MIND: A Distributed Multi-Dimensional Indexing System for Network Diagnosis

Xin Li, Fang Bian, Hui Zhang,
Christophe Diot, Ramesh Govindan, Wei Hong and Gianluca Iannaccone

***Abstract*—**
Detecting coordinated attacks on Internet resources requires a distributed network monitoring infrastructure. Such an infrastructure will have two logically distinct elements: distributed monitors that continuously collect traffic information, and a distributed query system that allows network operators to efficiently correlate information from different monitors in order to detect anomalous traffic patterns. In this paper, we discuss the design and implementation of MIND, a distributed index management system that supports the creation and querying of multiple distributed indices. We validate MIND using traffic traces from two large backbone networks, then examine the performance of a MIND prototype on more than 100 PlanetLab machines. Our experiments show that MIND can detect and report network anomalies in about one second on an inter-continental backbone. We also analyze the efficiency of our load balancing mechanism and evaluate the robustness of MIND to node failure.

I. Introduction

Distributed attacks and self-propagating worms have spurred researchers and industry alike to explore innovative intrusion or anomaly detection systems. The traffic is generally analyzed buy an independent physical system located on the link or by the router itself. The state of the art in operational anomaly detection is either to apply pre-defined rules corresponding to known anomalies [19], [23] or to identify unusual patterns [13], [14], [15] in the traffic.

The current systems have two important drawbacks. First, they have no easy way to correlate observations on multiple monitoring points, except by exporting summarized traffic information to a central location, making the overall system vulnerable to failure, attacks or congestion.Second, they often do not support historical analysis of network traffic for reconstructing past network events, identifying all compromised hosts, or detecting vulnerabilities in the network infrastructure that could lead to future intrusions and attacks.

A robust network monitoring and anomaly diagnosis system should therefore have the following properties: (i) store some historical traffic information, (ii) be robust to network failures and to attacks, and (iii) exhibit low response time when a problem is detected. These properties imply that moving all traffic information to a central location is not an option as it could create congestion, slow down network diagnosis, and make the system very vulnerable.

This work is supported by a grant from Intel Corporation.

X. Li, F. Bian, and R. Govindan are with the University of Southern California ({xinli, bian, ramesh}@usc.edu). H. Zhang is with NEC Labs America (huizhang@nec-labs.com). C. Diot is with Thomson Paris Research Lab (christophe.diot@thomson.net). W. Hong is with Arched Rock Corporation (whong@archedrock.com). G. Iannaccone is with Intel Research (gianluca.iannaccone@intel.com).

We propose to structure such a network-wide monitoring and anomaly detection system as two logical components: (1) a set of distributed traffic monitors deployed on as many network links as possible, and (2) a querying system that enables fast correlation of traffic data. The traffic monitors collect and store network traces. They can also generate traffic summaries in the form of flow records or suitably aggregated and filtered versions thereof. The querying system allows users (or monitoring systems) to efficiently query these traffic summaries. In addition, it should make possible to design decentralized anomaly detection algorithms and to add some redundancy in order to improve system's robustness and availability.

The querying system will likely support several kinds of queries, ranging from exact match to flexible pattern matching on flow records. However, it is crucial for such a system to efficiently support multi-dimensional range queries. Many queries that attempt to correlate traffic summaries are naturally expressed as multi-dimensional range queries. For example, a query of the form "was there a flow of size greater than 100MB to prefix X in time interval T" can be used to monitor traffic volumes to a collection of customers. Similarly, a query of the form "what is the count of distinct sources sending small flows to port 3306 to destination prefix X" can be used to detect suspicious port scanning activity.

In this paper, we explore the design and implementation of MIND, a *distributed* system that supports the network-wide monitoring and anomaly detection system described above. MIND is a hypercube overlay that supports the creation and querying of multi-dimensional indices with features that are specific to network wide anomaly detection. Traffic summaries expressed as multi-attribute data records and generated at network monitors can be inserted into one or more MIND indices. MIND routes these tuples to nodes such that tuples near each other in the attribute space are likely to be stored at the same node, making multi-dimensional range searches efficient. In addition, MIND contains a novel load balancing technique that avoids storage hotspots arising from skewed data distributions. Last, MIND replicates indicies for improved robustness and availability, in the face of attacks (potentially directed against the anomaly detection system itself) and node failure.

Inserting all flow records from each network monitor into MIND could incur significant traffic overhead and could impact network performance. Instead, we see MIND as being used in much the same way database administrators build centralized indices. A network administrator performs careful off-line analysis to decide the attributes to be indexed, and the granularity of traffic summaries to be inserted into MIND.

This database design analysis is based on the trade-off between the cost of building the index, and the expected frequency of querying the system. MIND could also be made adaptable to network conditions.

We have implemented a full-featured MIND prototype and have extensively experimented on PlanetLab [20] using traffic flow records from two large academic backbones Abilene [1] and GÉANT [7]. In one of these experiments, we constructed a MIND overlay containing 34 nodes that matched the geographical and topological distribution of Abilene and GÉANT backbone routers and inserted over 9 million flow records into the system. In another experiment, we evaluated the performance and robustness of MIND at a scale of more than 100 nodes. Finally, we compared the anomalies detected by an online MIND system to those detected by an independently designed off-line trace analysis algorithm [15].

Our results reveal that MIND can support median insertion and query latencies under 1 second, and can provide, with each data item being replicated once, accurate replies to queries even when 15% of the nodes fail. MIND exhibited perfect recall on all the anomalies we searched for, with average response times on the order of a second.

The contribution of this work is not-yet-another overlay hypercube validated on PlanetLab. Instead, MIND is the first operational overlay specifically designed and deployed to support network wide anomaly diagnosis on the Internet with the following new features: (i) it allows fast routing of multi-dimensional range queries, (ii) it load balances indices traffic in order to minimize search time and risks of congestion, and (iii) it is robust to attacks and network failures. In addition, our validation is performed with real traffic traces collected on two backbone networks and re-deployed on PlanetLab nodes located near the original backbones' PoPs.

This paper is organized as follows. We discuss the motivation and design choices of MIND in Section II. In Section III, we describe the design and implementation details. Section IV shows the results of our experiments for evaluating the performance of MIND. In Section V, we demonstrate that one can quickly retrieve real-world anomalies using MIND queries. We discuss related work in Section VI and conclude in Section VII.

II. Design Considerations

As explained earlier, we consider a network of traffic monitors where each monitor has access to the entire sequence of packets observed at each link attached to the monitor. These monitors can either store full packet traces and/or compute metrics of interests to network operators and generate multi-attribute *flow records*. A flow record typically represents the summary of traffic aggregated over a specific time window. For example, flow records $(destIP = www.foo.com, destport = 80, octets = 1K, timewin = 30)$ measures the volume of Web traffic to that destination per 30 seconds. Below, We discuss the design choices we encountered during MIND system construction. We believe that these design alternatives also play critical roles for designs of the systems with similar structures and for similar applications.

A. Architecture

There are three possible ways in which one can structure MIND and similar systems. A *query flooding* architecture keeps the flow records at or near the monitors and floods each query to every monitor. A *centralized* architecture moves the data to one node (or a cluster for redundancy), and queries are sent to that cluster. Finally, in a *distributed* architecture traffic monitors store summaries in a specialized routing structure that can forward queries where the summary resides.

