus provide a suite of heuristic algorithms that ensure data quality (specifically, granularity and timeliness) while preserving bandwidth. Our evaluation shows that group-aware stream filtering is effective in trading CPU time for bandwidth savings, compared with self-interested filtering.

1. Introduction

Recent years have seen data-intensive applications that feed on near-real time "context" information, such as location, environmental status, and surrounding resources, collected from distributed data sources leveraging sensor networks. At the scene of a large fire, we imagine, fire-spread prediction require sub-second updates on temperature, wind speed and direction from the sensor networks deployed near the fire; command-and-control applications need frequent updates on first responders' locations. Systems that disseminate data for those monitoring applications often use wireless networks for rapid deployment and cost-effectiveness. In the emergency-response scenario, such an infrastructure can be a wireless mesh network formed by computers on police cars or fire trucks on the scene. It is well recognized that the effective bandwidth of a wireless network is usually much lower than its link capacity, and that the high-volume data acquisition needs of monitoring applications may push the envelope of the bandwidth-constrained network.

Two classical approaches often used in event dissemination for saving bandwidth consumption are multicast and in-network filtering. Multicast eliminates redundant communications at network links. In-network filtering pushes the computation close to the data source to discard unnecessary data before transporting them. However, many applications that feed on the same data source may use data differently and thus require different filters. We can combine filtering and multicast by multiplexing the results of application-specific filters at the source node before multicasting. Figure 1 shows this process: two applications, A and B, share the same data source D, but each application's filter selects a different subset on the source node. 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.

Filtering is a common way to ensure data granularity, an important quality measure for the level of details of domain-specific features embedded in a source data stream. Using more aggressive filtering can reduce the data bandwidth consumed by applications, but it may also degrade the data granularity. When the resulting bandwidth consumption of the filter-then-multicast approach reaches the limit of the network, rather than resorting to more aggressive filters that may severely reduce the data quality, we propose a *group-aware stream filtering* approach, combined with multicast, to explore further bandwidth-saving opportunities within the same level of data granularity required by applications.

Our group-aware stream filtering approach 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

DEBS '08, July 1-4, 2008, Rome, Italy

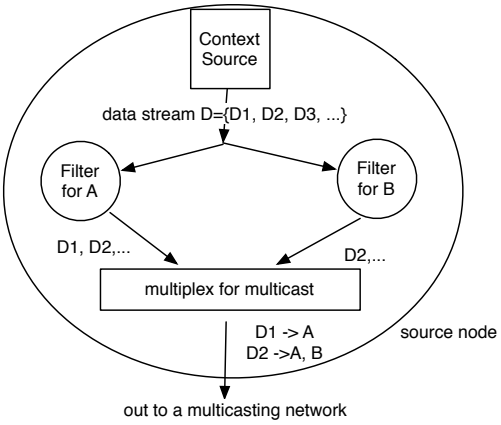

Figure 1. Filtering for multicasting.

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. This paper makes the following key contributions:

- This is the first paper that thoroughly treats the group-aware stream filtering problem; we provide a suite of heuristic algorithms, which ensure data timeliness and data granularity. Thus our approach is quality-managed.
- We provide a general framework of group-aware filtering to support a variety of filtering needs.
- We built a prototype system for evaluation. Our results, based on real-world data sets, show that group-aware filtering can effectively save bandwidth with low CPU overhead when compared with self-interested filtering.

In the next section, we describe the foundation of group-aware filtering. In Section 3, we formally define the problem and introduce the framework and algorithms for group-aware stream filtering. In Section 4, we show that our framework is extensible to support diverse filters. In Section 5, we evaluate our approach with a prototype system. We discuss related work in Section 6 and summarize in Section 7.

2. Two key observations

We base our filtering approach on two key observations about data-quality requirements of monitoring applications. Data quality is normally measured as the *accuracy*, *granularity*, *timeliness*, and *completeness* of the data. Implications of data quality at different parts of the data acquisition process may be different. For filtering, ensuring accuracy and completeness may mean that filters must not tamper with the input data (enforcing accuracy), and that filters must output all tuples in the input data stream that satisfy applications' needs (enforcing completeness). We assume that the chosen filters can always ensure these two qualities. The filters' main job is to select an appropriate subset of input data that meets the applications' data granularity requirement. 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* (DC) filter that removes values that have changed less than n units from the filter's previous output, in effect compressing the stream data at "delta", in this case n, units. (We consider other types of filters in Section 4.) The higher the data granularity is (in the case of DC filters, the lower the "delta" interval), the more output a filter should normally produce. Data granularity thus directly affects bandwidth consumption. The timeliness requirement at the filter can be measured by the amount of delay introduced by filtering. The faster a filter processes and outputs the data, the more timely is the data delivered to applications.

First observation. *Monitoring applications 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, $\{0, 35, 29, 45, 50, 59, 80, 97, 100\}$,[1] the output that satisfies compression at 50-unit granularity is $\{0, 50, 100\}$. 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. We denote such filters as a *(slack, delta) Delta-Compression filter,* which selects data at delta-unit with slack-unit of quality deviation.

Second observation. *There may exist multiple sequences from a data source that 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 each also satisfy the approximate granularity requirements: $\{0, 45, 100\}$, $\{0, 59, 100\}$, $\{0, 50, 97\}$, $\{0, 45, 97\}$, $\{0, 59, 97\}$, as 45, 59 are close-by tuples within 10-degree deviation from "ideal" output 50 after the output 0, and 97 from "ideal" output 100 as a third output.

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: $\{0, 45, 97\}$, $\{0, 50, 97\}$, $\{0, 50, 100\}$, $\{0, 45, 100\}$.

[1] 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.

