

**DISTRIBUTED DATA COLLECTION: ARCHIVING, INDEXING,
AND ANALYSIS**

A Dissertation Presented

by

PETER DESNOYERS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2008

Computer Science

© Copyright by Peter Desnoyers 2008

All Rights Reserved

**DISTRIBUTED DATA COLLECTION: ARCHIVING, INDEXING,
AND ANALYSIS**

A Dissertation Presented

by

PETER DESNOYERS

Approved as to style and content by:

Prashant Shenoy, Chair

Deepak Ganesan, Member

James Kurose, Member

Tilman Wolf, Member

Andrew Barto, Department Chair
Computer Science

To Elizabeth, Christopher, Madeline, and Little Boy.

ACKNOWLEDGMENTS

I would like to thank the following people who played a role in my graduate career, as well as many who are not named here.

My advisor, Prof. Prashant Shenoy, whose support, collaboration, and advice have been invaluable. Prof. Deepak Ganesan, whose collaboration, advice, ideas, and bike riding have all been appreciated. Prof. Jim Kurose for his valuable discussions over the years, and for his role on my thesis committee with Prof. Tilman Wolf; their comments and advice have contributed significantly to this work. Thanks are due Prof. David Jensen for his advice on the modeling work presented here, as well as his wonderful Research Methods course.

Sharon Mallory, Leeanne Leclerc, Karren Sacco, Pauline Hollister, and Tyler Trafford have all helped me to surmount various administrative and technical hurdles. The graduate students of the LASS and Networks labs have provided advice, friendship, and encouragement during these years.

And finally, I would like to thank my wife, Elizabeth Glaser, who supported me and our family financially and emotionally during these years, and our children, Christopher and Madeline, who put up with my long absences and the move to Amherst and back. Without their efforts, this work would not have been possible.

ABSTRACT

DISTRIBUTED DATA COLLECTION: ARCHIVING, INDEXING, AND ANALYSIS

FEBRUARY 2008

PETER DESNOYERS

S.B., MASSACHUSETTS INSTITUTE OF TECHNOLOGY

S.M., MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

As computing hardware becomes more powerful and systems become bigger, the amount of data we can collect within a system grows seemingly without bounds. These systems share a common characteristic: the volume of raw data available is far higher than can be handled by the application or user. The key problem is thus to identify some small amount of data which is of interest, obtaining it and presenting it to the user.

In this thesis we examine distributed data collection environments, and advocate *data-aware* and *storage-centric* approaches. Collected raw data is stored within the network, and only the most relevant information is presented to the application. We apply two strategies: (i) archiving, indexing and querying to retrieve discrete portions of data, and (ii) mathematical modeling to extract relationships spread more diffusely across the data. We propose mechanisms for archiving, indexing, and analysis, and describe them in the context of systems incorporating them.

First we address storage and indexing of high-speed event data in a resource-rich environment. A disk-based storage system operates on commodity hardware, yet guarantees writing at high rates to avoid loss. A *signature file*-based index allows indexing of high-speed data in real time, for efficient *ad hoc* querying. A network monitoring system based on these mechanisms is presented and evaluated.

Next we leverage the resources of a few resource-rich systems within a resource-constrained environment to index and route queries to remote sensors. These sensors send summaries of stored data to more capable *proxy* nodes, which use a novel search structure, the Interval Skip Graph, representing multiple records by a single imprecise key. We present a prototype implementation of a sensor network storage system based on these mechanisms.

Lastly we address analysis and model-building from feature-rich but poorly structured events. In our approach, statistical machine learning techniques are used to build models of application and system behavior in a data center, relying on an automated feature identification mechanism to identify model inputs from within the raw data stream. We present and evaluate *Modellus*, a data center monitoring and analysis system based on these mechanisms.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	v
ABSTRACT	vi
LIST OF TABLES	xii
LIST OF FIGURES	xiii
 CHAPTER	
1. INTRODUCTION	1
1.1 Data Management in Distributed Data Collection Systems	2
1.1.1 Distributed Data Archiving	2
1.1.2 Indexing and Querying	3
1.1.3 Analysis, Modeling, and Prediction	3
1.2 Thesis Contributions	4
1.3 Thesis Outline	6
2. NETWORK PACKET MONITORING	7
2.1 Introduction	7
2.1.1 Background and motivation	8
2.1.2 Research Contributions	9
2.2 Requirements and Challenges	10
2.2.1 Hyperion Design principles	12
2.3 Hyperion Stream File System	13
2.3.1 Stream Storage	14
2.3.2 Why Not a General Purpose File system?.....	14

2.3.3	StreamFS Design	17
2.3.4	StreamFS Organization	18
2.3.5	Striping and Speed Balancing	19
2.3.6	Read Addressing and Implementation	20
2.4	Indexing Archived Data	21
2.4.1	Signature Index	22
2.4.2	Multi-Level and Bit-Sliced Signature Indexes	23
2.4.3	Distributed Index and Query	24
2.5	Implementation	25
2.6	Experimental Results	26
2.6.1	Experimental Setup	26
2.6.2	Database and General-Purpose File System Evaluation	27
2.6.3	StreamFS Evaluation	29
2.6.4	Index Evaluation	32
2.6.5	Prototype Evaluation	35
2.7	Related Work	39
2.8	Summary	41
3.	WIRELESS SENSOR NETWORK STORAGE	42
3.1	Introduction	42
3.1.1	Background and Motivation	43
3.1.2	Contributions	44
3.2	Requirements and Design	45
3.2.1	System Model	45
3.2.2	Usage Models	46
3.2.3	Design Principles	47
3.2.4	System Design	48
3.3	Proxy Level - Indexing	49
3.3.1	Summary Index	49
3.3.2	Index structure	50
3.3.2.1	Skip Graph Overview	50
3.3.2.2	Interval Skip Graph	53
3.3.2.3	Sparse Interval Skip Graph	56
3.3.3	Low-Resolution Redundant Indexing	58

3.4	Index Evaluation	59
3.4.1	Insert/Delete Performance	60
3.4.2	Lookup Performance	60
3.4.3	Variations	61
3.5	Sensor Level - Data Handling	62
3.5.1	Data Sensing and Storage	63
3.5.2	Summarization	64
3.5.3	Query handling	65
3.5.4	Adaptive summarization	65
3.6	Prototype Evaluation	66
3.6.1	Prototype Implementation	66
3.6.2	Query processing latency	67
3.6.3	Adaptive Summarization	69
3.7	Related Work	70
3.8	Conclusion	72
4.	DATA CENTER PERFORMANCE ANALYSIS	73
4.1	Introduction	73
4.1.1	Background and Motivation	74
4.1.2	Contributions	75
4.2	Data Center Modeling Methods	76
4.2.1	Problem Formulation	76
4.2.2	Workload-to-utilization Model	79
4.2.3	Workload-to-workload Model	80
4.2.4	Model Composition	81
4.3	Automated Model Generation	83
4.3.1	Feature Selection	84
4.3.2	Stepwise Linear Regression	87
4.4	Accuracy, Efficiency and Stability	89
4.4.1	Model Validation and Adaptation	90
4.4.2	Limiting Cascading Errors	91
4.4.3	Stability Considerations	92

4.5	From Theory to Practice: Modellus Implementation.....	92
4.5.1	Modellus Nucleus	93
4.5.2	Monitoring and Scheduling Engine	95
4.5.3	Modeling and Validation Engine	96
4.6	Results	97
4.6.1	Experimental Setup	97
4.6.2	Model Generation Accuracy	99
4.6.3	Model Composition	102
4.6.4	Cascading Errors	103
4.6.5	System Overheads	105
4.6.6	Limitation of our Approach	107
4.7	Data Center Analysis	107
4.7.1	Online Retail Scenario	108
4.7.2	Financial Application Analysis	109
4.8	Related Work	110
4.9	Conclusions	112
5.	CONCLUSIONS	114
5.1	Conclusion	114
5.2	Summary of the Thesis	114
5.3	Future Work	115
	BIBLIOGRAPHY	117

LIST OF TABLES

Table	Page
2.1 StreamFS Test System	27
2.2 Postgres Insertion Performance	27
2.3 Index Calculation Performance Parameters	33
2.4 Hyperion communication overhead overhead in K bytes/sec	37
2.5 Network Monitoring and Query Systems	39
3.1 Proxy and sensor platforms used in prototype	66
3.2 Comparison of Distributed Index Structures	71
4.1 Data Center Testbed and Applications	98
4.2 CPU and data overhead for Modellus log processing on several workloads	105
4.3 Model training times for Workload-to-Utilization (W2U) models	106
4.4 Model training times for Workload-to-Workload (W2U) models	106
4.5 Case study: Predicted impact of workload changes	108
4.6 Trading system traces - feature-based and naïve rate-based estimation vs. measured values	110

LIST OF FIGURES

Figure	Page
2.1 Write arrivals and disk accesses for single file per stream	15
2.2 Logfile rotation	15
2.3 Log allocation in StreamFS, LFS	16
2.4 StreamFS metadata structures. A record header is prepended to record written by the application; block headers are written for each fixed-length block, block map is written for every N (256) blocks, and one file system root.	18
2.5 Hyperion multi-level signature index	22
2.6 Streaming write-only throughput by file system	28
2.7 XFS vs. StreamFS write only throughput, showing 30 second and mean values	29
2.8 Scatter plot of StreamFS and XFS write and read performance.	30
2.9 Streaming file system read performance	31
2.10 Sensitivity of performance to number of streams	32
2.11 Signature density when indexing source/destination addresses, vs. number of addresses hashed into a single 4KB index block	34
2.12 Single query overhead for summary index	35
2.13 Single query overhead for bit-sliced index	36
2.14 Packet arrival and loss rates	37
2.15 Forensic query example	38

2.16	Outline of Queries for Forensic Case Study	39
3.1	Architecture of a multi-tier sensor network	45
3.2	A skip list of sorted elements	51
3.3	Skip List Search Algorithm	52
3.4	Skip Graph of 8 Elements showing search operation with two messages.....	52
3.5	Interval Skip Graph and search for intervals containing key <i>13</i>	54
3.6	Distributed Interval Skip Graph	55
3.7	Sparse Interval Skip Graph.....	58
3.8	Skip Graph Insert Performance	60
3.9	Skip Graph Lookup Performance	61
3.10	Skip Graph Overheads	62
3.11	Sensor Summarization	64
3.12	Query Processing Latency	67
3.13	Query Latency Components.....	68
3.14	Impact of Summarization Granularity.....	70
4.1	Application models	78
4.2	Example 2-tier request flow	80
4.3	Composition of the basic models.....	81
4.4	HTTP feature enumeration algorithm	86
4.5	SQL Feature Enumeration	86
4.6	Stepwise Linear Regression Algorithm.....	88

4.7	Modellus components	93
4.8	Model training states	95
4.9	Data Center testbed	99
4.10	Request breakdown over time during a single test run	100
4.11	Prediction performance for TPC-W and OpenForBusiness sharing the same back-end database	101
4.12	Model composition - errors in predicting MySQL server utilization from tier 1 (HTTP) input features	101
4.13	Error CDF when predicting RuBIS server utilization from HTTP traffic	101
4.14	Error CDF when predicting MySQL server utilization from query traffic	101
4.15	Learning curve for predicting RUBiS server utilization from HTTP features	102
4.16	Learning curves for composed and direct prediction	103
4.17	Web services - cascade errors. Table entries at (\mathbf{x}, \mathbf{y}) give the absolute RMS error of prediction for utilization at \mathbf{y} given inputs at \mathbf{x}	103
4.18	Errors in prediction from composed model due to multiple downstream servers	104
4.19	Errors in prediction from composed model due to summing inputs from multiple upstream models	104
4.20	Growth of prediction error as data variability increases	105

CHAPTER 1

INTRODUCTION

With the growth in size and speed of networked and distributed systems, the volume of raw data we may record within a single system, network, or installation has increased almost without bound in recent years. The result in many cases is a system overloaded with raw data, perhaps to the extent that we can no longer locate the information the user actually needs. Data collection in such systems remains a challenge, due not to the difficulty of sensing or acquiring the data, but of locating and transmitting the small subset of that data needed by the user.

Data collection and selection is common to many systems, which vary greatly not only in scale, but in sensing modalities—i.e. what data is collected, and what mechanism is used acquire that data. Some sensor networks, for instance, monitor only simple physical phenomena such as temperature and humidity. In others the data source may be of higher speed and richer content, providing far more information than a slowly changing scalar value. The nature of the data being collected varies as well, from physical processes like temperature or radar to collection of output from computer applications.

Because of the wide variety in scale and modality in data collection systems, there is no common set of components and protocols which can serve for all. However, there are certain characteristics which are common to most, if not all, such systems:

- **Data acquisition:** Data collection is different from “data processing”. The inputs to a system come from outside that system; they are acquired in some fashion, and then some function of these inputs is presented to the user.

- **Data transformation:** In a wide class of data collection networks—the class considered in this thesis—the network does not just carry data, but rather it understands it. The collection system is able to use this knowledge of the schema or semantics of acquired data to filter, transform, or otherwise process the data being collected.
- **Data transmission:** In order to collect data, ultimately some of it must be transmitted to the user or application which is collecting it. In systems with high volumes of raw data, this transmission step is often problematic due to data volumes exceeding the capacity available for transmission.

1.1 Data Management in Distributed Data Collection Systems

There are many other aspects to such systems; for instance, the entire field of sensor network research focuses on particular forms of data collection networks. This thesis focuses on data management, and in particular on methods for coping with data overload in data collection systems. In the systems we examine there are two strategies for handling this data overload, depending on the structure of the data, as in some cases the application requires a tiny part of the data, while in others it requires a small summary over a larger part of the data. In the first case the strategy we advocate is one of archiving and indexing data as it is acquired, within the network, and only retrieving it when needed by the application. In the second case we argue for the use of mathematical modeling techniques to extract user-meaningful information from unstructured data. We next discuss the components of these strategies in more detail.

1.1.1 Distributed Data Archiving

Data archiving is an important function in any application which requires access to past data; for instance, the importance of a piece of data may not be known until after the fact. In a pure streaming system, where persistent queries are posted and act as filters on arriving data, implementing such an application would be difficult. In particular, it would be

necessary to request all the data which *might* be of use, to avoid missing anything that turns out to be important. In this case a central application itself is acting as an archival store, keeping information against the possibility that it might be needed. If communications bandwidth is limited, especially in contrast to storage bandwidth local to the remote system, it will be far more efficient to store data in a distributed fashion, and only retrieve it for the application when necessary.

The nature of such a distributed store varies with the scale of the system. In small wireless sensor networks, flash memory (particularly NAND flash) is the most appropriate local storage medium. In resource-rich systems, such as those composed of server-class machines, the storage resource of choice is magnetic disk. In each case a network storage system must cope with the particular challenges posed by the underlying technology, while still meeting application goals such as storage rate.

1.1.2 Indexing and Querying

If data is to be archived, then a query mechanism is needed to allow an application to retrieve it after the fact. The challenges in implementing such a query mechanism will vary according to the characteristics of the system in which it is implemented. In cases where large numbers of nodes each store moderate amounts of data, the primary challenge will be to efficiently locate query matches and route queries without exhaustively searching across nodes. If the amount of data archived on each node is large, as is the case in some higher-speed applications, and the number of nodes is modest, then forwarding a query to the appropriate node will be easier, and the challenge will be to locate the requested information once we get there.

1.1.3 Analysis, Modeling, and Prediction

Indexing and querying represents a simple form of data analysis, organizing data according to its characteristics and then dividing it by whether or not it matches an application-specified query. Deeper data analysis may be performed within a data col-

lection system, as well. Constructing models of the monitored phenomena or process is one form of such analysis, providing information useful for prediction and compression. Predictive models may even be considered a counterpart to archiving, as where one allows queries to be made into the past, the other allows queries of the (predicted) future.

In simple cases identifying the variables to be included in the model is straightforward; if there is only one measured value, for instance, then it will be the model input. However, in many cases there are a number of monitored variables or *features* which may be used for prediction, and it is necessary to choose the significant features to incorporate into the model, while ignoring irrelevant ones. For systems where these features do not change over time or from one instance to another, this may be done manually by human experts; an automated process at run-time can then learn model parameters. However, in other cases these features change over time or from case to case, and such manual feature identification is infeasible. What is needed, instead, is an automated method for model derivation. We propose the use of *statistical learning techniques* for building system response models based on collected raw data, in order to create system-specific models for behavior prediction.

1.2 Thesis Contributions

Each of these strategies is only a component of a system, and cannot be deployed and effectively evaluated in isolation. In this thesis we present a number of advances in the areas of data archiving, indexing, and analysis, embodied in three distributed data collection systems.

The first set of mechanisms address the issue of storage and indexing of high-speed sensed data in a resource-rich environment. A task-specific file system is used to provide both the performance guarantees and storage management functions needed by this application; for this task it is shown to offer significant performance increases over general-purpose file systems. A streaming index based on *signature files* [28] is used to index data at high speed for query and retrieval, and is also used as part of a distributed index

which routes queries across multiple monitors. We present the *Hyperion*¹ system, a network packet monitoring system with archiving and online query capability based on these mechanisms.

Next, we present a system which leverages the resources of a non-homogeneous sensor network to efficiently index and route queries to where data is stored on resource-poor sensing nodes. These nodes archive sensed data to flash storage, and then summarize information to more capable *proxy* nodes. The proxies, in turn, use a novel distributed search structure, the Interval Skip Graph, to index and search this summarized information. The summarization process reduces bandwidth consumption for indexing by representing multiple data items as a single inexact range, introducing a trade-off in overheads between index updates and queries, and an adaptation mechanism is used to tune the degree of summarization for optimal efficiency. We evaluate these mechanisms in *TSAR*,² a storage and retrieval system for low-power wireless sensor networks.

Lastly we address the problems of analysis and model-building when predictive data consists of feature-rich events. We monitor and collect resource consumption and application behavior across a set of data center servers, and then use statistical machine learning techniques to build predictive models of application and system behavior. These models retain predictive power across significant shifts in application load and request mix. Rather than requiring by-hand analysis of applications, we use an automated feature extraction and selection mechanism to analyze application monitoring data and identify elements which comprise the model. We present and evaluate *Modellus*,³ a data center monitoring and analysis system based on these mechanisms.

¹Hyperion, a Titan, was the Greek god of observation

²Tiered Storage ARchitecture

³Latin, root of “model”

1.3 Thesis Outline

The remainder of this thesis is structured as follows. In Chapter 2 we describe our file system and index approaches for high speed network monitoring, and evaluate the Hyperion implementation. Chapter 3 presents storage and retrieval mechanisms for resource-constrained networks like wireless sensor networks, which are evaluated in TSAR. Chapter 4 explores methods for feature extraction and model derivation from data center monitoring logs, and evaluate these methods in the Modellus system for data center monitoring and analysis. Lastly, we conclude in Chapter 5.

CHAPTER 2

NETWORK PACKET MONITORING

Network packet monitoring is an application involving extremely high-speed streams of relatively structured data. The amount of data captured on a single interface is far more than can be handled by the user, who typically is only interested in a tiny subset of the entire packet event stream. In this chapter we present Hyperion,¹ a network packet monitoring system capable of archiving captured traffic at high speed and performing online queries against this data, selecting and returning only those records of interest to the user.

2.1 Introduction

By *network packet monitoring* we refer to the passive collection of packet information such as headers, flow identifiers, etc. at various points within a communications network. Such information has become useful for a multitude of purposes. In security applications, packet-level monitoring can detect intruders and malware, or unauthorized usage. For network management and troubleshooting, it can provide visibility into network interactions, allowing communications errors to be located and corrected. In these and many other applications, one of the key features of a packet monitor is the ability to “find a needle in a haystack”—i.e. to locate what may be a very small number of packets within a stream of uninteresting data.

¹Hyperion, a Titan, was the Greek god of observation

2.1.1 Background and motivation

Network packet monitors, such as the widely used Tcpdump [42] and Snort [50], are used to capture and display network packets for various purposes. They may capture packet headers only, thus identifying most characteristics of a traffic flow or application, or examine the entire packet. Typically they incorporate a sophisticated filter capability in order to identify packets or flows of interest, which may be hidden in a high-speed stream of data. Such systems may range from single systems to large, distributed monitoring systems such as AT&T's GigaScope [20].

Perhaps the primary challenge in creating a packet monitoring system is that of handling network speeds. At speeds of 100s of kilobits or even megabits, the capture, storage, and analysis of network data could be performed trivially on modern hardware by any number of general-purpose software tools. However, a single heavily loaded gigabit link might carry 800mbit/s of bi-directional traffic; with a typical mean packet size of 500 bytes, this corresponds to a rate of 200,000 packets per second. Capturing and filtering traffic at this speed is a challenging problem, and has spawned a number of innovative technical solutions, up to and including special-purpose hardware [91].

Almost all packet monitor systems to date have coped with this challenge by implementing streaming query systems—a query or filter is specified *a priori*, and those packets matching the query are returned as results. In some systems, captured and possibly filtered data may be stored for later offline analysis; however, such analysis requires either exhaustive search or the construction of indexes on this data, both of which are highly time-consuming. In many cases, however, the data being sought is not known until after the fact: e.g. in a security application, the need to search for an intruder may not be known until after a compromised system is discovered. Streaming query systems such as implemented in most network monitors, however, are unable to provide the interactive *retrospective queries*—ready access to past data—which would be needed to perform such after-the-fact analysis.

The data storage capacity to build a system for archiving high-speed network traffic is available today. If we sample 90-byte packet headers from the gigabit link above, the data rate to be archived is about 18Mbyte/sec, which is well within the range of disk storage systems. To store 24 hours of this traffic would require 1.5TB, or about \$300 worth of raw storage at today's prices; building a system with the storage capacity to archive a week of gigabit traffic is clearly possible with standard computing equipment. However, to date there appear to be no systems which are able to archive and query data at these rates and volumes. Several commercial network forensics systems (e.g. Sandstorm NetIntercept [75]) use conventional file systems and databases for storage and look-up, limiting their capture rate and ability to handle simultaneous queries. The Network Time Machine (Kronox1, Paxson et al. [48]) trims the data stream instead, storing and indexing only the first few packets from high-volume flows.

2.1.2 Research Contributions

The Hyperion system described in this chapter is a network packet monitor which performs comprehensive archiving and indexing of captured packet headers, and provides on-line query capability for searching this stored information. Three components of this system are significant: (i) A special-purpose streaming file system provides guaranteed storage bandwidth for incoming data, even while queries are being performed. (ii) A signature file-based index, which can be computed efficiently and stored to disk as packets arrive, while providing efficient after-the-fact search capability. (iii) Lastly, a distributed index, based on the same structure as the local on-disk index, is used for efficient query routing in a distributed system of Hyperion monitors.

The remainder of this chapter describes the goals and constraints, design, and implementation of the Hyperion system, followed by an evaluation of its performance. In Section 2.2 we discuss the goals of the Hyperion system in more detail, and examine the performance implications and requirements of these goals. In Section 2.3 we present the

Hyperion stream file system, StreamFS, and describe its design and structure. Section 2.4 describes our use of *signature index* structures to accelerate user queries without burdening the data collection and storage process unduly. Experimental results for StreamFS, our index system, and Hyperion as a whole are presented in Section 2.6. In Section 2.7 we review related work in comparison to Hyperion, and finally we conclude in Section 2.8.

2.2 Requirements and Challenges

We begin by examining the requirements for a network monitoring tool such as Hyperion. In order to fulfill its purpose as a monitor with retrospective query capability, we believe that it needs to meet the following goals:

- Continuously monitor multiple interfaces in multiple locations. Network events such as attacks may take place at different locations in the network, and detecting or tracing them may require correlating information from multiple locations.
- Provide on-line, interactive querying of monitored data. On-line access to prior data is the primary function of Hyperion, and to do this we need a query mechanism which allows only data of interest to be retrieved and examined.
- Store data for a reasonable (but limited) period of time. In order to examine past data it will be necessary to store it until it is needed. However, during continuous operation the total volume of data which may be captured is unbounded, and as new data arrives it will be necessary to replace older data.
- Run on commodity hardware. In order to maximize its utility, we strive to implement Hyperion on readily available systems and hardware.

These goals, in turn, imply several additional requirements. Monitoring in multiple locations requires a distributed system, able to search across data recorded by different monitors in these locations. Interactive querying poses search performance goals, necessitating some form of index structure for search acceleration. Yet, if such searches are to

be done on-line, any index must be updated as fast as data is stored, rather than after the fact. Storing streaming data as it arrives and managing data lifetime results in usage patterns which are far different from typical file and application behavior. Finally, the use of commodity hardware dictates disk-based storage, and imposes some additional limits on processing available at each monitoring node.

In order to meet these requirements, Hyperion addresses the following challenges:

Archive multiple, high-volume streams. As described above, a single heavily loaded gigabit link may easily produce monitor data at a rate of almost 20Mbyte/sec; one system may need to monitor several such links, and thus scale far beyond this rate. Merely storing this data on a disk-based system as it arrives may be a problem; although the peak speed of such a system is sufficient, the worst-case speed is far lower than is required. In order to achieve the needed speeds, it is necessary to tailor our system to the performance behavior of modern disks and disk arrays, as well as the sequential append-only nature of archival writes.

Maintain indexes on archived data. The cost of exhaustive searches through archived data would be prohibitive, so an index is required to support most queries. Over time, updates to this index must be stored at wireline speed, as packets are captured and archived, and thus must support especially efficient updating. This high update rate (e.g. 220K pkts/sec in the example above) rules out many index structures; e.g. a B-tree index over the entire stream would require one or more disk operations per insertion. Unlike storage performance requirements, which must be met to avoid data loss, retrieval performance is not as critical; however, our goal is that it be efficient enough for interactive use. The target for a highly selective query, returning very few data records, is that it be able to search an hour of indexed data in 10 seconds.

Reclaim and re-use storage. Storage space is limited in comparison to arriving data, which is effectively infinite if the system runs long enough. This calls for a mechanism for reclaiming and reusing storage. Data aging policies that delete the oldest or the least-

valuable data to free up space for new data are needed, and data must be removed from the index as it is aged out.

Coordinate between monitors. A typical monitoring system will comprise multiple monitoring nodes, each monitoring one or more network links. In network forensics, for instance, it is sometime necessary to query data archived at multiple nodes to trace events (e.g. a worm) as they move through a network. Such distributed querying requires some form of coordination between monitoring nodes, which involves a trade-off between distribution of data and queries. If too much data or index information is distributed across monitoring nodes, it may limit the overall scale of the system as the number of nodes increase; if queries must be flooded to all monitors, query performance will not scale.

2.2.1 Hyperion Design principles

To address these goals we have applied the following guiding design principles for our system design:

P1: *Support queries, not reads:* A general-purpose file system supports low-level operations such as reads and writes. However, the nature of monitoring applications dictates that data is typically accessed in the form of queries; in the case of Hyperion, for instance, these queries would be predicates identifying values for particular packet header fields such as source and destination address. Consequently, a stream archiving system should support data accesses at the level of queries, as opposed to raw reads on unstructured data. Efficient support for querying implies the need to maintain an index and one that is particularly suited for high update rates.

P2: *Exploit sequential, immutable writes:* Stream archiving results in continuous sequential writes to the underlying storage system; writes are typically immutable since archived data is not modified. The system should employ placement techniques that exploit these I/O characteristics to reduce disk seek overheads and improve system throughput.

P3: *Archive locally, summarize globally.* There is an inherent conflict between the need to scale, which favors local archiving and indexing to avoid network writes, and the need to avoid flooding to answer distributed queries, which favors sharing information across nodes. We “resolve” this conflict by advocating a design where data archiving and indexing is performed locally and a coarse-grain summary of the index is shared between nodes to support distributed querying without flooding.

We apply each of these design principles to both the storage and indexing subsystems, as described below.

2.3 Hyperion Stream File System

The requirements for the Hyperion storage system are: storage of multiple high-speed traffic streams without loss, re-use of storage on a full disk, and support for concurrent read activity without loss of write performance. The main barrier to meeting these requirements is the variability in performance of commodity disk and array storage; although storage systems with best-case throughput sufficient for this task are easily built, worst-case throughput can be three orders of magnitude worse.

In this section we first consider implementing this storage system on top of a general-purpose file system. After exploring the performance of several different conventional file systems on stream writes as generated by our application, we then describe StreamFS, an application-specific file system for stream storage.²

²Specialized file systems for specific application classes (e.g. streaming media) have a poor history of acceptance. However, file systems specific to a single application, often implemented in user space, have in fact been used with success in a number of areas such as web proxies [79] and commercial databases such as Oracle [61].

2.3.1 Stream Storage

In order to consider these issues, we first define a stream storage system in more detail. Unlike a general purpose file system which stores *files*, a stream storage system stores *streams*. These streams are:

- *Recycled*: when the storage system is full, writes of new data succeed, and old data is lost (i.e. removed or overwritten in a circular buffer fashion). This is in contrast to a general-purpose file system, where new data is lost and old data is retained.
- *Immutable*: an application may append data to a stream, but does not modify previously written data.
- *Record-oriented*: attributes such as timestamps are associated with ranges in a stream, rather than the stream itself. Optionally, as in StreamFS, data may be written in records corresponding to these with boundaries which are preserved on retrieval.