The trade-offs between these architectures are well-understood, but in the network monitoring context we find the distributed architecture to be the most attractive. Query flooding scales well in that it does not involve movement of flow records, but can lead to poor performance for high query loads since all queries must be evaluated at each node. Furthermore, to make this architecture robust to failure, it may be necessary to replicate the flow records in a topologically different location, requiring movement of data. The centralized approach lacks the physical redundancy necessary in an operational network monitoring system. It is always possible to replicate the central flow record repository, but such a system will not be that much easier to manage or provision when compared to a distributed architecture.

A distributed querying system can be organized hierarchically or in a peer-to-peer overlay. Both these structures have good scaling properties and are natural candidates for the querying system. Hierarchical systems like Irisnet [8] organize the namespace hierarchically, and can efficiently support hierarchical wild-carded queries. Such structures can only efficiently support multi-dimensional range queries *that are well-aligned with the naming hierarchy*. However, traffic data cannot be organized in a manner required by systems like [8]. For example, even if a network prefix is contained in another, the packets belonging to the two prefixes may be observed in two completely different parts of the network. For this reason, we only consider peer-to-peer overlays as the basis for our design.

B. Building Indices

Traffic data can be indexed at various granularities. At one extreme, one can envision building a single index of every TCP flow observed on each link in the network. While bandwidth is not a scarce resource in today's backbone networks, per flow data is voluminous (*e.g.*, more than 1 TB/day for GÉANT) and most of it is likely to be "normal" traffic and therefore uninteresting. For this reason, MIND is designed to allow users to dynamically create multiple indices on arbitrary subsets of the traffic attributes. Such a capability is analogous to the creation of indices in classical databases, and will enable a more bandwidth-efficient and responsive system. We delay the details about using multiple indices to detect various anomalies until Section IV and Section V.

C. Hashing

The design of overlays for supporting range queries has started to receive some attention in the literature [17], [4], [21],

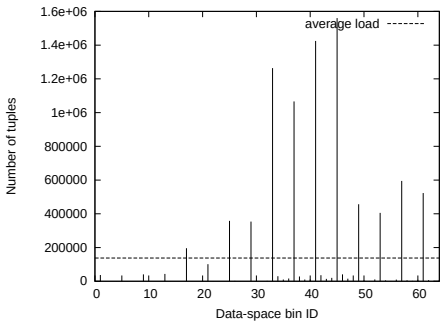

Fig. 1. The data distribution from one-day real-world network traces.

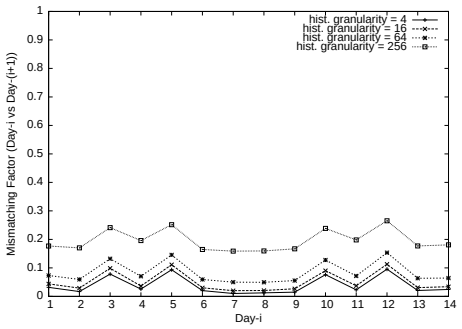

Fig. 2. The daily mismatch of the two backbones' data in two weeks.

[3]. Two classes of approaches have been studied. The first attempts to build support for such queries on top of DHTs. The second stores data on the overlay in a manner that preserves *locality*—data records are routed to nodes such that records stored at a node are "near" each other in the attribute space. This *locality-preserving hashing* is more naturally suited than the first approach to efficient multi-dimensional querying, since queries can be routed directly to nodes that contain the relevant parts of the attribute space. For this reason, MIND employs the latter approach. In Section VI we more carefully contrast previously proposed systems with MIND and explain why prior work is unsuitable for network monitoring and anomaly detection.

D. Load balancing.

Our choice of locality-preserving hashing can, with a naive design, lead to poor system performance. This is because network traffic data is, in general, significantly skewed. Figure 1 plots the number of flow records that would fit into a 64-bin multi-dimensional histogram constructed on three different indices over a one day traffic summary from Abilene and GÉANT (the indices are described in detail in Section IV). This figure illustrates that without a careful systems design, the amount of data stored at different nodes within the network can vary by an order of magnitude.

MIND's load balancing scheme leverages the fact that network traffic is approximately stationary over diurnal timescales. It uses an approximate histogram of the data distribution from a 24-hour period to govern how data storage is distributed across overlay nodes (this is described in detail in Section III-E).

To validate that the distribution of traffic summaries is indeed stationary over different timescales, we analyzed Netflow records from *all* routers on Abilene and GÉANT over a two-week period (December 1–14, 2004). We indexed the Netflow records aggregated (but not filtered) over 30-second intervals on six attributes (consisting of the source and destination addresses, timestamp, the total number of bytes transferred, the number of distinct connections, and the average size of a connection). Figure 2 depicts how the distribution of this index varies day-to-day. We have used different time window size, including 15 seconds, 60 seconds, 5 minutes, and 15 minutes, and found that the data stationarity always hold.

We define a *mismatch* metric to compare multi-dimensional histograms representing the data distribution. For a $d-$*dimensional* index, we partition it into k^d (k is called histogram granularity) equal-sized bins. For bin x ($1 \leq x \leq k^d$), we define

$$L_k^D(i,x) = \text{percentage of Day-i tuples fallen into bin x.}$$

The mismatch between the data distributions of Day-i and Day-j, $MF_k^D(i,j)$, is then defined as

$$MF_k^D(i,j) = \sum_{x=1}^{k^d} \frac{|L_k^D(i,x) - L_k^D(j,x)|}{2}.$$

When k^d is high enough, a daily balanced data allocation can be generated by directly assigning the bins to network nodes. $MF_k^D(i,j)$ is the *upper bound* of the re-balancing cost to have Day-j data allocated evenly among the nodes with Day-i's allocation scheme as the base. One can analogously define the mismatch at different timescales (*e.g.*, hourly).

We find that, even with the finest grain histogram, no more than 20% of the data items need be moved from one day's histogram to another. By contrast, the traffic distribution can vary significantly hour-to-hour (mismatch close to 1 when the histogram has a 64 or higher granularity), indicating that the continuous re-balancing proposed in some other systems [4] might be highly inefficient. We have also verified that this behavior holds qualitatively for filtered indices built on fewer attributes.

E. Robustness

By design, MIND is more robust than centralized schemes. The redundant routing paths provided by the hypercube overlay of MIND enable a simple probing-based detection and recovery mechanism. However, events affecting links or nodes such as transient malfunctions or malicious attacks can compromize the functioning and performance of MIND. Therefore, we decided to add to MIND a replication mechanism to increase availability of critical performance and traffic information.

The MIND replication mechanism is designed in such a way that the replica can be accessed (stored/retrieved) within limited overlay hops from the original data. With the hypercube structure it is natural and efficient to locate replicas on the hypercube neighbors. First, the hypercube neighbors of a node are all one overlay hop away from the node. Second, by design, the hypercube neighbors are the nodes that will consecutively take the place of the original node under failures. Finally, this gives the user a tunable parameter as to the number of

replicas, traded off between data availability and replication overhead. We discuss the details of robustness design in MIND in Section III-F.