Individually, A may choose $\{0, 50, 100\}$ as its output; B may choose $\{0, 45, 97\}$ as its output; there are thus 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

In this section, we prove the group-aware stream filtering problem NP-hard and provide a framework based on a suite of heuristic algorithms to solve the problem approximately.

3.1 Problem definition

We assume that the input data stream is a time-ordered continuous sequence of tuples. Each tuple consists of several attributes, including a unique time stamp. We assume that the output of a filter is a subset of its input data. We abstract our group-aware stream filtering method into two stages: In the first stage, *compute candidate outputs*, each filter processes the input stream and computes a set of candidate outputs; in the second stage, *decide on final outputs*, an output decider chooses a candidate output for each filter for multicasting. For a continuous stream, group-aware filtering iteratively goes through these two stages, processing one segment at a time.

Reference-based candidate sets. For the first stage, there exist many domain-specific ways for quality-slack tolerating filters to compute the candidate outputs. Here we introduce a *reference-based approach* to find candidate outputs for filters. The idea is for the filter to compute a *candidate set* for each tuple that the self-interested filter would select. We call the tuples that would be chosen by a self-interested filter *reference tuples*. Choosing any tuples from the candidate set corresponding to a reference tuple would be quality-equivalent to choosing the corresponding reference tuple for the output. Figure 2 shows how a Delta Compression (DC) filter that can tolerate 10-unit slack in 50-unit compression can select a vicinity of tuples around the reference tuple, here tuple 50, that are no more than 10 units from the reference tuple, to form its candidate set. In this case, tuples whose values are 45, 50, 59 are within the 10-unit vicinity of, and contiguous with, the reference tuple 50, and thus make the candidate set. We assume for now that every candidate set is finite or *closable*.

The group-aware filters with reference-based candidate sets exhibit the following properties. First, a group-aware filter has to choose all candidates of a reference output before choosing any candidates of its next reference output. Second, a group-aware filter is able and obligated to finish choosing candidates of an output, if the system asks it to do so (for timeliness, as we show later in the section). Finally, a group-aware filter computes the candidates for an output in an on-line fashion. It includes the possibility for a filter

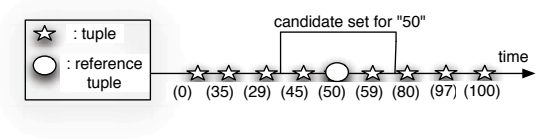

Figure 2. Candidate set of a reference tuple.

to adjust the set of candidates for an output before moving on to computing the set for the next output.

Definition 1. *A time cover, TC_i, of a candidate set i is [min $\{\forall t_j | t_j$ is time stamp of tuple j in candidate set $i\}$, max $\{\forall t_j | t_j$ is time stamp of tuple j in candidate set $i\}$]*

Axiom 1. *Time covers of a filter's candidate sets do not intersect.*

We assert this requirement to ensure that candidate tuples remain in temporal order; that is, if a reference tuple A's time stamp is smaller than reference tuple B's, we make sure the time stamp of any of the candidates for A is smaller than that for all the candidates for B. In a delta-compression filter, the axiom requires that the quality slack is less than half of the delta value, which is normally desirable.

Problem Definition 1. *Given an input stream segment S, n filters $F_1, F_2, ..., F_n$ in the group, and a collection C containing all candidate sets produced by the filters. The objective is to pick a tuple o_j from each candidate set in C, such that the set Output $= \bigcup_j \{o_j\}$ has minimal size.*

Theorem 1. *Group-aware stream filtering is NP-hard.*

Proof: We prove this property by reducing the problem to the minimum hitting-set problem, which is a classic NP-hard problem [8]. Consider a special instance of the group-aware filtering problem in which each filter F_i has exactly one reference point and thus exactly one candidate set $cands_i$ for input stream S. Suppose we have n filters, so there are n candidate sets to choose output from. Since each candidate set is a subset of the tuples in S, this problem has a solution if and only if the minimum hitting-set problem with these n sets has a solution, that is, the output O of the minimum hitting-set problem makes sure that every of the n sets intersects with (or "hits") O, and O's size is smallest among all solutions. □

The minimal hitting-set problem has been studied extensively in the computer science literature. It is proved that the greedy algorithm produces a $\rho(n)$ approximation to the

optimal solution [8], where $\rho(n) = H(max\{|C| : C$ is a set in the hitting-set problem$\})$, and where H is a harmonic function, and n is the total number of sets in the problem. We can apply this bounded approximation algorithm directly to the group-aware filtering problem.

3.2 Region-base group-aware filtering

Notice the problem we defined assumes that the input is a finite-length time series. For a continuous event stream that is potentially infinite in length, we consider a group-aware filtering optimization problem for all its finite prefixes of a time-ordered input data sequence.

For a long stream, it is not time-efficient, if not impossible, to collect all the data before applying filtering algorithms. So we consider the problem of whether there exists a way to segment the input time series in such a way that the segmentation does not affect the optimality of the solution. Here we propose *region-based segmentation* for applying minimum hitting-set algorithms.