This stream abstraction provides the features needed by Hyperion, while lacking other features (e.g. mutability) which are not.

2.3.2 Why Not a General Purpose File system?

We first examine constructing such a stream storage system on top of a general-purpose file system. To do this we need a mapping between streams and files; several such mappings are possible, and we examine two of them below. In this examination we ignore the use of buffering and RAID, which may be used to improve the performance of each of these methods but will not change their relative efficiency.

File-per-stream: A naïve implementation of stream storage uses a single large file for each data stream. When storage is filled, the beginning of the file cannot be deleted if the most recent data (at the end of the file) is to be retained, so the beginning of the file is overwritten with new data in circular buffer fashion. A simplified view of this implementation and the resulting access patterns may be seen in Figure 2.1. Performance of this method

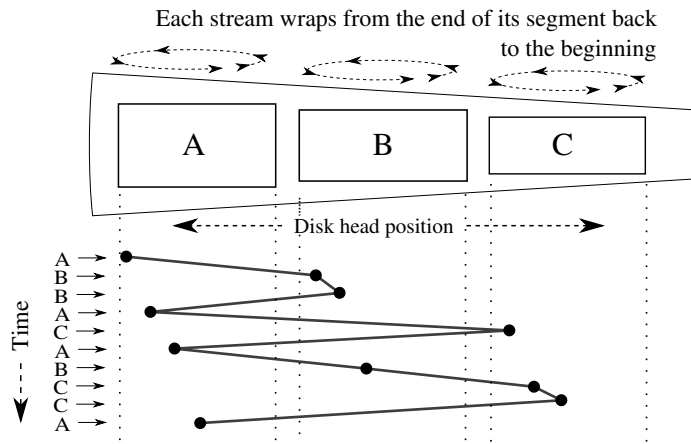


Figure 2.1. Write arrivals and disk accesses for single file per stream. Writes for streams A, B, and C are interleaved, causing most operations to be non-sequential.

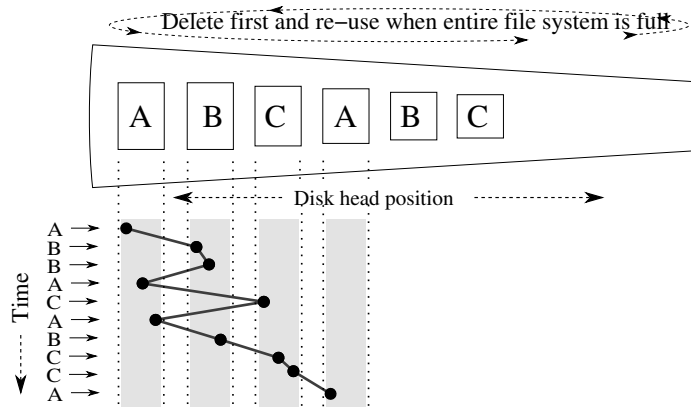


Figure 2.2. Logfile rotation. Data arrives for streams A, B, and C in an interleaved fashion, but is written to disk in a mostly sequential order.

is poor, as with multiple simultaneous streams the disk head must seek back and forth between the write position on each file. In particular, the disk position of any block of data is predetermined, before that data is received, so the disk access pattern is at the mercy of data arrivals. In addition, storage flexibility is poor, as once a file is created it in general cannot be shrunk or increased in size without data loss.

Log files: A better approach to storing streams is known as *logfile rotation*, where a new file is written until it reaches some maximum size, and then closed; the oldest files are then deleted to make room for new ones. Simplified operation may be seen in Figure 2.2,

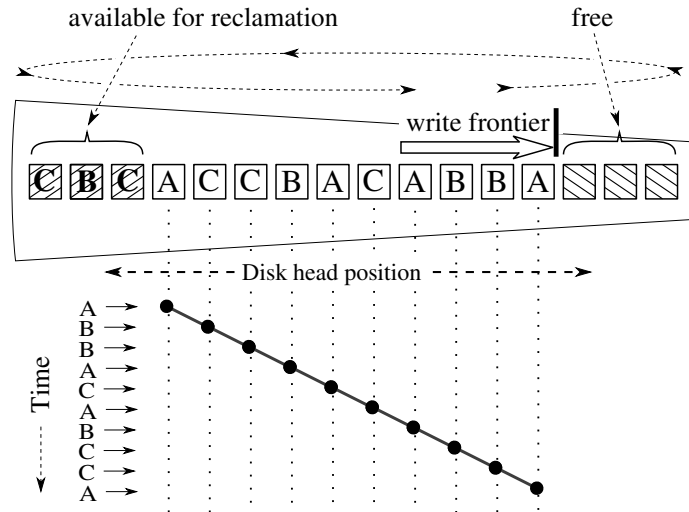


Figure 2.3. Log allocation - StreamFS, LFS. Data arrives in an interleaved fashion and is written to disk in that same order.

where files are allocated as single extents across the disk. This organization is much better at allocating storage flexibly, as allocation decisions may be revised dynamically when choosing which file to delete. In addition, since files are always appended to rather than over-written, the file system need not be bound by previous allocation decisions when placing newly arriving data. Note, however, that file systems which allocate extents of storage in advance of writing them will still suffer poor locality when handling interleaved arrivals on different streams.

Log-Structured File System: The highest write throughput will be obtained if storage is allocated sequentially as data arrives, as illustrated in Figure 2.3. This is the method used by a log-structured file system such as LFS [72], and when logfile rotation is used on such a file system, interleaved writes to multiple streams will be allocated closely together on disk.

Although write allocation in log-structured file systems is straightforward, *cleaning*, or the garbage collecting of storage space after files are deleted, has however remained problematic [76, 90]. Cleaning in a general-purpose LFS must handle files of vastly different sizes and lifetimes, and all existing solutions involve copying data to avoid fragmentation.

The FIFO-like Hyperion write sequence is a very poor fit for such general cleaning algorithms; in Section 2.6 below we present results indicating that it results in significant cleaning overhead.

2.3.3 StreamFS Design

Hyperion StreamFS uses an LFS-like log structure, but avoids this cleaning overhead by eliminating the segment cleaner, and avoiding data copying entirely. This is done by moving the delete or truncate function into the file system; applications write data, but the file system (using application-provided policies) decides when to delete it.

Like LFS, all writes take place at the *write frontier*, which advances as data is written, as shown in Figure 2.3. Since the delete decision is made in the file system, it can be performed as the write frontier advances, so that it can advance to the next segment eligible to be deleted.

A trivial implementation of this strategy would be to over-write *all* data as the write frontier advances, treating the file system as a simple circular buffer and implicitly establishing a single age-based expiration policy for all streams. In a practical system, however, it may be necessary to retain data from some streams for longer periods of time, while records from other streams may be discarded quickly. For this reason, StreamFS provides each stream a storage *guarantee*, which defines a window of records which will not be reclaimed or over-written. As new data is written to the stream, the oldest records fall outside of this guarantee window and become *surplus*, or eligible to be deleted.

As the write frontier advances, the next segment is examined to determine whether it is surplus. If so, it is simply overwritten, as seen in Figure 2.3; if not it is skipped and will expire later. Skipping segments which are not surplus will result in disk seeks, this policy is not as efficient as blind overwrite. However, in practice the overhead due to these seeks is minor, as has been shown in simulation [23]. Intuitively, the reason this is the case is that those segments most likely to be skipped will belong to lower-rate streams, and thus will

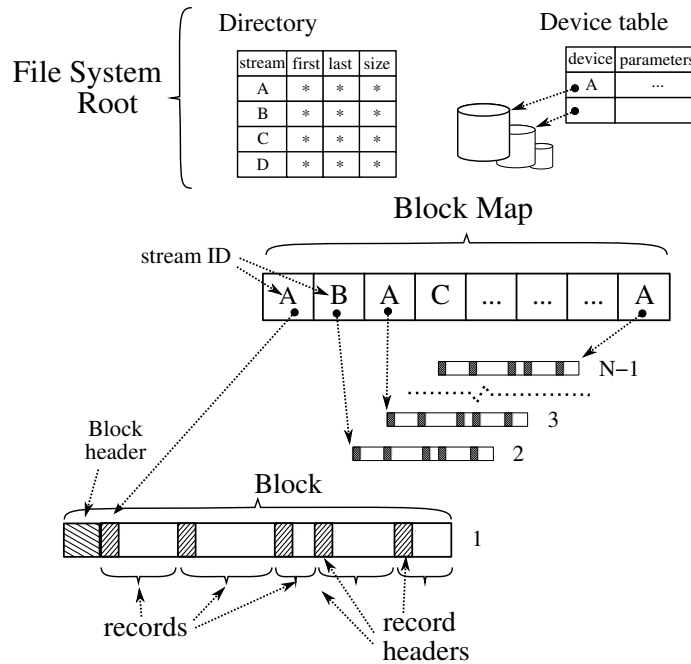


Figure 2.4. StreamFS metadata structures. A record header is prepended to record written by the application; block headers are written for each fixed-length block, block map is written for every N (256) blocks, and one file system root.

be infrequent; segments typically belong to high-rate streams, and will have expired by the time the write frontier wraps around to them.

2.3.4 StreamFS Organization

The data structures which implement this stream file system are illustrated in Figure 2.4. The structures and their fields and functions are as follows:

- *record*: Each variable-length record written by the application corresponds to an on-disk record and *record header*.
- *segment*: Multiple records from the same stream are combined in a single fixed-length segment, by default 1Mbyte in length. The *segment header* identifies the stream to which the segment belongs, and record boundaries within the segment.

- *segment map*: Every N^{th} segment (default 256) is used as a *segment map*, indicating the associated stream and an in-stream sequence number for each of the preceding $N - 1$ segments.
- *file system root*: The root holds the stream directory, metadata for each stream (head and tail pointers, size, parameters), and a description of the devices making up the file system.

Each record corresponds to a stream record as described above in Section 2.3.1; it is a single write (or group of related writes) with associated attributes such as timestamp and length stored in the record header. The segments correspond to the segments in the description of the write frontier and storage allocation above. Finally, the segment map is used in making allocation decisions as the write frontier advances. Rather than caching information about all segments in memory, or incurring the seek overhead of checking each segment, we occasionally read a segment map, and are able to use the map information to make allocation decisions for the next N segments.

2.3.5 Striping and Speed Balancing

StreamFS supports a number of other features to enhance streaming performance; two of these are multiple device support (striping) and speed balancing. Multiple devices are handled directly; data is distributed across the devices in units of a single block, much as data is striped across a RAID-0 volume, and the benefits of single disk write-optimizations in StreamFS extend to multi-disk systems as well. Since successive blocks (e.g., block i and $i + 1$) map onto successive disks in a striped system, StreamFS can extract the benefits of I/O parallelism and increase overall system throughput. Further, in a d disk system, blocks i and $i + d$ will map to the same disk drive due to wrap-around. Consequently, under heavy load when there are more than d outstanding write requests, writes to the same disk will be written out sequentially, yielding similar benefits of sequential writes as in a single-disk system.

Speed balancing, in turn, addresses performance variations across a single device. Modern disk drives are *zoned* in order to maintain constant linear bit density; this results in disk throughput which can differ by a factor of 2 between the innermost and the outermost zones. If StreamFS were to write out data blocks sequentially from the outer to inner zones, then the system throughput would drop by a factor of two when the write frontier reached the inner zones. This worst-case throughput, rather than the mean throughput, would then determine the maximum loss-less data capture rate of the monitoring system.

StreamFS employs a balancing mechanism to ensure that system throughput remains roughly constant over time, despite variations across the disk platter. This is done by appropriately spreading the write traffic across the disk and results in an increase of approximately 30% in worst-case throughput. The disk is divided into three³ zones R , S and T , and each zone into large, fixed-sized regions (R_1, \dots, R_n) , (S_1, \dots, S_n) , (T_1, \dots, T_n) . These regions are then used in the following order: $(R_1, S_1, T_n, R_2, S_2, T_{n-1}, \dots, R_n, S_n, T_1)$; data is written to blocks within each region sequentially. The effective throughput is thus the average of throughput at 3 different points on the disk, and close to constant.

When accessing the disk sequentially, a zone-to-zone seek will be required after each region; the region size must thus be chosen to balance seek overhead with buffering requirements. For disks used in our experiments, a region size of 64MB results in one additional seek per second (degrading disk performance by less than 1%) at a buffering requirement of about 16MB per device.

2.3.6 Read Addressing and Implementation

Hyperion uses StreamFS to store packet data and indexes to that data, and then handles queries by searching those indexes and retrieving matching data. This necessitates a mechanism to identify a location in a data stream by some sort of offset or handle which

³The original choice of 3 regions was selected experimentally, but later work [23] demonstrates that this organization results in throughput variations of less than 4% across inner-to-outer track ratios up to 4:1.

may be stored in an index (e.g. across system restart), and then later used to retrieve the corresponding data. This value could be a byte offset from the start of the stream, with appropriate provisions (such as a 64-bit length) to guard against wrap-around. However, the pattern of reads generated by the index is highly non-sequential, and thus translating an offset into a disk location may require multiple accesses to on-disk tables. We therefore use a mechanism similar to a SYN cookie [10], where the information needed to retrieve a record (i.e. disk location and approximate length) is safely encoded and given to the application as a handle, providing both a persistent handle and a highly optimized random read mechanism.

Using application-provided information to directly access the disk raises issues of robustness and security. Although we may ignore security concerns in a single-application system, we still wish to ensure that in any case where a corrupted handle is passed to StreamFS, an error is flagged and no data corruption occurs. This is done by using a *self-certifying* record header, which guarantees that a handle is valid and that access is permitted. This header contains the ID of the stream to which it belongs and the permissions of that stream, the record length, and a hash of the header fields (and a *file system secret* if security is of concern) allowing invalid or forged handles to be detected. To retrieve a record by its persistent handle, StreamFS decodes the handle, verifies that the resulting address lies within the volume, reads from the indicated address and length, and then verifies the record header hash. At this point a valid record has been found; permission fields may then be checked and the record returned to the application if appropriate.

2.4 Indexing Archived Data

It is not enough to merely be able to store and retrieve data quickly; an Hyperion monitor needs to maintain an index in order to support efficient retrospective queries. Because these queries are received while the system is online and recording data, the index must be created and stored at high speed, as data comes in. Disk performance significantly limits

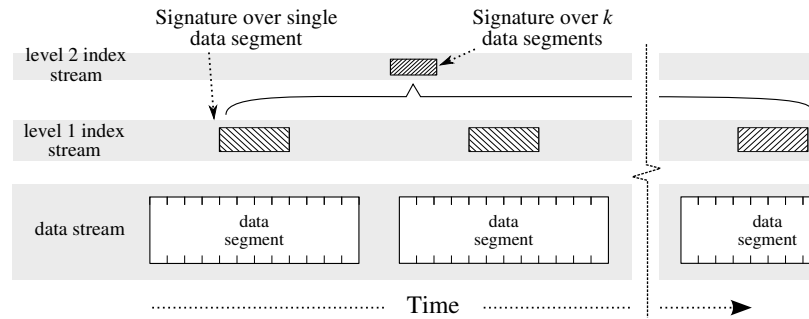


Figure 2.5. Hyperion multi-level signature index, showing two levels of signature index plus the associated data records.

the options available for the index; although minimizing random disk operations is a goal in any database, here multiple fields must be indexed in records arriving at a rate of over 100,000 per second per link. To scale to these rates, Hyperion relies on index structures that can be computed online and then *stored immutably*.

2.4.1 Signature Index

Hyperion partitions a stream into intervals and computes one or more *signatures* [28] for each interval. The signatures can be tested for the presence of a record with a certain key in the associated data interval. Unlike a traditional B-tree-like structure, a signature only indicates whether a record matching a certain key is present; it does not indicate where in the interval that record is present. Thus, the entire interval needs to be retrieved and scanned for the result. However, if the key is not present, the entire interval can be skipped.

Signature indexes are computed on a per-interval basis; no stream-wide index is maintained. This organization provides an index which may be streamed to disk along with the data—once all data within an interval have been examined (and streamed to storage), the signature itself can also be streamed out and a new signature computation begun for the next interval. This also solves the problem of removing keys from the index as they age out, as the signature associated with a data interval ages out as well.

Hyperion uses a multi-level *signature index*, the organization of which is shown in detail in Figure 2.5. A signature index, the most well-known of which is the Bloom Filter [11], creates a compact signature for one or more records, which may be tested to determine whether a particular key is present in the associated records. To search for records containing a particular key, we first retrieve and test only the signatures; if any signature matches, then the corresponding records are retrieved and searched.

Signature functions are typically inexact, with some probability of a *false positive*, where the signature test indicates a match when there is none. This will be corrected when scanning the actual data records; the signature function cannot generate *false negatives*, however, as this will result in records being missed. Search efficiency for these structures is a trade-off between signature compactness, which reduces the amount of data retrieved when scanning the index, and false positive rate, which results in unnecessary data records being retrieved and then discarded.

The Hyperion index uses Bloom’s hash function, where each key is hashed into a b -bit word, of which k bits are set to 1. The hash words for all keys in a set are logically OR-ed together, and the result is written as the signature for that set of records. To check for the presence of a particular key, the hash for that key h_0 is calculated and compared with the signature for the record, h_s ; if any bit is set in h_0 but not set in h_s , then the value cannot be present in the corresponding data record. To calculate the false positive probability, we note that if the fraction of 1 bits in the signature for a set of records is r and the number of 1 bits in any individual hash is k , then the chance that a match could occur by chance is r^k ; e.g. if the fraction of 1 bits is $\frac{1}{2}$, then the probability is 2^{-k} .

2.4.2 Multi-Level and Bit-Sliced Signature Indexes

Hyperion employs a two-level index [74], where a level-1 signature is computed for each data interval, and then a level-2 signature is computed over k data intervals. A search scans the level-2 signatures, and when a match is detected the corresponding k level-1

signatures are retrieved and tested; data blocks are retrieved and scanned only when a match is found in a level-1 signature.

When no match is found in the level-2 signature, k data segments may be skipped; this allows efficient search over large volumes of data. The level-2 signature will suffer from a higher false positive rate, as it is k times more concise than the level-1 signature; however, when a false positive occurs it is almost always detected after the retrieval of the level-1 signatures. In effect, the multi-level structure allows the compactness of the level-2 signature, with the accuracy of the level-1 signature.

The description thus far assumes that signatures are streamed to disk as they are produced. When reading the index, however, a signature for an entire interval—thousands of bytes—must be retrieved from disk in order to examine perhaps a few dozen bits.

By buffering the top-level index and writing it in *bit-sliced* [27] fashion we are to retrieve only those bits which need to be tested, thus possibly reducing the amount of data retrieved by orders of magnitude. This is done by aggregating N signatures, and then writing them out in N -bit *slices*, where the i^{th} slice is constructed by concatenating bit i from each of the N signatures. If N is large enough, then a slice containing N bits, bit i from each of N signatures, may be retrieved in a single disk operation. (although not implemented at present, this is a planned extension to our system.)

2.4.3 Distributed Index and Query

Our discussion thus far has focused on data archiving and indexing locally on each node. A typical network monitoring system will comprise multiple nodes and it is necessary to handle distributed queries without resorting to query flooding. Hyperion maintains a distributed index that provides an integrated view of data at all nodes, while storing the data itself and most index information locally on the node where it was generated. Local storage is emphasized for performance reasons, since local storage bandwidth is more

economical than communication bandwidth; storage of archived data which may never be accessed is thus most efficiently done locally.

To create this distributed index, a coarse-grain summary of the data archived at each node is needed. The top level of the Hyperion multi-level index provides such a summary, and is shared by each node with the rest of the system. Since broadcasting the index to all other nodes would result in excessive traffic as the system scales, an *index node* is designated for each time interval $[t_1, t_2)$. All nodes send their top-level indexes to the index node during this time-interval. Designating a different index node for successive time intervals results in a temporally-distributed index. Cross-node queries are first sent to an index node, which uses the coarse-grain index to determine the nodes containing matching data; the query is then forwarded to this subset for further processing.

2.5 Implementation

We have implemented a prototype of the Hyperion network monitoring system on Linux, running on commodity Intel-architecture servers; it currently comprises 7000 lines of code.

The StreamFS implementation takes advantage of Linux asynchronous I/O and raw device access, and is implemented as a user-space library. In an additional simplification, the file system root resides in a file on the conventional file system, rather than on the device itself. These implementation choices impose several constraints: for instance, all access to a StreamFS volume must occur from the same process, and that process must run as root in order to access the storage hardware. These limitations have not been an issue for Hyperion to date; however, a kernel re-implementation of StreamFS would address them if they become problematic in the future.

The index is a two-level signature index with linear scan of the top level (not bit-sliced) as described in Section 2.4. Multiple keys may be selected to be indexed on, where each key may be a single field or a composite key consisting of multiple fields. Signatures for

each key are then superimposed in the same index stream via logical OR. Query planning is not yet implemented, and the query API requires that each key to be used in performing the query be explicitly identified.

Packet input is supported from trace files and via a special-purpose gigabit ethernet driver, `sk98_fast`, developed for `nProbe` at the University of Cambridge [59].

The Hyperion system is implemented as a set of modules which may be controlled from a scripting language (Python) through an interface implemented via the SWIG wrapper toolkit. This design allows the structure of the monitoring application to be changed flexibly, even at run time—as an example, a query is processed by instantiating data source and index search objects and connecting them. Communication between Hyperion systems is by RPC, which allows remote query execution or index distribution to be handled and controlled by the same mechanisms as configuration within a single system.

2.6 Experimental Results

In this section we present operational measurements of the Hyperion network monitor system. Tests of the stream file system component, `StreamFS`, measure its performance and compare it to that of solutions based on general-purpose file systems. Micro-benchmarks as well as off-line tests on real data are used to test the multi-level indexing system; the micro-benchmarks measure the scalability of the algorithm, while the trace-based tests characterize the search performance of our index on real data. Finally, system experiments characterize the performance of single Hyperion nodes, as well as demonstrating operation of a multi-node configuration.

2.6.1 Experimental Setup

Tests reported below (unless otherwise noted) were performed on the system described in Table 2.1, a dual-processor Xeon server with fast SCSI disks. File system tests wrote dummy data (i.e. zeros), and ignored data from read operations. Most index tests, however,

Hardware	2 × 2.4GHz P4 Xeon, 1 GB memory	
Storage	4 × Fujitsu MAP3367NP Ultra320 SCSI, 10K RPM	
OS	Linux 2.6.9 (CentOS 4.4)	
Network	SysKonnnect SK-9821 1000mbps	

Table 2.1. StreamFS Test System

data set	158 seconds	14.5M records
table load	252 s (± 7 s)	1.56 × real-time
query	50.7 s (± 0.4 s)	0.32 × real-time

Table 2.2. Postgres Insertion Performance

used actual trace data from the link between the University of Massachusetts and the commercial Internet [88]. These trace files were replayed on another system by combining the recorded headers (possibly after modification) with dummy data, and transmitting the resulting packets directly to the system under test.

2.6.2 Database and General-Purpose File System Evaluation

Our first tests establish a baseline for evaluating the performance of the Hyperion system. Since Hyperion is an application-specific database, built on top of an application-specific file system, we compare its performance with that of existing general-purpose versions of these components. In particular, we measure the speed of storing network traces on both a conventional relational database and on several conventional file systems.

Database Performance: We briefly present results of bulk loading packet header data on Postgres 7.4.13. Approximately 14.5M trace data records representing 158 seconds of sampled traffic were loaded using the COPY command; after loading, a query retrieved a unique row in the table. To speed loading, no index was created, and no attempt was made to test simultaneous insert and query performance. Mean results with 95% confidence intervals (8 repetitions) are shown in Table 2.2.

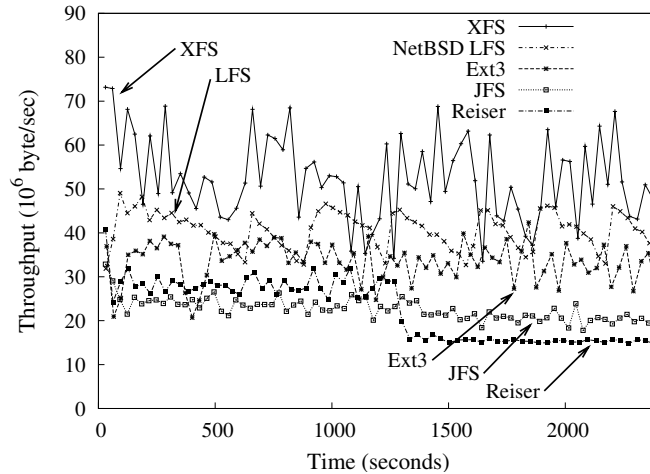


Figure 2.6. Streaming write-only throughput by file system. Each trace shows throughput for 30s intervals over the test run.

Postgres was not able to load the data, collected on a moderately loaded (40%) link, in real time. Query performance was much too slow for on-line use; although indexing would improve this, it would further degrade load performance.

Baseline File system measurements: These tests measure the performance of general-purpose file systems to serve as the basis for an application-specific stream database for Hyperion. In particular, we measure write-only performance with multiple streams, as well as the ability to deliver write performance guarantees in the presence of mixed read and write traffic. The file systems tested on Linux are ext3, ReiserFS, SGI’s XFS [85], and IBM’s JFS; in addition LFS was tested on NetBSD 3.1.

Preliminary tests using the naïve single file per stream strategy from Section 2.3.2 are omitted, as performance for all file systems was poor. Further tests used an implementation of the log file strategy from Section 2.3.2, with file size capped at 64MB. Tests were performed with 32 parallel streams of differing speeds, with random write arrivals of mean size 64KB. All results shown are for the steady state, after the disk has filled and data is being deleted to make room for new writes.

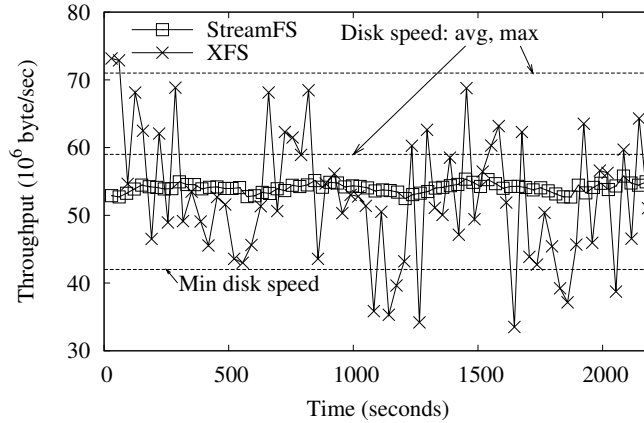


Figure 2.7. XFS vs. StreamFS write only throughput, showing 30 second and mean values. Straight lines indicate disk throughput at outer tracks (max) and inner tracks (min) for comparison. Note the substantial difference in worst-case throughput between the two file systems.

The clear leader in performance is XFS, as may be seen in Figure 2.6. It appears that XFS maintains high write performance for a large number of streams by buffering writes and writing large extents to each file – contiguous extents as large as 100MB were observed, and (as expected) the buffer cache expanded to use almost all of memory during the tests. (Sweeney *et al.* [85] describe how XFS defers block assignment until pages are flushed, allowing such large extents to be generated.)

LFS performance was best of the remaining file systems. We hypothesize that a key factor in its somewhat lower performance was the significant overhead of the segment cleaner. Although we were not able to directly measure I/O rates due to cleaning, the system CPU usage of the cleaner process was significant: approximately 25% of that used by the test program.

2.6.3 StreamFS Evaluation

In light of the above results, we evaluate the Hyperion file system StreamFS by comparing it to XFS.

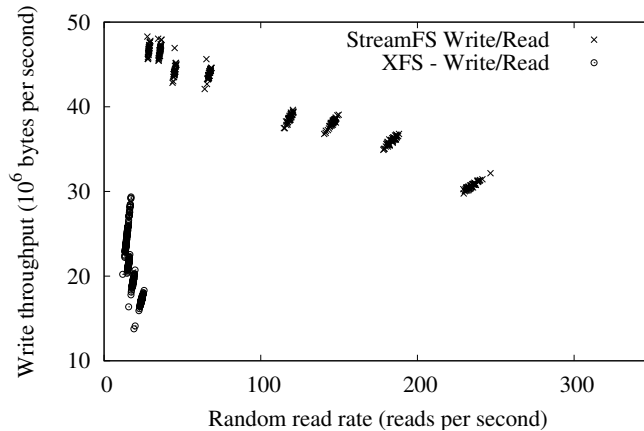


Figure 2.8. Scatter plot of StreamFS and XFS write and read performance.