III. Design Details

MIND is a distributed system that provides applications with the abstraction of multi-dimensional indices. Applications (or users) can create indices with specific schemas from any MIND node. Any MIND node can insert data into an index, or query an index. Finally, users can delete MIND indices. Thus, in the context of our driving application, a network operator may install a distributed MIND system, then create indices for detecting anomalies. A network operator (or perhaps individual customers) could script periodic queries that poll for the likely existence of anomalies in different traffic volumes. That script could also be automated to drill down (by issuing a sequence of queries) once a potential anomaly was detected.

Note that although we propose to use MIND for network anomaly detection, the MIND system represents a generic distributed substrate for multi-dimensional range queries. As such, it can be used for other tasks, such as root cause analysis of Internet routing problems, performance analysis or troubleshooting.

For each index, MIND defines a mapping between the multi-dimensional data-space and the underlying hypercube. MIND's novelty lies in its de-coupling the data-space mapping from the underlying routing structure. This allows the MIND system to naturally balance the storage load across all the overlay nodes while maintaining the conceptual simplicity of the overall routing scheme. MIND replicates the record on a natural "sibling", in a manner to be described later, to allow robust retrieval of data.

A. Overlay Construction

The logical structure of an N-node MIND is a N-node hypercube where each vertex corresponds to one MIND node. The choice of a hypercube is not fundamental in MIND; we have not investigated other overlay routing geometries, such as the ring, but we expect that they can be used for MIND, perhaps with different trade-offs [9].

Each MIND node uses its hypercube vertex code as its "address" on the overlay. MIND attempts to maintain a *balanced* hypercube, to the extent possible. A balanced hypercube minimizes the maximum code length of any node, resulting in about $\log N$ neighbors per node. Since the neighbor list is the only state each MIND node needs to maintain in order to make routing decisions, a balanced hypercube will even out the routing table size at all nodes.

There are various ways to establish a balanced hypercube with a given set of nodes. One approach, for example, would use a set of *well-known* hosts to record the current hypercube topology. Other nodes can join the hypercube by querying a well-known host to find an appropriate neighborhood of nodes to join. Adler *et al.* [2] propose a distributed randomized node join procedure as follows. A host that wants to join randomly chooses a node in the hypercube. In the neighborhood of the selected node including the node itself, it chooses the node with the shortest code to join and becomes that node's sibling node. This join procedure guarantees that the resulting hypercube, after a sequence of node joins, is balanced with high probability. Because it is amenable to a more robust implementation, we adopted Adler's algorithm in MIND.

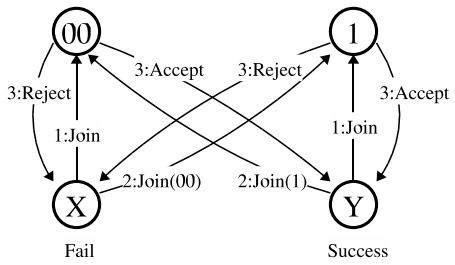

Fig. 3. The deadlock free concurrent join procedure in MIND.

However, our implementation modifies the original join algorithm to serialize concurrent node joins to the same neighborhood in a manner that does not introduce deadlocks. In our modification, a node may optimistically accept multiple join requests. However, a new join request preempts an ongoing join process that has not been committed if it joins a shallower node. Figure 3 illustrates the concurrent join process. Consider the case where node X and Y simultaneously join the overlay. In step 1, node X and Y send join requests to node 00 and node 1, respectively, at the same time. In step 2, X joins node 1 with code 00 and Y joins node 00 with code 1. In step 3, unless X's requests have been committed by all nodes X has contacted, only Y's join requests are accepted while X are rejected by both node 00 and node 1 because Y is joining a shallower neighbor. The procedure shown can be easily extended to the cases of more than two concurrent node joins.

B. Index Creation and Data Space Embedding

MIND allows the construction of several indices, and *maps* each index independently onto the overlay. The procedure for creating an index in MIND is relatively straightforward. When an application calls `create_index`,[1] the index creation request, together with the schema for the index, is flooded on the overlay. When nodes join the overlay, they obtain the current set of defined indices from the neighbor to which they attach. Similarly, when an index is dropped, a message flooded throughout the overlay causes all state at each overlay node pertaining to the index to be deleted.

In MIND, the data space associated with an index is stored on the overlay in the following manner. Consider an index built on k-dimensions. For example, for our alpha flow monitoring application, k can be 4 (source IP address, destination IP address, flow size, and time). Conceptually, the data inserted into such an index can be described as points in a k-dimensional space. MIND essentially maps hyper-rectangles in this space into individual overlay nodes, in a manner described below. A

[1] We describe MIND algorithms without giving their pseudo code for lack of space.

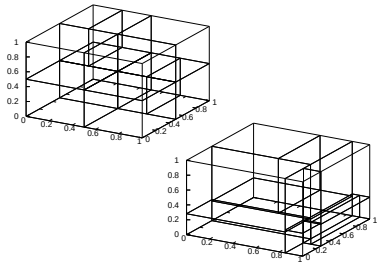

Fig. 4. The equal data space division (top left) vs. the histogram based data space division (bottom right) in a 3-D normalized space for 10 nodes.

key property of this algorithm is that k can be independent of the dimensionality of the hypercube.

Take the k-dimensional data space and "cut" it (for now, assume that these cuts divide the data-space equally, but we shall remove this assumption shortly) along each of these k dimensions using a $k-1$ dimensional hyper-plane. This results in 2^k hyper-rectangle. At each cut, define one half of the data-space to have a code bit of 0, and the other a code bit of 1. Thus, following this sequence of cuts, each 2^k hyper-rectangle is associated with a bit-string "code". Now, repeat this procedure until the number of hyper-rectangles equals the number of nodes (Figure 4, top left). Then, the data points in a MIND index falling into a hyper-rectangle A are stored (we describe later how index data is routed) in the overlay node whose code maximally matches the code of A. When a node joins an existing overlay and takes over a part of the data-space from a sibling, data already stored in existing indices are not moved from the sibling to the joiner. Rather, the joiner maintains a pointer to the sibling and forwards queries to it. The pointer will be dropped once the data have aged.

When the data distribution is skewed, the number of data items in different hyper-rectangles can be different. In that case, the storage across nodes is not balanced. However, if the global data distribution were known, even approximately, then we could "cut" the data space to allow for a more equitable distribution of data items (Figure 4, bottom right, illustrates this). We describe this load balancing scheme in greater detail in Section III-E below.

C. Data Insertion

User applications can insert data (*e.g.,* flow records for network monitoring) into MIND from any MIND node. When an application calls `insert_data` at a node, the corresponding MIND instance computes the hyper-rectangle that the data item would fall into. To do this, each MIND node would have to know the hyper-planes that cut the index data space; we describe how a MIND node knows this in Section III-E. Furthermore, because the cuts will, in general, generate more hyper-rectangles than there are nodes (a node may not know the exact number of nodes in the overlay, but computes an estimate of this from its own code length), the computed code for the data item may not exactly match the code associated with a node. In this case, the data item is stored at the node whose code maximally matches the computed code.

To find this node, MIND greedily routes the data item on the hypercube using the computed code. In this procedure, at each step the data item is forwarded to that neighbor whose code maximally matches the computed code until the data item reaches a node which is assigned the hyper-rectangle containing the data item. During routing transients, the routing procedure might hit a dead-end. In such cases, MIND employs a recovery procedure that attempts to find an alternate route (Section III-F).

