# Distributed Data Streams Indexing using Content-based Routing Paradigm *

Ahmet Bulut and Ambuj K. Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106
{bulut,ambuj}@cs.ucsb.edu

Roman Vitenberg
Haifa Research Labs, IBM
Haifa, Israel
romanv@il.ibm.com

## Abstract

*In recent years, we have seen a dramatic increase in the use of data-centric distributed systems such as global grid infrastructures, sensor networks, network monitoring, and various publish-subscribe systems. The realization of this potential requires adequate support from middleware that could be used to deploy and support such systems. In this regard, we propose an integrated distributed indexing architecture that supports scalable handling of intense dynamic information flows. The architecture is geared towards providing timely responses to queries of different types while minimizing the use of network and computational resources. The underlying communication framework ensures scalability and load balancing of communication as well as adaptivity in presence of dynamic changes. We elaborate on database and content-based routing methodologies used in the integrated solution as well as non-trivial interaction between them, and thereby provide a valuable feedback to the designers of these techniques. We demonstrate the effectiveness of our architecture with performance results that we obtained using our prototype implementation on top of the Chord system simulator.*

## 1. Introduction

Data stream systems such as sensornets, network monitoring systems, and security sensors in military applications consist of a multitude of streams at different locations. The core of such system architectures is formed by *data centers*, which coordinate their data handling for providing collaborative data mining and fusion, and for responding to various types of user queries. Typically, data centers can be dispersed over significant distances and communication between them is expensive.

Continuous queries that run indefinitely, unless a lifetime has been specified, fit naturally into the mold of data stream applications. Monitoring a set of conditions or events to occur, detecting a certain trend in the underlying raw sensor data, or in general discovering relations between various components of a large distributed real time system are of profound importance in most real world applications. We consider two types of queries in this paper: (1) *Inner product* queries are important for statistics computation and condition specification. An example query is "Which links or routers in a network monitoring system have been experiencing significant fluctuations in the packet handling rate over the last 5 minutes?". (2) *Similarity* queries are important for trend analysis and pattern recognition. An example query is "Which temperature sensors in a weather observatory exhibit some temperature behavior pattern?".

Centralized solutions with a single dedicated data center collecting information about all streams and answering all queries are impractical for current and future data stream systems: such server and the network in its vicinity would have to handle dozens of thousand of messages every second. While such loads could be tolerated by performance-oriented cluster services, it would render a typical low-resource data stream system non-scalable. Furthermore, the dedicated data center becomes a single point of failure.

Emerging content-based routing technologies such as CAN [19], Chord [24], Pastry [21], and Tapestry [26] that are introduced as general solutions for Internet-based peer-to-peer applications, bear the potential to lay ground for the underlying communication stratum in distributed data stream systems as they provide the means to uniformly distribute the load of processing stream data and queries as well as the burden of communication due to data propagation across all nodes and links in the system. Furthermore, these technologies aim at adaptive accommodation of dynamic changes, which facilitates a seamless addition of new data streams and data centers to the system as well as handling of various possible failures, e.g., data center crashes.

We exploit the scalability and load balancing of communication as well as adaptivity in presence of dynamic changes provided by content-based routing schemes, and

* Work supported partially by NSF under grants EIA-0080134 and ANI-0123985.

propose an adaptive and scalable middleware for data stream processing in a distributed environment. Our solution relies on the standard distributed hash table interface, which makes it more general and portable since it can be used on top of virtually any existing content-based routing implementation tailored to a specific data stream environment. The proposed architecture handles the most popular types of queries in complex data stream applications, i.e., similarity queries and inner product queries, while minimizing the amount of network and computational resources consumed by data centers and network links.

In the rest of the paper, we will elaborate on both database and content-based routing methodologies used in the integrated solution as well as non-trivial interaction between them. Furthermore, we outline possible extensions to the existing content-based routing schemes in Section 4. Our previous work [7] outlines the general vision of this architecture. In this paper, however, we describe the details of our prototype that we built on top of the open source Chord simulator [22]. In Section 5, we present performance results with varying number of nodes on different scalability characteristics, such as the system load, effectiveness, and responsiveness. The results we obtain confirm our claims, and allow us to identify potential bottlenecks under non-favorable conditions. These bottlenecks can be overcome using several extensions to our middleware, which are discussed in Section 6.

## 2. Related Work

### 2.1. Data stream management systems

Design and implementation issues for building Data Stream Management Systems [2] for sensor networks [11, 13, 16], Internet databases [9], and telecommunications networks [10] have been explored in the database community. A significant amount of previous work addresses query processing over dynamic information sources [3, 17]. However, to the best of our knowledge, none of these works consider similarity queries over distributed data streams.

Several approaches were proposed in the past towards solving the problem of mining multiple data streams in centralized settings in which all streams and client queries arrive at a single location [5, 6, 27]. While developing important techniques aimed at conserving local computational resources, these solutions did not take the distributed nature of data stream systems into account. In this paper, we fill this gap by proposing an architecture that handles complex user queries over distributed and dynamic information sources.