StreamFS Write Performance: In Figure 2.7 we see representative traces for 32-stream write-only traffic for StreamFS and XFS. Although mean throughput for both file systems closely approaches the disk limit, XFS shows high variability even when averaged across 30 second intervals. Much of the XFS performance variability remains within the range of the disk minimum and maximum throughput, and is likely due to allocation of large extents at random positions across the disk. A number of 30s intervals, however, as well as two 60s intervals, fall considerably below the minimum disk throughput; we have not yet determined a cause for these drop-offs in performance. The consistent performance of StreamFS, in turn, gives it a worst-case speed close to the mean — almost 50% higher than the worst-case speed for XFS.

Read/Write Performance: Useful measurements of combined read/write performance require a model of read access patterns generated by the Hyperion monitor. In operation, on-line queries read the top-level index, and then, based on that index, read non-contiguous segments of the corresponding second-level index and data stream. This results in a read access pattern which is highly non-contiguous, although most seeks are relatively small. We model this non-contiguous access stream as random read requests of 4KB blocks in our measurements, with a fixed ratio of read to write requests in each experiment.

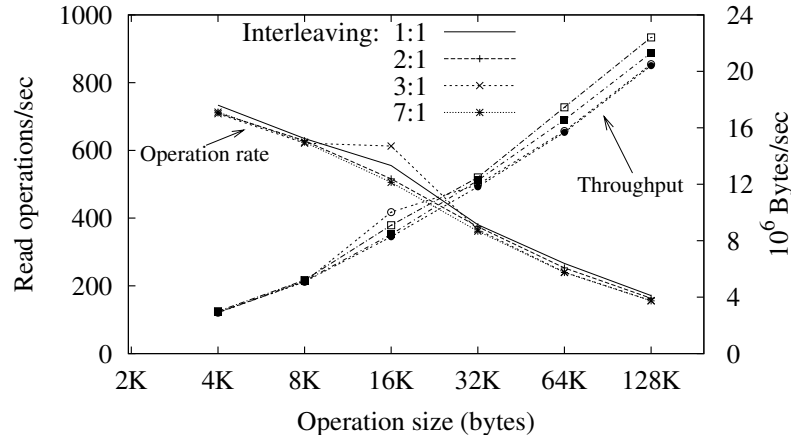


Figure 2.9. Streaming file system read performance. Throughput (rising) and operations/sec (falling) values are shown. The interleave factor refers to the number of streams interleaved on disk - i.e. for the 1:1 case the stream being fetched is the only one on disk; in the 1:7 case it is one of 7.

Figure 2.8 shows a scatter plot of XFS and StreamFS performance for varying read/write ratios. XFS read performance is poor, and write performance degrades precipitously when read traffic is added. This may be a side effect of organizing data in logfiles, as due to the large number of individual files, many read requests require opening a new file handle. It appears that these operations result in flushing some amount of pending work to disk; as evidence, the mean write extent length when reads are mixed with writes is a factor of 10 smaller than for the write-only case.

StreamFS Read Performance: We note that our prototype of StreamFS is not optimized for sequential read access; in particular, it does not include a read-ahead mechanism, causing some sequential operations to incur the latency of a full disk rotation. This may mask smaller-scale effects, which could come to dominate if the most significant overheads were to be removed.

With this caveat, we test single-stream read operation, to determine the effect of record size and stream interleaving on read performance. Each test writes one or more streams to an empty file system, so that the streams are interleaved on disk. We then retrieve the records of one of these streams in sequential order. Results may be seen in Figure 2.9,

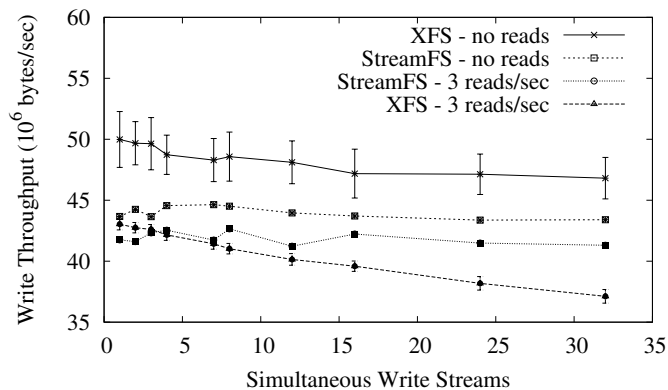


Figure 2.10. Sensitivity of performance to number of streams

for record sizes ranging from 4KB to 128KB. Performance is dominated by per-record overhead, which we hypothesize is due to the lack of read-ahead mentioned above, and interleaved traffic has little effect on performance.

Additional experiments: These additional tests measured changes in performance with variations in the number of streams and of devices. Figure 2.10 shows the performance of StreamFS and XFS as the number of simultaneous streams varies from 1 to 32, for write-only and mixed read/write operations. XFS performance is seen to degrade slightly as the number of streams increases, and more so in the presence of read requests, while StreamFS throughput is relatively flat.

Multi-device tests with StreamFS on multiple devices and XFS over software RAID show almost identical speedup for both. Performance approximately doubles when going from 1 to 2 devices, and increases by a lesser amount with 3 devices as we get closer to the capacity of the 64-bit PCI bus on the test platform.

2.6.4 Index Evaluation

The Hyperion index must satisfy two competing criteria: it must be fast to calculate and store, yet provide efficient query operation. We test both of these criteria, using both generated and trace data.

variable	coeff.	std. error	<i>t</i> -stat
index (N)	132 ns	6.53 ns	20.2
bytes hashed (B_i)	9.4 ns	1.98 ns	4.72
bit generated (k)	43.5 ns	2.1 ns	21.1

Table 2.3. Index Calculation Performance Parameters

Signature Index Computation: The speed of this computation was measured by repeatedly indexing sampled packet headers in a large (\gg cache size) buffer. on a single CPU. Since the size of the computation input — i.e. the number of headers indexed — is variable, linear regression was used to determine the relationship between computation parameters and performance.

In more detail, for each packet header we create N indexes, where each index i is created on F_i fields (e.g. source address) totaling B_i bytes. For index i , the B_i bytes are hashed into an M -bit value with k bits set, as described in Section 2.4.1. Regression results for the significant parameters are shown in Table 2.3.

As an example, if 7 indexes are computed per header, with a total of 40 bytes hashed and 60 signature bits generated, then index computation would take $7 \cdot 132 + 40 \cdot 9.4 + 60 \cdot 43.5 = 3910\text{ns}$ or $3.9 \mu\text{s}/\text{packet}$, for a peak processing rate of 256,000 headers per second on the test CPU, a 2.4GHz Xeon. Although not sufficient to index minimum-sized packets on a loaded gigabit link, this is certainly fast enough for the traffic we have measured to date. (e.g. 106,000 packets/sec on a link carrying approximately 400Mbit/sec.)

Signature Density: This next set of results examines the performance of the Hyperion index after it has been written to disk, during queries. In Figure 2.11 we measure the signature *density*, or the fraction of bits set to 1, when summarizing addresses from trace data. On the X axis we see the number of addresses summarized in a single hash block, while the different traces indicate the precision with which each address is summarized. From Bloom [11] we know that the efficiency of this index is maximized when the fraction of 1 (or 0) bits is 0.5; this line is shown for reference.

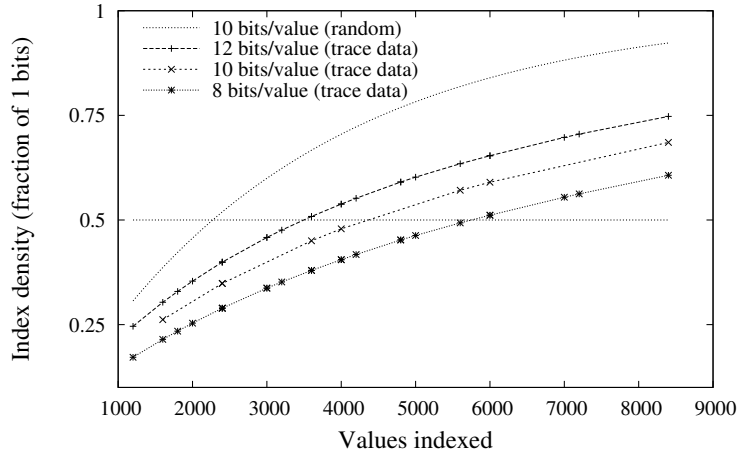


Figure 2.11. Signature density when indexing source/destination addresses, vs. number of addresses hashed into a single 4KB index block. Curves are for varying numbers of bits (k) per hashed value; top curve is for uniform random data, while others use sampled trace data.

From the graph we see that a 4KB signature can efficiently summarize between 3500 and 6000 addresses, depending on the parameter k and thus the false positive probability. The top line in the graph shows signature density when hashing uniformly-distributed random addresses with $k = 10$; it reaches 50% density after hashing only half as many addresses as the $k = 10$ line for real addresses. This is to be expected, due to repeated addresses in the real traces, and translates into higher index performance when operating on real, correlated data.

Query overhead: Since the index and data used by a query must be read from disk, we measure the overhead of a query by the factors which affect the speed of this operation: the volume of data retrieved and the number of disk seeks incurred. A 2-level index with 4K byte index blocks was tested, with data block size varying from 32KB to 96KB according to test parameters. The test indexed traces of 1 hour of traffic, comprising 26GB, $3.8 \cdot 10^8$ packets, and $2.5 \cdot 10^6$ unique addresses. To measure overhead of the index itself, rather than retrieval of result data, queries used were highly selective, returning only 1 or 2 packets.

Figures 2.12 and 2.13 show query overhead for the simple and bit-sliced indexes, respectively. On the right of each graph, the volume of data retrieved is dominated by sub-

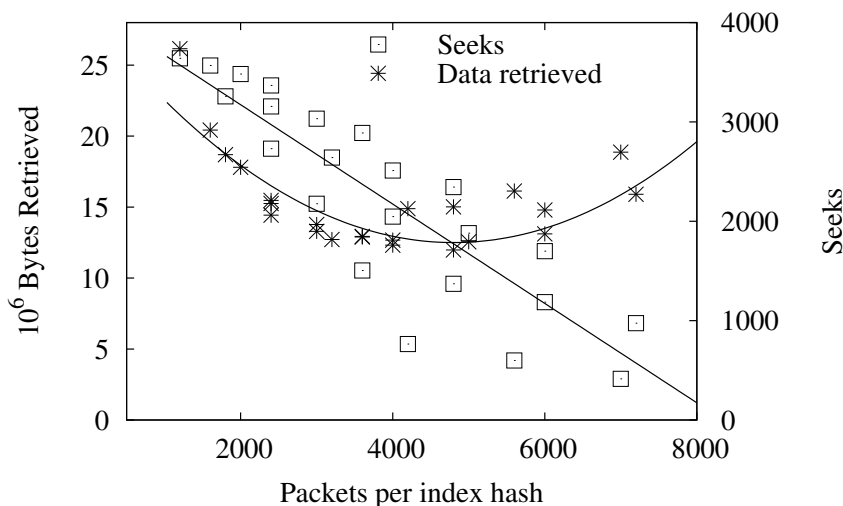


Figure 2.12. Single query overhead for summary index, with fitted lines. N packets (X axis) are summarized in a 4KB index, and then at more detail in several (3-6) 4KB sub-indexes. Total read volume (index, sub-index, and data) and number of disk seeks are shown.

index and data block retrieval due to false hits in the main index. To the left (visible only in Figure 2.12) is a domain where data retrieval is dominated by the main index itself. In each case, seek overhead decreases almost linearly, as it is dominated by skipping from block to block in the main index; the number of these blocks decreases as the packets per block increase. In each case there is a region which allows the 26GB of data to be scanned at the cost of 10-15 MB of data retrieved, and 1000-2000 disk seeks.

2.6.5 Prototype Evaluation

After presenting test results for the components of the Hyperion network monitor, we now turn to tests of its performance as a whole. Our implementation uses StreamFS as described above, and a 2-level index without bit-slicing. The following tests for performance, functionality, and scalability are presented below:

- performance tests: tests on single monitoring node which assess the system's ability to gather and index data at network speed, while simultaneously processing user queries.

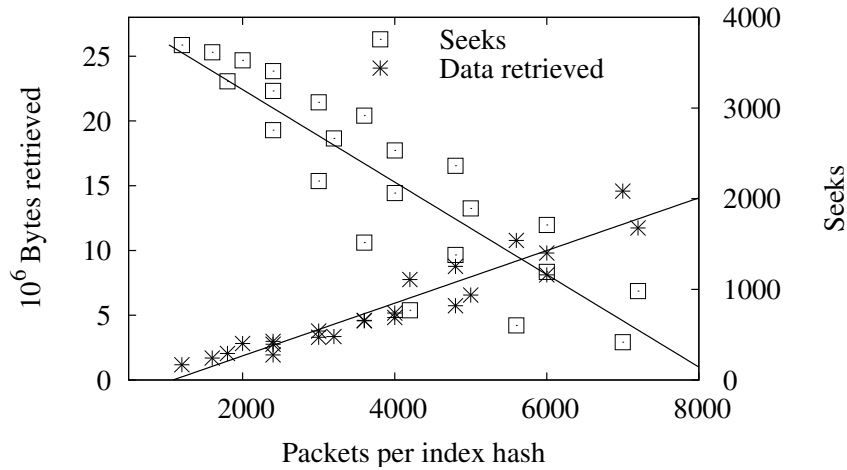


Figure 2.13. Single query overhead for bit-sliced index. Identical to Figure 2.12, except that each index was split into 1024 32-bit slices, with slices from 1024 indexes stored consecutively by slice number.

- functionality testing: three monitoring nodes are used to trace the origin of simulated malicious traffic within real network data.
- scalability testing: a system of twenty monitoring nodes is used to gather and index trace data, to measure the overhead of the index update protocol.

Monitoring and Query Performance: These tests were performed on the primary test system, but with a single data disk. Traffic from our gateway link traces was replayed over a gigabit cable to the test system. First the database was loaded by monitoring an hour's worth of sampled data — $4 \cdot 10^8$ packets, or 26GB of packet header data. After this, packets were transmitted to the system under test with inter-arrival times from the original trace, but scaled so as to vary the mean arrival rate, with simultaneous queries. We compute packet loss by comparing the transmit count on the test system with the receive count on Hyperion, and measure CPU usage.

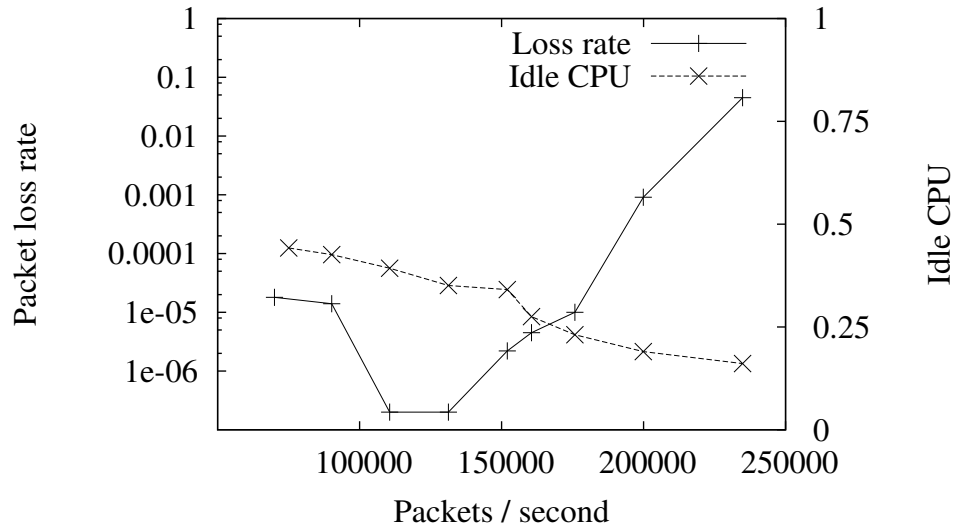


Figure 2.14. Packet arrival and loss rates

	leaf	cluster head	net head
transmit	102 KB/s	102 KB/s	
receive		408 KB/s	510 KB/s

Table 2.4. Hyperion communication overhead overhead in K bytes/sec

Figure 2.14 shows packet loss and free CPU time remaining as the packet arrival rate is varied.⁴ Although loss rate is shown on a logarithmic scale, the lowest points represent zero packets lost out of 30 or 40 million received. The results show that Hyperion was able to receive and index over 200,000 packets per second with negligible packet loss. In addition, the primary resource limitation appears to be CPU power, indicating that it may be possible to achieve significantly higher performance as CPU speeds scale.

System Scalability: In this test a cluster of 20 monitors recorded trace information from files, rather than from the network itself. Tcpdump was used to monitor RPC traffic between the Hyperion processes on the nodes, and packet and byte counts were measured. Each of the 20 systems monitored a simulated link with traffic of approximately 110K

⁴Idle time is reported as the fraction of 2 CPUs which is available. Packet capture currently uses 100% of one CPU; future work should reduce this.

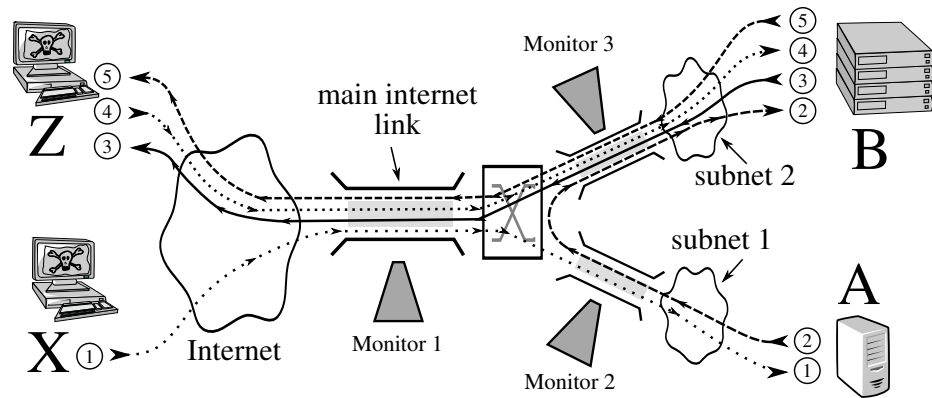


Figure 2.15. Forensic query example. (1) attacker X compromises system A, providing an inside platform to (2) attack system B, installing a bot. The bot (3) contacts system Z via IRC, (4) later receives a command, and begins (5) relaying spam traffic.

pkts/sec, with a total bit rate per link of over 400 Mbit/sec. Level 2 indexes were streamed to a *cluster head*, a position which rotates over time to share the load evenly. A third level of index was used in this test; each cluster head would store the indexes received, and then aggregate them with its own level 2 index and forward the resulting stream to the current *network head*. Results are shown in Table 2.4. From these results we see that scaling to dozens of nodes would involve maximum traffic volumes between Hyperion nodes in the range of 4Mbit/s; this would not be excessive in many environments, such as within a campus.

Forensic Query Case Study: This experiment examines a simulated 2-stage network attack, based on real-world examples. Packet traces were generated for the attack, and then combined with sampled trace data to create traffic traces for the 3 monitors in this simulation, located at the campus gateway, the path to target A, and the path to B respectively.

Abbreviated versions of the queries for this search are shown in Figure 2.16. We note that additional steps beyond the Hyperion queries themselves are needed to trace the attack; for instance, in step 3 the search results are examined for patterns of known exploits, and the results from steps 5 and 6 must be joined in order to locate X. Performance of this search (in particular, steps 1, 4, and 7) depends on the duration of data to be searched,

1	SELECT p WHERE src=B,dport=SMTP,t ≤ T _{now}	Search outbound spam traffic from B, locating start time T ₀ .
2	SELECT p WHERE dst=B,t ∈ T ₀ ··· T ₀ + Δ	Search traffic into B during single spam transmission to find control connection.
3	SELECT p WHERE dst=B,t ∈ T ₀ - Δ ··· T ₀	Find inbound traffic to B in the period before T ₀ .
4	SELECT p WHERE s/d/p=Z/B/P _x , syn, t ≤ T ₀	Search for SYN packet on this connection at time T ₋₁ .
5	SELECT p WHERE dst=B,t ∈ T ₋₁ - Δ ··· T ₋₁	Search for the attack which infected B, finding connection from A at T ₂ .
6	SELECT p WHERE dst=A,t ∈ T ₋₂ - Δ ··· T ₋₂ + Δ	Find external traffic to A during the A-B connection to locate attacker X.
7	SELECT p WHERE src=X,syn,t ≤ T ₋₂	Find all incoming connections from X

Figure 2.16. Outline of Queries for Forensic Case Study

System	Query support	Storage	Local Index	Dist. Index	flow only
PIER	yes	yes	yes	yes	yes
MIND	yes	yes	yes	yes	yes
GigaScope	yes	no	no	no	no
CoMo	yes	yes	no	no	no
Hyperion	yes	yes	yes	yes	no

Table 2.5. Network Monitoring and Query Systems

which depends in turn on how quickly the attack is discovered. In our test, Hyperion was able to use its index to handle queries over several hours of trace data in seconds. In actual usage it may be necessary to search several days or more of trace data; in this case the long-running queries would require minutes to complete, but would still be effective as a real-time forensic tool.

2.7 Related Work

Although the structure of captured network trace information—a header for each packet consisting of a set of fields—naturally lends itself to a database view, the rate at which data is generated (over 100,000 packets/sec for a single gigabit link) precludes the use of conventional database systems.

Table 2.5 displays the approaches used by a range of existing systems to meet these performance demands. PIER [39] and MIND [51] are peer-to-peer systems, and are only able to archive and index data at a limited rate; they therefore are restricted to processing flow-level instead of packet-level information. A number of systems such as the Endace DAG [26] have been developed for wire-speed collection and storage of packet monitoring data, but these systems are designed for off-line analysis of data, and provide no mechanisms for indexing or even querying the data. CoMo [40] addresses high-speed monitoring and storage, with provisions for both streaming and retrospective queries; however, it lacks any mechanism for indexing.

Several commercial network forensics systems (e.g. Sandstorm NetIntercept [75] and Niksun NetDetector [62]) provide archiving of network traffic, with indexing and on-line search. However, these products use conventional databases and file systems and are unable to support simultaneous capture and query at the speeds targeted by Hyperion [31].

GigaScope [20] is a streaming system, where queries are evaluated as data is received, as in general-purpose stream databases [60, 83, 1]. These queries, however, may only be applied over incoming data streams; there is no mechanism in such a system for *retrospective queries*, or queries over past data. Streambase [84] is another general-purpose stream database, with the addition of persistent tables; however these tables are conventional hash or B-tree indexed tables.

The structure of Hyperion StreamFS bears a debt to the original Berkeley log-structured file system [73], and especially to the V System WORM file system [16]. In addition there has been significant work in the past on streaming file systems for multimedia (e.g. [86]); however, these systems have typically been designed for read-dominated workloads, rather than archival storage.

2.8 Summary

In this chapter we have presented Hyperion, a system for distributed monitoring and archiving of network traffic with online query capability. The core components of this system are a local archival storage subsystem, StreamFS, and a multi-level local and distributed index. We have implemented Hyperion on commodity Linux servers, and have used our prototype to conduct a detailed experimental evaluation using both synthetic data and real network traces. We present experiments showing StreamFS achieving a worst-case streaming write performance of 80% of the mean disk speed, or almost 50% higher than for the best general-purpose Linux file system. In addition, StreamFS is shown to be able to handle a workload equivalent to streaming a million packet headers per second to disk while responding to simultaneous read requests. Our multi-level index, in turn, scales to data rates of over 200K packets/sec while at the same time providing interactive query responses, searching an hour of trace data in seconds. Finally, we examine the overhead of scaling a Hyperion system to tens of monitors, and demonstrate the benefits of our distributed storage system using a real-world example.

CHAPTER 3

WIRELESS SENSOR NETWORK STORAGE

Wireless sensor networks, comprised of small battery-powered *sensor* devices such as the Mica2[21] and Telos [68] “motes”, can scale to hundreds or even thousands of data-gathering systems. Depending on the sensing modality, these nodes may collect and store large amounts of raw data, especially for high data volume sensing such as image capture.

Due to resource constraints it is not possible to retrieve more than a small fraction of this data for the user, thus posing the problem of locating and retrieving only the most valuable portions. In this chapter we describe a data management system specialized for storage and retrieval in these networks, and examine the performance of our system in an environmental sensing application.

3.1 Introduction

The emergence of small, long-lived battery-powered sensing devices has enabled the proliferation of networked data-centric sensor applications. These applications have ranged from wildlife habitat sensing [56, 81] to monitoring buildings [93], and are typically environmental monitoring systems, where sensors report local measurements of physical parameters.

One important characteristic of these systems is their extreme resource limitations, in contrast to the computing devices common in most fields. Processors are small and slow: floating point speed is measured in thousands of operations per second. Memory may be as small as 4096 bytes (Mica2), and network speed may be tens of small (100 bytes or less) packets per second. The most critical resource, however, is energy. A sensor

node such as the Telos or Mica2, powered by two AA batteries and using current radio technologies, will exhaust its power source in less than 3 days in continuous receive mode, and in perhaps a week or two if the radio is turned off but the CPU remains active. It is thus crucial to conserve both communication and computation in order to implement long-lived monitoring applications.

3.1.1 Background and Motivation

The growing use of networks of such resource-limited wireless sensors has turned the problem of gathering and processing data under severe resource constraints into an area of active research. Much of the work in this area has focused on the trade-off between computation and communication, exploiting the fact that computation is many orders of magnitude less expensive than radio communication. This *computation vs communication trade-off* has had a tremendous influence both on algorithm design as well as on the design of sensor network platforms themselves.

However, there are several developments which appear promising for sensor network systems which look beyond optimizing the communication/computation tradeoff in homogeneous networks of small sensors.

The first is the availability of low-power, high-capacity flash memory. NAND flash storage, under rapid development due to demand from the camera and portable music markets, is readily adaptable to small sensor platforms. Previous work has shown it to be as much as two orders of magnitude more energy-efficient, on a byte-per-byte basis, than radio transmission on currently available sensor platforms [58]. Without sensor-local storage, an application must request data in advance, and in order to ensure that it receives the information it needs in all cases, often must receive much unneeded data as well. If instead this data could be cheaply stored locally at the sensor until the application can determine precisely which parts are needed, then it should be possible to reduce the amount of communication and thus energy usage.

A second development which we believe should be considered in the design of wireless sensor networks is the availability of relatively resource-rich systems within a deployment, allowing a hierarchical organization. In many sensor deployments to date ([56, 87]), higher-powered “micro-servers” have been deployed to aggregate data from a number of sensors. Such systems typically have greater communications bandwidth and computational resources, and may derive energy from solar cells, the power grid, or other means. Yet little work to date has focused on such inhomogeneous networks, and on mechanisms for using additional computation and communications resources in order to reduce corresponding burdens on resource-constrained sensors.

3.1.2 Contributions

In this chapter we present TSAR, a data storage and retrieval architecture for energy-constrained wireless sensor networks. The TSAR system takes advantage of both low-power flash storage, in order to store data locally at sensors, and resource-rich proxy systems, to index summaries of this data and efficiently forward application queries to those sensors containing responses. At the sensor level, information is both stored locally and summarized into concise metadata which is forwarded to a proxy. The proxies index this data using a novel distributed approximate search structure, the Interval Skip Graph, and then route application queries via this index and cache results. The use of an approximate index results in a trade-off between index update overhead, which goes up as the index precision increases, and search overhead, which is reduced by index precision. The summarization precision therefore adapts according to the access patterns of the particular application, in order to achieve optimal performance.

The remainder of this chapter is structured as follows: In Section 3.2 we discuss the requirements and target applications for TSAR, and give an overview of its design. In Section 3.3 we describe the proxy design and implementation, and in particular the index structure; in Section 3.4 we give results from experimental evaluation of this index.

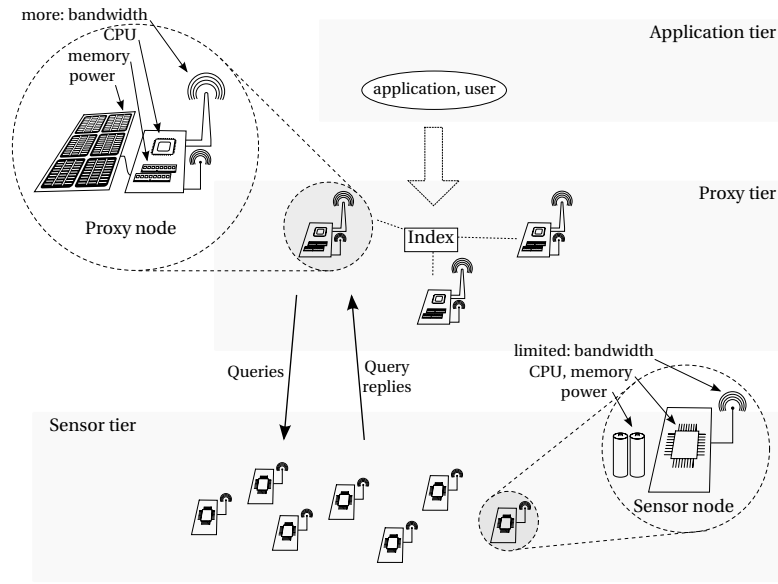


Figure 3.1. Architecture of a multi-tier sensor network

Section 3.5 describes the sensor node mechanisms and implementation, and Section 3.6 presents results from evaluation of a prototype TSAR system. Finally, in Section 3.7 we survey related work, and conclude in Section 3.8.