D. Query Processing

A query on a MIND index is described by a range of values for each attribute of the index (of course, any of the attributes can be wild-carded). Put differently, a query in MIND represents a hyper-rectangle in the data-space. Depending on the size of the query, this hyper-rectangle may be contained within a hyper-rectangle resulting from the data-space cuts described above, or may contain many such hyper-rectangles.

When an application calls `query_index` at a MIND node, that node computes a code for the query in a manner similar to computing codes for data items. However, given that a large query may cover more than one data-space hyper-rectangle, the computed code can be a prefix of the codes assigned to overlay nodes. The query is routed using exactly the same strategy described above for routing data items, but with one important difference. When the query reaches the first node whose associated hyper-rectangle (perhaps partially) intersects the query hyper-rectangle, that node *splits* the query into sub-queries covering data-space hyper-rectangles. These sub-queries are independently routed using the strategy described below.

For simplicity, results from all sub-queries are directly transferred to the originator rather than being routed on the overlay. The originator can then determine, by examining which nodes responded, when the query response is complete. If a node is assigned a hyper-rectangle in the data-space covered by the query, but has no matching data, it responds negatively to the query.

E. Balancing Data Storage in MIND

MIND uses locality-preserving hashing. If the data-space is embedded on the overlay by recursively cutting the data-space evenly across each dimension, then a skewed data distribution (where the data items are unevenly distributed across the hyper-rectangles) can result in significant storage imbalance across the nodes. This imbalance can be a significant practical problem for applications like network monitoring that can store large amounts of data items in different indices.

Classical approaches to re-balancing centralized index data structures usually attempt to do so *on the fly*, *i.e.,* as data items are inserted into the index. In an overlay based system like MIND, this approach can introduce significant complexity. MIND would have to dynamically re-partition the data-space among the overlay nodes, and migrate data items from one node to the other in order to do this. Care must be taken to process queries correctly and to robustly handle node failures during re-balancing. Furthermore, for network monitoring applications, the re-balancing operations must be

invoked relatively infrequently to avoid moving large volumes of data between overlay nodes.

For all these reasons, MIND uses a much simpler approach to load balancing that leverages on two properties of the data: (i) network traffic data is approximately stationary on diurnal time-scales (see Section II); (ii) the cuts on the MIND data-space can be scaled so that the hyper-rectangles contain approximately the same amount of data. We now describe the mechanics of this approach.

In MIND, a designated node collects, once a day, an approximate multi-dimensional *histogram* of the data distribution on each index, by aggregating the individual data distributions observed at each node. These per-index distributions are then sent to each node, which independently computes a *balanced cut* on the data-space. In such a balanced cut, a hyper-plane cuts a hyper-rectangle such that the number of data items on each division of the hyper-rectangle is approximately the same. The histogram of the data distribution is used to effect this, and the efficacy of load balancing depends upon the granularity of the bins in the histogram.

Having computed the balanced cut, MIND does not attempt to *migrate* historical data between overlay nodes. Rather, it maintains daily versions of each index (we have deferred an examination of version storage management to future work) separately, and the histogram describing the data distribution on one day is used to store data for the next. This is conceptually easy to do, since a histogram completely defines the balanced cuts and the relevant index versions that should be used will be evident from the query itself (*e.g.*, by examining the time interval described in the query as in our current implementation).

Our current prototype does not implement the on-line histogram collection algorithm mentioned above. In the experiments in this paper, we compute the balanced cuts off-line and install them at nodes. However, implementing the on-line histogram collection should not be a major endeavor.

F. Robustness to Node and Link Failure

MIND incorporates two classes of mechanisms to make the system robust to un-anticipated failures of nodes or overlay links.

Dealing with routing transients In our experiments, we have often observed transient overlay link failures, presumably caused by routing failures in the underlying network. Restoring a link by repeatedly attempting to reconnect to the peer node suffices in most cases and has the attractive property that it minimally perturbs the overlay. When successive reconnection attempts fail, the MIND node first attempts to discover whether the peer is alive or not, by routing a probe message on the hypercube overlay to its peer. When a neighbor of the peer receives this message and can attest to the peer's liveness, the node continues to attempt re-connection. The system attempts to repair the overlay (as described below) only if the peer is found to have failed.

During these transient link failures, a query may encounter a failure in greedy routing. The failed link may have been the only exit from the node towards the destination of the query. There is a fairly substantial literature on the subject of robust hypercube routing (see [16] for examples), but our current implementation uses a rather simple approach. When a query reaches a MIND node at which greedy routing fails, that node sends out an expanding-ring scoped broadcast along the overlay to find another node whose address overlaps the query's code to an equal or greater extent to that of the broadcast originator. Query forwarding resumes from that overlay node. We have seen several instances of this procedure being invoked during our experiments.

Dealing with node failure MIND also attempts to reconfigure the overlay after a node failure. Recall, that MIND nodes have addresses on the hypercube. MIND's recovery procedure is best understood by thinking of the overlay node addresses as forming a virtual binary tree whose leaves are the nodes themselves. On this tree, for example, nodes with addresses 000000 and 000001 are *siblings*. Furthermore, a node with an address 000010 is said to be on a sibling sub-tree of these nodes.

In MIND, when a node fails, its sibling takes over the hyper-rectangle associated with the failed node. It does this simply by shortening its code. Thus, in our example, if node 000000 fails, its sibling shortens its code to 00000. If *both* a node and its sibling fail, then a node in the sibling sub-tree (in our example, the node 000010) takes over. This procedure can be applied recursively. Notice that this recovery strategy may not preserve the balance in the hypercube. We have postponed the design of a balance-preserving recovery strategy pending a better understanding of the frequency of node failure—our sense is that because MIND is intended to be a managed infrastructure (as opposed to a truly peer-to-peer system in which nodes may join and leave at will), our current approach will suffice.

In MIND, this recovery procedure is closely aligned with our data replication strategy. Notice that the set of nodes that could potentially take over a node's hyper-rectangle are its neighbors on the hypercube overlay. A MIND index's availability can be tuned by carefully replicating data items at these neighbors. With the right choice of neighbors at which to replicate, the failover to replicas is made transparent. For example, let m be the desired level of replication and k be the code length of the original node. Then the replication nodes are those neighbors that have common prefixes of length $k-1, k-2, \ldots, k-m$ with the original node, and the system is robust to failure of m nodes. For example, consider node with code 000000 and $m = 3$, then the replication nodes are those with codes 000001, 000010, and 000100.

G. Software Structure and Implementation Details

We have implemented a MIND prototype using Java. Figure 5 describes the software structure of our prototype. Conceptually, our prototype consists of two separable components (separated with a dashed line in the figure), one dealing with network communication (the left half of the figure), and the other with data storage (the right half). Our implementation is largely event-driven, but uses three long-lived threads: one for

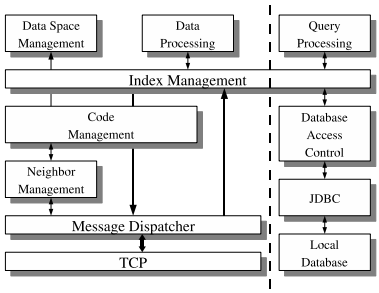

Fig. 5. The MIND node architecture.

TCP communication, one for message dispatch, and one for managing storage.