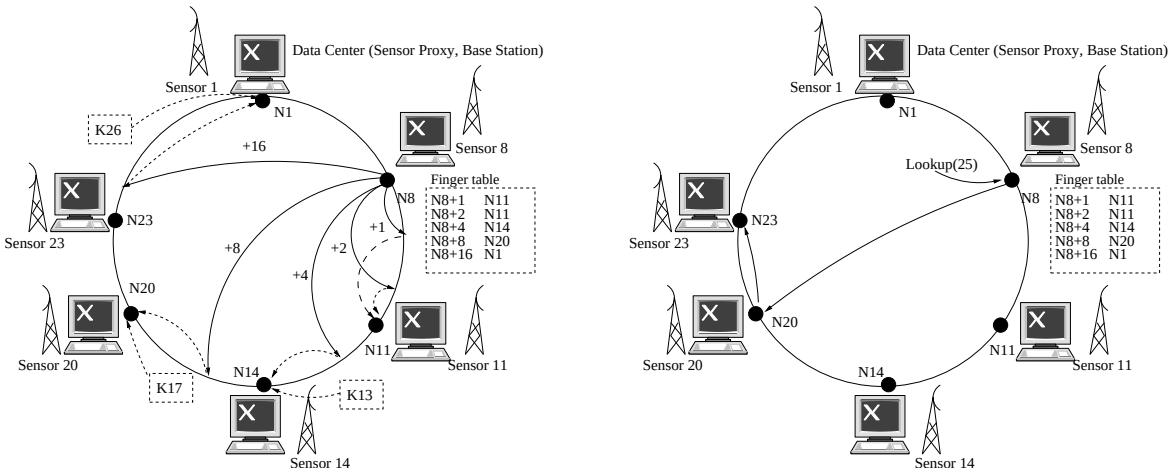
### 2.2. Content-based routing paradigm

Various content-based routing approaches have been designed for different types of networks: GHT [20] and NanoPeers [25] were introduced specifically for sensornets while CAN [19], Chord [24], Pastry [21] and Tapestry [26] were introduced as general solutions for Internet-based peer-to-peer applications. However, the common idea behind most schemes is to map both messages and data centers to the same universe of integer *keys*. Each key is *covered* by some data center (e.g., the one which maps to the closest key value.) A message containing application data is sent not to a specific data center but rather to the key to which the summary maps. The system automatically routes the message to the data center, which covers the key in the message. Such key-based routing allows the system to quickly adapt to dynamic changes, such as a failure or addition of individual data centers to the system.

Furthermore, virtually all content-based routing schemes provide the same interface for the applications, which consists of the following basic primitives: a) `send` operation to send a message to a destination determined by the given key, b) `join` and `leave` operations for a node to join or leave the system, and c) `deliver` operation that invokes an application upcall upon message delivery. This interface is used to implement standard structured overlay applications, such as the put/get DHT functionality, peer-to-peer storage systems, or publish-subscribe. The solution we propose relies on this interface built on top of Chord routing protocol. However, our solution can use virtually any content-based routing protocol mentioned above. In Section 2.2.1, we present the key features of the Chord protocol. The interested reader can refer to [24] for an in-depth analysis.

**2.2.1. The Chord protocol.** The Chord protocol specifies how to find the locations of the keys in a dynamic system where new nodes join and some existing nodes leave. A Chord node maintains information about $O(\log N)$ other nodes in an $N$-node network. A consistent hash function, e.g., SHA-1 [1], is used to assign an $m$-bit identifier to each node and key. These identifiers are ordered on an identifier circle (*Chord ring*) modulo $2^m$. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Key $k$ is assigned to the first node called the *successor node* of key $k$, whose identifier is equal to or follows $k$ in the ring. Figure 1(a) shows a Chord ring with $m = 5$. Keys with identifiers $13, 17$ and $26$ are assigned to nodes with identifiers $14, 20$ and $1$.

For efficient lookup, Chord maintains a table called *finger table*. The $i^{th}$ entry in the table at node $n$ is the successor node $s$ of the identifier $(n+2^{i-1})$ modulo $2^m$. The node $s$ is called the $i^{th}$ *finger* of node $n$. Each entry also contains the IP address and the port number of the relevant node. Figure 1(a) shows an example finger table for node $8$. For example, the $4^{th}$ entry is $N20$ which is the successor node of identifier $(8 + 2^{4-1})$ mod $2^5 = 16$.




(a) An identifier circle with 6 nodes storing 3 keys

(b) Scalable key look-up using finger table at node 8

**Figure 1. Chord: a content-based routing protocol for P2P networks**

Consider the Chord ring in Figure 1(b), and suppose that node 8 wants to find the successor of key 25. Node 8 finds the node that most immediately precedes key 25, which is node 20. The query is forwarded to node 20. This node in turn finds the closest preceding node ($N23$) in its own table for key 25, and forwards the query to this node. Node 23 finds that key 25 falls within itself and its immediate successor, node 1. Therefore, node 1 is returned as the location of key 25.

## 3. Data, Query and Computation Models

In this section, we present the preliminaries of our data stream management application for sensor networks in terms of data, query and computation models used.

### 3.1. Stream data model

Sensing nodes are basic components in sensor networks. Each sensor integrated in a sensing node is a separate data source and monitors the physical environment by sampling physical signals. In other words, each sensor generates a discrete time series, an ordered sequence of data points $\langle \ldots, x_i, \ldots \rangle$ such that the value of each data point $x_i$ lies in a bounded range, $[R_{min}, R_{max}]$. For all practical purposes, we will be interested in only those values of each stream that falls into a given time frame of size $N$. This model corresponds to the "sliding-window" model with window parameter $= N$.

### 3.2. Stream query models

**3.2.1. Inner product queries.** An inner product query is a quadruple $(S_{id}, I, W, T)$, where $S_{id}$ denotes the identifier for the stream in the system, $I$ denotes the index vector (the data items of interest), $W$ denotes the weight vector (the individual weights corresponding to each data item), and $T$ denotes the lifespan of the query [5]. For example, $(0, [0, 1, 2, 3], [8, 4, 2, 1], \infty)$ is an inner product query, which computes the weighted average of data items at indices $0, \ldots, 3$ of the stream with $id = 0$ indefinitely.

**3.2.2. Similarity queries.** A similarity query is a triplet $(Q, r, T)$, where $Q$ is the query sequence (a pattern or a trend), $r$ is the threshold value (similarity score), and $T$ is the lifespan of the query. All sequences in the given set of sensor streams that are within Euclidean distance $r$ to the query sequence $Q$ are output during $T$ time units. We establish the notion of similarity between sequences as follows: given two sequences $x$ and $y$, $x$ is considered to be $r$-*similar* to $y$ if $L_2(\hat{x}, \hat{y}) = \sqrt{\sum_{i=0}^{N-1} (\hat{x_i} - \hat{y_i})^2} \leq r$ where $L_2$ denotes the Euclidean distance, and $\hat{x}$ denotes the unit-normalized sequence for $x$.