DEFINITION 2. *If A and B are candidate sets from two filters, and the time covers of A and B intersect, we say A and B are connected.*

DEFINITION 3. *If A and B are connected candidate sets, and B and C are connected candidate sets, we deem A and C to be connected.*

DEFINITION 4. *A region is a maximum family of candidate sets such that each set is connected with every other in the family.*

DEFINITION 5. *A time cover for a region is the union of all time covers of the candidate sets contained in the region.*

Figure 3 shows that there are two regions based on three DC filters' candidate sets: $region_1 = \{$cands1-1, cands2-1, cands3-1$\}$, $region_2 = \{$cands1-2, cands2-2, cand3-2, cands1-3, cands2-3$\}$. Each filter continuously compute its candidate sets one after another. In this example, we assume that the closure of a candidate set is signaled by the first tuple that is not a candidate. Thus, cands2-1 is closed when tuple 35 comes, as 35 is more than 5 units away from the reference tuple 0. Filter B now anticipates the next reference tuple to be at least 40 and it admits tuple 35 into its next candidate set as the tuple is within 5 units away from 40. When tuple 29 comes, it is not qualified as a candidate tuple, and it also invalidates tuple 35's candidacy, as in this example we assume that a candidate set is made of tuples that comes consecutively in time. When tuple 45 comes, it is at least 40 unit away from the previous reference tuple and thus is admitted into B's candidate set. B admits tuple 50 and closes the candidate set when tuple 59 comes. Note that before a candidate set is closed, a filter has the

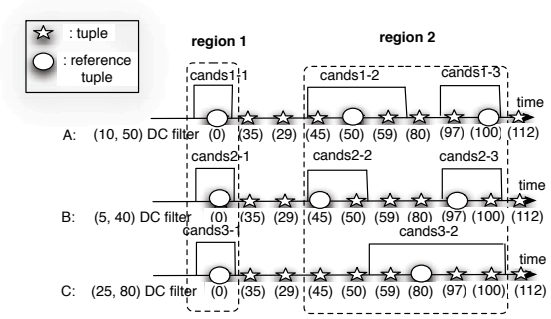

Figure 3. Two regions for three DC filters.

ability to adjust its current candidate set by removing invalid candidates, for instance when a filter come cross the real reference tuple in a candidate set, or find unqualified tuples. It is easy to verify that adding any candidate set outside a region to the region will invalidate the region, as the added candidate set is not connected with the rest of the sets in the region.

AXIOM 2. *Different regions' time covers do not intersect.*

Proof: we prove it by contradiction. Suppose the time covers of two regions, A and B, intersect. It is easy to see that at least one candidate set, say $cands_0$ in B, is connected with a candidate set in A. Then adding a new candidate set $cands_0$ to A will still make A a region, which directly contradicts the assumption that A is a maximum collection of the connected candidate sets. □

THEOREM 2. *Given an input time sequence S, applying divide-and-conquer approach for the group-aware filtering to each region in S will not affect the optimality of the solution.*

Proof: We need to prove that a set-union of the optimal solutions from each region on the input stream S is an optimal solution for S. We prove it by contradiction: that is, we suppose the opposite is true: given a total of n regions on S, and each region has an optimal solution O_i, the cardinality of the optimal solution O' of S is smaller than the size of the set-union U of all O_i, thus U is not an optimal solution for S. Now we divide O' into n distinctive subsets such that each subset is a group-aware filtering solution on each region, that is, each subset is a hitting set of a region.

To find such n subsets, we can first initialize n empty auxiliary sets, one for each region; then, for each tuple in O' that is contained by one of the candidate sets in a region, we put it in the auxiliary set of that region. We can see each tuple fall into exactly one auxiliary set; otherwise if a tuple belongs to two auxiliary sets, then there must be two candidate sets from two different regions containing the tuple,

which means that the two candidate sets are connected and thus belong to the same region, which contradicts the assumption that they are from two different regions. In the end, we get n distinct subsets of O' in the auxiliary sets. We can prove that each auxiliary set is a hitting-set solution to its corresponding region. We prove it by contradiction. Suppose the opposite is true: that is, at least one candidate set in a region does not intersect with ("hit") the auxiliary set corresponding to the region. We know none of the other auxiliary sets hit this candidate set, otherwise there must be a candidate set in another region that intersects with this candidate set, which means that they are connected and are in the same region, which reaches a contradiction to our assumption.

As the size of O' is smaller than that of U, there must be at least one of the n subsets of O' whose size is smaller than that of the optimal solution O_j of that region, which contradicts the optimality of O_j for the region. □

For heuristics-based algorithms that find sub-optimal solutions for the group-aware filtering problem, region-based segmentation preserves the approximation ratio of the solution as shown in the theorem below (see [10] for the proof).

THEOREM 3. *Region-based segmentation preserves the maximum approximation ratio of a heuristics-based algorithm. That is, given an input source segment S, which is segmented into n regions, if a heuristics-based algorithm has approximate ratio r_1, r_2,..., r_n on each region respectively, the approximation ratio r of the algorithm on the overall segment S satisfies the property that $r \leq \max(r_1, r_2, \cdots, r_n)$.*

Now we introduce REGION-BASED-GREEDY-FILTERING, a region-based greedy algorithm, for a continuous stream S in Figure 4. First, assume that we have instantiated 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 and updated. We use a global object *globalState* to coordinate the filtering. The global state mainly consists of 1) the *group utility* of each tuple, which counts the number of filters that have included the tuple in their candidate set, and 2) the current *region* that keeps track of the connected candidate sets since the last region. Each filter uses its *isAdmissible* (line 3) method to decide whether a tuple is admissible to its candidate set. If so, the tuple is added to the filter's candidate set (line 5), and the tuple's group utility is incremented in the *globalState* (line 7). A filter's *isAdmissible* method may trigger the filter to find the next reference tuple as internal state for admitting its candidate tuples. Next, if the filter finishes computing the current candidate set (line 8) when detecting that the current tuple does not belong to the current candidate set, the filter's current candidate set is *closed*. It then checks whether all connected candidate sets are closed. This check is done at the *globalState*, which keeps track of currently closed candidate sets not included in the previous region and tracks the group utilities of each tuple. If the utility of any tuple in a closed candidate set is greater than the number of currently closed candidate sets, then the region is not closed, as there must be a not-yet-closed candidate set admitting this tuple (line 10). When the current region is closed, it consists of all the closed candidate sets that are connected. Next, we apply a greedy hitting-set algorithm, GREEDY-HITTING-SET in Figure 5, to the current region (line 12) and send the solution for multicast (line 13). The solution contains a set of tuples chosen from the region that have high group utilities and hit all candidate sets in the region.