3.2 Requirements and Design

We begin by describing in more detail the problems TSAR is designed to solve and the environments it is intended to be deployed in. We then describe the design of the system in relation to these factors.

3.2.1 System Model

We envision a sensor network comprising three tiers — a bottom tier of untethered remote sensor nodes, a middle tier of tethered sensor proxies, and an upper tier of applications and user terminals (see Figure 3.1).

Low-power sensors form the bottom tier. These nodes may be equipped with sensing equipment, a microcontroller, and a radio, as well as a significant amount of flash memory

(e.g., 1GB). Devices at this tier are frequently battery powered and thus energy-limited, and the active lifespan of the deployment may be limited by this energy constraint. The use of radio, processor, and the flash memory all consume energy, which needs to be limited. In general, we can assume that computation (e.g. a single instruction) requires significantly less energy than storage (writing or reading a byte), which in turn is significantly cheaper than transmitting or receiving a byte over the radio.

More powerful micro-servers form the second or *proxy* tier, so called because they relay or proxy requests from the application to the sensors. These proxy systems have significant computation, memory and storage resources in comparison to the sensors. In addition, they have greater energy reserves, and in the case of tethered or solar-powered systems may be able to operate continuously. In urban environments with grid power available, the proxy tier could be comprised of tethered base-station class nodes (e.g., Crossbow Stargate), each with with multiple radios—an 802.11 radio that connects it to a wireless mesh network and a low-power radio (e.g. 802.15.4) that connects it to the sensor nodes. In remote sensing applications [32], this tier could comprise a similar node with a solar array.

The proxy nodes would preferably be distributed geographically within the sensor field, with each proxy associated with or managing perhaps tens to several hundreds of sensor nodes. For instance, in a building monitoring application, one sensor proxy might be placed per floor or hallway to monitor temperature, heat and light sensors in their vicinity.

At the highest tier of our infrastructure are the applications themselves, which query the sensor network through a query interface [55].

3.2.2 Usage Models

In designing this system, we first consider the ways in which we expect it to be used. Sensors in such a system typically provide information about the physical world; two key attributes of this information are *when* something happened and *where* it occurred. We therefore expect a large fraction of queries on sensor data to be spatio-temporal in nature.

In addition, it is often desirable to support efficient access to data in a way that maintains spatial and temporal ordering, and in particular to support queries for ranges of these parameters. There are several ways of implementing range queries in an index system, such as locality-preserving hashes as in DIMS [52]. However, the most straightforward mechanism, and one which naturally provides efficient ordered access, is via the use of *order-preserving* data structures. Order-preserving structures such as the well-known B-Tree maintain relationships between indexed values, and thus allow natural access to ranges, as well as predecessor and successor operations on their key values.

3.2.3 Design Principles

To address the requirements identified in the system and usage models above, we base our design on the following set of principles.

- **Principle 1:** *Store locally, access globally:* Since local flash storage is much more energy efficient than radio communications, for maximum network life a sensor storage system should leverage this storage to archive data locally on each sensor, substituting cheap flash operations for expensive radio transmission. However, without efficient mechanisms for retrieval, the energy gains of local storage may be outweighed by communication costs incurred by the application in searching for data. We believe that if the data storage system provides the abstraction of a single logical store to applications, as does TSAR, then it will have additional flexibility to optimize communication and storage costs.
- **Principle 2:** *Distinguish data from metadata:* Data must be identified so that it may be retrieved by the application without exhaustive search. To do this, we associate *metadata* with each data record — data fields of known syntax which serve as identifiers and may be queried by the storage system, such as location and time, or selected or summarized data values. We leverage the presence of resource-rich proxies to index metadata for resource-constrained sensors. The proxies share this

metadata index to provide a unified logical view of all data in the system, thereby enabling efficient, low-latency look-ups. Such a tier-specific separation of data storage from metadata indexing enables the system to exploit the idiosyncrasies of multi-tier networks, while improving performance and functionality.

- **Principle 3:** *Provide data-centric query support:* Unlike a general-purpose file system, in a sensor network the application does not write the data, and thus is unable to supply precise location information for retrieving it. If exhaustive search is not feasible, then applications will be best served by a query interface which allows them to locate data by value or attribute (e.g. location and time)—i.e. an interface like a database, rather than a file system. This in turn implies the need to identify and organize data in a way which allows the network to perform such queries.

3.2.4 System Design

TSAR embodies these design principles by employing local storage at sensors and a distributed index at the proxies. The key features of the system design are as follows:

Sensed data is written at the sensor nodes as a combination of opaque application data and structured *metadata*, a tuple of known types which may be used to locate and identify data records. In a camera-based sensing application, for instance, this metadata might include coordinates describing the field of view or basic image recognition results. Depending on the application, this metadata may be two or three orders of magnitude smaller than the data itself, for instance if the metadata consists of features extracted from image or acoustic data.

In addition to storing data locally, each sensor periodically sends a metadata summary to a nearby proxy. The summary contains information such as the sensor ID, timestamps for the beginning and end of the time period summarized, and a coarse-grained representation of the metadata associated with the record. The precise data representation used in the summary is application-specific; for instance, a temperature sensor might choose to report

the maximum and minimum temperature values observed in an interval as a summary of the actual time series.

3.3 Proxy Level - Indexing

We begin a more detailed description of the TSAR system with the proxy tier, and in particular its index structure.

3.3.1 Summary Index

The proxy nodes implement a distributed index in order to locate sensor data records in response to application queries. The keys in this index are taken from metadata records reported by the sensors themselves, describing data which is stored on those sensor nodes. In order to reduce the communications overhead of index creation at the sensor nodes, metadata from multiple records is represented by a single summary, a compact representation of multiple values. Although many summarization mechanisms are available, for indexing purposes we assume that each summary can be considered to identify an interval or range of values, matching any value falling within that range.

The proxies use these summaries to construct a global index, storing information from all sensors in the system and distributed across the various proxies in the system. Thus, applications see a unified view of distributed data, and can query the index at any proxy to get access to data stored at any sensor. Specifically, each query triggers lookups in this distributed index; the proxy is then responsible for querying the sensors identified in the matching index records, and assembling their results.

There are several distributed index and lookup methods which might be used in this system; however, the index structure described below in Section 3.3.2 is highly suited for the task. In particular, it is able to index *intervals*, and the index may be queried to find all intervals overlapping a given value. This is a natural fit for our summarization mechanism, as each summary corresponds to a single index entry.

Since the index is constructed using a coarse-grained summary, index lookups will yield approximate matches. The interval-based summarization mechanism may result in *false positives*, where a summary matches the query, but the value is not in fact there. In that case, when the query is routed to the sensor itself, no matching record will be found. Such false positives cause no harm, other than network overhead, as they will not affect the result set being assembled by the proxy. If a summary were to result in a *false negative* it could result in valid query matches being skipped over; however, the interval summarization mechanism used in TSAR guarantees that this will not occur.

3.3.2 Index structure

To implement this index, TSAR employs a novel structure called the Interval Skip Graph, which is an ordered, distributed data structure for finding all intervals that intersect a particular point or range of values. Interval skip graphs combine Interval Trees [18], an interval-based binary search tree, with Skip Graphs [7], a ordered, distributed data structure for peer-to-peer systems [36]. The resulting data structure has two properties that make it ideal for sensor networks. First, it has $O(\log n)$ search complexity for accessing the first interval that matches a particular value or range, and constant complexity for accessing each successive interval. Second, indexing of intervals rather than individual values makes the data structure ideal for indexing summaries over time or value. Such summary-based indexing is a more natural fit for energy-constrained sensor nodes, since transmitting summaries incurs less energy overhead than transmitting all sensor data.

In the remainder of this section we will first explain the building blocks of interval skip graphs — skip lists and simple skip graphs — and then describe interval skip graphs themselves, and finally compare them to alternate data structures.

3.3.2.1 Skip Graph Overview

A *skip graph* is a distributed version of a *skip list* [69], a randomized balanced tree which offers similar performance to balanced binary trees without needing complex bal-

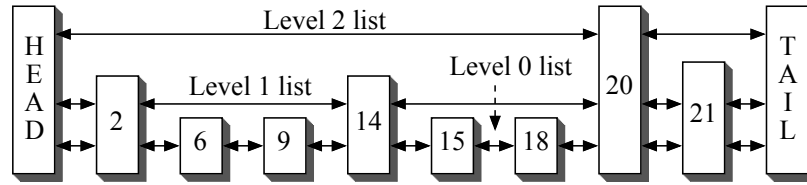


Figure 3.2. A skip list of sorted elements

ancing operations. A skip list is constructed as a multiply-linked list, as shown in Figure 3.2, beginning with a head and ending with a tail. At the lowest level, 0, all N elements are linked in sorted order. At each successive level i , approximately $N/2^i$ of the nodes are randomly chosen to be linked together; the highest level at which the head is linked to any data nodes is thus on average approximately $\log_2 N$.

Searching a skip list is analogous to a binary search on the sorted list of nodes, except that at each step the remaining interval is split at a random point (i.e. at the destination of a link traversing on average half the interval) rather than splitting exactly in two. The search algorithm is shown in Figure 3.3; we begin at the head and proceed downwards, at each level making a comparison and deciding whether to skip forwards or remain at the same entry. Insertion of a particular key begins with the same search algorithm, in order to locate the proper position in the list for the new key. The new element is then inserted into linked lists at levels 0 through m , where m is geometrically distributed with $p = 1/2$.

Skip *graphs* extend this basic idea, to create a structure where each node (and key) may reside on an independent system, and where searches or inserts may begin at any of the nodes. Besides the need to replace pointers with tokens which can identify a node residing on a remote system, the following changes are required:

Double-linking: In a skip list, all searches begin at a root, e.g. the left-most node in Figure 3.2, and all link traversals are in a single direction. In order to allow operations to begin at any node, it must be possible to traverse links in a skip graph in either direction.

Multiple roots: As may be seen in Figure 3.4, where at level ℓ a skip list has a single chain linking roughly $N/2^\ell$ nodes, each node in a skip graph is linked at all $\log_2 N$ levels,

```

1:  $\ell \leftarrow height$ 
2:  $n \leftarrow head$ 
3: while  $\ell > 0$  do
4:   while  $n.next_\ell < key$  do
5:      $n = n.next_\ell$ 
6:      $\ell \leftarrow \ell - 1$ 
7:   end while
8: end while

```

Figure 3.3. Skip List Search Algorithm

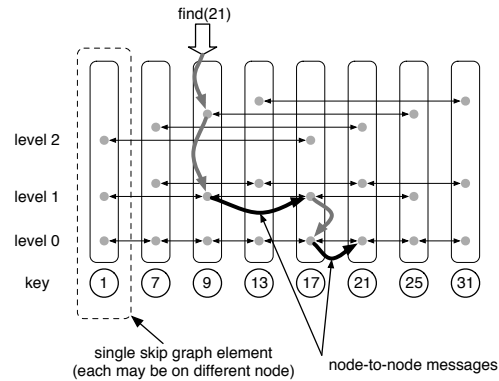


Figure 3.4. Skip Graph of 8 Elements showing search operation with two messages

resulting in 2^ℓ separate chains at each level. This allows searches to begin at any node in the graph.

A skip graph contains a complete skip list rooted at each node in the graph, so the search algorithm is almost identical to that for skip lists. This is shown in Figure 3.4, where it may be seen that pointer traversals are replaced with messages. Note, however, that it is no longer possible to access two nodes simultaneously, which is necessary in line 4 of the algorithm above. This is solved most efficiently by exchanging key values when creating links, so that both values are available at either node.

To construct this structure, each node is assigned a bit vector V , and links at level ℓ with the nearest nodes sharing the first ℓ bits of V . To do this a search is performed to find the node's two neighbors in the level 0 list, and then links are established from the bottom up, passing a message via level ℓ neighbors to find the nearest level $\ell + 1$ neighbor. (This neighbor will share the first $\ell + 1$ bits of V , and will thus be linked on the list of nodes sharing only ℓ bits.)

The resulting data structure has the following properties:

- **Ordering:** As in other tree search structures, searches are done by ordered comparisons of keys such as integers. The structure allows keys to be easily and efficiently

retrieved in sorted order, allowing range queries to be implemented in a straightforward manner.

- **In-place indexing:** Data elements remain on the nodes where they were inserted, and messages are sent between nodes to establish links between those elements and others in the index.
- **Log N height:** There are $H \approx \log_2 N$ pointers associated with each element, where N is the number of data elements indexed. Each pointer belongs to a level ℓ in $[0 \dots H - 1]$, and together with some other pointers at that level forms a chain of approximately $N/2^\ell$ elements.
- **Probabilistic balance:** Because the tree structure is determined by a random number stream, rather than data keys, it is approximately balanced with high probability. Thus, there is no need for global re-balancing operations during inserts or deletes, and the tree is unaffected by the structure of data values.
- **Redundancy and resiliency:** Each data element forms an independent search tree root, so searches may begin at any node in the network, eliminating hot spots at a single search root. In addition the index is resilient against node failure; data on the failed node will not be accessible, but remaining data elements will be accessible through search trees rooted on other nodes.

3.3.2.2 Interval Skip Graph

A skip graph as described above is designed to store single-valued entries. In this section, we introduce a novel data structure that extends skip graphs to store intervals $[low_i, high_i]$ and allows efficient searches for all intervals covering a value v , i.e. $\{i : low_i \leq v \leq high_i\}$. Our data structure can be extended to range searches in a straightforward manner.

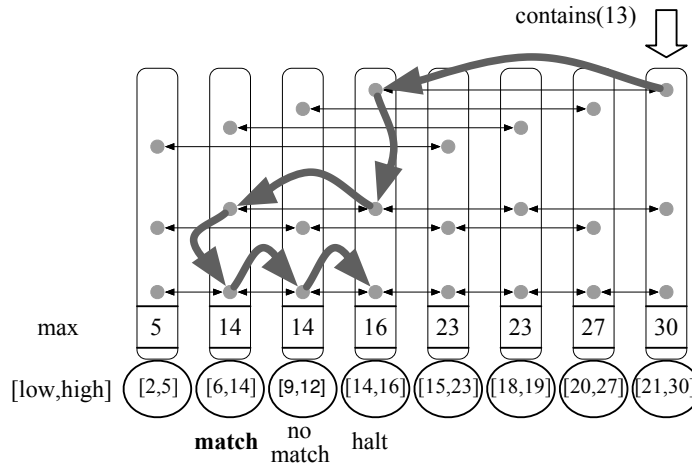


Figure 3.5. Interval Skip Graph and search for intervals containing key 13

The interval skip graph is constructed by applying the method of augmented search trees, as described by Cormen, Leiserson, and Rivest [18] and applied to binary search trees to create an Interval Tree. The method is based on the observation that a search structure based on comparison of ordered keys, such as a binary tree, may also be used to search on a secondary key which is non-decreasing in the first key.

Given a set of intervals sorted by lower bound – $low_i \leq low_{i+1}$ – we define the secondary key as the cumulative maximum, $max_i = \max_{k=0..i} (high_k)$. The set of intervals intersecting a value v may then be found by searching for the first interval (and thus the interval with least low_i) such that $max_i \geq v$. We then traverse intervals in increasing order lower bound, until we find the first interval with $low_i > v$, selecting those intervals which intersect v .

Using this approach we augment the skip graph data structure, as shown in Figure 3.5, so that each entry stores a range (lower bound and upper bound) and a secondary key (cumulative maximum of upper bound). To efficiently calculate the secondary key max_i for an entry i , we take the greatest of $high_i$ and the maximum values reported by each of i 's left-hand neighbors.

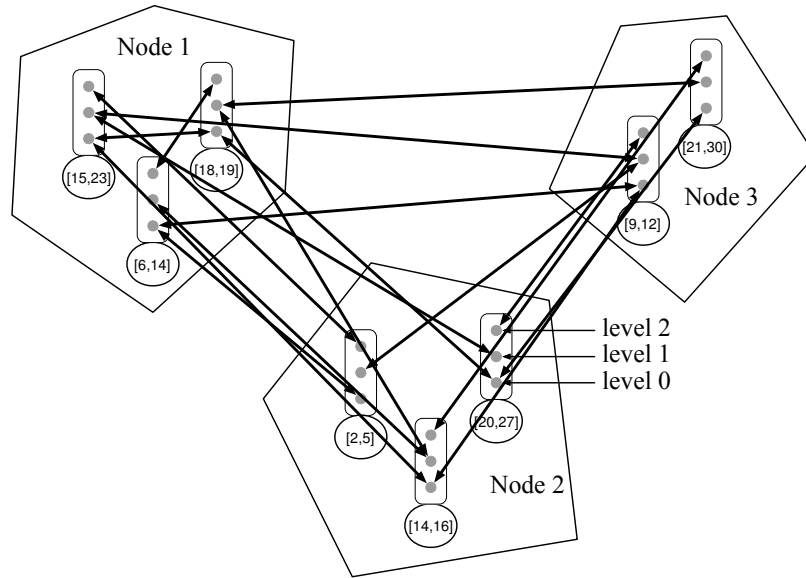


Figure 3.6. Distributed Interval Skip Graph. Note that this structure is exactly the same as that in Figure 3.4, except that it is shown in place on 3 nodes rather than in data order

To search for those intervals containing the value v , we first search for v on the secondary index, max_i , and locate the first entry with $max_i \geq v$. (by the definition of max_i , for this data element $max_i = high_i$.) If $low_i > v$, then this interval does not contain v , and no other intervals will, either, so we are done. Otherwise we traverse the index in increasing order of min_i , returning matching intervals, until we reach an entry with $min_i > v$ and we are done. Searches for all intervals which overlap a query range, or which completely contain a query range, are straightforward extensions of this mechanism.

Lookup Complexity: Lookup for the first interval that matches a given value is performed in a manner very similar to an interval tree. The complexity of search is $O(\log n)$. The number of intervals that match a range query can vary depending on the amount of overlap in the intervals being indexed, as well as the range specified in the query.

Insert Complexity: In an interval tree or interval skip list, the maximum value for an entry need only be calculated over the subtree rooted at that entry, as this value will be examined only when searching within the subtree rooted at that entry. For a simple interval skip graph, however, this maximum value for an entry must be computed over all entries

preceding it in the index, as searches may begin anywhere in the data structure, rather than at a distinguished root element. It may be easily seen that in the worst case the insertion of a single interval (one that covers all existing intervals in the index) will trigger the update of all entries in the index, for a worst-case insertion cost of $O(n)$.

3.3.2.3 Sparse Interval Skip Graph

The final extensions we propose take advantage of the difference between the number of items indexed in a skip graph and the number of systems on which these items are distributed. The cost in network messages of an operation may be reduced by arranging the data structure so that most structure traversals occur locally on a single node, and thus incur zero network cost. In addition, since both congestion and failure occur on a per-node basis, we may eliminate links without adverse consequences if those links only contribute to load distribution and/or resiliency within a single node. These two modifications allow us to achieve reductions in asymptotic complexity of both update and search.

As may be in Section 3.3.2.2, insert and delete cost on an interval skip graph has a worst case complexity of $O(n)$, compared to $O(\log n)$ for an interval tree. The main reason for the difference is that skip graphs have a full search structure *rooted at each element*, in order to distribute load and provide resilience to system failures in a distributed setting. However, in order to provide load distribution and failure resilience it is only necessary to provide a full search structure *for each system*. If as in TSAR the number of nodes (proxies) is much smaller than the number of data elements (data summaries indexed), then this will result in significant savings.

Implementation: To construct a sparse interval skip graph, we ensure that there is a single distinguished element on each system, the *root element* for that system; all searches will start at one of these root elements. When adding a new element, rather than splitting lists at increasing levels l until the element is in a list with no others, we stop when we find that the element would be in a list containing no root elements, thus ensuring that the

element is reachable from all root elements. An example of applying this optimization may be seen in Figure 3.7. (In practice, rather than designating existing data elements as roots, as shown, it may be preferable to insert null values at startup.)

When using the technique of membership vectors as in [7], this may be done by broadcasting the membership vectors of each root element to all other systems, and stopping insertion of an element at level l when it does not share an l -bit prefix with any of the N_p root elements. The expected number of roots sharing a $\log_2 N_p$ -bit prefix is 1, giving an expected height for each element of $\log_2 N_p + O(1)$. An alternate implementation, which distributes information concerning root elements at pointer establishment time, is omitted due to space constraints; this method eliminates the need for additional messages.

Performance: In a (non-interval) sparse skip graph, since the expected height of an inserted element is now $\log_2 N_p + O(1)$, expected insertion complexity is $O(\log N_p)$, rather than $O(\log n)$, where N_p is the number of root elements and thus the number of separate systems in the network. (In the degenerate case of a single system we have a skip list; with splitting probability 0.5 the expected height of an individual element is 1.) Note that since searches are started at root elements of expected height $\log_2 n$, search complexity is not improved.

For an interval sparse skip graph, update performance is improved considerably compared to the $O(n)$ worst case for the non-sparse case. In an augmented search structure such as this, an element only stores information for nodes which may be reached from that element—e.g. the subtree rooted at that element, in the case of a tree. Thus, when updating the maximum value in an interval tree, the update is only propagated towards the root. In a sparse interval skip graph, updates to a node only propagate towards the N_p root elements, for a worst-case cost of $N_p \log_2 n$.

Shortcut search: When beginning a search for a value v , rather than beginning at the root on that proxy, we can find the element that is closest to v (e.g. using a secondary local index), and then begin the search at that element. The expected distance between

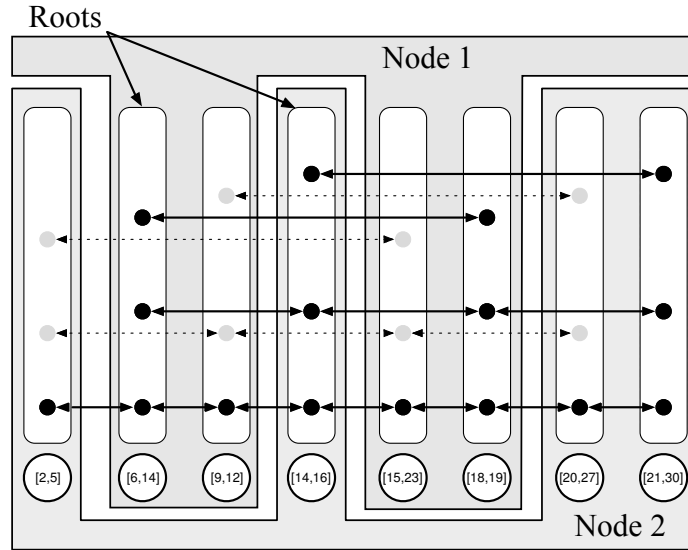


Figure 3.7. Sparse Interval Skip Graph

this element and the search terminus is $\log_2 N_p$, and the search will now take on average $\log_2 N_p + O(1)$ steps. To illustrate this optimization, in Figure 3.6 depending on the choice of search root, a search for $[21, 30]$ beginning at node 2 may take 3 network hops, traversing to node 1, then back to node 2, and finally to node 3 where the destination is located, for a cost of 3 messages. The shortcut search, however, locates the intermediate data element on node 2, and then proceeds directly to node 3 for a cost of 1 message.

Performance: This technique may be applied to the primary key search which is the first of two insertion steps in an interval skip graph. By combining the short-cut optimization with sparse interval skip graphs, the expected cost of insertion is now $O(\log N_p)$ messages, independent of the size of the index or the degree of overlap of the inserted intervals.

3.3.3 Low-Resolution Redundant Indexing

The analysis above considers the case where all parts of the network are operational. However, failure handling is an important issue in a multi-tier sensor architecture, since it relies on many components—proxies, sensor nodes and routing nodes can fail, and wire-

less links can fade. Handling of many of these failure modes is outside the scope of this paper; however, we consider the case of resilience of skip graphs to proxy failures. In this case, skip graph search (and subsequent repair operations) can follow any one of the other links from a root element. Since a sparse skip graph has search trees rooted at each node, searching can then resume once the lookup request has routed around the failure. Together, these two properties ensure that even if a proxy fails, the remaining entries in the skip graph will be reachable with high probability—only the entries on the failed proxy and the corresponding data at the sensors becomes inaccessible.

To further improve on this resiliency, we would like to ensure the availability of the data indexed by the failed proxy, as well. To do this we incorporate redundant index entries, using a simple scheme where additional coarse-grained summaries are used to protect the regular index entries. Each sensor sends summary data periodically to its local proxy, but less frequently sends a lower-resolution summary to a backup proxy. This backup summary represents the same data as the finer-grained ones, but in a less accurate fashion. This results in higher read overhead (due to false hits) if the backup summary is used, but conserves bandwidth in the expected case where the primary index node is available, providing resiliency at low cost.

3.4 Index Evaluation

We next present results from a series of experiments designed to evaluate the properties of our Interval Skip Graph index. Experiments were performed in simulation, using the actual code from our prototype TSAR implementation. This allowed us to obtain precise measurements of packet transmissions, and thus message complexity, which would not have been possible with more general simulations or with actual over-the-air operation.

Experiments were performed with two different data sets. The first is a temperature dataset from James Reserve [92] that includes data from eleven temperature sensor nodes over a period of 34 days. The second dataset is synthetically generated, using the same

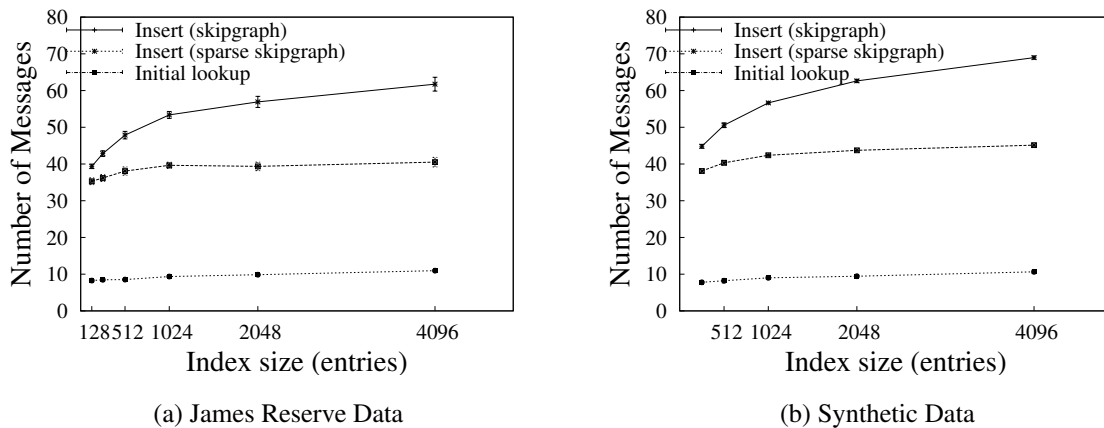


Figure 3.8. Skip Graph Insert Performance

value range as found in the James River data; the trace for each sensor is generated using a uniformly distributed random walk though that data range.

3.4.1 Insert/Delete Performance

We first examine insert and delete overheads of our index. We assume a proxy tier with 32 proxies and construct sparse interval skip graphs of various sizes using our datasets. For each skip graph, we evaluate the cost of inserting a new value into the index. After inserting all entries, all entries were then deleted, enabling us to measure the delete overhead as well. Figures 3.8(a) and (b) quantify the insert overhead for our two datasets: each insert entails an initial traversal that incurs $\log n$ messages, followed by neighbor pointer update at increasing levels, incurring a cost of $4 \log n$ messages. Our results demonstrate this behavior, and show as well that performance of delete—which also involves an initial traversal followed by pointer updates at each level—incurs a similar cost.

3.4.2 Lookup Performance

Next, we evaluate the lookup performance of the index structure. Again, we construct skip graphs of various sizes using our datasets and evaluate the cost of a lookup on the index structure. Figures 3.9(a) and (b) depict our results. There are two components for

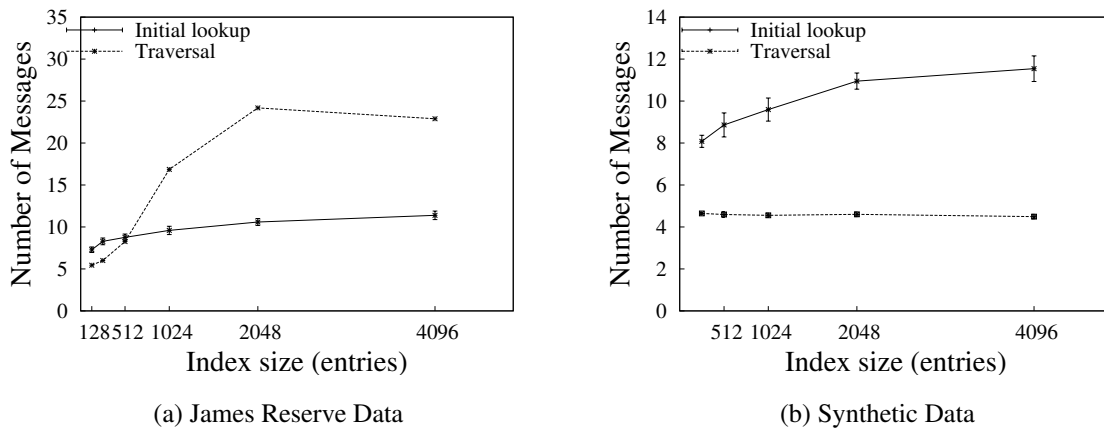


Figure 3.9. Skip Graph Lookup Performance

each lookup—the lookup of the first interval that matches the query and, in the case of overlapping intervals, the subsequent linear traversal to identify all matching intervals. The initial lookup can be seen to take $\log n$ messages, as expected. The costs of the subsequent linear traversal, however, are highly data dependent. For instance, temperature values for the James Reserve data exhibit significant spatial correlations, resulting in significant overlap between different intervals and variable, high traversal cost (see Figure 3.9(a)). The synthetic data, however, has less overlap and incurs lower traversal overhead as shown in Figure 3.9(b).