### 3.3. Stream computation model

System devices that are in use in today's data stream systems such as battery powered sensors in an embedded sensor network, and routers in a telecommunications network have limited resources. Therefore, the state information maintained for a stream of data has to be small in space, and has to be updated fast with each incoming data item. This is essential for normal functioning of the whole system. We use Discrete Fourier Transform (DFT) to capture the salient features of the underlying data stream on the fly. The N-point DFT of a signal $\vec{x} = [x_t]$, $t = 0, \ldots, N-1$ is defined to be a sequence $\vec{X}$ of $N$ complex numbers $X_f$ such that

$$X_f = \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} x_t e^{-j2\pi ft/N} \quad f = 0, \ldots, N-1 \qquad (1)$$

where $j = \sqrt{-1}$. The inverse Fourier transform of $\vec{X}$ is given by

$$x_t = \frac{1}{\sqrt{N}} \sum_{f=0}^{N-1} X_f e^{j2\pi ft/N} \quad t = 0, \ldots, N-1 \qquad (2)$$

DFT is an orthogonal transformation; hence, it preserves the energy of the signal: $\sum_{t=0}^{N-1} x_t^2 = \sum_{f=0}^{N-1} X_f^2$.

If we compute the coefficients from scratch with each new data arrival, the per-item processing time can be prohibitive for large $N$. However, due to the updateability of DFT [12], each coefficient $X'_f$, $f = 0, \ldots, N-1$, for the signal $\vec{x} = [x_t]$, $t = 1, \ldots, N$ can be computed in constant time using the previously computed coefficients $X_f$ and stream values $x_0$ and $x_N$ as follows:

$$X'_f = e^{\frac{j2\pi f}{N}} (X_f + \frac{x_N - x_0}{\sqrt{N}}) \quad f = 0, \ldots, N-1 \qquad (3)$$

For most real time series, the first $k$ ($k << N$) DFT coefficients retain most of the energy of the signal. Therefore, we can safely disregard all but the very first few DFT coefficients, effectively reducing the dimensionality of the space to work with. The overall approach requires $O(k)$ in time to retain the salient features (e.g., the overall trend) of the original time series. We call this state information maintained "summary" or "synopsis" in the rest of the paper.

## 4. Proposed Solution

### 4.1. Distributed indexing approaches

The main design goal of a distributed indexing scheme is to provide fast and efficient stream data mining and timely response to queries while minimizing the amount of network and computational resources consumed by data centers and network links. It is imperative for such a scheme to uniformly distribute the load of processing stream data and queries as well as the burden of communication due to data propagation across all nodes and links in the system. Furthermore, the system must be able to adaptively accommodate dynamic changes: data centers and links may fail and new data centers and streams may be added without the need to temporarily block the normal system operation.

The naive approach to this problem would be to store all data for a particular stream at the nearest data center. In this solution, all queries to this stream would need to be sent to this data center. While this approach is adequate for inner product queries that are interested in individual values of a specific stream, it does not provide a viable solution for similarity queries, which will require communication with every data center in the system with the purpose of collecting information: Thus, data centers have to preprocess data streams and match them against each other or against a pattern (*flooding* the query to the *entire* network for computing an answer) in order to be able to answer such queries in a timely manner. If such preprocessing is done by a single data center, this data center will immediately become a bottleneck in the system: in addition to being overloaded with the burden of queries and data processing thereby limiting the system scalability, a failure of this single node will render the whole system completely non-functional for the duration of the recovery operation. We refer the interested reader to [14] for a detailed discussion of fault-tolerance issues in various schemes.

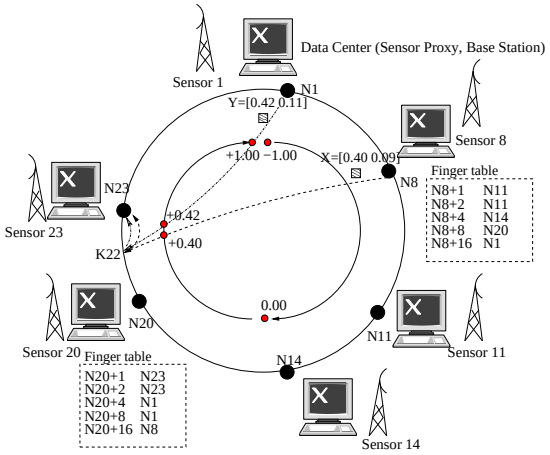



**Figure 2. Content-based routing of stream summaries**

However, if each stream summary is stored locally, and also routed to another data center based on its content, we can avoid flooding the similarity query to the entire network. We can identify the relevant nodes that contain similar content to the query by a scalable content-lookup that will be explained in Section 4.2. The overall approach behaves as a distributed index structure in order to prune the whole search space (the entire network) to a small candidate set (sub-net), thereby reducing the message communication considerably.

### 4.2. Mapping stream summaries to nodes

In the heart of our approach lies the ability to route messages containing stream summaries based on the summary content. A summary computed as described in Section 3.3 over a *normalized* stream of data is a vector $X \in \Re^k$ in a k-dimensional *unit* feature space. Each newly computed feature vector $X$ is routed to a data center, which is the successor node of a $2^m$-bit identifier $i$ determined by hashing $X$ to a valid identifier on the Chord ring. For this purpose, we provide a mapping function $h : \Re^k \rightarrow \{0, \ldots, 2^m\text{-}1\}$ that takes a feature vector $X \in \Re^k$ and returns a valid Chord identifier $i \in \{0, \ldots, 2^m\text{-}1\}$. We use the real value com-

ponent of $X_0$ (or of $X_1$ if the streams are z-normalized to have mean $\mu_x = X_0 = 0$) for key computation. If we know the probability distribution function $f(X_0)$ of $X_0$ *a priori*, we can compute intervals $[a_i, b_i]$ to be assigned to each one of a total of $M$ nodes in the system such that $\int_{a_i}^{b_i} f(X_0)dX_0 = 1/M$. This approach results in uniform load balancing. In this paper, we assume that the distribution is uniform, and confirm the validity of this assumption in Section 5.