GREEDY-HITTING-SET (in Figure 5) picks the tuple with the highest group utility (line 3). If multiple tuples have the same highest utility, we use tuples' time stamps to break the ties and choose the tuple with the latest time stamp to favor time freshness. Then, remove all the candidate sets that contain the chosen tuple (line 5). The group utility of any tuple included in the removed candidate sets is decremented by the number of removed candidate sets containing the tuple (line 6). The algorithm then greedily picks the next tuple with the highest utility and the same hitting-set process continues until no candidate set is left to be hit. The chosen tuples constitute the solution.

In the previous example with three DC filters A, B, and C (see Figure 3), when the tuple 35 comes, cands1-1, cand2-1, and cand3-1 are closed as the tuple 35 is more than the tolerable slack away from the reference tuple for each filter. Region 1 closes at the moment when previously open candidate sets contained in the region are now all closes. The greedy algorithm then runs on the closed region and the tuple 0 is output to all three filters. After Region 1, Region 2 starts when the group finds the first tuple whose utility is not 0. Region 2 ends when all its contained candidate sets are closed, when tuple 112 comes. By running the greedy hitting-set algorithm on the region, tuple 100 is chosen as an output first as it is one of the tuples with the highest group utility. That is, cands1-3, cands2-3, cands3-2 are "hit" by tuple 100. Next, tuple 50 is chosen, as it has the next highest group utility. cands1-2 and cands2-2 are both hit by tuple 50. Now, all candidate sets have been hit in Region 2. Thus, the outputs for Region 2 are tuple 100 for filter A, B and C, and tuple 50 for filter A and B.

3.3 Region's timely cuts

The region-based group-aware algorithm computes the smallest input stream segment to apply the hitting-set algorithm so that the optimality of the solution will not be affected. Beyond preserving bandwidth, we aim to ensure data timeliness as well.

Long candidate sets affect the timeliness of the output, because a region has to wait for all its member candidate sets to close before choosing outputs. In the case of a delta-compression filter, after admitting a tuple in the candidate set, it waits for the first tuple that does not fall into the valid range for this candidate set to close the current candidate set. If the stream data changes little, and the filter's quality slack is relatively large, the candidate set can grow long, which affects the timely outputs of all its connected candidate sets.

```
REGION-BASED-GREEDY-FILTERING(S)
 1  while ( (currentTuple ← S.getNextTuple()) ! = null)
 2  do for each filter f in the group
 3          do if f.isAdmissible(currentTuple)
 4                  then ▷ first stage: admit candidates
 5                       f.candidateSet.add(currentTuple)
 6                       f.state.update(currentTuple)
                         ▷ increment group utility of currentTuple
 7                       globalState.groupUtility.increment(currentTuple)
 8                  if f.candidateSet.closed(currentTuple)
 9                     then globalState.addClosedCandidateSet(f.candidateSet)
            ▷ second stage: if current region is ready, decide output based on globalState
10          if (region ← globalState.getCurrentRegion()) ! = null)
11             then ▷ apply greedy algorithm to the region to decide the output
12                  output ← GREEDY-HITTING-SET(region, globalState.groupUtility)
                    ▷ multicast output
13                  multicaster.send(output)
```

Figure 4. Region-based greedy algorithm for group-aware stream filtering.

```
GREEDY-HITTING-SET(region, groupUtility)
1  resultSet.init(∅)
2  while (region.hasMoreCandidateSets())
   ▷ greedily pick the tuple with max groupUtility; use time stamp to break ties, if there are any
3  do maxUtilityTuple ← getMax(groupUtility)
4          resultSet.add(maxUtilityTuple)
           ▷ get all the candidate sets that are hit by maxUtilityTuple
5          hitCandidateSets ← region.removeAllCandidateSetsHitBy(maxUtilityTuple)
           ▷ decrement groupUtility for each tuple in the hitCandidateSets
6          groupUtility.decrUtilityFor(hitCandidateSets)
7  return resultSet
```

Figure 5. Greedy hitting-set algorithm.

Here we propose a mechanism, *cuts*, to curb the computation of long candidate sets according to filters' time constraints. We assume that each filter specifies a maximum tuple delay for group-aware filtering and we simply use the minimum of all the time specifications, we call it the *groupTime*, to enforce the data timeliness for the group. To derive the time cover of a region that satisfies *groupTime*, we build a latency model based on on-line observations of the most recent ten regions' performance, specifically the correlation between region sizes and CPU time for computing the regions and choosing output for the regions. From our experiments, we found that a linear model was a pretty accurate fit. The last tuple in a region should not have timestamp that exceeds $(groupTime - intercept)/slope$, where *intercept* and *slope* are coefficients of the current linear time model.

To enforce timely cuts to our previous algorithm, we extend it to check the time constraint after each filter finishes processing the new input tuple (after line 7 in Figure 4). Then, if the time constraints are about to be violated if we wait any longer, we force all open candidate sets to close. These closures will make the current region close automatically and then we can apply the GREEDY-HITTING-SET to choose the output, as before. Finally, after line 13 in Figure 4, we let the *globalState*, which keeps track of CPU time for computing a region, update the time model to compute a new group time constraint, which will be used in the next region.

It should be easy to see that cuts may reduce the sizes of candidate sets and thus reduce the likeliness of overlapping candidate sets, which may reduce the bandwidth-saving performance of group-aware filtering. Nevertheless, we can prove that the worst case for group-aware filtering algorithms with timely cuts is that each candidate set contains only one tuple and thus it is no different from self-interested filtering. Thus, although a timely cut may affect the bandwidth-saving performance of group-aware filtering, it performs no worse than self-interested filtering in terms of bandwidth consumption.