At the lowest level of the communication component is a dispatcher that sends and receive messages. Network maintenance messages including node join and leave, and neighbor status update are processed within the neighbor management and the code management components. Should a node's code (address) change, the code management module sends a request to the data space management module for remapping the multi-dimensional data space to the code. Application data and control messages are delivered to the index management component where messages are further classified and handed off to individual indices. Routing decisions for data and queries are made within the data insertion and query processing modules. The code management module interacts with the neighbor management module to maintain the overlay. The data space management module is also responsible for collecting and distributing histograms and rebuilding indices for load balancing purposes (as explained earlier).

The system also implements several modules that maintain and manage access to the data. Central to this functionality is the database access control (DAC) module, one for each index, which buffers database access requests in a queue and communicates with the local database via JDBC. MIND uses a mySQL backend to store the data items. When invoked, the DAC processes each pending insertion received from the network. DAC is tuned to handle the relatively high data insertion rates seen in network monitoring applications. Another important function of the DAC is to resolve queries. DAC builds an SQL statement for each pending query and sends it via JDBC to the local database. Upon receiving a response from the local database, the DAC builds one or more response messages and sends them directly to the query originator. Finally, since our robustness mechanisms affect many components, handling failure recovery has not been modularized in our implementation.

IV. Performance Evaluation

We have deployed our MIND prototype on PlanetLab. Our prototype is robust enough to handle insertions of tens of millions of flow records over several days. We have conducted several controlled experiments using our implementation, most of them on PlanetLab, to evaluate MIND's behavior at the scale of modern academic backbones, as well as its behavior at larger scales and under greater churn. This section reports the results of these experiments.

A. Methodology

For most of our experiments, we deployed a number of MIND instances on PlanetLab nodes (the node deployment strategy is discussed when we describe the experiments). In all experiments, we inserted aggregated flow records from two large backbone networks: Abilene and GÉANT. The flow records were obtained by processing NetFlow [6] or cflowd [11] information collected continuously from all the routers of the two research networks.

On the resulting MIND overlay, we created three indices, each corresponding to a real network monitoring or anomaly detection task.

1) Index-1 for port scan detection: This 3-dimensional index is built from the first three attributes of the aggregated flow records of the form:

$$(destination, timestamp, fanout, source, node),$$

where fanout determines the number of short connection attempts made by the source to the destination, and the other attributes are self-explanatory. Such an index is useful for detecting port scanning activity, using a query of the form:

> Find all the sources that attempted to connect to more than F hosts in destination D within the time period T.

2) Index-2 for unusually large flow detection: This 3-dimensional index is also built from the first three attributes of the aggregated flow records of the form:

$$(destination, timestamp, octets, source, node),$$

where *octets* denotes the total size of the flow in bytes. Such an index can be used to find unusually large flows between aggregates, using queries of the form:

> Find all flows destined for D that have carried at least O octets (or in between O_1 and O_2) within time period T.

3) Index-3 for camouflaged application detection: Finally, this 3-dimensional index is built from the first three attributes of the aggregated flow records of the form:

$$(destination, timestamp, flow\_size, source, dest\_port, node)$$

where the $flow\_size$ in a time window is the average traffic sent on each distinct connection from a source to a destination within the time window. Such an index can be used to identify network applications that use well-known ports to work around firewalls and packet filters (e.g., peer-to-peer applications [12]) or that tunnel their traffic using other application layer protocols to avoid connection charges (e.g., DNS tunneling [18]). The hallmark of such applications is an unexpected amount of traffic to ports on which one would not expect large traffic volumes. Such a pattern can be detected using a query of the form:

> Find all the flows either from source S, or to destination D, or both, that have flow size more than X and/or destination port P within time period T.

In addition, we also filter out small and "uninteresting" flow records. Specifically, we discard flow records with fanout less than 16, bytes less than 80 KB, and flow size less than 1.5 KB,

Days	Raw data	Index-1	Index-2	Index-3	Total
2004-9-1	863 M	2.18 M	2.70 M	1.54 M	6.42 M
2004-9-2	877 M	2.66 M	2.66 M	1.50 M	6.82 M
2004-9-3	815 M	2.67 M	2.74 M	1.43 M	6.84 M

TABLE I

ABILENE NETWORK DATA AMOUNT. ALL NUMBERS ARE IN MILLIONS.

Days	Raw data	Index-1	Index-2	Index-3	Total
2004-9-1	394 M	1.21 M	0.74 M	0.44 M	2.39 M
2004-9-2	388 M	1.21 M	0.75 M	0.46 M	2.42 M
2004-9-3	367 M	1.14 M	0.77 M	0.45 M	2.36 M

TABLE II

GÉANT NETWORK DATA AMOUNT. ALL NUMBERS ARE IN MILLIONS.

respectively for each index[2]. Furthermore, in our experiments the aggregation time window is chosen as 30 seconds for all three indices. Note that the time window size is a tunable MIND parameter depending on specific applications. In each case, we deem these thresholds as low enough to not be interesting from an anomaly detection perspective. In Table I and Table II, we list the summary of the data amount, including the raw data and the flow records after aggregation, we used for our experiments.

Finally, in each experiment, we issued queries that were uniformly sized with respect to all attributes other than the timestamp. For the timestamp, we always used a time interval of the last five minutes *i.e.,* all queries are interested in events that occurred in the immediately past 5 minutes. The uniform choice of attributes in other dimensions stresses the system by including some large and some small queries. The 5 minute query interval is representative of periodic monitoring queries that are continuously assessing the state of the network to determine potential anomalies.

In the following subsections, we measure several aspects of MIND performance: *insertion path length* (the number of overlay hops for tuple insertions), data *insertion latency*[3] , *query cost* (the number of nodes visited in order to resolve queries), the *query latency*, and the *data and traffic distribution* across nodes and links.

B. The Baseline Experiment

Our baseline experiment was designed to assess the realistic performance of the MIND prototype at the scale of network backbones. To achieve this, we carefully selected 34 PlanetLab nodes (11 from North America and 23 from Europe) such that their geographic locations were close (to within a city granularity in most cases) to the routers of Abilene and GÉANT backbones, respectively (recall that our traffic trace data sets also come from these backbones). This way we could emulate the network communication that an actual MIND deployment on those networks would experience, since most PlanetLab nodes communicate through these networks.

We inserted data collected during a period of three days (September 1-3, 2004) for a total of nearly 9 million flow records per day. We inserted the aggregated and filtered flow records at the same timescales as they would have been inserted into the real network: a few filtered flow records from each MIND node every 30 seconds. Due to the different packet sampling rates configured in the two networks (1/100 in Abilene and 1/1000 in GÉANT), more flow record tuples were injected from Abilene nodes than from GÉANT nodes.

[2]We chose 1024, 2MB and 128 KB as the maximum ranges respectively for these attributes after analysis revealed that these bounds were exceeded by less than 0.1% of the tuples. For these tuples, we assigned them the largest possible values.

[3]To measure this, we used the fact that PlanetLab nodes are synchronized using NTP. As such, our measurements might be affected by NTP errors, typically around tens of ms.

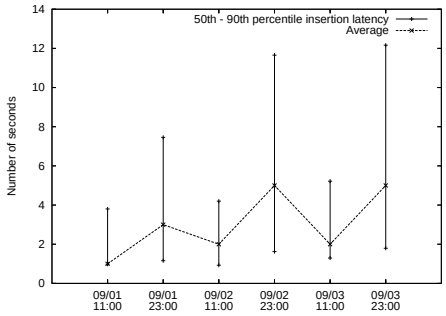

Fig. 6. Tuple 50th – 90th percentile insertion latency distribution.