Since all streams are projected on to the *unit* feature space, we have $\sum_{i=0}^{k-1} X_i^2 \leq \sum_{i=0}^{N-1} X_i^2 = 1$, which implies $-1 \leq X_i \leq 1$ for $i = 0, \ldots, k-1$. We note that a tighter bound of $-0.7 \leq X_i \leq 0.7$ holds if each stream $x$ is $z$-normalized to have mean $\mu_x = 0$ and standard deviation $\sigma_x = 1$ [27]. To compute the identifiers, we scale the interval $[-1, 1]$ to $[0, 2^{m-1}]$ as follows:

$$i = \lfloor (X_0 + 1) * 2^{m-1} \rfloor \bmod 2^m \tag{4}$$

According to Equation 4, $X_0 = -1$, $X_0^{'} = 0$, and $X_0^{''} = 1$ maps to $i = 0$, $i' = 2^{m-1}$ and $i'' = 0$ respectively. For example, the feature vector $X = [0.40\ 0.09]$ in $\Re^2$ maps to 22 on the Chord ring in Figure 1(a), which can be verified easily by $\lfloor (0.4 + 1) * 16 \rfloor \bmod 32 = 22$.

Figure 2 gives a simplistic overview of the system in operation. The feature vector $X = [0.40\ 0.09]$ computed at node 8 ($N8$) over the data stream produced by sensor 8 hashes to key 22 ($K22$) as indicated by the dashed-line in Figure 2. Node 8 checks its finger table, and finds that node 20 is the closest preceding node to $K22$. Therefore, it routes $X$ to node 20. Node 20 finds that its immediate successor, node 23, is the successor node of $K22$. The feature vector is stored at node 23 at the end of this operation. The feature vector $Y$ computed at node 1 has its $0^{th}$ coefficient ($Y_0$) numerically close to $X_0$. Therefore it either maps to node 23 or to a neighbor of this node. In Figure 2, it hashes to node 23.

Observe that this summary-based routing is used both for stream updates and queries, which can be perceived as "put" and "get" operations in the common DHT functionality.

### 4.3. Extending content-based routing

As described in section 2.2, our solution is based on the standard {`join,leave,send,deliver`} interface of content-based routing schemes. However, in many cases, our schemes involve sending messages to a range of keys rather than to a single key. In other words, all nodes that cover the given key range should receive the message. For example, a message sent to range $[K5, K13]$ in Figure 2 need to be delivered to $N8$, $N11$, and $N14$.

Unfortunately, none of the popular content-based routing architectures provides native support for multicasting a message to a range of keys (e.g., the Internet Indirection infrastructure [23] aims at facilitating generic one-to-many communication using multicast groups in the Internet as opposed to the "lightweight" application-level multicast that our application requires). Therefore, we need to provide this functionality in our solution by using the fact that most schemes including Chord allow a node to send a message to its successor. We exploit this fact to implement multicast to a range of keys in the following way: we send the message to the lowest key in the range. The node that receives such a message, both delivers it locally and forwards it to the successor. The message is forwarded further until the entire key range is covered.

While this implementation is efficient in terms of the number of messages being sent, it induces long delays in propagating the message because the propagation is completely sequential. If the content-based routing system supports sending a message to the predecessor of a node, we can send the message to the middle node in the range and this node could forward the message to both its successor and predecessor. While the difference in the propagation method is insignificant for small ranges, it starts playing important role for wide ranges and systems with a large number of nodes as we show in Section 5. Additionally, efficient native support of multicast to a range of keys can substantially contribute to improving the scalability of our system.