3.4 Stateful candidate sets

The above algorithms compute candidate sets based on reference tuples that are chosen by the self-interested filters with predicates. In other words, computing a filter's current candidate set does not depend on the chosen output of its previous candidate set. We call this *stateless computation of candidate sets* for a filter. For some applications, an alternative semantics for computing a candidate set is to base its reference on the chosen output of the previous can-

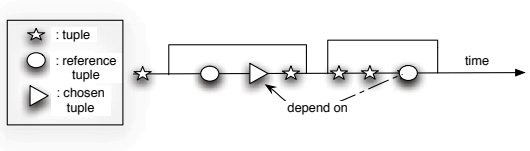

Figure 6. Stateful candidate sets.

didate set. We call this *stateful computation of candidate sets*, and call the candidate sets *stateful candidate sets*.

For stateful candidate sets, the filter needs to choose the output as soon as its current candidate set closes, as the reference for the next candidate set depends on the chosen output (Figure 6). We use group state to track already-chosen tuples of each stateful candidate set in addition to the group utilities of tuples, and propose the following two heuristics for choosing output tuples from stateful candidate sets: **(1) choose the tuple that has been chosen by other filters**, and **(2) choose the tuple that has the highest group utility**.

The first heuristic takes precedence over the second heuristic. Both are subject to the tie-breaking rule, preferring the more recent tuple. After a filter chooses a tuple from a candidate set, the group utilities of all tuples in its candidate set are decremented by 1. Group state will keep track of the tuples chosen by each filter. If there are stateless filters in the group, identifying regions is still useful, as it is the earliest possible time to multicast decided tuples that have not yet been output in the region, when a region closes. In that case, we can still apply the greedy hitting-set algorithm upon the time the region is closed, only that the stateful filters' candidate sets become singleton sets with one chosen tuple in each. The logic for computing regions and timely cuts is the same as in the previous algorithm.

3.5 Output Strategy

There are several output strategies we can use to enforce different output patterns. First, by computing regions, we get the *earliest possible* time for output tuples of a region without hurting the optimality of the solution. Second, by enforcing group time constraints, we get the *earliest possible subject to group time constraint* output pattern. Third, filters may opt for a *batched output* pattern, that is, for a fixed-sized (time or tuple) batch of the input stream, select and output tuples.

In the case of a group with all stateful filters, it may be desirable to output tuples at the time each candidate set is closed, if the applications can tolerate disordered output within the predefined time frame. We call this *per-candidate set* output pattern. The benefit of using this pattern is that the delay of average tuple is less than that with a region-based earliest possible output strategy. The downside of it is that it may cause disorder in the output for the candidate sets in a region.

4. Extensible framework

Group-aware filtering supports a variety of filters beyond simple delta-compression filters.

Filters of special interest to many exploratory data-analysis applications are *sampling* filters, which derive interesting properties by choosing a small set of data from a 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 whole data sequence in a predefined window. 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*, as it first decides strata of data with different characteristics and then samples each stratum with a different sample rate.

Our framework is general enough to support those sophisticated filtering requirements. It supports the same two-stage process as with delta-compression filters. At the first stage, we allow each filter to extend the basic group-aware filter by implementing an *isAdmissible* method to apply domain-specific functions (perhaps with non-trivial states) in candidate admission. At the second stage, group-aware filtering chooses a required number of outputs from each candidate set. The greedy algorithm for region-base filtering now needs to "hit" a candidate set k times (i.e., the final output intersects the candidate set with at least k tuples), where k is the total number of required outputs for the candidate set, before removing the set from the set-hitting process. For stateful candidate sets, the second rule for choosing the outputs of a candidate set becomes choosing tuples with the top-k ($k \geq 1$) highest group utilities. This extended group-aware filtering problem is also NP-hard, as it is more general than the previous problem, but the heuristics in our framework work well. We have implemented these extensions for diverse filters, including stratified sampling filters. We provide a library of customizable filters and distance or member functions to facilitate applications to specify their filtering needs.

5. Evaluation

The main goal for our evaluation is to see how well group-aware filtering works in comparison to self-interested filtering, in terms of network bandwidth consumption and its effect on data timeliness.

5.1 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 in which 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 for pre-processing the source data.

For our testing, we replay real-world 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. To compute end-to-end latency based on time stamps, we deploy the subscribing applications on the same node as the data source to eliminate time skew in a network. Here we assume the real end-to-end latency is the time difference we will measure between a tuple published from the source and the time it arrives in an application, plus a constant number that captures overlay multicasting cost. In past deployments of Solar in a small (7-node) overlay network in Emulab,[2] Solar's overlay multicasting delay was about 130 ms. This paper does not focus on the network aspects of group-aware filtering and we do not measure network behavior while performing group-aware filtering. We thus measure the performance on the node where stream data are filtered. The source node is an Apple Powerbook with 1.67 GHz PowerPC G4 and 1 GB memory. Our code is written in Java and ran with Java 1.5.0 on Mac OS 10.4.9.

5.2 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 fluoro-meters, among others, installed on buoys of the lake. The data traces have data rates of 100 measurements per second 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 *tmpr* readings), one reading from a fluoro-meter (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.

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

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

GROUP NAME	FILTER
DC_Fluoro	DC(fluoro, 0.0301, 0.0150)
	DC(fluoro, 0.0702, 0.0301)
	DC(fluoro, 0.0500, 0.0250)
DC_Hybrid	DC(fluoro, 0.0702, 0.0100)
	DC(tmpr2, 0.0460, 0.0153)
	DC(tmpr4, 0.0310, 0.0103)
DC_Tmpr	DC(tmpr4, 0.0310, 0.0155)
	DC(tmpr4, 0.0620, 0.0310)
	DC(tmpr4, 0.0480, 0.0240)

Table 1. Specifications for groups of filters.