3.4.3 Variations

Since the previous experiments assumed 32 proxies, we evaluate the impact of the number of proxies on skip graph performance. We vary the number of proxies from 10 to 48 and distribute a skip graph with 4096 entries among these proxies. We construct regular interval skip graphs as well as sparse interval skip graphs using these entries and measure the overhead of inserts and lookups. Thus, the experiment also seeks to demonstrate the benefits of sparse skip graphs over regular skip graphs. Figure 3.10(a) depicts our results. In regular skip graphs, the complexity of insert is $O(\log_2 n)$ in the expected case (and $O(n)$

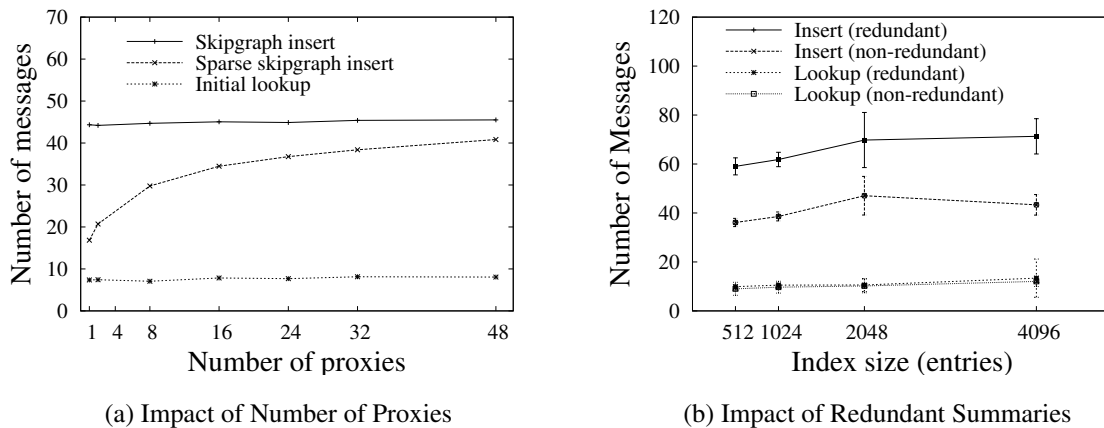


Figure 3.10. Skip Graph Overheads

in the worst case) where n is the number of *elements*. This complexity is unaffected by changing the number of proxies, as indicated by the flat line in the figure. Sparse skip graphs require fewer pointer updates; however, their overhead is dependent on the number of proxies, and is $O(\log_2 N_p)$ in the expected case, independent of n . This can be seen to result in significant reduction in overhead when the number of proxies is small, which decreases as the number of proxies increases.

Finally, we measure the overhead of low-resolution redundant indexing for recovery from proxy failures. We see the overhead of this redundancy scheme in Figure 3.10(b), with one coarse summary being sent to a backup for every two summaries sent to the primary proxy. Since a redundant summary is sent for every two summaries, the insert cost is 1.5 times the cost in the normal case. However, these redundant entries result in only a negligible increase in lookup overhead, due the logarithmic dependence of lookup cost on the index size, while providing full resilience to any single proxy failure.

3.5 Sensor Level - Data Handling

Having described and evaluated the proxy-level index structure, we turn to the design of the sensor tier. TSAR implements two key mechanisms at this tier: a local archival store

at each sensor node that is optimized for resource-constrained devices, and an adaptive summarization algorithm that enables each sensor to adapt to changing data and query characteristics. The rest of this section describes these features in more detail.

3.5.1 Data Sensing and Storage

We begin with sensing and storage. As TSAR sensor nodes receive measurement or sensing data, they divide it into application data and structured, searchable metadata. In some cases, simple values such as temperature may serve directly as metadata. For e.g. vision or vibration sensors, application-specific computation will be needed to determine values such as movement or amplitude which can be searched on. A record consisting of both metadata and any additional information is then stored locally on the sensor node.

In the TSAR implementation this is done by writing to an append-only store, and for each record written a handle is returned which identifies the storage location of the record. When metadata summaries are reported to the proxies, a range of handles is reported as well. We use these handles to reduce the amount of searching and indexing needed on the sensor nodes. In particular, when a match is found in the index, the corresponding handle range can be passed to the sensor node along with the query. The sensor is then able to directly search the matching records, avoiding the need for either exhaustive search or complex index structures.

While we anticipate local storage capacity to be large, eventually there might be a need to overwrite older data, especially in high data rate applications. This may be done via techniques such as multi-resolution storage of data [30], or just simply by overwriting older data. In addition, recent work on data storage for wireless sensor networks (e.g. CAPSULE [57]) has resulted in data structures which could be utilized both for storage reclamation and for efficient indexing and search of sensor-local storage.

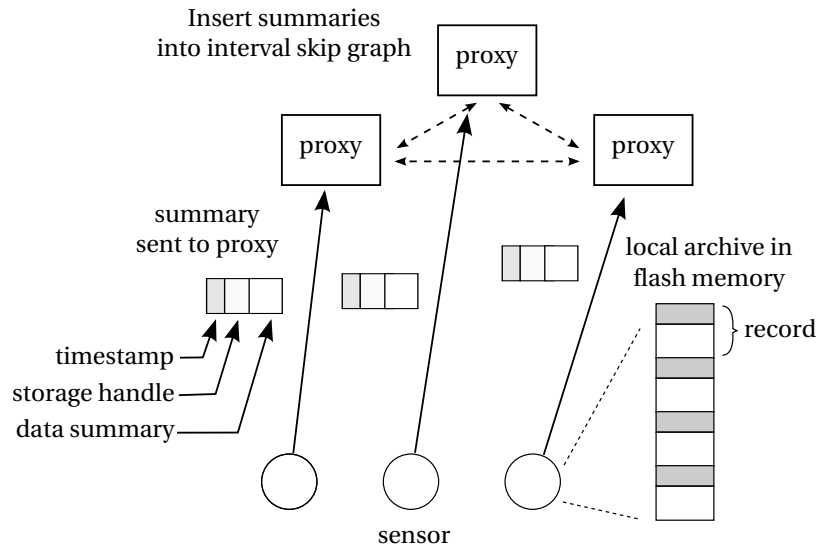


Figure 3.11. Sensor Summarization

3.5.2 Summarization

After data is received, it is summarized; as described previously in Section 3.3, multiple records into a single summary in order to reduce the overhead of index update messages from sensors to proxies. This relationship may be seen in Figure 3.11. The summarization algorithm is simple, representing multiple scalar values by an interval containing all of the values. In cases where the data values are not simple scalars, it may be necessary to apply an appropriate transform in order to summarize in this fashion; for instance, z-coding [64] could be used on 2-dimensional data.

The simplest way of implementing interval summarization is by collecting a fixed number of data values, or for a fixed length of time, and then reporting the interval containing these values. More sophisticated mechanisms could be used as well to improve the precision of the summaries, by e.g. varying the number of samples in each summary to avoid creating overly broad intervals. For the remainder of this chapter we assume that simple fixed windows are used for summarization.

These data summaries serve as glue between the storage at the remote sensor and the index at the proxy. Each update from a sensor to the proxy includes the summary (i.e. data

interval), and the timestamps of the earliest and latest data points summarized. In general, the proxy can index the time interval representing a summary or the value range reported in the summary (or both). The former index enables quick lookups on all records seen during a certain interval, while the latter index enables quick lookups on all records matching a certain value.

3.5.3 Query handling

This index is searched when application queries are received by a proxy, in order to locate matching summaries. Each such match corresponds to a report from one of the sensor nodes, representing a set of data records. The record in the index will identify the corresponding sensor, as well as providing a handle to locate the matching records on the sensor. The query and corresponding handle will then be forwarded to the sensing node. Upon receiving the query, the sensor node locates the corresponding range of records based on the handle information, and then searches them for matches to the query. A message indicating whether any matching records were found, and if so how many, is sent back to the proxy, followed by the matching records.

3.5.4 Adaptive summarization

In the case where no records are found, due to a false hit in the summary index, the query still must be transmitted to the sensor and a response returned, consuming energy. Since the chance of a false hit increases as the resolution of the summaries decreases, there is a trade-off between energy used in sending summaries (and thus their frequency and resolution) and the cost of forwarding queries due to false hits in the index. The coarser and less frequent the summary information, the less energy will be required for updates, and the more energy that will be wasted looking for non-existent data during queries.

To address this, TSAR employs an adaptive summarization technique that balances the cost of sending updates against the cost of false positives. The key intuition is that each sensor can independently identify the fraction of false hits and true hits for queries that

	Proxy	Sensor
CPU	400MHz XScale	8MHz ATmega 128L
Memory	64M	4K RAM / 128K ROM
OS	Linux 2.4.19/EmStar 2.1	TinyOS 1.1.8
Network	802.11b, UDP/IP	CC1000, BMAC

Table 3.1. Proxy and sensor platforms used in prototype

access its local archive. If most queries result in true hits, then the sensor determines that the summary can be coarsened further to reduce update costs without adversely impacting the hit ratio. If many queries result in false hits, then the sensor makes the granularity of each summary finer to reduce the number and overhead of false hits.

3.6 Prototype Evaluation

In this section we present results from the evaluation of a prototype of the TSAR system, consisting of both proxy and sensor nodes.

3.6.1 Prototype Implementation

Our prototype implements proxies and sensor nodes on Crossbow Stargate and Mica2/Mica2Dot systems, respectively. Detailed specifications for these platforms may be seen in Table 3.1; on-board NOR flash was used for storage, while in actual deployment more efficient NAND flash would be used.

Since sensor nodes may be several hops away from the nearest proxy, the sensor tier employs multi-hop routing to communicate with the proxy tier. In addition, to reduce the power consumption of the radio while still making the sensor node available for queries, low power listening is enabled, in which the radio receiver is periodically powered up for a short interval to sense the channel for transmissions, and the packet preamble is extended to account for the latency until the next interval when the receiving radio wakes up. Our prototype employs the MultiHopLEPSM routing protocol with the BMAC layer configured in the low-power mode with a 11% duty cycle, one of the default BMAC [67] parameters.

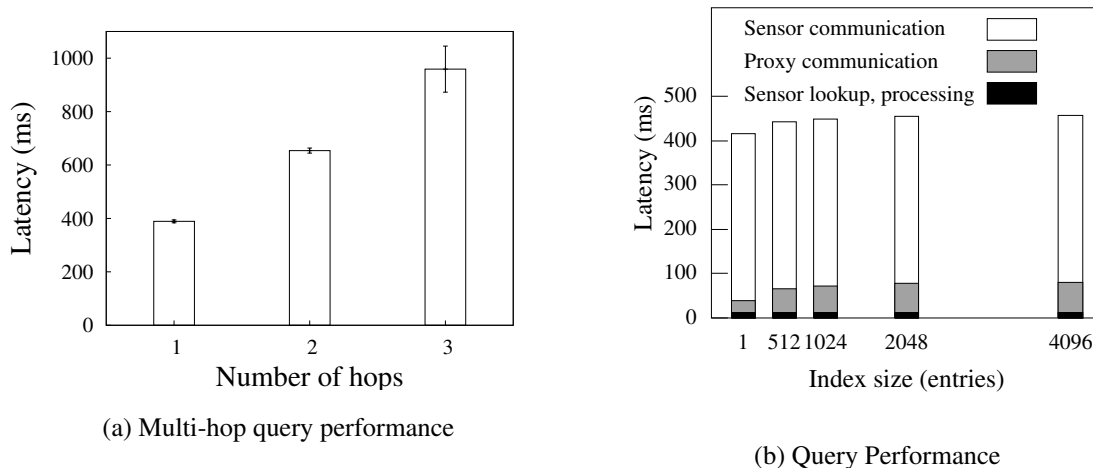


Figure 3.12. Query Processing Latency

3.6.2 Query processing latency

We first present results from an end-to-end evaluation of the TSAR prototype involving both tiers. Four proxies are connected via 802.11 links, with three sensors associated with each proxy. By hand-configuring the routing, the sensor node topology was configured to connect each group of three sensors in a line, forming a “tree” of depth 3. Due to resource constraints we were unable to perform experiments with dozens of sensor nodes, however this topology ensured that the network diameter was as large as for a typical network of significantly larger size.

The evaluation metric used is the end-to-end latency of query processing. Each query first incurs the latency of a sparse skip graph lookup, followed by routing to the appropriate sensor node(s). The sensor node reads the required page(s) from its local archive, processes the query on the page that is read, and transmits the response to the proxy, which then forwards it to the user. We first measure query latency for different sensors in our multi-hop topology. Depending on which of the sensors is queried, the total latency increases almost linearly from about 400ms to 1 second, as the number of hops increases from 1 to 3 (see Figure 3.12(a)).

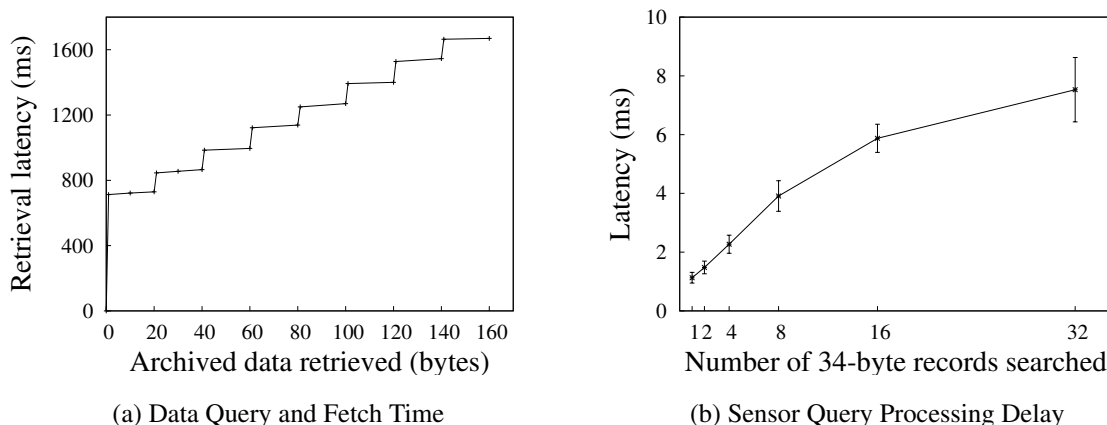


Figure 3.13. Query Latency Components

Figure 3.12(b) provides a breakdown of the various components of the end-to-end latency. The dominant component of the total latency is the communication over one or more hops. The typical time to communicate over one hop is approximately 300ms. This large latency is primarily due to the use of a duty-cycled MAC layer; the latency will be larger if the duty cycle is reduced (e.g. the 2% setting as opposed to the 11.5% setting used in this experiment), and will conversely decrease if the duty cycle is increased. The figure also shows the latency for varying index sizes; as expected, the latency of inter-proxy communication and skip graph lookups increases logarithmically with index size. Not surprisingly, the overhead seen at the sensor is independent of the index size.

The latency also depends on the number of packets transmitted in response to a query—the larger the amount of data retrieved by a query, the greater the latency. This result is shown in Figure 3.13(a). The step function is due to packetization in TinyOS; TinyOS sends one packet so long as the payload is smaller than 30 bytes and splits the response into multiple packets for larger payloads. As the data retrieved by a query is increased, the latency increases in steps, where each step denotes the overhead of an additional packet.

Finally, Figure 3.13(b) shows the impact of searching and processing flash memory regions of increasing sizes on a sensor. Each summary represents a collection of records in

flash memory, and all of these records need to be retrieved and processed if that summary matches a query. The coarser the summary, the larger the memory region that needs to be accessed. For the search sizes examined, amortization of overhead when searching multiple flash pages and archival records, as well as within the flash chip and its associated driver, results in the appearance of sub-linear increase in latency with search size. In addition, the operation can be seen to have very low latency, in part due to the simplicity of our query processing, requiring only a compare operation with each stored element. More complex operations, however, will of course incur greater latency.

3.6.3 Adaptive Summarization

We next present experiments showing the performance of the TSAR summarization mechanism. When data is summarized by the sensor before being reported to the proxy, information is lost. With the interval summarization method we are using, this information loss will never cause the proxy to believe that a sensor node does not hold a value which it in fact does, as all archived values will be contained within the interval reported. However, it does cause the proxy to believe that the sensor may hold values which it does not, and forward query messages to the sensor for these values. These *false positives* constitute the cost of the summarization mechanism, and need to be balanced against the savings achieved by reducing the number of reports. The goal of adaptive summarization is to dynamically vary the summary size so that these two costs are balanced.

Figure 3.14(a) demonstrates the impact of summary granularity on false hits. As the number of records included in a summary is increased, the fraction of queries forwarded to the sensor which match data held on that sensor (“true positives”) decreases. Next, in Figure 3.14(b) we run the a EmTOS simulation with our adaptive summarization algorithm enabled. The adaptive algorithm increases the summary granularity (defined as the number of records per summary) when $\frac{Cost(updates)}{Cost(falsehits)} > 1 + \epsilon$ and reduces it if $\frac{Cost(updates)}{Cost(falsehits)} > 1 - \epsilon$, where ϵ is a small constant. To demonstrate the adaptive nature of our technique, we plot

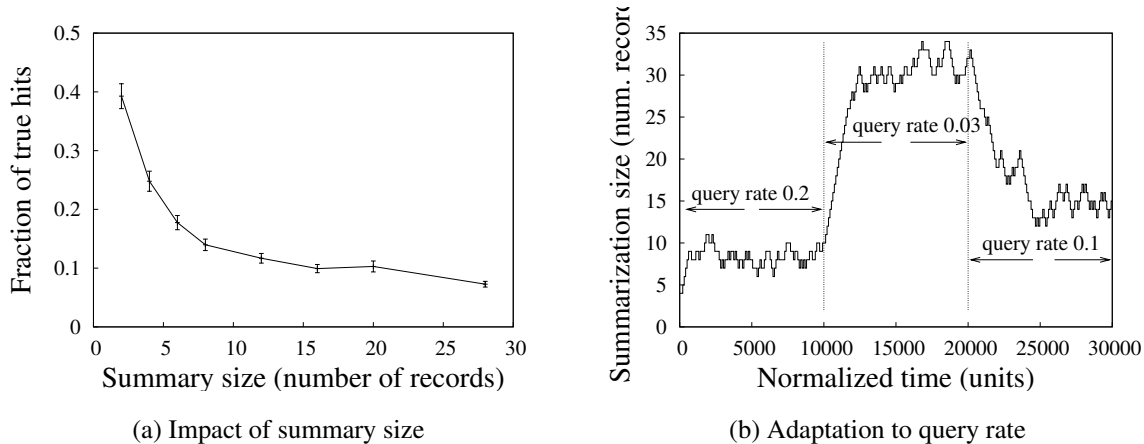


Figure 3.14. Impact of Summarization Granularity

a time series of the summarization granularity. We begin with a query rate of 1 query per 5 samples, decrease it to 1 every 30 samples, and then increase it again to 1 query every 10 samples. As shown in Figure 3.14(b), the adaptive technique adjusts accordingly by sending more fine-grain summaries at higher query rates (in response to the higher false hit rate), and fewer, coarse-grain summaries at lower query rates.

3.7 Related Work

Having presented the design, analysis, and evaluation of the TSAR system, we next review prior work in comparison. In particular we survey work in the areas of storage and indexing for sensor networks; we note that while our work addresses both problems jointly, most prior work has considered the two issues in isolation.

Beginning with basic storage systems for wireless sensor network platforms, we see file systems such as Matchbox [38] and ELF [22]. Matchbox is a very minimal file system which is part of the TinyOS distribution; ELF is a more fully-featured log-structured file system. MicroHash [94] provides a storage and indexing structure for wireless sensor nodes which is more closely tailored to the needs of sensor network applications. CAPSULE [57] goes beyond this to offer composable higher-level storage abstractions which

	Range Query Support	Interval Representation	Re-balancing	Resilience	Small Networks	Large Networks
DHT, GHT	no	no	no	yes	good	good
Local index, flood query	yes	yes	no	yes	good	bad
P-tree, RP* (dist. B-Trees)	yes	possible	yes	no	good	good
DIMS	yes	no	yes	yes	yes	yes
Interval Skipgraph	yes	yes	no	yes	good	good

Table 3.2. Comparison of Distributed Index Structures

can be adapted to different sensing applications. Neither of these systems, however, addresses the issues of coordinating data access and querying across multiple sensors.

Examining distributed storage systems, multi-resolution storage [30] is designed for in-network storage and search in systems where the volume of data is large in comparison to storage resources; it does not address the issues which arise when we are able to place large amounts of storage at sensors. Directed Diffusion [41] can be compared with TSAR, as well; there, queries must be spatially scoped if we are to avoid flooding the network, as unlike TSAR there is no global index structure.

We compare TSAR and its Interval Skip Graph index in more detail to other index-based sensor network storage systems in Table 3.2.

The hash-based systems, DHT [70] and GHT [71], lack the ability to perform range queries and are thus not well-suited to indexing spatio-temporal data. Indexing locally using an appropriate single-node structure and then flooding queries to all proxies is a competitive alternative for small networks; for large networks the linear dependence on the number of proxies becomes an issue. Two distributed B-Trees were examined - P-Trees [19] and RP* [53]. Each of these supports range queries, and in theory could be modified to support indexing of intervals; however, they both require complex re-balancing, and do not provide the resilience characteristics of the other structures. DIMS [52] provides the ability to perform spatio-temporal range queries, and has the necessary resilience to failures; however, it cannot be used index intervals, which are used by the TSAR data summarization algorithm.

In addition to storage and indexing techniques specific to sensor networks, many distributed, peer-to-peer and spatio-temporal index structures are relevant to our work. DHTs [70] can be used for indexing events based on their type, quad-tree variants such as R-trees [35] can be used for optimizing spatial searches, and K-D trees [9] can be used for multi-attribute search. While this paper focuses on building an ordered index structure for range queries, we will explore the use of other index structures for alternate queries over sensor data.

3.8 Conclusion

In this chapter, we argued that existing sensor storage systems are designed primarily for flat hierarchies of homogeneous sensor nodes and do not fully exploit the multi-tier nature of emerging sensor networks. We presented the design of *TSAR*, a fundamentally different storage architecture that envisions separation of data from metadata by employing local storage at the sensors and distributed indexing at the proxies. At the proxy tier, *TSAR* employs a novel multi-resolution ordered distributed index structure, the Sparse Interval Skip Graph, for efficiently supporting spatio-temporal and range queries. At the sensor tier, *TSAR* supports energy-aware adaptive summarization that can trade-off the energy cost of transmitting metadata to the proxies against the overhead of false hits resulting from querying a coarser resolution index structure. We implemented *TSAR* in a two-tier sensor testbed comprising Stargate-based proxies and Mote-based sensors. Our experimental evaluation of *TSAR* demonstrated the benefits and feasibility of employing our energy-efficient low-latency distributed storage architecture in multi-tier sensor networks.

CHAPTER 4

DATA CENTER PERFORMANCE ANALYSIS

Large-scale data centers generate large volumes of performance and operational data, in the form of monitored variables, application event logs, and other instrumentation and system output. Collection and processing of this raw data is complicated by several factors, including the unstructured nature of the raw log data. In addition, the information of value to the user is often not concentrated in a small number of records, but is instead found in the relationships of many different records.

One method of distilling this information and returning it to the user is by constructing mathematical models with which the user may predict or examine system performance. In this chapter we examine mechanisms for doing so, and for overcoming some of the obstacles which arise in the process.

4.1 Introduction

A *data center* is a large collection of computers managed by a single organization. These computers implement *applications* which are accessed remotely by *users*, over a corporate network or the Internet. The data center will have resources for computation (i.e. the servers themselves), storage, local communication (typically a high-speed LAN), and remote communication with the application end-users. Organizationally, such an installation may be a managed hosting provider, in which the data center, applications, and users all belong to different organizations; an enterprise data center, where all three are part of the same organization, or some point on the spectrum in between.

4.1.1 Background and Motivation

Management of such a data center can be a difficult task. A large data center may consist of hundreds or even thousands of servers, running as many as several hundred applications. The components of these applications—web servers, databases, application servers, etc.—interact with each another in a multitude of ways. Hardware and software components in a data center evolve through incremental modifications, resulting in a system that becomes more and more complex and difficult to manage over time. At the same time, effective management of these systems is often crucial. Data center applications are often subject to service level agreements (SLAs), requiring that the application meet a specified level of performance: for example, that the 95th percentile response time for a particular web page not exceed 2 seconds. Managing these systems to meet such SLAs is difficult, however, as they have become too complex to be comprehended by a single human.

We propose a new approach for conquering this complexity, using statistical methods from data mining and machine learning. These methods create predictive models which capture interactions within a system, allowing the user to relate input (i.e. user) behavior to interactions and resource usage. Data from existing sources (log files, resource utilization) is collected and used for model training, so that models can be created “on the fly” on a running system. From this training data we then infer models which relate events or input at different tiers of a data center application to resource usage at that tier, and to corresponding requests sent to tiers further within the data center. By composing these models, we are able to examine relationships across multiple tiers of an application, and to isolate these relationships from the effects of other applications which may share the same components.

The nature of data center applications, however, makes this analysis and modeling difficult. To create models of inputs and responses, we must classify them; yet they are typically unique to each application. Classifying inputs by hand will not scale in practice, due to the huge number of unique applications and their rate of change. Instead, if we are to use mod-

eling as a tool in data center management, we must *automatically* learn not only system responses but input classifications themselves.

The benefits of an automated modeling method are several. It relieves humans from the tedious task of tracking and modeling complex application dependencies in large systems. The models created may be used for the higher-level task of analyzing and optimizing data center behavior itself. Finally, automated modeling can keep these models up-to-date as the system changes, by periodic testing and repetition of the learning process.

4.1.2 Contributions

In this chapter we present *Modellus*, a system which monitors data center servers and applications, and applies statistical machine learning techniques—in particular variants of step-wise regression—to infer models of application behavior. We focus on models relating workload at an application tier to resource usage at that tier, and to workload at the next tier. These models may be used for prediction of system response to changing workloads or resource availability, or for troubleshooting, by giving insight into application behavior and performance. Although prior work has applied similar techniques to these problems, the key contribution in *Modellus* is an automated mechanism for selecting model *features*—i.e. classifications of input requests. By doing this we are able to create models for applications under typical data center conditions of complex, changing applications and hands-off management, rather than static applications analyzed by experts under research lab conditions.

In addition to the key feature extraction and modeling features, *Modellus* also incorporates mechanisms to ensure scalability under large data center conditions. Model accuracy is measured and tracked, and this information is used to bound cascading prediction errors due to the composition of multiple models. A fast, distributed model testing algorithm imposes negligible load on the servers themselves, but allows quick response to changing conditions without burdening the central analysis system. Finally, back-off heuristics iden-

tify fundamentally random or unpredictable applications, both to avoid making predictions based on poor models and to avoid expending excessive time attempting to model them.

We have implemented a prototype of Modellus using a Python framework connecting C++ and Numeric Python modules, consisting of both a nucleus running at the monitored systems and a control plane for model generation and testing. We conducted detailed experiments on a prototype Linux data center running a mix of realistic applications; our results show that in many cases we are able to predict server utilization within 5% or less based on measurements of the input to either that server or upstream servers. In addition, further experiments demonstrate the utility of our modeling techniques in predicting application response to future traffic loads and patterns for use in capacity planning.

The rest of this chapter is structured as follows. In Section 4.2 we provide an overview of data center modeling methods we employ in Modellus, and some of the challenges they present. We then discuss the particular techniques and algorithms used in Modellus. Section 4.5 covers the implementation, and Section 4.6 presents experimental results. In addition, Section 4.7 presents a case study of using Modellus to predict changes in system utilization due to shifts in user request mix. Finally, we present related work in Section 4.8 and our conclusions in Section 4.9.

4.2 Data Center Modeling Methods

Mathematical models such as the ones we construct in Modellus allow complex systems to be better understood and their behavior predicted, within limits. In this section we focus on the process of creating such mathematical models from data center monitoring information.