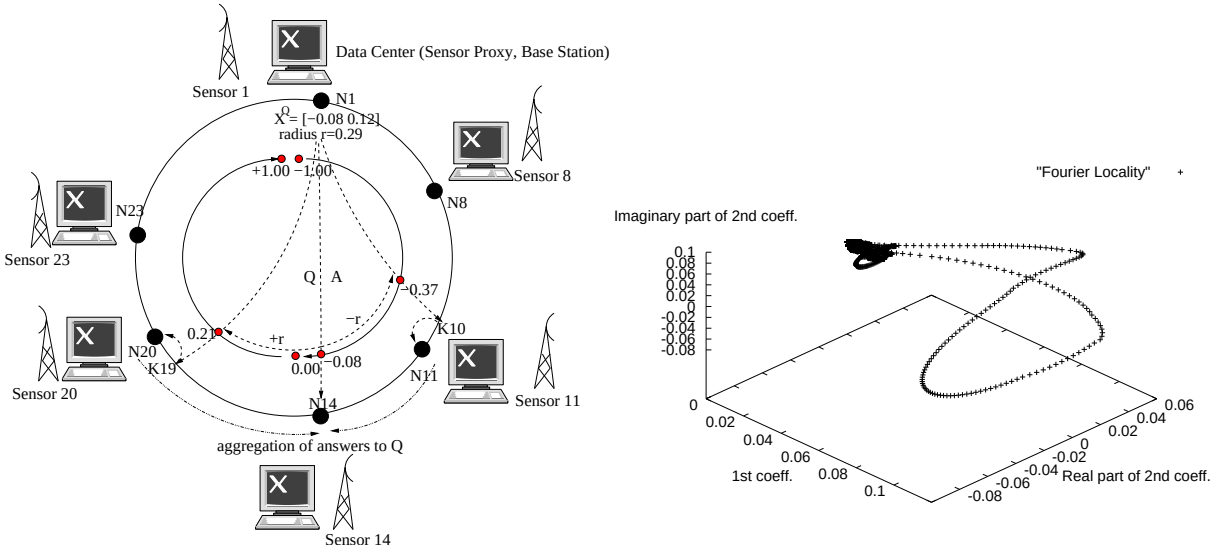
### 4.4. Handling inner product queries

For handling inner product queries, we exploit the underlying content-based routing scheme for implementing a location service, which is the standard way of how it is used in many other applications. To this end, we use a mapping function $h_2$ from the universe of stream identifiers to the universe of keys. Note that we do not use stream features for mapping, since the stream identifier will be enough to locate the source. When a stream with an identifier $S_{id}$ is registered in the system, its stream source $n$ "puts" the $\langle S_{id}, n \rangle$ pair at the node determined by $h_2(S_{id})$. When an inner product query $(S_{id}, I, W, T)$ is posed at node $n_2$, $n_2$ first "gets" the key of the stream source $n$ by sending a message to $h_2(S_{id})$ and then sends the query to $n$. In addition, $n_2$ remembers the mapping between $S_{id}$ and $n$ so that next time it does not need to retrieve it.

When node $n$ receives the query, it performs an inverse transformation on $X^{S_{id}}$ (the feature computed over the stream $S_{id}$) to reconstruct an approximate signal $\vec{x}$ as follows

$$x_t \approx \frac{1}{\sqrt{N}} \sum_{f=0}^{k-1} X_f^{S_{id}} e^{j2\pi ft/N} \quad t = 0, \ldots, N-1 \tag{5}$$

Let $l$ denote the length of the vectors $I$ and $W$. Using the approximate stream values $x_t$, we can compute the weighted inner product as $\sum_{i=0}^{l} x_{I_i} W_i$.




(a) A scalable lookup of similarity queries

(b) Locality of summaries computed on Host Load trace data

**Figure 3. A scalable middleware for distributed stream indexing**

### 4.5. Handling similarity queries

Given a similarity query $(Q, r, T)$ posed at node $n$, first the feature vector $X^Q$ is extracted. An arbitrary feature vector $X$ at a node is considered to be a candidate for similarity with $Q$ if

$$X_0^Q - r \leq X_0 \leq X_0^Q + r \tag{6}$$

Therefore, the query is sent to the key range of $[h(X_0^Q - r), h(X_0^Q + r)]$, as described in Section 4.3. Figure 3(a) shows how a query $Q$ posed at node 1 with the feature vector $X^Q = [-0.08\ 0.12]$ and with the threshold (radius) $r = 0.29$ is routed to the relevant nodes. In this case, the high boundary $X_0^Q + r$ with numerical value -0.08+0.29=0.21, hashes to key 19, and the low boundary $X_0^Q - r$ with numerical value -0.08-0.29=-0.37, hashes to key 10. Therefore, the query is replicated at nodes 11, 14 and 20 as shown pictorially in Figure 3(a).

### 4.6. Propagating responses to the queries

In response to an inner-product query, the node that receives the query sends the reply to the requesting node. Similarity queries cause all nodes in the range to send the detected similarities to the middle node periodically, and the middle node in turn regularly sends response messages to the client. In the example depicted in Figure 3(a), nodes 11 and 20 periodically route the local candidates found to node 14, which in turn aggregates the results and propagates them to node 1, the querying site, during the lifespan $T$ of the query.

### 4.7. Reducing the communication overhead

In our proposed approach, stream summaries are sent to a data center that may be located far away from the stream itself. If every new value generated by the stream caused updated summary information to be sent to a remote data center, this would incur high bandwidth consumption and potentially long lags in query responses. We now consider how to minimize this overhead.

The consecutive feature vectors computed on the same stream exhibit strong locality solely due to the feature extraction scheme used. For example, if a feature vector is computed at time $t$ on a stream segment $x_t, \ldots, x_{t+N-1}$, the next feature vector will be computed on the stream segment $x_{t+1}, \ldots, x_{t+N}$, which overlaps with the previous stream segment in $N-1$ entries. Therefore, there is a strong temporal correlation between the feature vectors computed at successive time units. We justify this claim on the summaries computed on a given trace from Host Load dataset [15], collected in late August 1997 at Carnegie Mellon University (CMU) on a group of machines, as shown in Figure 3(b). We can exploit this correlation in order to reduce the communication overhead by sending batch-updates to remote data centers: we group every $c$ of the feature vectors into a set called Minimum Bounding Rectangle (MBR) $B$, and route this MBR instead of propagating individual feature vectors.

With this optimization, the routing of features needs to be modified to reflect this change. An MBR $B$ in $\Re^k$ is specified by two points $X^{lo}$ and $X^{hi}$ in $\Re^k$ such that for each feature vector $X$ that $B$ contains, we have

$$X_i^{lo} \leq X_i \leq X_i^{hi} \text{ for } i = 0, \ldots, k-1 \tag{7}$$

In order not to incur any false dismissals during similarity query execution, each MBR $B$ has to be replicated at all nodes that is a successor of a key in $[h(X_0^{lo}), h(X_0^{hi})]$. Fig-