We ran MIND continuously for three days, but, for brevity, present results for two different hour-long periods (11am to noon, and 11pm to midnight GMT) on each day.

Figure 6 shows the insertion latency for these six time intervals. Most of the insertions were accomplished within a few seconds with the median varying between 1 and 2 seconds and the mean varying between 1 and 5 seconds.

However, the insertion latency distribution has a long tail in many cases, as evidenced by the rather high 90th-percentile numbers. Two factors contributed to the rather long insertion latencies: queueing of data at overlay nodes caused by a transient hotspot, or by network dynamics (failed links or changes in the overlay topology). In particular, we observed that there were several tuples whose insertion took up to several tens of seconds. In one particularly pathological case, queueing delays at successive links delayed the tuple insertion by 48 seconds! We examined the slowest link on this path, plotting the transmission delays (Figure 7) observed on this link over the entire hour. In the first half of the hour, the transmission latencies across that link increased dramatically, impacting the queue lengths at the overlay node. We have found several such instances of poor performance and conjecture that they may be due to PlanetLab artifacts, *e.g.,* competing experiments or slow access links to PlanetLab.

The query cost measures the number of overlay nodes visited by a query, regardless of whether the query is forwarded or resolved by those nodes. Figure 8 shows the overall query cost distribution for all three indices measured during the 11:00 to noon hour on the first day. Recall that except for time interval fixed at 5 minutes, the ranges for all the other attributes were randomly selected, so that some queries were for relatively large parts of the data-space. Even so, MIND indexing is quite efficient at preserving the data locality, and over 90% of the queries involved 4 overlay nodes or less while retrieving the results.

Figure 9 depicts the query latency statistics for this ex-

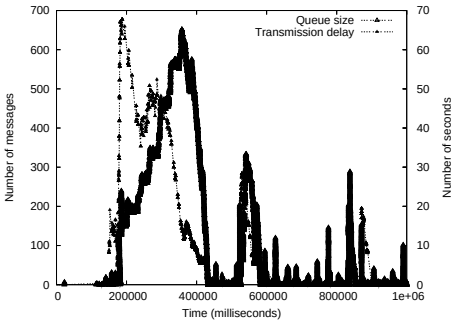

Fig. 7. Transmission delay and the sending side queue length distribution at the "slowest" link.

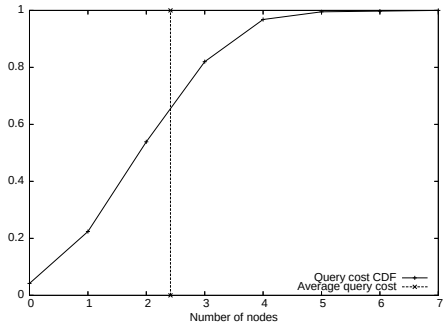

Fig. 8. Cumulative query cost distribution.

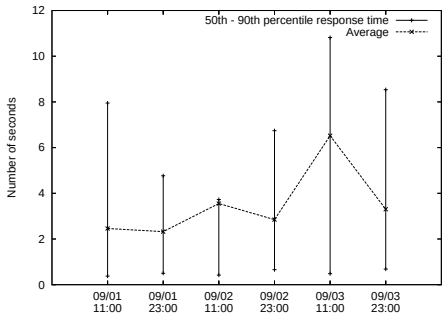

Fig. 9. 50th – 90th percentile query response time distribution.

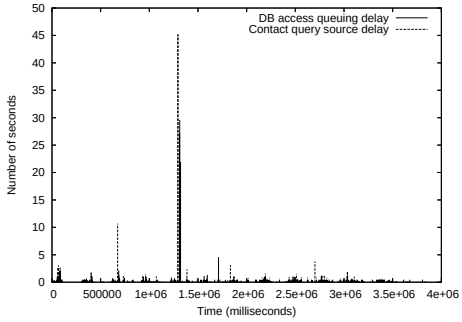

Fig. 10. Processing delay per query at the query hotspot node.

periment. Notice the low median query latency in the figure, about 500 ms. This is very encouraging from the perspective of anomaly detection: it indicates that it is feasible to deploy a distributed system based on MIND for on-line detection of Internet anomalies.

This figure also indicates that the query latency distribution is skewed—notice the high 90th-percentiles and means. In MIND there are two components to query latency: the time to route the query on the overlay, and the time to send the response directly back to the originator. The former component is qualitatively similar to insertion latencies (a fact borne out by the rough similarity between this plot and Figure 6). Furthermore, in general, the latter component is usually negligible (dominated by the round-trip time between the originator and the responder).

However, there were several instances in our experiments where the query responder could not connect to the query originator as a result of routing outages. In these cases, the time to send the response back to the query originator was significantly large. Figure 10 shows the time spent in resolving all queries during the time period from 23:00 to midnight on day 3 at a MIND node. There are two spikes back to back for two different indices corresponding to attempts by the query responder to contact the query originator. During this network outage, it took 45 seconds to establish the connection to the query source. This long delay is an artifact of our implementation, which attempts 3 times to establish a connection for sending query results back to the query source. Additionally, one of these queries was queued behind the other and suffered an additional delay, also an artifact of the MIND implementation where query database access is not interleaved with network transmission of query results.

Finally, we are interested in validating the balancing of network and storage load across the MIND infrastructure. Clearly, if a few nodes have to carry the bulk of the flow records and answer most queries then there would be no benefit in using a distributed solution when compared to a centralized approach.

In Figure 11 we plot the traffic load over each link as measured by the number of flow records that traverse the link. The figure consider the entire day of September 1, 2004. The traffic load is not distributed evenly across links mainly because of the difference in flow record volumes generated by the Abilene and GÉANT nodes. However, we can see that with MIND the busiest link carries an order of magnitude fewer records than what a centralized system would observe (i.e., nearly 9 million tuples!).

Furthermore, to validate the performance of our load-balancing strategy, we compute the number of flow records stored at each node for Index-1 at the end of Sep 2, 2004. The partition of the data space across nodes is based on the data distribution observer during the previous day. For comparison, we also compute the storage load when no load balancing is used, *i.e.*, the partition follows a "plain" binary-recursive scheme.

Figure 12 shows the number of records stored on each node using the two schemes. Without load balancing, the majority (90%) of the flow records are stored on just 8 nodes while 17 nodes are nearly empty! Instead, our load balancing scheme spreads the flow records more evenly across nodes: the overall average is 113825.

Note that there is still room for improvement in the load balancing scheme. Our goal here is to show that even a simple scheme can achieve extremely good results. However, one could identify better predictors for the distribution of flow records in the data space, rather than using the previous day

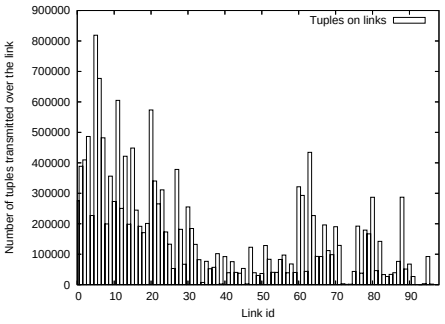

Fig. 11. The number of tuples traversing each link, Sep 01, 2004.