5.3 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 granularity of the collected buoy data for these applications.

Due to limited space, we only show our experiments with Delta-Compression (DC) filters (For experiments with other types of filters, see [**?**]). Each DC filter has three parameters: the data attribute(s) that the filter is interested in, a delta value for compression, and a corresponding quality slack it can tolerate. Table 1 shows the groups of filters we used for our testing. To set parameters for the *DC_Fluoro* and *DC_Tmpr* filter groups, we computed the average changes, *srcStatistics*, of two consecutive tuples in the source time series and then randomly picked delta values in the range of *srcStatistics* and 3**srcStatistics*, which ensured a reasonable data compression that had a non-trivial output data volume. Then we set slack values to be about 50% of the corresponding delta values. This approach prevented a tuple from being included in more than one candidate set for a filter, and also ensured large candidate sets for us to see the benefit of group-aware filtering. For the *DC_Hybrid* filter group, we randomly picked delta values from the range of *srcStatistics* and 20**srcStatistics* and randomly picked slack values that were less than 50% of corresponding delta values. Below, we also evaluate slack's effect on the performance of the delta-compression filtering.

5.4 Metrics

ABBREVIATION	MEANING
SI	Self-Interested filter
RG	Region-based Greedy filter
PS	Per-candidate-Set greedy filter
+C	with timely Cuts
+C(x)	with timely Cuts, x is the name of a time spec.
(B)	with Batched output strategy
(B)-x	with Batched output strategy, x is input tuple window
(Pcs)	with Per-candidate-set output strategy

Table 2. Filter type notations

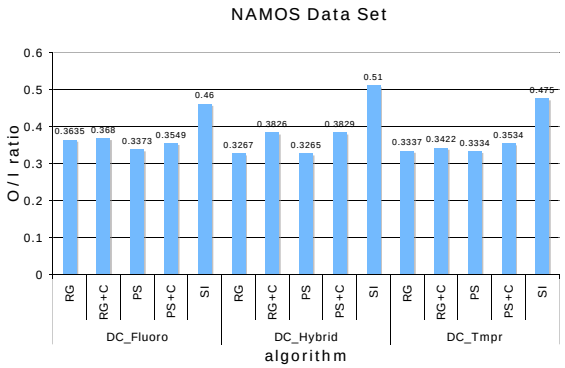

Figure 7. O/I ratios for three groups of group-aware filters.

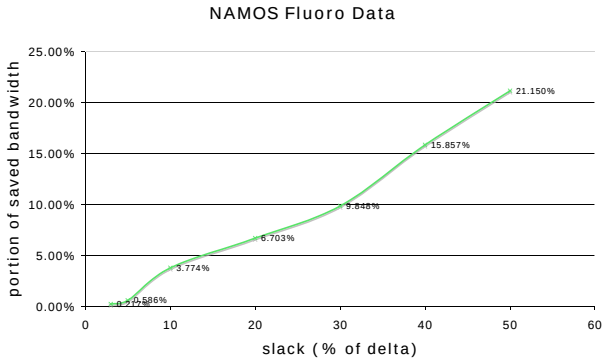

Figure 8. Slack's effect on O/I ratio

The metric we use to measure the benefit of our group-aware filtering approach is the *O/I ratio*, that is, the output vs. input ratio defined as the total number of output tuples over the number of input tuples. A lower O/I ratio means low bandwidth consumption. It measures the bandwidth-saving benefit of group-aware filtering. We expect group-filtering should have an O/I ratio no more than that of self-interested filtering. We measured the filtering cost with *CPU time per tuple*, representing the CPU overhead of group-aware filtering. We also measured data timeliness with source-to-application *latency per tuple*, which shows the delay induced by group-aware filtering to each output tuple. Again, this paper does not focus on the network aspects of group-aware filtering and we do not measure network behavior while performing group-aware filtering.

Table 2 shows the notations we use for filters in the results. Figure 7 shows the O/I ratios for three groups of filters. All group-aware filtering algorithms consumed less than 80% of the bandwidth consumed by self-interested filters. PS filters had a performance comparable to RG filters, which in theory should have better performance guarantee. The addition of timely cuts had little impact on O/I ratio in this experiment, as we set the group time constraint big enough that few regions were cut.

It is easy to imagine that the bigger the slack value is, the more likely the candidate sets of the delta-compression filters overlap. Figure 8 shows that when we decreased the slack values from 50% to 3% of the corresponding delta values in *DC_Fluoro*'s specifications, the portion of saved bandwidth, that is, $1 - \frac{\text{O/I ratio}_{\text{groupAware}}}{\text{O/I ratio}_{\text{selfInterested}}}$ also decreased from 21.15% to 0.217%, almost linearly. When the slack value is decreased to 0, the group-aware filtering is in effect the same as self-interested filtering: with 0% saved bandwidth.

Figure 9 shows the CPU cost per tuple for the *DC_Fluoro* group. (The results are in a box-plot, which plots a summary of the minimum, 25% quartile, median, 75% quartile, and maximum of the ten results. The circles represent outliers.) Group-aware filters were more than 10 times more expensive than self-interested filters. However, it took only 1ms for processing each tuple for group-aware filters, which is fast enough for an input stream with a data rate of 100 tuples per second. Figure 10 shows the latency per tuple for the *DC_Fluoro* group. Since the group-aware filtering gathers tuples in a region before releasing output, it is understandable that the latency incurred for group-aware filters (about 70ms per tuple) was much greater than that for self-interested DC filters (about 12ms). The average region size of the filters was about 6 tuples; since tuples arrived at 10ms intervals, it is clear that the 58ms difference of latency was mainly due to waiting for the tuples to arrive for processing in segments. Due to limited space, we omit the CPU and latency results for the other two groups, but the conclusion was similar to that of *DC_Fluoro* group.