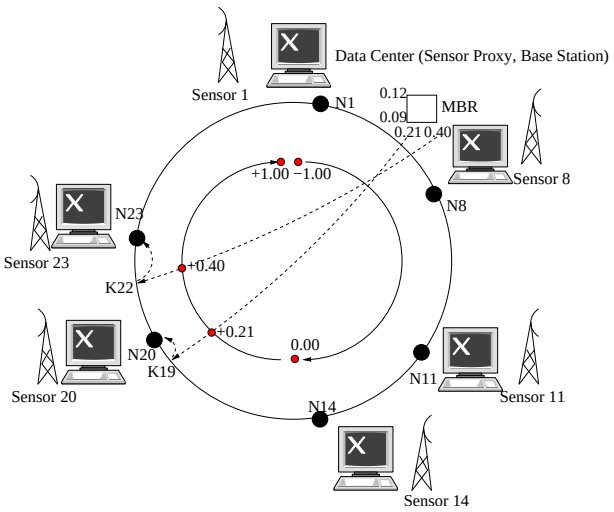



**Figure 4. Content based routing of MBRs that contain a set of stream summaries**

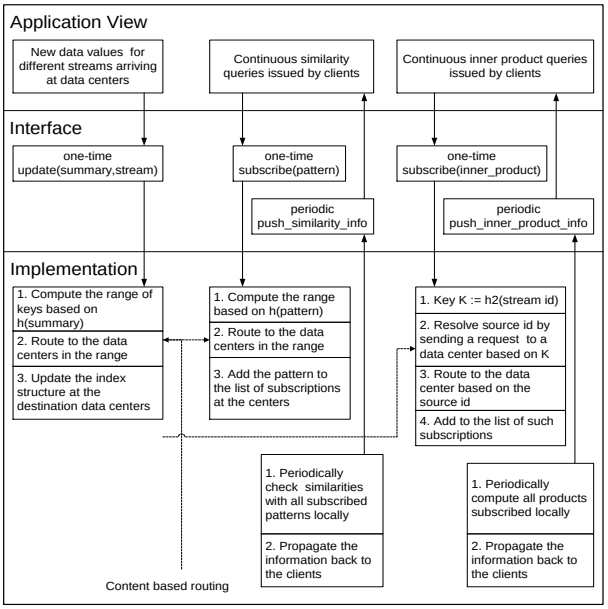



**Figure 5. The overview of various processes that occur in the system in response to queries of different types**

ure 4.7 shows an example MBR $B$ with $X^{lo} = [0.21\ 0.09]$ and $X^{hi} = [0.40\ 0.12]$ coordinates. As one can verify, the low boundary, $X_0^{lo}$, hashes to $K19$, and the high boundary, $X_0^{hi}$, hashes to $K22$. Therefore, $B$ is replicated at nodes 20 and 23, which are the only successor nodes for keys in $[19, 22]$, using the mechanism described in Section 4.3.

## 5. Performance Evaluation

We implemented a prototype of our distributed indexing scheme in a highly portable manner with the purpose of making it independent of the underlying content-based routing scheme. In order to obtain a performance testbed, we linked our implementation with the open source Chord simulator [22], which replays a given workload by simulating a) the Chord routing protocol and b) the execution of timed events on all nodes in the system. In our case, the two main types of input events are new data items generated by the streams and new queries posted by the nodes (see the application view in Figure 5). The code of our implementation is online [8].

The middleware we propose for the distributed stream model provides the following primitives for data stream application (see Figure 5, the application view layer): one primitive to post a new stream data value, and several primitives for client queries of different types. We assume that the infrastructure of data centers provides the necessary geographical coverage. Every stream value and client query arrives at the closest data center for this stream or client by using the means that are external to the proposed middleware while the middleware is responsible for the information exchange between data centers with the purpose of efficient and scalable data handling.

We use synthetic data in our experiments. The synthetic data streams are generated using the random walk model. For a stream $x$, the value at time $i$, $x[i]$ $(0 < i)$, equals to $R + \sum_{j=1}^{i}(u_j - 0.5)$ where $R$ is a constant uniform random number in $[0, 100]$, and $u_j$ is a set of uniform random real numbers in $[0, 1]$. In order to test the scalability of our system, we run a series of experiments in which the number of nodes varied from 50 to 500. We can extrapolate the system behavior to bigger systems that contain a larger number of nodes, since the results we obtain clearly indicate the type of various dependencies in the system. In all our tests we assume that each node is a source of exactly one stream. In contrast, every query is issued by a random node. Queries are generated synthetically by using a uniform distribution. We use similarity queries with radius $0.1$ for most of the experiments.

Here are the main parameters of our workload and runtime configuration: a stream is simulated as a periodic process such that the period for each stream is chosen randomly in the range of *150–250ms* and fixed for this stream. Every MBR or query is stored at nodes only for a certain life span. The life span of an MBR is five seconds while the life span for a query is chosen randomly from the range of 20–100 seconds. Query arrivals are modelled by the Poisson processes with the average rate of two queries per second. Responses to client queries and information exchanges between neighbor nodes are sent periodically with the period of two seconds. Since our analysis includes the notion

of time, it is important to mention that the Chord simulator simulates a constant *50ms* delay per hop when routing a message to the destination.

The three main characteristics we measure in our experiments are: (1) the *load* of messages on a node per unit of time, (2) the *system efficiency*, i.e., the number of messages sent in response to each input event in the system, and (3) the *responsiveness* of the system measured in the number of hops that a message has to traverse.

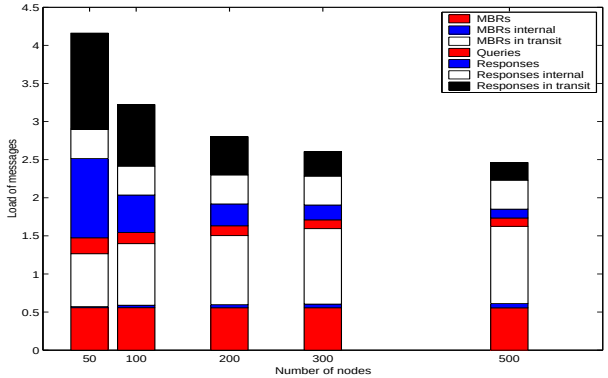
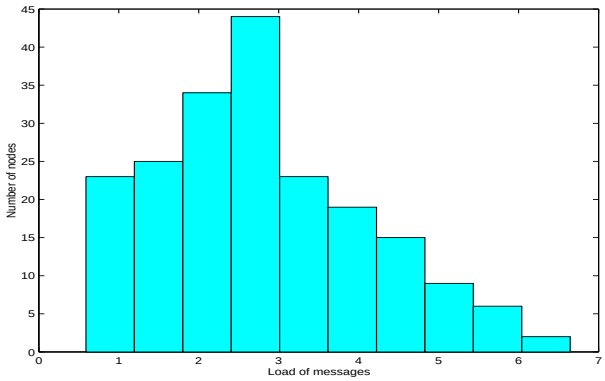