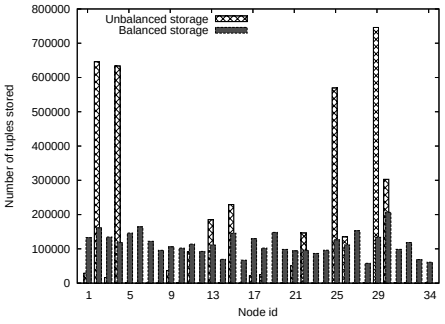

Fig. 12. The distribution of data storage across MIND nodes.

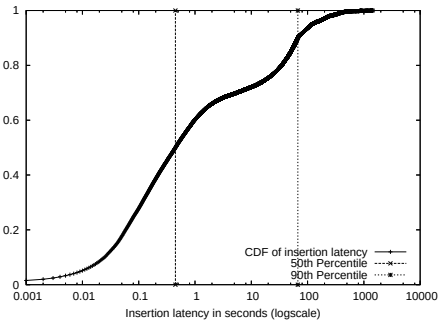

Fig. 13. The cumulative distribution of insertion latency on the 102-node MIND overlay.

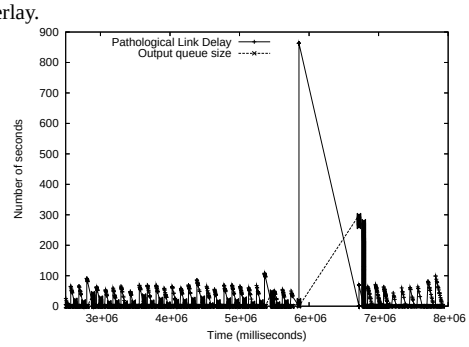

Fig. 14. Pathological link delays during our large-scale experiments with 102 nodes.

as we have done in this work. In addition, the choice of using 256 bins for the histograms is somewhat arbitrary. A larger number of bins would allow a better load balancing at the cost of higher communications overhead.

To conclude, our results are extremely encouraging. When nodes are deployed geographically close to the Abilene and GÉANT routers, our experiments show that a few nodes need to participate in the queries allowing MIND to provide low insertion and response times. We have also identified a few pathological events related to our current prototype implementation and PlanetLab-specific artifacts that result in larger response times. We believe that such artifacts will not arise when MIND is deployed on a dedicated infrastructure.

C. A Large-Scale Experiment

To explore MIND's scalability, we deployed the MIND prototype on 102 nodes. Unlike our baseline experiment, these nodes were arbitrarily chosen but were distributed across North America and Europe. During the course of our experiment, several nodes failed and re-joined the overlay so that the actual number of operational nodes varied from 70 to 102. In addition to scaling the network size, we also scaled up the network traffic inserting aggregated flow records from three days' worth of Abilene and GÉANT traces (about 11 million records of Index-1) at the rate of 1 record per second per node.

The results from this experiment are qualitatively similar to those obtained in the baseline experiment. Nearly 90% of insertions incur less than 5 hops on the overlay, but some of the insertions incur 1-2 hops more than the network diameter because MIND re-routed those insertions around failures. Furthermore, 90% of the queries visited less than 5 nodes, and at most 12 nodes were visited by any query. We have omitted graphs describing these results for reasons of space. Instead, we focus on the distribution of insertion latencies (the query latency distribution is qualitatively similar). Figure 13 describes the insertion latency distribution across the entire experiment. While the median latency is below 1 second, the tail of the distribution is long. As in our baseline experiment, there existed "slow" links in this experiment which significantly increased the insertion latency for several tuples. Figure 14 plots the link latency and queue lengths across one link which exhibited a fairly pathological behavior. One readily apparent feature of this plot is the large network outage about 2/3rds of the way into the trace. More interestingly, though, the plot exhibits very interesting periodic spikes in the link latency (causing an attendant increase in queue lengths). An examination of PlanetLab ping latency measurements reveals no such behavior, so we conjecture that these spikes are caused by PlanetLab scheduling. As we have discussed before, a more aggressive re-routing strategy than the one our prototype currently employs can alleviate this problem.

D. Robustness

Finally, we evaluated the robustness of the MIND design to node failure. To do this, we deployed a 102-node MIND prototype on a local cluster of nodes, with several instances of MIND running on each physical node. This deployment enabled us to fail individual nodes in a controlled fashion and observe the availability of data in MIND, at various levels of replication (Section III-F).

We inserted all three days' aggregated and filtered flow records for Index-1, but varied the degree of replication and measured the fraction of successfully completed queries at varying levels of random node failures. Figure 15 shows

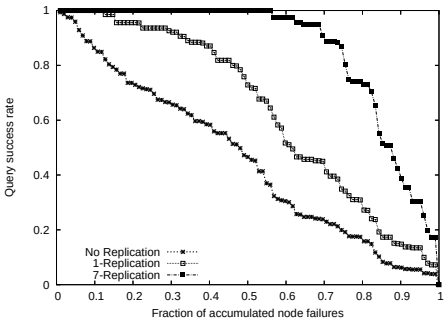

Fig. 15. Neighbor replication for data availability robustness under random accumulated node failures.

the results of our experiments when using 0, 1, and "full" replication per flow record. In full replication, each item is replicated at all overlay neighbors.

Without replication, the fraction of successful queries decreases almost linearly with the number of node failures. This is because each node failure will make approximately $\frac{1}{N}$ of the total amount of data unavailable after the load has been balanced among N nodes. With one replica per data item, MIND can sustain the failure of 15% of the nodes without compromising the data availability. Figure 15 also shows that with full replication, MIND can easily survive over 50% node failures without loss of data availability. In general, for a N-node hypercube with k replicas, $0 < k \leq \log N$, the probability that data loss will occur under a series of k random failures is $1/\binom{n}{k+1}$. Of course, replication storage and transmission cost scales linearly with the degree of replication.

V. MIND and Anomaly Detection

MIND is intended to support flexible examination of traffic volumes in order to detect potential network anomalies. The results of MIND queries can be used to *drill-down* into the data, *i.e.,* trigger a more detailed examination of flow records or packet traces. To get an initial sense of how well MIND achieves this goal, we used results from Lakhina *et al.*'s work on the efficacy of *off-line* centralized anomaly detection [15]. They detected several anomalies on the Abilene backbone on December 18th, 2003. These anomalies were of three types: alpha flows, DoS attacks and port scans [15].

We conducted an experiment to see if (and how well) MIND is able to capture the same anomalies. We constructed an 11-node MIND overlay on PlanetLab congruent to the Abilene backbone topology, and built two indices on this overlay (Index-1 and Index-2 from Section IV-A). We then inserted aggregated flow records from about 25 minutes worth of Abilene backbone traces during which several anomalies were observed by [15]. Finally, we issued queries on traffic volumes (multi-dimensional hyper-rectangles) circumscribing the observed anomalies. Our goal was to see how quickly MIND responded to those queries (the average response time when the query is issued from every node), and how closely our traffic volumes circumscribed the records corresponding to the anomaly.

To detect DoS attacks and port scans, we issued a query on Index-1 of the form:

> Find all flow records whose *fanout* is greater than 1500, and whose *timestamp* fell within a specified 5-minute interval.