Next, we compare the performance of algorithms that enforce timely cuts. By decreasing the maximum time for closing a region from 125ms in $RG + C(01)$ filters, to a time 16-fold less in $RG + C(05)$ (8ms), the resulting average latency per tuple consistently dropped from above

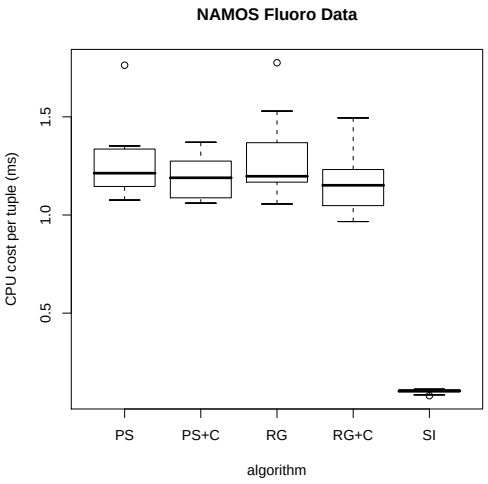

Figure 9. CPU cost for DC˙Fluoro.

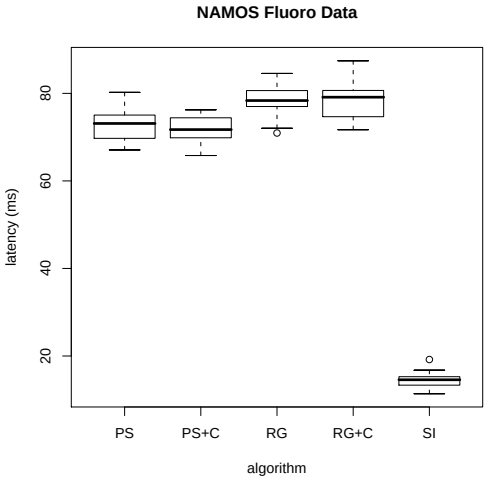

Figure 10. Latency for DC˙Fluoro.

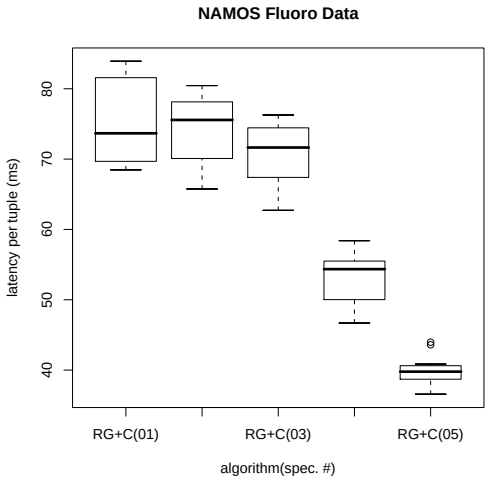

Figure 11. Cuts affect latency for DC˙Fluoro.

70ms/tuple to about 20ms/tuple (see Figure 11), thus proving that timely cuts were effective. Figure 12 shows that the CPU cost to enforce cuts, less than 1.5ms, is acceptable for a fast stream with a 10ms tuple interval. The percentage of regions cut consistently increased by decreasing the maximum time allowed for a region (Figure 13). Cuts affected the O/I ratio only slightly (less than 3% difference), which is understandable because cutting a region will affect the optimality of the solutions found and it is a necessary trade-off for a latency-sensitive filter.

Finally, we evaluate the output strategies with *DC_Fluoro* filter group (Figure 14 and Figure 15). The latency was affected mostly by the size of the average region a group-aware filter used before producing outputs. In the batched output pattern, when the batch size was much bigger than the size of a natural region, the latency increased dramatically due to backlogging of the tuples in the filters until enough tuples were processed. The per-candidate-set output strategy helped to decrease the latency from above 70ms to a little above 50ms. In terms of CPU cost, the batched output pattern did not require sophisticated checking on whether a natural region is closed, which cut 1ms from the original 1.3ms CPU time.

To sum up, our prototype-based experiments validated the effectiveness of group-aware stream filtering in further saving of the bandwidth, compared with self-interested filtering. Its low CPU overhead justified that group-aware stream filtering is suitable for fast stream processing. The increased delay in output tuples was due to the batch processing in group-aware filtering. Compared with application-level multicast delay we measured, it is considered minor. Timely cuts were effective in curbing the latency in output tuples, yet its CPU overhead and its effect on O/I ratio were both small. Output strategies had the anticipated effect on output's timeliness, thus they provide knobs for the system to tune the performance of group-aware stream filtering for

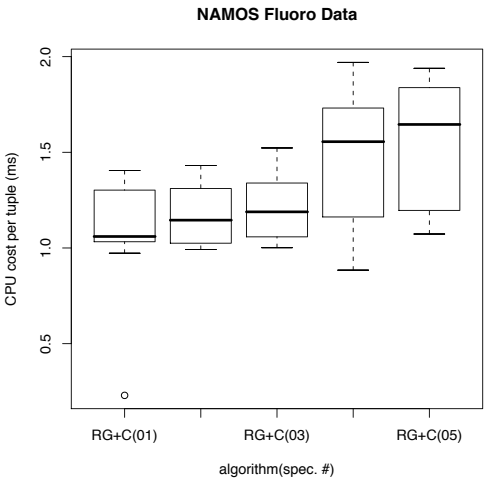

Figure 12. CPU cost of cuts for DC˙Fluoro.