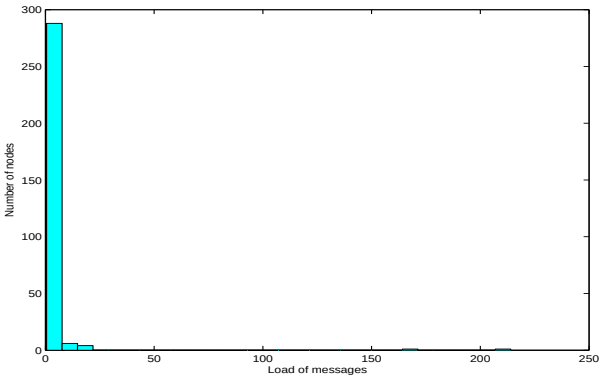

**Figure 6. Average load of messages on a node per second**

Figure 6 presents the average number of messages that an individual node sends or receives per second, as a function of the number of nodes in the system. This number is further broken down into seven components: a) MBR messages originated by the node as a stream source, b) additional messages in case an MBR key range spans multiple nodes as described in Section 4.7, c) MBR messages by intermediate nodes on the route from the stream source to the destination where the MBR is stored, d) all query messages, e) response messages from the notifying node to the client, f) information exchange about detected similarities between the neighbor nodes, and g) response messages by intermediate nodes on the route from the notifying node to the client that posed the query.

As we can see, messages due to MBR and response propagation contribute the most into the load while query-related messages constitute just a small fraction of the total number. This is due to the fact that queries are posed once and run continuously, and each such query typically results in many periodic responses being sent to the client. Furthermore, the query rate is independent of the number of nodes, while each node represents a stream source, thereby regularly sending new MBR messages. Since response messages are sent periodically by the node that receives a query, their total number is linearly proportional to the number of queries, which is why it is constant in all experiments and component e) is linearly decreasing. Notification messages to the neighbors are sent by each node periodically so that component f) is constant. The same argument explains why component a), i.e., the number of MBRs sent by a node as a stream source is not affected by the number of nodes. Our mechanism of MBR creation generated MBRs with relatively small ranges so that the contribution of component b) is also negligible. Thus, the only significant component which is increasing with the number of nodes is c), i.e., messages due to overlay routing. However, this component grows only logarithmically due to the fundamental property of all content-based routing systems such as Chord. Therefore, the load caused by this component cannot render the system non-scalable.




(a) Load distribution for the distributed solution




(b) Load distribution for the centralized solution

**Figure 7. Comparison of load balancing between competing techniques.**

We confirm the validity of our uniformity assumption in Section 4.2 with the load distribution for 200 nodes as shown in Figure 7(a). The distribution is not heavytailed, which indicates that the load is indeed distributed evenly. For illustrative comparison, Figure 7(b) shows the distribution of load for a centralized solution. Note that in practice,

the centralized server will employ a backup scheme in order to avoid being a single point of failure. This will further increase the load of messages on the server and the network in its vicinity. This load may reach thousands of messages per second in a real system. Handling such loads would require high-end dedicated servers/clusters, networks, and technologies instead of commodity mid-level hardware.

Next, we measure *system efficiency* in terms of the number of messages the system sends in order to handle an input event of each type such as a new MBR, a query, or a response. Obviously, every event results in at least one message being sent by the origin node of this event. Figure 8(a) shows the overhead of messages, i.e., the number of additional messages that the system sends. For each number of nodes, we present the number of a) MBR messages in case an MBR key range spans multiple nodes, b) MBR messages in transit that are sent by intermediate nodes, c) query messages in case a query radius spans multiple nodes, d) query messages in transit, e) messages with information about detected similarities that are exchanged between the neighbors, and f) response messages in transit. As we can see, the system efficiently handles messages of all types except for internal query messages. To see why this happens, observe that as the system grows in the number of nodes, the nodes are more densely distributed over the universe of keys. Therefore, the same key range of a query covers more nodes, all of which have to learn about this query in our implementation as described in Section 4.3. This dependency is linear with the number of nodes, which is also confirmed by Figure 8(a).

Figure 8(b) shows the message overhead for a larger query radius. The most significant difference here is in an even higher number of query messages because a two times larger query radius spans twice as many nodes. However, even this high query message load does not have a significant overhead on our system, thereby proving its scalability.

Another important system characteristic is how responsive the system is and how fast each event is taken into account. To this end, Figure 9 presents the number of hops each MBR, query, or response message traverses before reaching the destination and being processed. This is not the same as the number of messages induced by each event, since some of the messages are sent in parallel. Therefore, Figures 8 and 9 show related but different characteristics. While the system guarantees good responsiveness, the query messages take the longest time to propagate because of the linear increase in the number of nodes covered by the query range.

## 6. Future Work

Since the unit of communication between nodes are MBRs, we envision that the size of an MBR directly affects our system performance. Olston et al. [18] consider an adaptive technique for caching intervals in distributed settings. We will adapt their scheme to adjust the low and high boundaries of an MBR along each feature dimension.