4.2.1 Problem Formulation

Consider a data center consisting of a large collection of computers, running a variety of applications and accessed remotely via the Internet or an enterprise network. The data

center will have resources for computation (i.e. the servers themselves), storage, local communication (typically a high-speed LAN), and remote communication with end-users. Software and hardware components within the data center will interact with each other to implement useful services or *applications*.

As an example, a web-based student course registration system might be implemented on a J2EE server, passing requests to a legacy client-server backend and then to an enterprise database. Some of these steps or tiers may be shared between applications; for instance our example backend database is likely to be shared with other applications (e.g. tuition billing) which need access to course registration data. In addition, in many cases physical resources may be shared between different components and applications, by either direct co-location or through the use of virtual machines.

For our analysis we can characterize these applications at various tiers in the data center by their requests and the responses to them.¹ In addition, we are interested in both the computational and I/O load incurred by an application when it handles a request, as well as any additional requests it may make to other-tier components in processing the request. (e.g. database queries issued while responding to a request for a dynamic web page) We note that a single component may receive inter-component requests from multiple sources, and may generate requests to several components in turn. Thus the example database server receives queries from multiple applications, while a single application in turn may make requests to multiple databases or other components.

In order to construct models of the operation of these data center applications, we require data on the operations performed as well as their impact. We obtain request or event information from application logs, which typically provide event timestamps, client identity, and information identifying the specific request. Resource usage measurements are

¹This restricts us to request/response applications, which encompasses many but not all data center applications.

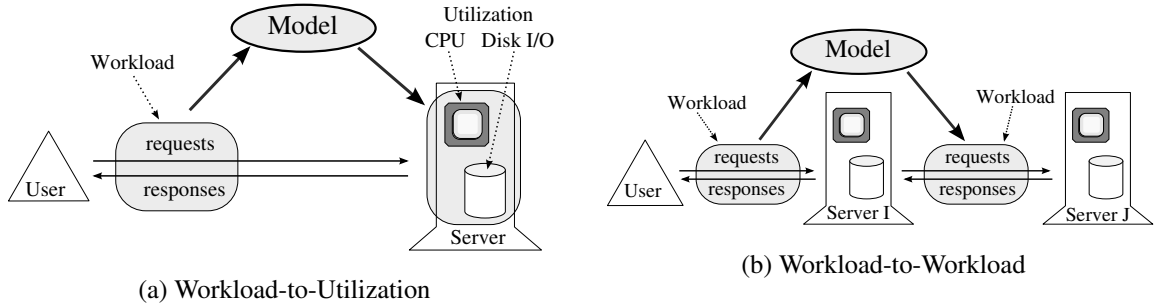


Figure 4.1. Application models

gathered from the server OS, and primarily consist of CPU usage and disk operations over some uniform sampling interval.

Problem Formulation: The automated modeling problem may be formulated as so. In a system as described, given the request and resource information provided, we wish to automatically derive the following models:²

(1) A *workload-to-utilization model*, which models the resource usage of an application as a function of its incoming workload. For instance, the CPU utilization and disk I/O operations due to an application μ_{cpu}, μ_{iop} can be captured as a function of its workload λ :

$$\mu_{cpu} = f_{cpu}(\lambda), \mu_{iop} = f_{iop}(\lambda)$$

(2) A *workload-to-workload model*, which models the outgoing workload of an application as a function of its incoming workload. Since the outgoing workload of an application becomes the incoming workload of one or more downstream components, our model derives the workload at one component as a function of another: $\lambda_j = g(\lambda_i)$

We also seek techniques to compose these basic models to represent complex system systems. Such model composition should capture *transitive behavior*, where pair-wise models between applications i and j and j and k are composed to model the relationship between i and k . Further, model composition should allow pair-wise dependence to be

²Workload-to-response time models are an area for further research.

extended to n -way dependence, where an application’s workload is derived as a function of the workloads seen by all its n upstream applications.

We next present the intuition behind our basic models, followed by a discussion on constructing composite models of complex data center applications.

4.2.2 Workload-to-utilization Model

Consider an application that sees an incoming request rate of λ over some interval τ . We may model CPU utilization as a function of the aggregate arrival rate and mean service time per request:

$$\mu = \lambda \cdot s \tag{4.1}$$

where λ is the total arrival rate, s is the mean service time per request, and μ is CPU usage per unit time, or utilization. By measuring arrival rates and CPU use over time, we may estimate the service time, \hat{s} , and predict utilization as arrival rate changes.

If each request takes the same amount of time and resources, then the accuracy of this model will be unaffected by changes in either the rate or request type of incoming traffic. However, in practice this is often far from true. Requests typically fall into classes with very different service times: e.g. a web server might receive requests for small static files and computationally-intensive scripts. Equation 4.1 can only model the average service time across all request types, and if the mix of types changes, it will produce errors.

Let us suppose, that the input stream consists of k distinct classes of requests, where requests in each class have similar service times—in the example above: static files and cgi-bin scripts. Let $\lambda_1, \lambda_2, \dots, \lambda_k$ denote the observed rates for each request class, and let s_1, s_2, \dots, s_k denote the corresponding mean service time. Then the aggregate CPU utilization over the interval τ is a linear sum of the usage due to each request type:

$$\mu = \lambda_1 \cdot s_1 + \lambda_2 \cdot s_2 + \dots + \lambda_k \cdot s_k + \epsilon \tag{4.2}$$

where ϵ is a error term assumed random and independent.

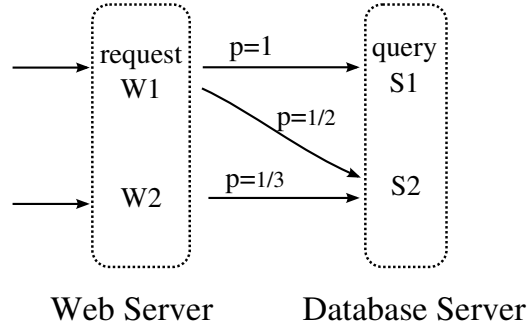


Figure 4.2. Example 2-tier request flow

If the request classes are well-chosen, then we can sample the arrival rate of each class empirically, derive the above linear model from these measurements, and use it to yield an estimate $\hat{\mu}$ of the CPU utilization due to the incoming workload λ . Thus in our example above, λ_1 and λ_2 might represent requests for small static files and scripts; s_2 would be greater than s_1 , representing the increased cost of script processing. The value of this model is that it retains its accuracy when the request mix changes. Thus if the overall arrival rate in our example remained constant, but the proportion of script requests increased, the model would account for the workload change and predict an increase in CPU load.

4.2.3 Workload-to-workload Model

We next consider two interacting components as shown in Figure 4.1(b), where incoming requests at i trigger requests to component j . For simplicity we assume that i is the source of all requests to j ; the extension to multiple upstream components is straightforward. Let there be k request classes at components i and m classes in the workload seen by j . Let $\lambda_I = \{\lambda_{i1}, \lambda_{i2}, \dots\}$ and $\lambda_J = \{\lambda_{j1}, \lambda_{j2}, \dots\}$ denote the class-specific arrival rates at the two components.

To illustrate, suppose that i is a front-end web server and j is a back-end database, and web requests at i may be grouped in classes $W1$ and $W2$. Similarly, SQL queries at the database are grouped in classes $S1$ and $S2$, as shown in Figure 4.2. Each $W1$ web request triggers an $S1$ database query, followed by an $S2$ query with probability 0.5. (e.g.

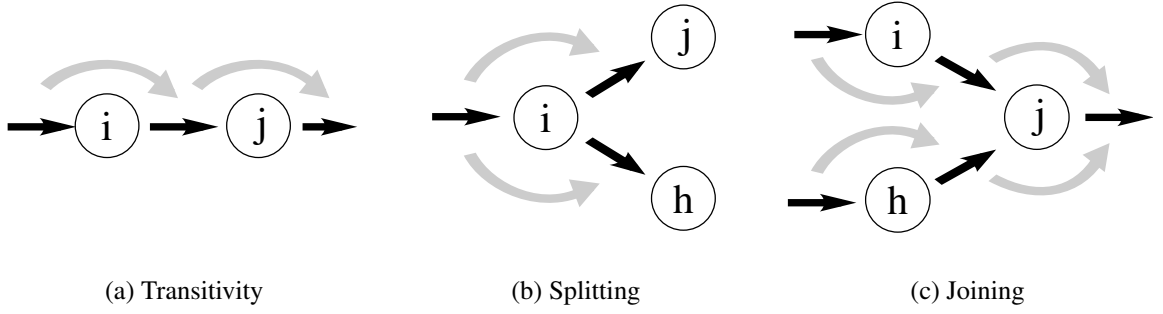


Figure 4.3. Composition of the basic models.

the second query may be issued when certain responses to the first query are received) In addition, each web request in class $W2$ triggers a type $S2$ query with probability 0.3.

We can thus completely describe the workload seen at the database in terms of the web server workload:

$$\lambda_{S1} = \lambda_{W1}; \quad \lambda_{S2} = 0.5\lambda_{W1} + 0.3\lambda_{W2} \quad (4.3)$$

More generally, each request type at component j can be represented as a weighted sum of request types at component i , where the weights denote the number of requests of this type triggered by each request class at component i :

$$\begin{aligned} \lambda_{j1} &= w_{11}\lambda_{i1} + w_{12}\lambda_{i2} + \dots + w_{1k}\lambda_{ik} + \epsilon_1 \\ \lambda_{j2} &= w_{21}\lambda_{i1} + w_{22}\lambda_{i2} + \dots + w_{2k}\lambda_{ik} + \epsilon_2 \\ \lambda_{jm} &= w_{m1}\lambda_{i1} + w_{m2}\lambda_{i2} + \dots + w_{mk}\lambda_{ik} + \epsilon_m \end{aligned} \quad (4.4)$$

where ϵ_i denotes an error term. Thus, Equation 4.4 yields the workload at system j , $\lambda_j = \{\lambda_{j1}, \lambda_{j2}, \dots\}$ as a function of the workload at system i , $\lambda_i = \{\lambda_{i1}, \lambda_{i2}, \dots\}$.

4.2.4 Model Composition

The workload-to-utilization (W2U) model yields the utilization due to an application j as a function of its workload: $\mu_j = f(\lambda_j)$; the workload-to-workload (W2U) model yields

the workload at application j as a function of the workload at application i : $\lambda_J = g(\lambda_I)$. Substituting allows us to determine the utilization at j directly as a function of the workload at i : $\mu_j = f(g(\lambda_I))$. Since f and g are both linear equations, the composite function, obtained by substituting Equation 4.4 into 4.2, is also a linear equation. This composition process is transitive: given cascaded components i , j , and k , it can yield the workload and the utilization of the downstream application k as a function of the workload at i . In a three-tier application, for instance, this lets us predict behavior at the database back-end as a function of user requests at the front-end web server.

Our discussion has implicitly assumed a linear chain topology, where each application sees requests from only one upstream component, illustrated schematically in Figure 4.3(a). This is a simplification; in a complex data center, applications may both receive requests from multiple upstream components, and in turn issues requests to more than one downstream system. Thus an employee database may see requests from multiple applications (e.g., payroll, directory), while an online retail store may make requests to both a catalog database and a payment processing system. We must therefore be able to model both: (i) “splitting” – triggering of requests to multiple downstream applications, and (ii) “merging”, where one application receives request streams from multiple others, as shown in Figure 4.3(b) and (c).

To model splits, consider an application i which makes requests of downstream applications j and h . Given the incoming request stream at i , λ_I , we consider the subset of the outgoing workload from i that is sent to j , namely λ_J . We can derive a model of the inputs at i that trigger this subset of outgoing requests using Equation 4.4: $\lambda_J = g_1(\lambda_I)$. Similarly by considering only the subset of the outgoing requests that are sent to h , we can derive a second model relating λ_H to λ_I : $\lambda_H = g_2(\lambda_I)$.

For joins, consider an application j that receives requests from upstream applications i and h . We first split the incoming request stream by source: $\lambda_J = \{\lambda_J|src = i\} + \{\lambda_J|src = h\}$. The workload contributions at j of i and h are then related to the input

workloads at the respective applications using Equation 4.4: $\{\lambda_J|src = i\} = f_1(\lambda_I)$ and $\{\lambda_J|src = h\} = f_2(\lambda_H)$, and the total workload at j is described in terms of inputs at i and h : $\lambda_J = f_1(\lambda_I) + f_2(\lambda_H)$. Since f_1 and f_2 are linear equations, the composite function, which is the summation of the two— $f_1 + f_2$ —is also linear.

By modeling these three basic interactions—cascading, splitting, and joining— we are able to compose single step workload-to-workload and workload-to-utilization models to model any arbitrary application graph. Such a composite model allows workload or utilization at each node to be calculated as a linear function of data from other points in the system.

4.3 Automated Model Generation

We next present techniques for automatically learning models of the form described above. In particular, these models require specification of the following parameters: (i) request classes for each component, (ii) arrival rates in each class, λ_i , (iii) mean service times s_i for each class i , and (iv) rates w_{ij} at which type i requests trigger type j requests. In order to apply the model we must measure λ_i , and estimate s_i and w_{ij} .

If the set of classes and mapping from requests to classes was given, then measurement of λ_i would be straightforward. In general, however, request classes for a component are not known *a priori*. Manual determination of classes is impractical, as it would require detailed knowledge of application behavior, which may change with every modification or extension. Thus, our techniques must *automatically determine an appropriate classification of requests* for each component, as part of the model generation process.

Once the request classes have been determined, we estimate the coefficients s_i and w_{ij} . Given measured arrival rates λ_i in each class i and the utilization μ within a measurement interval, Equations 4.2 and 4.4 yield a set of linear equations with unknown coefficients s_i and w_{ij} . Measurements in subsequent intervals yield additional sets of such equations;

these equations can be solved using linear regression to yield the unknown coefficients s_i and w_{ij} that minimize the error term ϵ .

A key contribution of our automated model generation is to combine determination of request classes with parameter estimation, in a single step. We do this by mechanically enumerating possible request classes, and then using statistical techniques to select the classes which are predictive of utilization or downstream workload. In essence, the process may be viewed as “mining” the observed request stream to determine features (classes) that are the best predictors of the resource usage and triggered workloads; we rely on step-wise regression—a technique also used in data mining—for our automated model generation.

In particular, for each request we first enumerate a set of possible features, primarily drawn from the captured request string itself. Each of these features implies a classification of requests, into those which have this feature and those which do not. By repeating this over all requests observed in an interval, we obtain a list of candidate classes. We also measure arrival rates within each candidate class, and resource usage over time. Step-wise regression of feature rates against utilization is then used to select only those features that are significant predictors of utilization and to estimate their weights, giving us the workload-to-utilization model.

Derivation of W2U models is an extension of this. First we create a W2U model at application j , in order to determine the significant workload features. Then we model the arrival rate of these features, again by using stepwise regression. We model each feature as a function of the input features at i ; when we are done we have a model which takes input features at i and predicts a vector of features $\hat{\lambda}_J$ at j .

4.3.1 Feature Selection

For this approach to be effective, classes with stable behavior (mean resource requirements and request generation) must exist. In addition, information in the request log must be sufficient to determine this classification. We present an intuitive argument for the exis-

tence of such classes and features, and a description of the feature enumeration techniques used in Modellus.

We first assert that this classification is possible, within certain limits: in particular, that in most cases, system responses to identical requests are similar, across a broad range of operating conditions. Consider, for example, two identical requests to a web server for the same simple dynamic page—regardless of other factors, identical requests will typically trigger the same queries to the back-end database. In triggering these queries, the requests are likely to invoke the same code paths and operations, resulting in (on average) similar resource demands.³

Assuming these request classes, we need an automated technique to derive them from application logs—to find requests which perform similar or identical operations, on similar or identical data, and group them into a class. The larger and more general the groups produced by our classification, the more useful they will be for actual model generation. At the same time, we cannot blindly try all possible groupings, as each unrelated classification tested adds a small increment of noise to our estimates and predictions.

In the cases we are interested in, e.g. HTTP, SQL, or XML-encoded requests, much or all of the information needed to determine request similarity is encoded convention or by syntax in the request itself. Thus we would expect the query `'SELECT * from cust WHERE cust.id=105'` to behave similarly to the same query with `'cust.id=210'`, while an HTTP request for a URL ending in `'images/map.gif'` is unlikely to be similar to one ending in `'browse.php?category=5'`.

Our enumeration strategy consists of extracting and listing *features* from request strings, where each feature identifies a potential candidate request class. Each enumeration strategy

³Caching will violate this linearity assumption; however, we argue that in this case behavior will fall into two domains—one dominated by caching, and the other not—and that a linear approximation is appropriate within each domain.

- 1: the entire URL:
/test/PHP/AboutMe.php?name=user5&pw=joe
- 2: each URL prefix plus extension:
/test/.php, /test/PHP/.php
/test/PHP/AboutMe.php
- 3: each URL suffix plus extension:
AboutMe.php, PHP/AboutMe.php
- 4: each query variable and argument:
/test/PHP/AboutMe.php?name=user5
/test/PHP/AboutMe.php?pw=joe
- 5: all query variables, without arguments:
/test/PHP/AboutMe.php?name=&pw=

Figure 4.4. HTTP feature enumeration algorithm

- 1: database:
TPCW
- 2: database and table(s):
TPCW:item,author
- 3: query “skeleton”:
SELECT * FROM item,author WHERE item.i_a_id=author.a_id AND i_id=?
- 4: the entire query:
SELECT * FROM item,author WHERE item.i_a_id=author.a_id AND
i_id=1217
- 5: query phrase:
WHERE item.i_a_id=author.a_id AND i_id=1217
- 6: query phrase skeleton:
WHERE item.i_a_id=author.a_id AND i_id=?

Figure 4.5. SQL Feature Enumeration

is based on the formal or informal⁴ syntax of the request and it enumerates the portions of the request which identify the class of operation, the data being operated on, and the operation itself, which are later tested for significance. We note that the feature enumeration algorithm must be manually specified for each application type, but that there are a relatively small number of such types, and once algorithm is specified it is applicable to any application sharing that request syntax.

The Modellus feature enumeration algorithm for HTTP requests is shown in Figure 4.4, with features generated from an example URL. The aim of the algorithm is to identify

⁴E.g. HTTP, where the hierarchical structure of requests, semantics of URI suffixes, and interpretation of query arguments are defined by convention rather than standard.

request elements which may identify common processing paths; thus features include file extensions and URL prefixes, and query skeletons (i.e. a query with arguments removed), each of which may identify common processing paths. In Figure 4.5 we see the feature enumeration algorithm for SQL database queries, which uses table names, database names, query skeletons, and SQL phrases (which may be entire queries in themselves) to generate a list of features. Feature enumeration is performed on all requests present in an application's log file over a measurement window, one request at a time, to generate a list of candidate features.

4.3.2 Stepwise Linear Regression

Once the enumeration algorithm generates a list of candidate features, the next step is to use training data to learn a model by choosing only those features whose coefficients s_i and w_{ij} minimize the error terms in Equations 4.2 and 4.4. In a theoretical investigation, we might be able to compose benchmarks consisting only of particular requests, and thus measure the exact system response to these particular requests. In practical systems, however, we can only observe aggregate resource usage given an input stream of requests which we do not control. Consequently, the model coefficients s_i and w_{ij} must also be determined as part of the model generation process.

One naïve approach is to use *all* candidate classes enumerated in the previous step, and to employ least squares regression on the inputs (here, arrival rates within each candidate class) and outputs (utilization or downstream request rates) to determine a set of coefficients that best fit the training data. However, this will generate spurious features with no relationship to the behavior being modeled; if included in the model they will degrade its accuracy, in a phenomena known as *over-fitting* in the machine learning literature. In particular, some will be chosen due to random correlation with the measured data, and will contribute noise to future predictions.

```

1: Let  $\Lambda_{model} = \{\}$ ,  $\Lambda_{remaining} = \{\lambda_1, \lambda_1, \dots\}$ ,  $e = \mu$ 
2: while  $\Lambda_{remaining} \neq \phi$  do
3:   for  $\lambda_i$  in  $\Lambda_{model}$  do
4:      $e(\lambda_i) \leftarrow error(\Lambda_{model} - \lambda_i)$ 
5:   end for
6:    $\lambda_i \leftarrow \min_i e(\lambda_i)$ 
7:   if F-TEST( $e(\lambda_i)$ ) = not significant then
8:      $\Lambda_{model} \leftarrow \Lambda_{model} - \lambda_i$ 
9:   end if
10:  for  $\lambda_i$  in  $\Lambda_{remaining}$  do
11:     $e(\lambda_i) \leftarrow error(\Lambda_{model} + \lambda_i)$ 
12:  end for
13:   $\lambda_i \leftarrow \min_i e(\lambda_i)$ 
14:  if F-TEST( $e(\lambda_i)$ ) = significant then
15:     $\Lambda_{model} \leftarrow \Lambda_{model} \cup \lambda_i$ 
16:  else
17:    return  $\Lambda_{model}$ 
18:  end if
19: end while

```

Figure 4.6. Stepwise Linear Regression Algorithm

This results in a data mining problem: out of a large number of candidate classes and measurements of arrival rates within each class, determining those which are predictive of the output of interest, and discarding the remainder. In statistics this is termed a *variable selection* problem [25], and may be solved by various techniques which in effect determine the odds of whether each input variable influences the output or not. Of these methods we use Stepwise Linear Regression, [54, 33] due in part to its scalability, along with a modern extension—the Foster and George’s risk inflation criteria [29].

A simplified version of this algorithm is shown in Figure 4.6, with input variables λ_i and output variable μ . We begin with an empty model; as this predicts nothing, its error is exactly μ . In the first step, the variable which explains the largest fraction of μ is added to the model. At each successive step the variable explaining the largest fraction of the remaining error is chosen; in addition, a check is made to see if any variables in the model have been made redundant by ones added at a later step. The process completes when no remaining variable explains a statistically significant fraction of the response.

4.4 Accuracy, Efficiency and Stability

We have presented techniques for automatic inference of our basic models and their composition. However, several practical issues arise when implementing these techniques into a system:

- *Workload changes:* Although our goal is to derive models which are resilient to shifts in workload composition as well as volume, some workload changes will cause model accuracy to decrease — for instance, the workload may become dominated by requests not seen in the training data. When this occurs, prediction errors will persist until the relevant models are re-trained.
- *Effective model validation and re-training:* In order to quickly detect shifts in system behavior which may invalidate an existing model, without un-necessarily retraining other models which remain accurate, it is desirable to periodically test each model for validity. The lower the overhead of this testing, the more frequently it may be done and thus the quicker the models may adjust to shifts in behavior.
- *Cascading errors:* Models may be composed to make predictions across multiple tiers in a system; however, uncertainty in the prediction increases in doing so. Methods are needed to estimate this uncertainty, so as to avoid making unreliable predictions.
- *Stability:* Some systems will be difficult to predict with significant accuracy. Rather than spending resources repeatedly deriving models of limited utility, we should detect these systems and limit the resources expended on them.

In the following section we discuss these issues and the mechanisms in Modellus which address them.

4.4.1 Model Validation and Adaptation

A trained model makes predictions by extrapolating from its training data, and the accuracy of these predictions will degrade in the face of behavior not found in the training data. Another source of errors can occur if the system response changes from that recorded during the training window; for instance, a server might become slower or faster due to failures or upgrades.

In order to maintain model accuracy, we must retrain models when this degradation occurs. Rather than always re-learning models, we instead test predictions against actual measured data; if accuracy declines below a threshold, then new data is used to re-learn the model.

In particular, we sample arrival rates in each class ($\hat{\lambda}_i$) and measure resource utilization $\hat{\mu}$. Given the model coefficients s_i and w_{ij} , we substitute $\hat{\lambda}_i$ and $\hat{\mu}$ into Equations 4.2 and 4.4, yielding the prediction error ϵ . If this exceeds a threshold ϵ_T in k out of n consecutive tests, the model is flagged for re-learning.

A simple approach is to test all models at a central node; data from each system is collected over a testing window and verified. Such continuous testing of tens or hundreds of models could be computationally expensive. We instead propose a fast, distributing model testing algorithm based on the observation that *although model derivation is expensive in both computation and memory, model checking is cheap*. Hence model validation can be distributed to the systems being monitored themselves, allowing nearly continuous checking.

In this approach, the model—the request classes and coefficients—is provided to each server or node and can be tested locally. To test workload-to-usage models, a node samples arrival rates and usages over a short window, and compares the usage predictions against observations. Workload-to-workload models are tested similarly, except that communication with the downstream node is required to obtain observed data. If the local tests fail in k out of n consecutive instances, then a full test is triggered at the central node; full testing in-

volves a larger test window and computation of confidence intervals of the prediction error. Distributed testing over short windows is fast and imposes minimal overheads on server applications; due to this low overhead, tests can be frequent to detect failures quickly.

4.4.2 Limiting Cascading Errors

In the absence of errors, we could monitor only the external inputs to a system, and then predict all internal behavior from models. In practice, models have uncertainties and errors, which grow as multiple models are composed.

Since our models are linear, errors also grow linearly with model composition. This may be seen by substituting Equation 4.4 into Equation 4.2, yielding a composite model with error term $\sum s_i \cdot \epsilon_i + \epsilon$, a linear combination of individual errors. Similarly, a “join” again yields an error term summing individual errors.

Given this linear error growth, there is a limit on the number of models that may be composed before the total error exceeds any particular threshold. Hence, we can no longer predict all internal workloads and resource usages solely by measuring external inputs. In order to scale our techniques to arbitrary system sizes, we must measure additional inputs inside the system, and use these measurements to drive further downstream predictions.

To illustrate how this may be done, consider a linear cascade of dependencies, and suppose ϵ_{max} is the maximum tolerable prediction error. The first node in this chain sees an external workload that is known and measured; we can compute the expected prediction error at successive nodes in the chain until the error exceeds ϵ_{max} . Since further predictions will exceed this threshold, a new measurement point must be inserted here to measure, rather than predict, its workload; these measurements drive predictions at subsequent downstream nodes.

This process may be repeated for the remaining nodes in the chain, yielding a set of measurement points which are sufficient to predict responses at all other nodes in the chain. This technique easily extends to an arbitrary graph; we begin by measuring all external

inputs and traverse the graph in a breadth-first manner, computing the expected error at each node. A measurement point is inserted if a node's error exceeds ϵ_{max} , and these measurements are then used to drive downstream predictions.

4.4.3 Stability Considerations

Under certain conditions, however, it will not be possible to derive a useful model for predicting future behavior. If the system behavior is dominated by truly random factors, for instance, model-based predictions will be inaccurate. A similar effect will be seen in cases where there is insufficient information available. Even if the system response is deterministically related to some attribute of the input, as described below, the log data may not provide that attribute. In this case, models learned from random data will result in random predictions.

In order to avoid spending a large fraction of system resources on the creation of useless models, Modellus incorporates backoff heuristics to detect applications which fail model validation more than k times within a period T . (e.g. 2 times within the last hour) These “mis-behaving” applications are not modeled, and are only occasionally examined to see whether their behavior has changed and modeling should be attempted again.

4.5 From Theory to Practice: Modellus Implementation

Our Modellus system implements the statistical and learning methods described in the previous sections. Figure 4.7 depicts the Modellus architecture. As shown, Modellus implements a *nucleus* on each node to monitor the workload and resource usage and for distributed model testing. The Modellus *control plane* resides on one or more dedicated nodes and comprises (i) a *Modeling and Validation Engine (MVE)* that implements the core numerical computations for model derivation and testing and may be distributed across multiple nodes for scalability, and (ii) a *Monitoring and Scheduling Engine (MSE)* which coordinates the gathering of monitoring data and scheduling of model generation

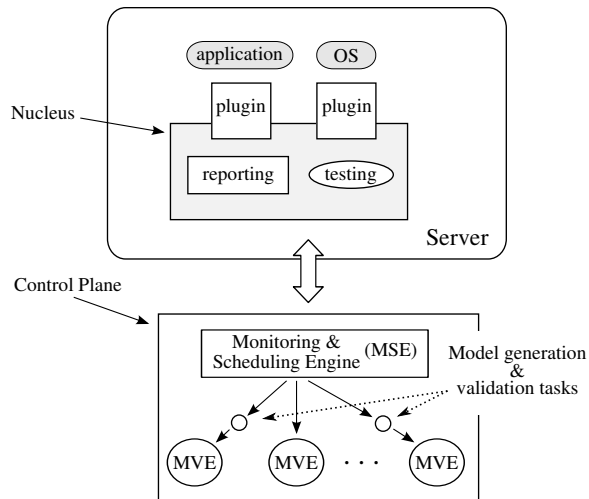


Figure 4.7. Modellus components. The *nucleus* is deployed on target systems, while the centralized *control plane* consists of the *modeling and validation engine* and *Monitoring and Scheduling Engine*.

and validation tasks when needed. The Modellus control plane also exposes a front-end that allows the derived models to be applied to data center analysis tasks; the current front-end is rudimentary and exports derived models and sampled statistics to a Matlab engine for interactive analysis.