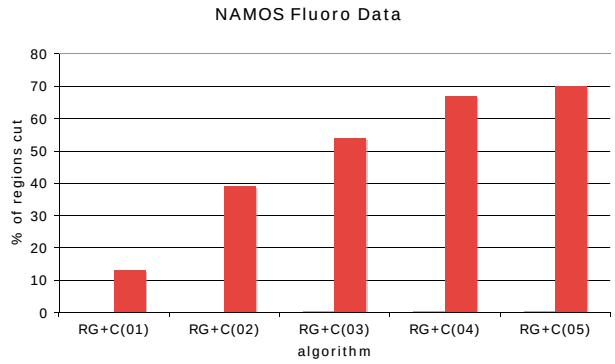

Figure 13. Percent of regions cut for DC˙Fluoro.

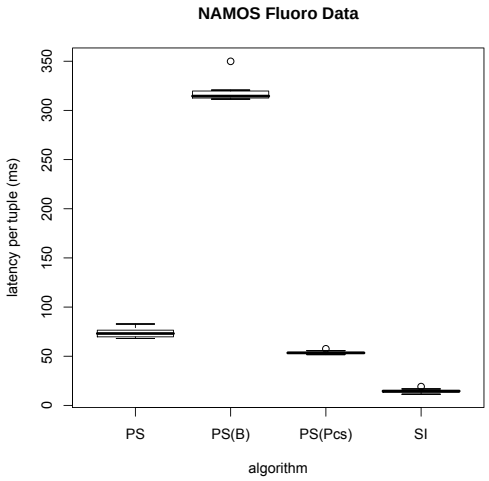

Figure 14. Output strategy affects data timeliness.

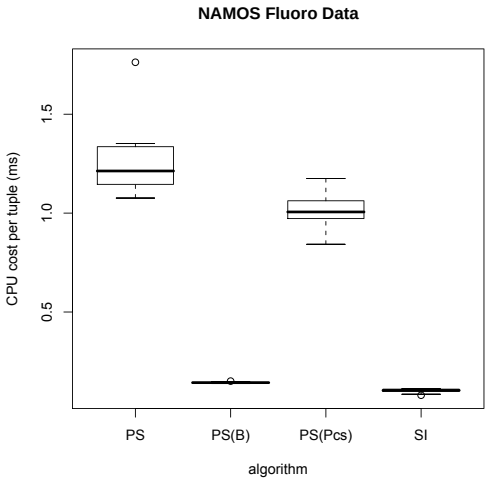

Figure 15. CPU cost of output strategies.

data timeliness.

6. Related work

Our work exploits the semantics of a stream processing application to improve resource management in a distributed dissemination system. IBM's Gryphon [14] also leverages the semantics of subscribing applications to compress a sequence of data updates that have the same effect on applications' ultimate states. Zhao et al. [15] propose a special rule-based language to specify an application's sophisticated processing needs, specifically, the semantic equivalence of outputs to a remote application in face of retransmitted and disordered data. Rather than using a complicated language to describe the needs, our implemen-

tation provides a simple framework with customizable filters and functions to facilitate applications to describe the approximate nature of their filtering requirements.

Bandwidth-reduction mechanisms, such as sampling, summarising, and filtering, have been actively studied in recent years in the systems community [2, 3, 4, 9, 12]. 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, 11, 13]. 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 CPU time [11, 13]. We have different objectives for our filtering; the goal of our work is to trade computation time for savings in communication.

Johnson et al. [9] summarized a general structure for sampling operators. The structure also contains candidate set admitting and output deciding stages, as we propose for the general group-aware filtering process. If we see the 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 for each filter. But our process involves coordination across a group of applications, which never occurs in Johnson's single-application sampling.

7. Summary

This paper provides a general framework that gives a complete treatment to the group-aware filtering problem. We formally define the optimization problem in group-aware filtering for continuous data streams, and prove its NP-hardness. We treat data quality management as the ultimate guidance to group-aware filtering: all our proposed heuristics-based algorithms for preserving bandwidth are subject to meeting the data granularity and timeliness requirements of the filters. We show that the group-aware filtering process is general enough to go beyond simple delta-compression filters, and supports many sophisticated data filters such as stratified sampling. We demonstrate the effectiveness of our algorithms with an implemented system for disseminating real-world data sets. The encouraging results show that group-aware filtering is a quality-managed tool in exploring further opportunities to preserve bandwidth for data dissemination in low-bandwidth networks.

8. References

[1] S. Aryangat, H. Andrade, and A. Sussman. Time and space optimization for processing groups of multi-dimensional scientific queries. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, pages 95–105, 2004.

[2] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634, 2002.

[3] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Sampling algorithms: lower bounds and applications. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing (STOC)*, pages 266–275, 2001.

[4] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 263–274, 1999.

[5] G. Chen, M. Li, and D. Kotz. Design and implementation of a large-scale context fusion network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pages 246–255. ACM Press, 2004.

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

[7] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu. Adaptive stream filters for entity-based queries with non-value tolerance. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 37–48, 2005.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[9] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 1–12. ACM Press, 2005.

[10] M. Li. *Group-Aware Stream Filtering*. PhD thesis, Dartmouth College Computer Science, Hanover, NH, May 2008. Available as Technical Report TR2008-621.

[11] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 49–60. ACM Press, 2002.

[12] D. P. Mitchell. Consequences of stratified sampling in graphics. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 277–280. ACM Press, 1996.

[13] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 563–574, 2003.

[14] 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)*, 1998.

[15] Y. Zhao and R. Strom. Exploiting event stream interpretation in publish-subscribe systems. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, 2001.