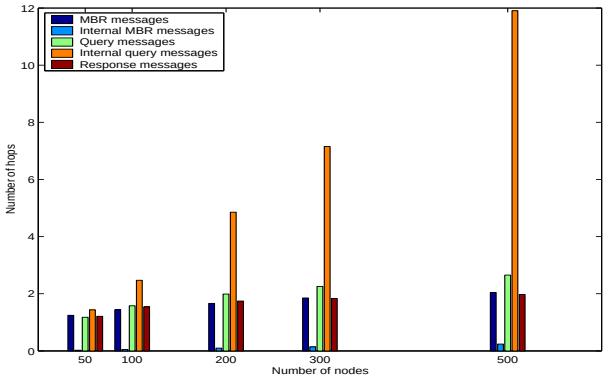



**Figure 9. Average number of hops traversed by a request**

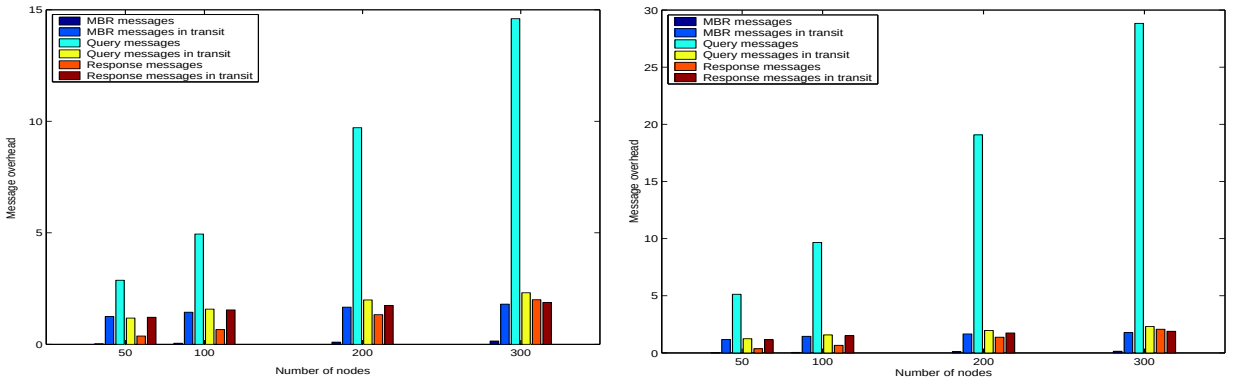
In order to provide an efficient similarity detection for queries with varying selectivity, we can design a feature space partitioning scheme such that data centers are organized into a hierarchy of clusters, similar to the one used in [4] for application layer multicast. At the bottom level, all data centers are divided into small constant size clusters of neighbor data centers. For each cluster, a leader is chosen to represent the nodes in this cluster. Then, all leaders of the bottom level clusters are divided into the next level clusters by their proximity to each other. This process is repeated until a single leader is chosen for all nodes at the topmost level.

When a new summary arrives at some data center that belongs to the bottom level cluster $C$, this data center (in addition to storing the summary locally) forwards it to the leader of $C$. The leader of $C$, in turn, forwards the summary to its own leader. As a result of this technique, the higher we go up the hierarchy of cluster leaders, the larger is the feature space covered.

## 7. Concluding Remarks

In this paper, we proposed a system solution that addresses the challenge of providing fast and efficient stream data mining and timely response to queries while minimizing the amount of network and computational resources consumed by data centers and network links. The system distributes the load of processing stream data and queries as well as the burden of communication due to data propagation uniformly across all nodes and links in the system.

The underlying communication stratum of our solution accommodates dynamic changes such as data center failures, link failures, and/or addition of new data centers as




(a) Message overhead, query radius=0.1

(b) Message overhead, query radius=0.2

**Figure 8. Effect of varying query selectivity on message load**

well as new streams, without the need to temporarily block the normal system operation. Since the proposed middleware relies on the standard distributed hashing table interface provided by content-based routing schemes, it can be used on top of any existing content-based routing implementation tailored to a specific data stream environment.

## References


[1] F. 180-1. Secure hash standard. In *US Department of Commerce/NIST, Springfi eld, VA*, April, 1995.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[3] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, pages 28–39, 2003.

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *SIGCOMM*, 2002.

[5] A. Bulut and A. Singh. SWAT: Hierarchical stream summarization in large networks. In *ICDE*, pages 303–314, 2003.

[6] A. Bulut and A. K. Singh. A unifi ed framework for monitoring data streams in real time. In *ICDE*, 2005.

[7] A. Bulut, R. Vitenberg, F. Emekci, and A. K. Singh. An adaptive and scalable middleware for distributed indexing of data streams. In *DBISP2P*, pages 123–137, 2003.

[8] A. Bulut, R. Vitenberg, and A. Singh. Prototype. http://amazon.cs.ucsb.edu/˜bulut/adidas/source.zip.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD*, pages 379–390, 2000.

[10] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: A language for extracting signatures from data streams. In *KDD*, pages 9–17, 2000.

[11] A. Deshpande, S. Nath, P. B. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. In *SIGMOD*, pages 503–514, 2003.

[12] D. Q. Goldin and P. C. Kanellakis. On similarity queries for time-series data: Constraint specifi cation and implementation. In *CP*, 1995.

[13] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A Distributed Index for Features in Sensor Networks. In *IEEE SNPA*, May 2003.

[14] I. Gupta, R. Renesse, and K. Birman. Scalable Fault-tolerant Aggregation in Large Process Groups. In *ICDSN*, 2001.

[15] http://www.cs.nwu.edu/∼pdinda/LoadTraces/. Load data.

[16] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[17] C. Olston, J. Jiang, and J. Widom. Adaptive fi lters for continuous queries over distributed data streams. In *SIGMOD*, pages 563–574, 2003.

[18] C. Olston, J. Widom, and B. Loo. Adaptive precision setting for cached approximate values. In *SIGMOD*, pages 355–366, 2001.

[19] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A Scalable Content-Addressable Network. In *SIGCOMM*, 2001.

[20] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A Geographic Hash Table for Data-Centric Storage in SensorNets. In *WSNA*, Sept. 2002.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM Middleware*, Nov. 2001.

[22] I. Stoica. Chord. http://www.pdos.lcs.mit.edu/chord/.

[23] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, 2002.

[24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[25] P. Triantafi llou, N. Ntarmos, S. Nikoletseas, and P. Spirakis. Nanopeer networks and p2p worlds. In *P2P*, pages 40–47, 2003.

[26] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Faultresilient Widearea Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.

[27] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.