The Modellus nucleus and control plane are implemented in a combination of C++, Python, and Numerical Python [6], providing an efficient yet dynamically extensible system. The remainder of this section discusses our implementation of these components in detail.

4.5.1 Modellus Nucleus

The nucleus is deployed on each target system, and is responsible for both data collection and simpler data processing tasks in the Modellus system. It monitors resource usage, tracks application events, and translates these events into rates. It reports usages and rates to the control plane when requested, and can also test a control plane-provided model against these usages and rates to provide local testing. A simple HTTP-based inter-

face is provided to the control plane, with commands falling into the following groups: (i) monitoring configuration, (ii) data retrieval, and (iii) local model validation.

Monitoring: The nucleus performs *adaptive monitoring* under the direction of the control plane—it is instructed which variables to sample and at what rate. It implements a uniform naming model for data sources, and an extensible plugin architecture allowing support for new applications to be easily implemented.

Resource usage is monitored via standard OS interfaces, and collected as counts or utilizations over fixed measurement intervals. Event monitoring is performed by plugins which process event streams (i.e. logs) received from applications. These plugins process logs in real time and generate a stream of request arrivals; class-specific arrival rates are then measured by mapping each event using application-specific feature enumeration rules and model-specified classes.

The Modellus nucleus is designed to be deployed on production servers, and thus must require minimal resources. By representing feature strings by hash values, we are able to implement feature enumeration and rate monitoring with minimal overhead, as shown experimentally in Section 4.6.5. We have implemented plugins for HTTP and SQL, with particular adaptations for Apache, MySQL, Tomcat, and XML-based web services.

Data retrieval: A monitoring agent such as the Modellus nucleus may either report data asynchronously (*push*), or buffer it for the receiver to retrieve (*pull*). In Modellus data is buffered for retrieval, with appropriate limits on buffer size if data is not retrieved in a timely fashion. Data is serialized using Python’s *pickle* framework, and then compressed to reduce demand on both bandwidth and buffering at the monitored system.

Validation and reporting: The nucleus receives model validation requests from the control plane, specifying input classes, model coefficients, output parameter, and error thresholds. It periodically measures inputs, predicts outputs, and calculates the error; if out of bounds k out of n times, the control plane is notified. Testing of workload-to-

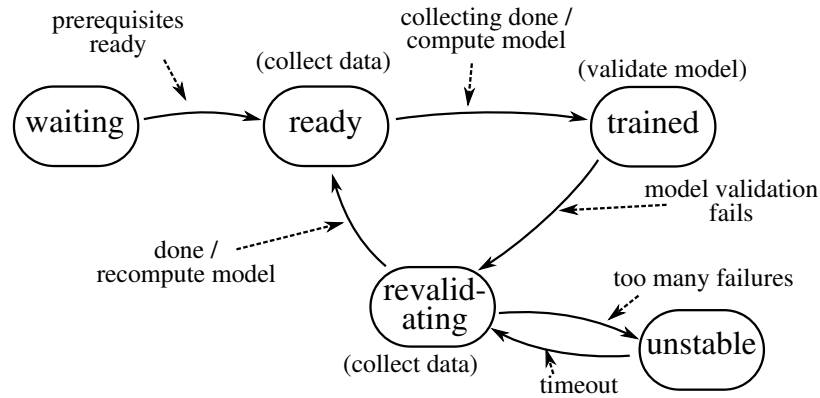


Figure 4.8. Model training states

workload models is similar, except that data from two systems (upstream and downstream) is required; the systems share this information without control plane involvement.

4.5.2 Monitoring and Scheduling Engine

The main goal of the Modellus control plane is to generate up-to-date models and maintain confidence in them by testing. Towards this end, the monitoring and scheduling engine (MSE) is responsible for (i) initiating data collection from the nuclei for model testing or generation, and (ii) scheduling testing or model re-generation tasks on the modeling and validation engines (MVEs).

The monitoring engine issues data collection requests to remote nuclei, requesting sampled rates for each request class when testing models, and the entire event stream for model generation. For workload-to-workload models, multiple nuclei are involved in order to gather upstream and downstream information. Once data collection is initiated, the monitoring engine periodically polls for monitored data, and disables data collection when a full training or testing window has been received.

The control plane has access to a list of workload-to-utilization and workload-to-workload models to be inferred and maintained; this list may be provided by configuration or discovery. These models pass through a number of states, which may be seen in Figure 4.8: *waiting* for prerequisites, *ready* to train, *trained*, *re-validating*, and *unstable*. Each

W2U model begins in the *waiting* state, with the downstream W2U model as a prerequisite, as the feature list from this W2U model is needed to infer the W2U model. Each W2U model begins directly in the *ready* state. The scheduler selects models from the ready pool and schedules training data collection; when this is complete, the model parameters may be calculated. Once parameters have been calculated, the model enters the *trained* state; if the model is a prerequisite for another, the waiting model is notified and enters the *ready* state.

Model validation as described above is performed in the *trained* state, and if at any point the model fails, it enters *revalidating* state, and training data collection begins. Too many validation failures within an interval cause a model to enter the *unstable* state, and training ceases, while from time to time the scheduler selects a model in the unstable state and attempts to model it again. Finally, the scheduler is responsible for distributing computation load within the MVE, by assigning computation tasks to appropriate systems.

4.5.3 Modeling and Validation Engine

The modeling and validation engine (MVE) is responsible for the numeric computations at the core of the Modellus system. Since this task is computationally demanding, a dedicated system or cluster is used, avoiding overhead on the data center servers themselves. By implementing the MVE on multiple systems and testing and/or generating multiple models in parallel, Modellus will scale to large data centers, which may experience multiple concurrent model failures or high testing load.

The following functions are implemented in the MVE:

Model generation: W2U models are derived directly; W2U models are derived using request classes computed for the downstream node's W2U model. In each case step-wise regression is used to derive coefficients relating input variables (feature rates) to output resource utilization (W2U models) or feature rates (W2U models).

Model validation: Full testing of the model at the control plane is similar but more sophisticated than the fast testing implemented at the nucleus. To test an W2U model, the sampled arrival rates within each class and measured utilization are substituted into Equation 4.2 to compute the prediction error. Given a series of prediction errors over successive measurement intervals in a test window, we compute the 95% one-sided confidence interval for the mean error. If the confidence bound exceeds the tolerance threshold, the model is discarded.

The procedure for testing an W2U model is similar. The output feature rates are estimated and compared with measured rates to determine prediction error and a confidence bound; if the bound exceeds a threshold, again the model is invalidated. Since absolute values of the different output rates in a W2U model may vary widely, we normalize the error values before performing this test, by using the downstream model coefficients as weights, allowing us to calculate a scaled error magnitude.

4.6 Results

In this section we present experiments examining various performance aspects of the proposed methods and system. To test feature-based regression modeling, we perform modeling and prediction on multiple test scenarios, and compare measured results with predictions to determine accuracy. Additional experiments examine errors under shifting load conditions and for multiple stages of prediction. Finally, we present measurements and benchmarks of the system implementation, in order to determine the overhead which may be placed on monitoring systems and the scaling limits of the rest of the system.

4.6.1 Experimental Setup

The purpose of the Modellus system is to model and predict performance of real-world applications, and it was thus tested on results from a realistic data center testbed and applications. A brief synopsis of the hardware and software specifications of this testbed is

CPU	1 x Pentium 4, 2.4 GHz, 512KB cache
Disk	ATA 100, 2MB cache, 7200 RPM
Memory	1, 1.2, or 2GB
OS	CentOS 4.4 (Linux kernel 2.6.9-42)
Servers	Apache 2.0.52 MySQL 5.1.11 (cluster), 4.1.20 (single)
Applications	Tomcat 5.0.30, Sun Java 1.5.0 RUBiS, TPC-W, OFBiz

Table 4.1. Data Center Testbed and Applications

given in Table 4.1, and the configuration of the systems illustrated in Figure 4.9. Three web applications were implemented: TPC-W [80, 12], an e-commerce benchmark, RUBiS [13], a simulated auction site, and Apache Open For Business (OFBiz) [63], an ERP (Enterprise Resource Planning) system in commercial use. TPC-W and OFBiz are implemented as 3-tier Java servlet-based applications, consisting of a front-end server (Apache) handling static content, a servlet container (Tomcat), and a back-end database (MySQL). RUBiS (as tested) is a 2-tier LAMP⁵ application; application logic written in PHP runs in an Apache front-end server, while data is stored in a MySQL database.

Both RUBiS and TPC-W have associated workload generators which simulate varying numbers of clients; the number of clients as well as their activity mix and think times were varied over time to generate non-stationary workloads. This is illustrated in Figure 4.10, where the frequency of several key requests is plotted as the traffic mix is varied during the course of a test run. A load generator for OFBiz was created using JWebUnit [44], which simulated multiple users performing shopping tasks from browsing through checkout and payment information entry.

Apache, Tomcat, and MySQL were configured to generate request logs, and system resource usage was sampled using the `sadc(8)` utility with a 1-second collection interval. Traces were collected and prediction was performed off-line, in order to allow re-use of the

⁵Linux/Apache/MySQL/PHP

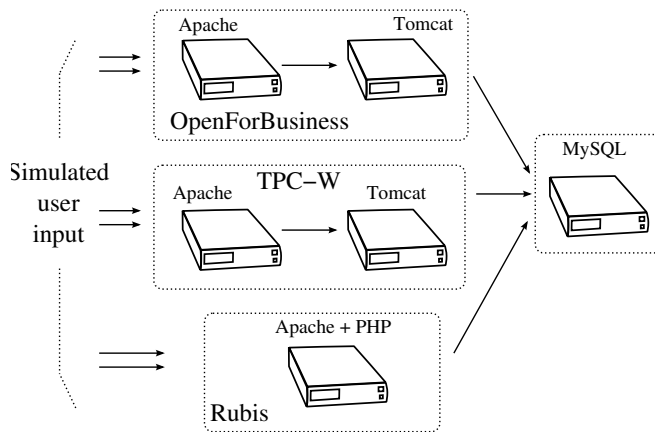


Figure 4.9. Data Center testbed. Testbed consists of a single back-end database cluster shared by three applications: OpenForBusiness, RUBiS and TPC-W.

same data for validation. Cross-validation was used for measurements of prediction error: each trace was divided into training windows of a particular length (e.g. 30 minute), and a model was constructed for each window. Each models was then used to predict each data point outside of the window on which it was trained. Deviations between predicted and actual values were then measured and statistics computed.

4.6.2 Model Generation Accuracy

To test W2U model accuracy, we tested OFBiz, TPC-W, and RUBiS configurations both alone and concurrently sharing the backend database. Using traces from these tests we compute models over 30-minute training windows, and then use these models to predict utilization $\hat{\mu}$ for 30-second intervals, using cross-validation as described above. We report the root mean square (RMS) error of prediction, and the 90th percentile absolute error ($|\mu - \hat{\mu}|$). For comparison we also show the standard deviation of the measured data itself, $\sigma(y)$.

In Figure 4.11 we see results from these tests. Both RMS and 90th percentile prediction error are shown for each server except the OFBiz Apache front end, which was too lightly loaded (< 3%) for accurate prediction. In addition we plot the standard deviation of the

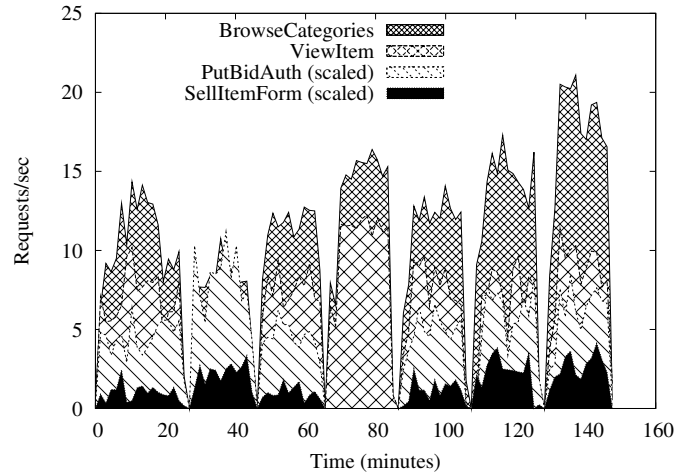


Figure 4.10. Request breakdown over time during a single test run. Arrival rates are shown for two shopping/browsing and two bidding/buying request URLs during the course of a test run.

variable being predicted (CPU utilization), in order to indicate the degree of error reduction provided by the model. In each case we are able to predict CPU utilization to a high degree of accuracy—less than 5% except for the TPC-W Tomcat server, and in all cases a significant reduction relative to the variance of the data being predicted.

We examine the distribution of prediction errors more closely in Figures 4.13 and 4.14, using the OFBiz application. For each data point predicted by each model we calculate the prediction error ($|\hat{y} - y|$), and display a cumulative histogram or CDF of these errors. From these graphs we see that about 90% of the Apache data points are predicted within 2.5%, and 90% of the MySQL data points within 5%.

In addition, in this figure we compare the performance of modeling and predicting based on workload features vs. predictions made from the aggregate request rate alone. Here we see that CPU on the OFBiz Tomcat server was predicted about twice as accurately using feature-based prediction, while the difference between naïve and feature-based prediction on the MySQL server was even greater.

To investigate the effect of the training window length, we plot a learning curve of error vs. window length in Figure 4.15. The RMS error is shown along with 90th and 95th

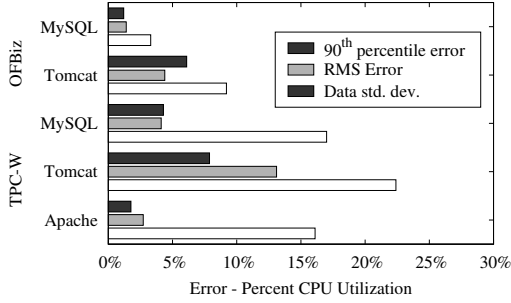


Figure 4.11. Prediction performance for TPC-W and OpenForBusiness sharing the same back-end database

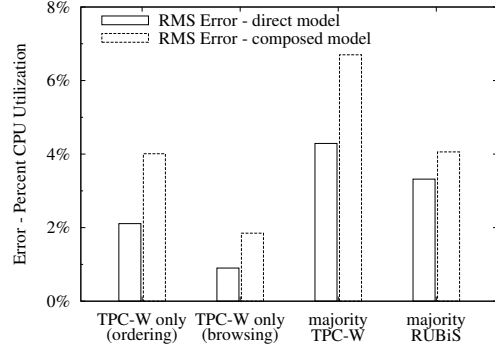


Figure 4.12. Model composition - errors in predicting MySQL server utilization from tier 1 (HTTP) input features

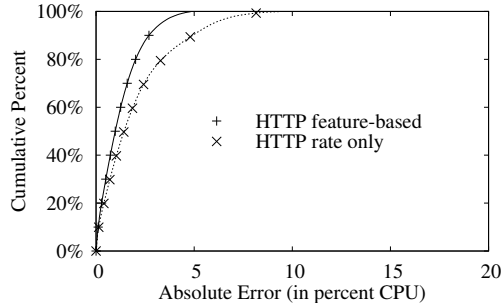


Figure 4.13. Error CDF when predicting RUBiS server utilization from HTTP traffic

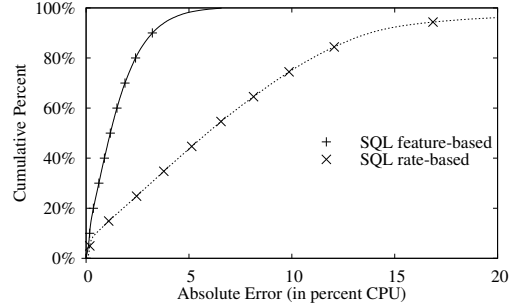


Figure 4.14. Error CDF when predicting MySQL server utilization from query traffic. Upper line is feature-based regression; lower line is naïve aggregate rate-based.

percentile errors; all are seen to drop steeply and then flatten out at somewhere between 10 and 20 minutes. These lines, however, give averages of the corresponding values across 30 test runs—i.e. the errors that may be expected from a typical model. However, we are also interested in the chance that training will generate a bad model, giving errors significantly worse than typical. To examine this, the RMS error line in Figure 4.15 is plotted with bars indicating the model-to-model standard deviation of the RMS error. Here we see that the variation across models in RMS prediction error continues to decrease until we reach a window of 25 minutes or more, well after the error curve itself has flattened out.

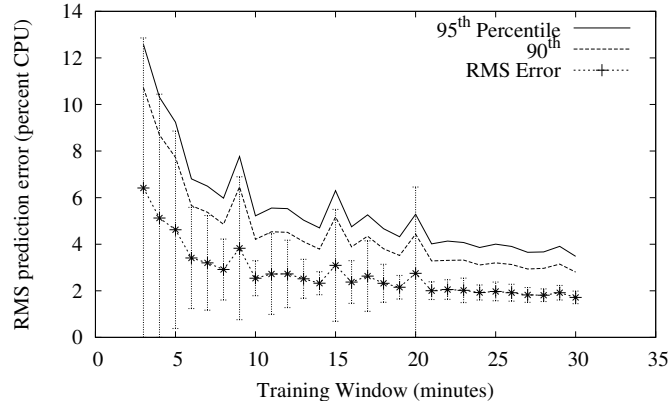


Figure 4.15. Learning curve for predicting RUBiS server utilization from HTTP features. Performance is shown vs. the length of the training window. We plot the 95th percentile, 90th percentile, and RMS prediction errors, as well as bars for the standard deviation (across models) of the RMS error.

4.6.3 Model Composition

The results presented above examine performance of single workload-to-utilization (W2U) models. We next examine prediction performance when composing workload-to-workload (W2U) and W2U models. We show results from the multi-tier experiments described above, but focus on cross-tier modeling and prediction.

As described earlier, the composition of two models is done in two steps. First, we train a W2U model for the downstream system (e.g. the database server) and its inputs. Next, we take the list of significant features identified in this model, and for each feature we train a separate upstream model to predict it. For prediction, the W2U model is used to predict input features to the W2U model, yielding the final result. Prediction when multiple systems share a single back-end resource is similar, except that the outputs of the two W2U models must be summed before input to the W2U model.

In Figure 4.12 we see the performance of this strategy in comparison with prediction directly from the inputs to the back-end system. The increase in error is seen to be modest, even when multiple applications are sharing the back-end.

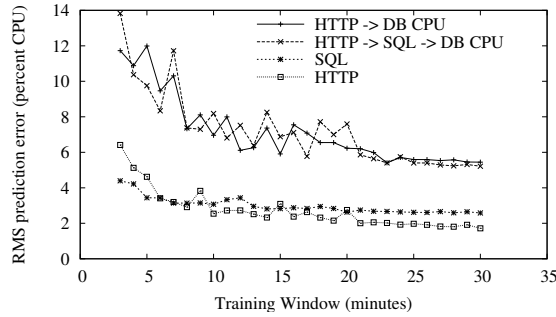


Figure 4.16. Learning curves for composed and direct prediction. Results are for RUBiS only, for a single workload-to-utilization model trained on HTTP features vs. database server utilization.

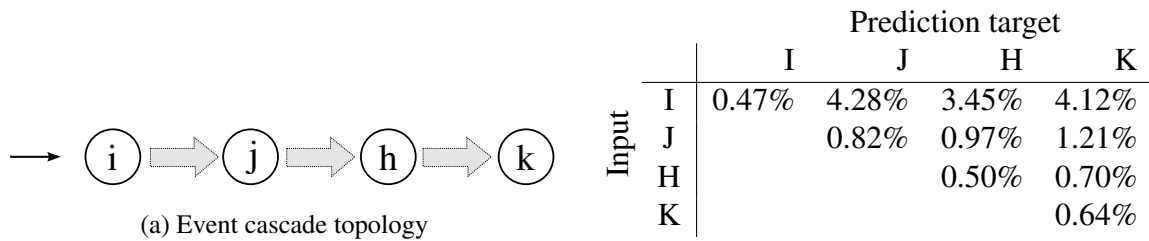


Figure 4.17. Web services - cascade errors. Table entries at (x,y) give the absolute RMS error of prediction for utilization at y given inputs at x .

In Figure 4.16 we compare learning curves for W2U models of the RUBiS-only experiment with the composed model for the RUBiS and MySQL servers, given HTTP input features; we note that the learning time necessary is comparable. In addition we validate our model composition approach by comparing its results to those of a model trained on HTTP inputs to the RUBiS server vs. CPU utilization on the MySQL server.

4.6.4 Cascading Errors

We measured prediction performance of an emulated web services application in order to investigate the relationship between model composition and errors. Three separate topologies were measured, corresponding to the model operations in Figure 4.3—cascade,

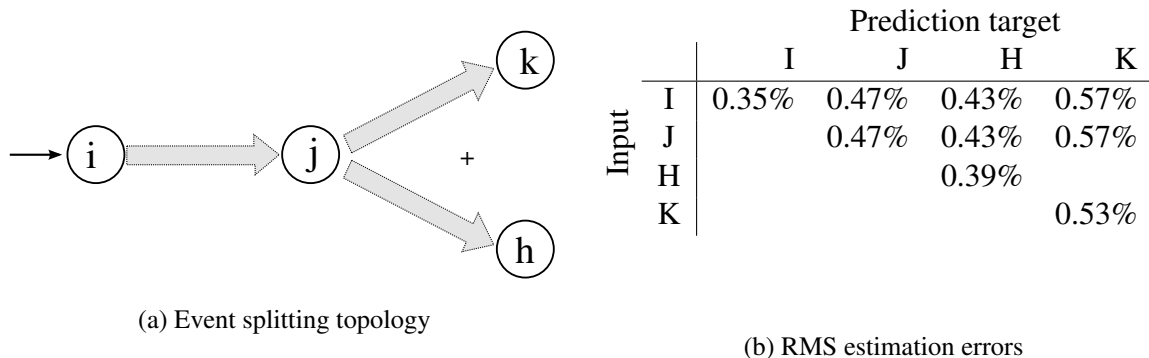


Figure 4.18. Errors in prediction from composed model due to multiple downstream servers

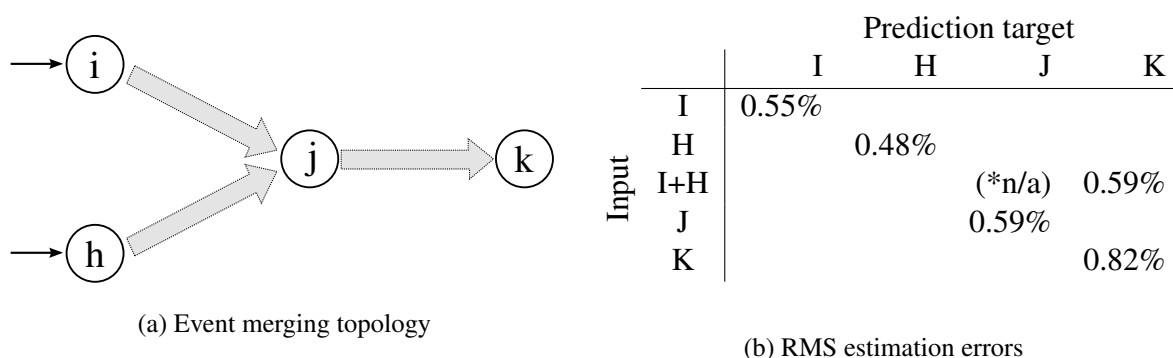


Figure 4.19. Errors in prediction from composed model due to summing inputs from multiple upstream models. (*note that results for $I+H \rightarrow J$ are not available.)

split, and join— and prediction errors were measured between each pair of upstream and downstream nodes. In Figure 4.17 we see results for the cascade topology, giving prediction errors for model composition across multiple tiers; errors grow modestly, reaching at most about 4%.

In Figure 4.18 we see results for the split topology, and the join case in Figure 4.19. In each case prediction errors are negligible. Note that in the join case, downstream predictions must be made using both of the upstream sources. This does not appear to affect accuracy; although the final prediction contains errors from two upstream models, they are

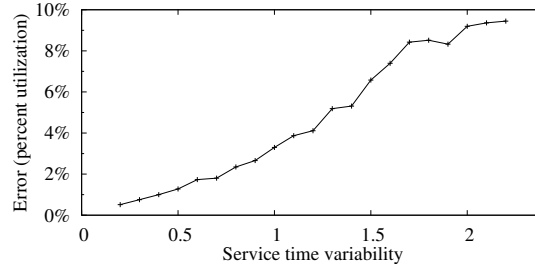


Figure 4.20. Growth of prediction error as data variability increases. Measured using synthetic CPU data generated from measured data traces. The X axis is the scaled standard deviation of the service times used to create the synthetic utilization trace.

	CPU/event	Equiv. overhead	Output data
HTTP (TPC-W)	16.5 μ s	2.7%	9.00 bytes/s
HTTP (World Cup)	11.8 μ s	n/a	9.05 bytes/s
SQL	23.8 μ s	2.6%	

Table 4.2. CPU and data overhead for Modellus log processing on several workloads

each weighted proportionally. We note that results are currently missing in the join scenario for estimating utilization of the second tier node, j , given the two first tier traffic streams.

4.6.5 System Overheads

We have benchmarked both the Modellus nucleus and the computationally intensive portions of the control plane. The nucleus was benchmarked on the testbed machines to determine both CPU utilization and volume of data produced. HTTP and SQL processing overheads were measured on log data from the TPC-W benchmark; in addition, HTTP measurements were performed for logfiles from the 1998 World Cup web site [4].

Based on the request rate in the trace logs and the CPU utilization while they were being generated, we report the estimated overhead due to Modellus event processing if the server were running at 100% CPU utilization. Figures include overhead for compressing the data before buffering; in addition, we report the rate at which compressed data is generated, as it must be buffered and then transmitted over the network. Results may be seen in Table 4.2.

W2U model	features considered		
Training window	500	1000	2000
short (8 min)	0.06s	0.12	0.24
medium (15 min)	0.10	0.20	0.42
long (30 min)	0.16	0.33	0.72

Table 4.3. Model training times for Workload-to-Utilization (W2U) models

W2U model	features considered		
Training window	500	1000	2000
short (8 min)	0.4s	0.3	0.3
medium (15 min)	0.8	0.7	0.8
long (30 min)	1.1	1.0	1.1

Table 4.4. Model training times for Workload-to-Workload (W2U) models

We measure the computationally intensive tasks of the Modeling and Validation Engine, to determine the scalability of the system. Tests were run using two systems: a 2.8GHz Pentium 4 with 512K cache, and a 2.3GHz Xeon 5140 with 4M cache. Results are reported below for only the Xeon system, which was approximately 3 times faster on this task than the Pentium 4. Each test measured the time to train a model; the length of the training window and the number of features considered was varied, and multiple replications across different data sets were performed for each combination.

Results for training W2U models are seen in Table 4.3. For 30 minute training windows and 1000 features considered, a single CPU core was able to compute 3 models per second. Assuming that at most we would want to recompute models every 15 minutes—i.e. overlapping half of the previous training window—a single CPU would handle model computation for over 2500 monitored systems. W2U model training is computationally more complex; results for the same range of model sizes are shown in Table 4.4. These measurements showed a very high data-dependent variation in computation time, as complexity of computing the first-tier model is directly affected by the number of significant features identified at the second tier. We see that computation time was primarily determined by the training window length. For 30 minute windows our system took about a

second to compute a model; if we calculate scaling as above, it could handle training data from nearly 1000 monitored systems.

Unlike model generation, model testing is computationally simple. Validation of a W2U model across a window of 30 minutes of data, for example, required between 3 and 6 milliseconds on the system used above.

4.6.6 Limitation of our Approach

Our approach to modeling system performance based on input features has a number of limitations, which we explore in this section. As with any statistical process, the larger the random component of a given amount of measured data, the higher the resulting error will be. In Modellus, such errors may be due to almost purely random factors (e.g. scheduling and queuing delays) or to “hidden variables” - factors which may deterministically affect system response, but which we are unable to measure. In this section we demonstrate the effects of such errors using by modification of testbed traces.

Simulated CPU traces were created from actual TPC-W data traces, by assigning weights to each of the TPC-W operations and then adding a lognormal-distributed random component to each processing time. Workload-to-utilization models were then trained on the original input stream and the resulting utilization data, and prediction results are reported. These may be seen in Figure 4.20, where prediction error for a fixed training window size may be seen to grow roughly linearly with the variability of the data. From this we see that increases in variability will result in either longer training windows, lower accuracy, or some combination of the two.

4.7 Data Center Analysis

In this section we apply the Modellus system and methodology to actual and simulated real-world scenarios.

Application	Mix	Web reqs/sec	Predicted	Measured	Error
Apache	Browsing	90	32.00%	20.00%	+12.00%
	Browsing	114	40.53%	31.40%	+9.13%
	Ordering	90	43.00%	41.00%	+2.00%
Tomcat	Browsing	90	37.00%	32.00%	+5.00%
	Browsing	114	46.87%	45.00%	+1.87%
	Ordering	90	56.00%	51.00%	+5.00%
MySQL	Browsing	90	25.00%	17.30%	+7.70%
	Browsing	114	31.67%	26.00%	+5.67%
	Ordering	90	66.00%	69.00%	-3.00%

Table 4.5. Case study: Predicted impact of workload changes. Model was built while running TPC-W mix 2 (shopping) and RUBiS. Predictions and measurements were made for TPC-W mix 1 (browsing) or 3 (ordering), running TPC-W only.