Anomaly Time window	Result size	Actual Anomaly size	Average Response time(s)
15:45	43	2 alpha flows	2.09
15:50	38	2 alpha flows	1.38
15:55	55	2 alpha flows	1.47
19:50	10	2 DoS, 1 scan	0.81
19:55	13	2 DoS	0.79

TABLE III
REAL WORLD ANOMALY DETECTIONS USING MIND INDICES

To detect alpha flows, we issued a query on Index-2 of the form:

> Find all flow records whose total size *fanout* is greater than 4000000, and whose *timestamp* fell within a specified 5-minute interval.

Table III summarizes the results of our experiment. The first column identifies the time of occurrence of the anomaly. In each case MIND returned a super-set of the flow records that constituted the anomaly. Notice how closely we can constrain the traffic volume: in all cases, the number of flow records returned by the MIND query is relatively small (tens of flow records). Of course, in our experiment, anomalies were known a priori. In practice, a network operator would arrive at this by programmatically querying progressively smaller traffic volumes. Furthermore, note that the average response times are encouraging, on the order of a second. Finally, a useful by-product of a MIND query response (which illustrates the fact that MIND essentially correlates data from different nodes) is the exact set of network monitors (Abilene routers) which observed the anomalous traffic. For example, for the 2 DoS flows captured in time window 19:55, the returned tuples show that the paths taken by the DoS flows included the Abilene backbone routers in the following cities:

DoS 1	Chicago, Denver, Indianapolis, Kansas City, Los Angeles, Sunnyvale
DoS 2	Chicago, Indianapolis

VI. Related Work

To our knowledge, MIND is the first system designed for distributed monitoring. Most closely related to MIND is the recent literature on support for range searches using overlays [4], [25], [21], or geographic embeddings [17]. But none of these have been specifically designed for wide-area network monitoring applications.

Mercury [4] replicates data on k separate rings where k is the number of indexed attributes. It can resolve multi-dimensional range queries by carefully querying one of the rings based on the selectivity of the data. It dynamically re-balances the storage load on the overlay by using random sampling to approximate the overall data distribution. Mercury's high degree of replication and dynamic load balancing mechanisms may not be suited to the network monitoring application, for reasons discussed previously.

As we have shown with our mismatch metric, traffic distributions can vary significantly hour to hour, so the kinds of continuous rebalancing proposed in this and like systems can be highly inefficient.

SkipIndex [25] is a distributed implementation of the SkipGraph [3]. Similar to MIND, SkipIndex follows the same data space embedding scheme described in [17]. Unlike MIND, however, which decouples the data-space embedding from the overlay construction, in SkipIndex the entire index may need to be rebuilt if the data distribution changes significantly. For high volume insertions as in traffic monitoring, for example, MIND's approach of histogram-based re-inserting data into a new version of the index might perform better.

PHT [21] is a qualitatively different approach that attempts to build support for range queries on an underlying DHT overlay. It treats the prefix of a key as a key, and consecutively looks up all prefixes of the key, increasing one bit each time, until the key is found. PHT has been shown able to support range queries in a 2-D space [5]. However, the performance of PHT, esp. its storage load balancing, under higher dimensional spaces has not been shown to our knowledge.

MIND is heavily influenced by the DHT literature [22], [24], [26]. Its geometry resembles that of a class of DHT systems that use hypercubic structures, but differs significantly from DHT systems in the way data items are hashed to nodes.

Finally, prior work on multi-dimensional indexing in wireless sensor networks (DIM [17]) is related to MIND. However, unlike MIND, DIM relies on a geographic embedding of the data space. This difference completely changes the overlay management and data routing mechanisms and, as such, MIND is a qualitatively different from DIM.

Ongoing research on distributed database query engines such as PIER [10] focuses on sophisticated query execution (*e.g.*, joins). Such a system might be a component of the network monitoring query sub-system discussed in Section II. Recent work on anomaly detection is also related to our work [13], [15]. Systems like these can be built on top of MIND, since MIND indices can first be used to detect possible anomalies, and these systems can then be used to improve the robustness of the detection by detailed traffic analysis from a subset of the traffic monitors.

VII. Conclusions and Future work

We have described the design of MIND, a distributed system to support fast and robust detection of network-wide anomalies. Our initial evaluations suggest that it might be feasible to use MIND to perform on-line near real-time anomaly detection in a large network. However, much remains to be done to achieve this goal, such as how to design queries for periodic monitoring and how to automate the process of drilling down to find potential anomalies. Then implementing the actual algorithms that examine detailed traffic traces based on MIND's query results, are all interesting challenges.

Another important issue is to study how real time can a system like MIND be. In general, the smaller the time windows are for aggregation, the more data are moved around. Finding the right compromise is an interesting issue as this compromise is not a static one.

We also intend to address other open issues in the overlay design area: fast re-routing to avoid re-connection latencies, more sophisticated but practical alternate path routing on the hypercube, and automated histogram collection and version management.

Acknowledgment

We would like to thank Anukool Lakhina for providing the data on traffic anomalies in the Abilene network, the other members of ENL group at USC for their feedback on the previous drafts of this paper, and the anonymous reviewers for their useful comments.

References

[1] Abilene Backbone Network. http://abilene.internet2.edu.
[2] M. Adler, E. Halperin, R. M. Karp, and V. V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proc. of STOC'03*, pages 575–584.
[3] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Baltimore, MD, USA, 12–14 Jan. 2003.
[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM 2004*.
[5] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker. A case study in building layered dht applications. In *Proceedings of ACM Sigcomm*, Aug. 2005.
[6] Cisco Systems. NetFlow services and applications. White Paper, 2000.
[7] GEANT Backbone Network. http://www.dante.net.
[8] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), Oct. 2003.
[9] K. Gummadi, R. Gummadi, S. Ratnasamy, S. Shenker, and I. Sotica. The impact of dht routing geometry on resilience and proximity. *ACM SIGCOMM, 2003.*
[10] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, 2003.
[11] Juniper Traffic Sampling. http://www.juniper.net/techpubs.
[12] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy. Transport layer identification of P2P traffic. In *Proceedings of ACM Sigcomm Internet Measurement Conference*, Oct. 2004.
[13] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium (Security 2004)*, Aug. 2004.
[14] C. Kreibich and J. Crowcroft. Creating intrusion detection signatures using honeypots. In *Proceedings of ACM HotNets*, Nov. 2003.
[15] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic distributions. Technical Report BUCS-TR-2005-002, Department of Computer Science, Boston University, Feb 2004.
[16] S. Lam and H. Liu. Failure recovery for structured p2p networks: protocol design and performance evaluation. *ACM SIGMETRICS, 2004*.
[17] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the ACM Sensys*, Nov. 2003.
[18] Name Server Transport Protocol. http://www.freshmeat.net/projects/nstx.
[19] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23), Dec. 1999.
[20] PlanetLab Network. http://www.planet-lab.org/.
[21] S. Ramabhadran, J. M. Hellerstein, S. Ratnasamy, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables, 2004.
[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM 2001*, pages 161–172.
[23] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of Usenix LISA*, 1999.
[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large–scale peer–to–peer systems. *International Conference on Distributed Systems Platforms (Middleware), Nov 2001.*
[25] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. http://www.cs.princeton.edu/ chizhang/research.html.
[26] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault–tolerant wide–area location and routing. *Tech. Report UCB/CSD-01-1141, U.C. Berkeley, April 2001.*