4.7.1 Online Retail Scenario

First we demonstrate the utility of our automatically derived models for “what-if” analysis of data center performance.

Consider an online retailer who is preparing for the busy annual holiday shopping season. We assume that the retail application is represented by TPC-W, which is a full-fledged implementation of an 3-tier online store and a workload generator that has three traffic mixes: *browsing*, *shopping* and *ordering*, each differing in the relative fractions of requests related to browsing and buying activities. We suppose that the shopping mix represents the typical workload seen by the application.

Suppose that the retailer wishes to analyze the impact of changes in the workload mix and request volume in order to plan to future capacity increases. For instance, during the holiday season it is expected that the rate of buying will increase and so will the overall traffic volume. We employ Modellus to learn models based on the typical shopping mix and use it to predict system performance for various what-if scenarios where the workload mix as well as the volume change.

We simulate this scenario on our data center testbed, as described in Section 4.6.1. Model training was performed over a 2 hour interval with varying TPC-W and RUBiS load, using the TPC-W “shopping” mixture. We then used this model to express utilization of each system in terms of the different TPC-W requests, allowing us to derive utilization as a function of requests per second for each of the TPC-W transaction mixes. The system was then measured with several workloads consisting of either TPC-W “browsing” or “ordering” mixtures.

Predictions are shown in Table 4.5 for the three traffic mixes, on the three servers in the system: Apache, which only forwards traffic; Tomcat, which implements the logic, and MySQL. Measurements are shown as well for two test runs with the browsing mixture and one with ordering. Measured results correspond fairly accurately to predictions, capturing both the significant increase in database utilization with increased buying traffic as well as the relative independence of the front-end Apache server to request mix.

4.7.2 Financial Application Analysis

The second case study examines the utility of our methods on a large stock trading application at a real financial firm, using traces of stock trading transactions executed at a financial exchange. Resource usage logs were captured over two 72-hour periods in early May, 2006; in the 2-day period between these intervals a hardware upgrade was performed. We learn a model of the application before the upgrade and demonstrate its utility to predict application performance on the new upgraded server. Event logs were captured during two shorter intervals, of 240,000 pre-upgrade and 100,000 post-upgrade events. In contrast to the other measurements in this paper, only a limited amount of information is available in these traces. CPU utilization and disk traffic were averaged over 60s intervals, and, for privacy reasons, the transaction log contained only a database table name and status (success/failure) for each event.

		cpu	preads	reads (-1000)
Pre-upgrade	Naive	38.95	6748	1151
	Feature-based	47.46	10654	1794
	Measured	47.20	8733	1448
Post-upgrade	Naive	17.472	4086	1170
	Feature-based	24.338	4819	1471
	Measured	31.03	6856	2061
	From pre-upgrade	71.18	4392	1369

Table 4.6. Trading system traces - feature-based and naïve rate-based estimation vs. measured values

In Table 4.6 we see predicted values for three variables—CPU utilization, physical reads, and logical reads—compared to measured values. Pre-upgrade and post-upgrade estimates are in all cases closer to the true values than estimates using the naïve rate-based model, and in some cases are quite accurate despite the paucity of data. In addition, in the final line we use the pre-upgrade model to predict post-upgrade performance from its input request stream. We see that I/O is predicted within about 30% across the upgrade. Predicted CPU utilization, however, has not been adjusted for the increased speed of the upgraded CPU. Hypothetically, if the new CPU was twice as fast, the predicted value would be accurate within about 15%.

4.8 Related Work

Data center and distributed system monitoring has a long history; however in this section we compare Modellus to systems which combine performance monitoring with analysis and/or model-building. We thus omit discussion of fault- and event-oriented monitoring systems such as SNMP or HP Openview [47]; these systems typically focus on modeling event correlation, which is a much different problem. In particular, in this section we contrast Modellus to two classes of existing modeling and analysis applications: ones which utilize *black-box* models, and *white-box* model-based systems. Black-box models describe only the externally visible performance characteristics of a system, with minimal assump-

tions about the internal operations. White-box models, in contrast, are based on knowledge of these internals, often at a detailed level.

Black-box models are used in a number of approaches to data center modeling and control via feedback mechanisms, using several different methods to close the feedback loop. MUSE [14] uses a market bidding mechanism to optimize utility, while Model-Driven Resource Provisioning (MDRP) [24] and Aron *et al.* [5] use dynamic resource allocation to optimize SLA satisfaction, effectively expanding the resources to fit the input. Several control theory-based systems use admission control of requests, instead, thus reducing the input to fit the resources available [45, 46]. Parekh *et al.* [65] use admission control to control queue lengths; Abdelzaher *et al.* [2] control server utilization.

While black-box models concern themselves only with the inputs (requests) and outputs (measured response and resource usage), white-box models are based on causal links between actions. *Event Relationship Networks* (ERNs) describe such relationships between fine-grained events or operations within a system. Perng *et al.* [66] use statistical data mining techniques to derive ERNs. Magpie [8] and Project5 [3] use temporal correlation on OS trace information and packet traces, respectively, to find event relationships. In a variant of these methods, Jiang *et al.* [43] use an alternate approach; viewing events of a certain type as a *flow*, sampled over time, they use regression to find invariant ratios between event flow rates, which are typically indicative of causal links.

Given knowledge of a system's internal structure (i.e. ERN), a queuing model may be created, which can then be calibrated against measured data, and then used for analysis and prediction. Stewart [82] uses this approach to analyze multi-component web applications with a simple queuing model. Urgaonkar [89] and Kounev [49] use more sophisticated product-form queuing network models to analyze and predict application performance for dynamic provisioning and capacity planning. Other work on SLA monitoring includes WebMon [34] and ETE [37], both of which merge multiple server information streams to derive user-visible performance information.

Other systems are predictive. NIMO, [78, 77] for example, uses statistical learning with active sampling to model application resource usage, and thus predict completion time for varying resource assignments. Their work focuses on the completion time of long-running applications; model composition such as done by Modellus is not applicable here as the workload is not request-based.

The work of Zhang, Cherkasova, and Smirni [95] is closely related to ours; they use regression to derive service times for queuing models of web applications, but require manual classification of events and do not compose models over multiple systems. Other work learns classifiers from monitored data: in Cohen [17] tree-augmented Bayesian Networks are used to predict SLA violations, and similarly in Chen [15] a K-nearest-neighbor algorithm is used to for provisioning to meet SLAs.

From this survey we see that modeling of event-to-event and event-to-utilization relationships has been studied in a number of ways. However, to our knowledge Modellus is the first such system to apply automated feature selection, thus transforming a tool for research analysis and insight into one which may be deployed for system management and optimization.

4.9 Conclusions

Modeling and prediction are important tools in many sensing systems, not just in data center monitoring. Many of these may be cases such as ours, where monitored data to be used for modeling and analysis is a series of information-rich events, and techniques are needed for incorporating these events into a model.

In this chapter we argue that modeling techniques relating workload volume and mix to resource utilization and tier-to-tier interactions can be useful tools in data center management. However, the complexity and rate of change of applications in these data centers makes the use of hand-constructed models difficult, as they will be of limited scope and quickly become obsolete. Therefore we propose a system which applies statistical machine

learning techniques to mine large amounts of incoming event data, discover predictive features within it, and build models from it. These models relate workload to resource utilization at a single tier, as well as tier-to-tier interactions, in a way which may be composed to examine the relationship between user input and behavior of tiers deeper in the application.

We have implemented a prototype of Modellus and deployed it on a Linux data center testbed. Our experimental results show the ability of this system to learn models and make predictions across multiple systems in a data center application, with accuracies in prediction of CPU utilization on the order of 5% in many cases. In addition, benchmarks show the overhead of our monitoring system to be low enough to allow deployment on heavily loaded servers.

CHAPTER 5

CONCLUSIONS

5.1 Conclusion

5.2 Summary of the Thesis

In this thesis we have described a number of different data management architectures and techniques for distributed data collection and presentation. These methods are described in the context of three different system designs, in which they play important roles; we summarize them below.

Archival storage for high-speed structured event streams is addressed in the Hyperion system in Chapter 2 by a file system designed for this purpose, taking advantage of the properties of sensed data streams to extract maximum performance out of standard disk hardware. By designing the Hyperion stream file system around the requirements of this task, we are able to achieve speeds of almost 50% higher than for the best general-purpose file system tested, when measured on streaming-specific benchmarks.

The second key issue addressed in Hyperion is the problem of indexing event data as it is received at high speed. By using a Bloom filter-like structure, the multi-level signature index, we are able to compute and store an index as fast as data is received. This index, in turn, provides high enough search speed to allow acceptable interactive query response across many tens of gigabytes of recorded data.

We discuss archival storage and retrieval for wireless sensor networks in Chapter 3. We present a novel data structure, the Interval Skip Graph, and its use for approximate indexing by use of intervals. The use of interval summarization allows us to construct an index where we can trade off the overhead of inserting into the index with the cost of searching,

a tradeoff which is particularly useful in archival storage applications with infrequent reads and low probabilities that any piece of data will ever be read. In addition, we describe an adaptive approach to summarization, where indexing precision is tailored to application behavior. The optimal tradeoff between index construction and lookup overhead will vary according to the request behavior of the application, and so rather than require this parameter to be set *a priori*, we instead adapt it according to current index performance to achieve the optimum balance. We present and evaluate the TSAR storage and retrieval system, which embodies these techniques.

Finally, in Chapter 4 we examine the problem of analysis and modeling in rich sensing environments, in particular in cases where we must infer models from information-rich event streams. We apply statistical learning techniques to the problem of analyzing and modeling data center application performance, allowing us to automatically construct relatively precise models without manual classification of event classes. We describe the Modellus system, incorporating these techniques, and present experimental results evaluating the performance of this approach, in both laboratory and near-real-world conditions, showing effective learning of accurate models in almost all cases.

5.3 Future Work

In Chapter 2 we present the Hyperion network monitoring system, with a focus on both its storage system and its index structure, both of which are optimized for speed. Consideration of each of these areas leads to a number of possibilities for future work. The record-oriented stream store provided by Hyperion StreamFS represents a significant break with current file system semantics; further exploration is needed to determine appropriate storage semantics for stream archival and to develop APIs to effectively deliver these semantics within the framework of modern operating systems. Conversely, the storage layout techniques used to obtain high performance bear examination to determine whether these

could be adapted in some way for use in conventional file systems when presented with appropriate usage.

The index structure used by the TSAR system in Chapter 3 presents a number of areas of future work. Can a distributed interval search structure achieve the $\log N$ update complexity bound of an Interval Tree? Are there methods of sparse tree construction which allow one to achieve better than $\log N$ amortized insertion cost, and what are the robustness penalties for doing so? Integration of TSAR with the CAPSULE storage system and other parts of the PRESTO architecture remains to be done in the future. Finally, the adaptive summarization algorithm is an area deserving of study, as although it adapts current storage to current query behavior, it is not as yet able to adapt indexing precision of previously stored data.

Finally, many areas of exploration remain in the Modellus system in Chapter 4. The effort to date has focused almost exclusively on model building; the use of these models is a rich area for future work. The use of alternate feature selection techniques, and testing of additional feature enumeration algorithms, would be valuable. Another area of possible exploration is the use of different modeling techniques, and especially data transforms before model inference; in particular, these avenues may allow us to examine the problem of modeling request latency as a function of input traffic. In the experimental area, more thorough evaluation of Modellus is needed on a spectrum of real-world data center applications and workloads.

BIBLIOGRAPHY

- [1] Abadi, Daniel J., Ahmad, Yanif, Balazinska, Magdalena, Çetintemel, Ugur, Cherniack, Mitch, Hwang, Jeong-Hyon, Lindner, Wolfgang, Maskey, Anurag S., Rasin, Alexander, Ryvkina, Esther, Tatbul, Nesime, Xing, Ying, and Zdoni, Stan. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)* (Asilomar, CA, Jan. 2005).
- [2] Abdelzaher, Tarek F., Shin, Kang G., and Bhatti, Nina. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems* 13, 1 (2002), 80–96.
- [3] Aguilera, Marcos K., Mogul, Jeffrey C., Wiener, Janet L., Reynolds, Patrick, and Muthitacharoen, Athicha. Performance debugging for distributed systems of black boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 74–89.
- [4] Arlitt, M., and Jin, T. Workload Characterization of the 1998 World Cup Web Site. Tech. Rep. HPL-1999-35R1, HP Labs, 1999.
- [5] Aron, Mohit, Iyer, Sitaram, and Druschel, Peter. A Resource Management Framework for Predictable Quality of Service in Web Servers. Tech. Rep. TR03-421, Rice University, 2003.
- [6] Ascher, D., Dubois, P.F., Hinsen, K., Hugunin, J., Oliphant, T., et al. Numerical Python. Available via the World Wide Web at <http://www.numpy.org> (2001).
- [7] Aspnes, James, and Shah, Gauri. Skip Graphs. In *ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, MD, USA, Jan. 2003), pp. 384–393.
- [8] Barham, Paul, Donnelly, Austin, Isaacs, Rebecca, and Mortier, Richard. Using Magpie for request extraction and workload modeling. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2004), pp. 259–272.
- [9] Bentley, Jon Louis. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (1975), 509–517.
- [10] Bernstein, D.J. Syn cookies. Published at <http://cr.yip.to/syncookies.html>.
- [11] Bloom, Burton. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (July 1970), 422–426.

- [12] Cain, Harold W., Rajwar, Ravi, Marden, Morris, and Lipasti, Mikko H. An Architectural Evaluation of Java TPC-W. In *Proceedings of the Intl. Symposium on High-Performance Computer Architecture* (2001). Code available at <http://www.ece.wisc.edu/~pharm/tpcw.shtml>.
- [13] Cecchet, Emmanuel, Chanda, Anupam, Elnikety, Sameh, Marguerite, Julie, and Zwaenepoel, Willy. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *ACM/IFIP/USENIX Intl. Middleware Conference* (June 2003).
- [14] Chase, Jeffrey S., Anderson, Darrell C., Thakar, Prachi N., Vahdat, Amin M., and Doyle, Ronald P. Managing energy and server resources in hosting centers. In *Proceedings of the ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 103–116.
- [15] Chen, Jin, Soundararajan, Gokul, and Amza, Cristiana. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *IEEE International Conference on Autonomic Computing (ICAC)* (June 2006), pp. 231–242.
- [16] Cheriton, David. The V Distributed System. *Communications of the ACM* 31, 3 (Mar. 1988), 314–333.
- [17] Cohen, Ira, Chase, Jeffrey S., Goldszmidt, Moisés, Kelly, Terence, and Symons, Julie. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004), pp. 231–244.
- [18] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, second edition ed. The MIT Press and McGraw-Hill, 2001.
- [19] Crainiceanu, Adina, Linga, Prakash, Gehrke, Johannes, and Shanmugasundaram, Jayavel. Querying Peer-to-Peer Networks Using P-Trees. Tech. Rep. TR2004-1926, Cornell University, 2004.
- [20] Cranor, Chuck, Johnson, Theodore, Spataschek, Oliver, and Shkapenyuk, Vladislav. Gigascope: a stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), ACM Press, pp. 647–651.
- [21] Crossbow Technology. *MICA2 Datasheet*, 2004. part number 6020-0042-06 Rev A.
- [22] Dai, Hui, Neufeld, Michael, and Han, Richard. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (New York, NY, USA, 2004), ACM Press, pp. 176–187.

- [23] Desnoyers, Peter, and Shenoy, Prashant. Hyperion: High Volume Stream Archival for Retrospective Querying. Tech. Rep. TR46-06, University of Massachusetts, Sept. 2006.
- [24] Doyle, Ronald P., Chase, Jeffrey S., Asad, Omer M., Jin, Wei, and Vahdat, Admin M. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (Mar. 2003).
- [25] Draper, Norman R., and Smith, Harry. *Applied Regression Analysis*. John Wiley & Sons, 1998.
- [26] Endace Inc. Endace DAG4.3GE Network Monitoring Card. available at <http://www.endace.com>, 2006.
- [27] Faloutsos, Christos, and Chan, Raphael. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison. In *Proceedings of the Intl. Conference on Very Large Data Bases (VLDB)* (1988), pp. 280–293.
- [28] Faloutsos, Christos, and Christodoulakis, Stavros. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.* 2, 4 (1984), 267–288.
- [29] Foster, Dean P., and George, Edward I. The Risk Inflation Criterion for Multiple Regression. *The Annals of Statistics* 22, 4 (1994), 1947–1975.
- [30] Ganesan, Deepak, Greenstein, Ben, Perelyubskiy, Denis, Estrin, Deborah, and Heidemann, John. An Evaluation of Multi-resolution Storage in Sensor Networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2003).
- [31] Garfinkle, Simpson. private communication, 2007.
- [32] Girod, L., Stathopoulos, T., Ramanathan, N., Elson, J., Estrin, D., Osterweil, E., and Schoellhammer, T. A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (Baltimore, MD, 2004).
- [33] Grechanovsky, Eugene. Stepwise Regression Procedures: Overview, Problems, Results, and Suggestions. *Reports from the Moscow Refusnik Seminar. Annals of the New York Academy of Sciences*, (1987).
- [34] Gschwind, Thomas, Eshghi, Kave, Garg, Pankaj K., and Wurster, Klaus. WebMon: A Performance Profiler for Web Transactions. In *Proceedings of the IEEE Intl. Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 171.
- [35] Guttman, Antonin. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1984), ACM Press, pp. 47–57.

- [36] Harvey, Nicholas, Jones, Michael B., Saroiu, Stefan, Theimer, Marvin, and Wolman, Alec. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (Seattle, WA, March 2003).
- [37] Hellerstein, Joseph L., Maccabee, Mark, Mills, W. Nathaniel, and Turek, John J. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *International Conference on Distributed Computing Systems* (1999).
- [38] Hill, Jason, Szewczyk, Robert, Woo, Alec, Hollar, Seth, Culler, David, and Pister, Kristofer. System architecture directions for networked sensors. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, MA, USA, Nov. 2000), ACM, pp. 93–104.
- [39] Huebsch, Ryan, Chun, Brent, Hellerstein, Joseph M., Loo, Boon Thau, Maniatis, Petros, Roscoe, Timothy, Shenker, Scott, Stoica, Ion, and Yumerefendi, Aydan R. The Architecture of PIER: an Internet-Scale Query Processor. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)* (Jan. 2005).
- [40] Iannaccone, Gianluca, Diot, Christophe, McAuley, Derek, Moore, Andrew, Pratt, Ian, and Rizzo, Luigi. The CoMo White Paper. Tech. Rep. IRC-TR-04-17, Intel Research, Sept. 2004.
- [41] Intanagonwiwat, Chalermek, Govindan, Ramesh, and Estrin, Deborah. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking* (Boston, MA, Aug. 2000), ACM Press, pp. 56–67.
- [42] Jacobson, V., Leres, C., and McCanne, S. *The Tcpdump Manual Page*. Lawrence Berkeley Laboratory, Berkeley, CA, June 1989.
- [43] Jiang, Guofei, Chen, Haifeng, and Yoshihira, K. Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management. In *IEEE International Conference on Autonomic Computing (ICAC)* (Dublin, Ireland, June 2006), pp. 199–208.
- [44] JWebUnit. <http://jwebunit.sourceforge.net>, 2007.
- [45] Kamra, A., Misra, V., and Nahum, E. Controlling the Performance of 3-Tiered Web Sites: Modeling, Design and Implementation. In *Proceedings of the ACM SIGMETRICS Conference* (2003).
- [46] Kamra, A., Misra, V., and Nahum, E. Yaksha: A Controller for Managing the Performance of 3-Tiered Websites. In *Proceedings of the 12th IWQoS* (2004).
- [47] Klemba, Keith S. Openview’s Architectural Models. In *Proc. of the Intl. Symp. on Integrated Network Management* (1989), IFIP, pp. 565–572.

- [48] Kornexl, S., Paxson, V., Dreger, H., Feldmann, A., and Sommer, R. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *Proc. ACM IMC* (Oct. 2005).
- [49] Kounev, Samuel, and Buchmann, Alejandro. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proc. Intl. CMG Conf. on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003* (Dec. 2003).
- [50] Kruegel, Christopher, Valeur, Fredrik, Vigna, Giovanni, and Kemmerer, Richard. Stateful Intrusion Detection for High-Speed Networks. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy* (May 2002), p. 285.
- [51] Li, Xin, Bian, Fang, Zhang, Hui, Diot, Christophe, Govindan, Ramesh, Hong, Wei, and Iannaccone, Gianluca. Advanced Indexing Techniques for Wide-Area Network Monitoring. In *Proc. IEEE Intl. Workshop on Networking Meets Databases (NetDB)* (2005).
- [52] Li, Xin, Kim, Young-Jin, Govindan, Ramesh, and Hong, Wei. Multi-Dimensional Range Queries in Sensor Networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2003). to appear.
- [53] Litwin, Witold, Neimat, Marie-Anne, and Schneider, Donovan A. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Intl. Conference on Very Large Data Bases (VLDB)* (San Francisco, CA, USA, 1994), pp. 342–353.
- [54] M. A. Efroymson, MA. Multiple regression analysis. *Mathematical Methods for Digital Computers 1* (1960), 191–203.
- [55] Madden, Samuel, Franklin, Michael, Hellerstein, Joseph, and Hong, Wei. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, 2002).
- [56] Mainwaring, A., Szewczyk, R., Culler, D., and Anderson, J. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)* (2002), IEEE.
- [57] Mathur, Gaurav, Desnoyers, Peter, Ganesan, Deepak, and Shenoy, Prashant. Capsule: An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (Nov. 2006).
- [58] Mathur, Gaurav, Desnoyers, Peter, Ganesan, Deepak, and Shenoy, Prashant. Ultra-low power data storage for sensor networks. In *Intl. Conference on Information Processing in Sensor Networks, Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)* (Nashville, TN, April 2006).

- [59] Moore, Andrew, Hall, James, Kreibich, Christian, Harris, Euan, and Pratt, Ian. Architecture of a Network Monitor. In *Passive & Active Measurement Workshop 2003 (PAM2003)* (2003).
- [60] Motwani, Rajeev, Widom, Jennifer, Arasu, Arvind, Babcock, Brian, Babu, Shivnath, Datar, Mayur, Maku, Gurmeet, Olston, Chris, Rosenstein, Justin, and Varma, Rohit. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)* (2003).
- [61] Ndiaye, Baila, Nie, Xumin, Pathak, Umesh, and Susairaj, Margaret. A Quantitative Comparison between Raw Devices and File Systems for implementing Oracle Databases. <http://www.oracle.com/technology/ deploy/performance/WhitePapers.html>, Apr. 2004.
- [62] Niksun, Inc. *Niksun NetDetector Data Sheet*. Monmouth Junction, New Jersey, 2005.
- [63] The Apache “Open For Business” project. <http://ofbiz.apache.org>, 2007.
- [64] Orenstein, J. A., and Merrett, T. H. A Class of Data Structures for Associative Searching. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)* (New York, NY, USA, 1984), ACM Press, pp. 181–190.
- [65] Parekh, Sujay, Gandhi, Neha, Hellerstein, Joe, Tilbury, Dawn, Jayram, T. S., and Bigus, Joe. Using Control Theory to Achieve Service Level Objectives In Performance Management. *Real-Time Systems* 23, 1 (2002), 127–141.
- [66] Perng, Chang-Shing, Thoenen, David, Grabarnik, Genady, Ma, Sheng, and Hellerstein, Joseph. Data-driven validation, completion and construction of event relationship networks. In *Proceedings of the ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining (KDD)* (New York, NY, USA, 2003), ACM Press, pp. 729–734.
- [67] Polastre, J., Hill, J., and Culler, D. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (Nov. 2004).
- [68] Polastre, Joseph, Szewczyk, Robert, and Culler, David. Telos: Enabling Ultra-Low Power Wireless Research. In *Intl. Conference on Information Processing in Sensor Networks, Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)* (Apr. 2005).
- [69] Pugh, William. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM* 33, 6 (1990), 668–676.
- [70] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM Conference* (2001).

- [71] Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S. GHT - A Geographic Hash-Table for Data-Centric Storage. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)* (2002).
- [72] Rosenblum, Mendel, and Ousterhout, John K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52.
- [73] Rosenblum, Mendel, and Ousterhout, John K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52.
- [74] Sacks-Davis, Ron, and Ramamohanarao, Kotagiri. A two level superimposed coding scheme for partial match retrieval. *Information Systems* 8, 4 (1983), 273–289.
- [75] SandStorm Enterprises: NetIntercept. <http://www.sandstorm.net/products/netintercept>.
- [76] Seltzer, Margo I., Smith, Keith A., Balakrishnan, Hari, Chang, Jacqueline, McMains, Sara, and Padmanabhan, Venkata N. File System Logging versus Clustering: A Performance Comparison. In *USENIX Winter Technical Conference* (1995), pp. 249–264.
- [77] Shivam, P., Babu, S., and Chase, J.S. Active Sampling for Accelerated Learning of Performance Models. In *Proc. 1st Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)* (June 2006).
- [78] Shivam, Piyuth, Babu, Shivnath, and Chase, Jeff. Learning Application Models for Utility Resource Planning. In *IEEE International Conference on Autonomic Computing (ICAC)* (Dublin, Ireland, June 2006), pp. 255–264.
- [79] Shriver, Elizabeth, Gabber, Eran, Huang, Lan, and Stein, Christopher A. Storage management for web proxies. In *Proceedings of the USENIX Annual Technical Conference* (2001), pp. 203–216.
- [80] Smith, W. TPC-W: Benchmarking An Ecommerce Solution. <http://www.tpc.org/information/other/techarticles.asp>.
- [81] Sorber, Jacob, Kostadinov, Alexander, Garber, Matthew, Brennan, Matthew, Corner, Mark D., and Berger, Emery D. eFlux: A language and runtime system for perpetual systems. Tech. Rep. TR 06-61, University of Massachusetts Amherst, Dec. 2006.
- [82] Stewart, Christopher, and Shen, Kai. Performance Modeling and System Management for Multi-component Online Services. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (May 2005).
- [83] Stonebraker, Michael, Cetintemel, Ugur, and Zdonik, Stan. The 8 requirements of real-time stream processing. *SIGMOD Record* 34, 4 (2005), 42–47.

- [84] StreamBase, Inc. StreamBase: Real-Time, Low Latency Data Processing with a Stream Processing Engine. from <http://www.streambase.com>, 2006.
- [85] Sweeney, Adam, Doucette, Doug, Hu, Wei, Anderson, Curtis, Nishimoto, Mike, and Peck, Geoff. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference* (Jan. 1996).
- [86] Tobagi, Fouad A., Pang, Joseph, Baird, Randall, and Gang, Mark. Streaming RAID: a disk array management system for video files. In *Proc. 1st ACM Intl. Conf. on Multimedia* (1993), pp. 393–400.
- [87] Tolle, Gilman, Polastre, Joseph, Szewczyk, Robert, Culler, David, et al. A macro-scope in the redwoods. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2005), ACM Press New York, NY, USA, pp. 51–63.
- [88] UMass Trace Repository. Available at <http://traces.cs.umass.edu>.
- [89] Urgaonkar, Bhuvan, Pacifici, Giovanni, Shenoy, Prashant, Spreitzer, Mike, and Tantawi, Assar. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proceedings of the ACM SIGMETRICS Conference* (Banff, Canada, June 2005).
- [90] Wang, Wenguang, Zhao, Yanping, and Bunt, Rick. HyLog: A High Performance Approach to Managing Disk Layout. In *USENIX Conference on File and Storage Technologies (FAST)* (2004), pp. 145–158.
- [91] Weaver, Nicholas, Paxson, Vern, and Gonzalez, Jose M. The shunt: an FPGA-based accelerator for network intrusion prevention. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays* (New York, NY, USA, 2007), ACM Press, pp. 199–206.
- [92] Xu, N., Osterweil, E., Hamilton, M., and Estrin, D. <http://www.lecs.cs.ucla.edu/~nxu/ess/>. James Reserve Data.
- [93] Xu, Ning, Rangwala, Sumit, Chintalapudi, Krishna Kant, Ganesan, Deepak, Broad, Alan, Govindan, Ramesh, and Estrin, Deborah. A Wireless Sensor Network For Structural Monitoring. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (New York, NY, USA, 2004), ACM Press, pp. 13–24.
- [94] Zeinalipour-Yazti, Demetrios, Lin, Song, Kalogeraki, Vana, Gunopulos, Dimitrios, and Najjar, Walid. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *USENIX Conference on File and Storage Technologies (FAST)* (Dec. 2005).
- [95] Zhang, Q., Cherkasova, L., and Smirni, E. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *IEEE International Conference on Autonomic Computing (ICAC)* (2007). to appear.