Technische Universität München
Fakultät für Informatik

Master's Thesis in Informatics

VAST: Analyse von Netzwerkverkehr unabhängig von Raum und Zeit

VAST: Network Visibility Across Space and Time

Bearbeiter:	Matthias Vallentin
Aufgabensteller:	Univ.-Prof. Dr. Uwe Baumgarten
Betreuer:	Dr. Robin Sommer
	International Computer Science Institute, Berkeley, USA
Abgabedatum:	15. Januar 2009

Affirmation

I assure the single handed composition of this master's thesis only supported by declared resources.

Eidesstattliche Erklärung

Ich versichere, dass ich diese Master-Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Berkeley, USA, den 15. Januar 2009

Matthias Vallentin

Abstract

Key operational networking tasks, such as troubleshooting and defending against attacks, greatly benefit from attaining views of network activity that are unified across *space and time*. This means that data from heterogeneous devices and systems is treated in a uniform fashion, and that analyzing past activity and detecting future instances follow the same procedures. Based on previous ideas that formulated principles for comprehensive network visibility [AKP+08], we present the design and architecture of *Visibility Across Space and Time* (VAST), an intelligent database that serves as a single vantage point into the network. The system is based on a generic event model to handle network data from disparate sources and provides a query architecture that allows operators or remote applications to extract events matching a given condition. We implemented a proof-of-principle prototype that can archive and index events from a wide range of sources. Moreover, we conducted a preliminary performance evaluation to verify that our implementation works efficient and as expected.

Zusammenfassung

Fehlersuche und Sicherheitsanalyse stellen integrale Aufgaben bei der Betreibung von Hochleistungsnetzwerken dar. Die dafür eingesetzten Methoden zur Analyse von Netzwerkverkehr sind jedoch fragmentiert in Bezug auf *Raum und Zeit*. Mit räumlicher Fragmentierung ist gemeint, dass die zu analysierenden Daten aus vielen verschiedenen Quellen stammen und in heterogenen Formaten vorliegen. Zeitliche Fragmentierung bedeutet, dass sich die Praktiken zur Analyse von Daten aus der Vergangenheit wesentlich von der Art und Weise unterscheiden, wie man Aktivität in der Zukunft beschreibt. Die Überwindung der räumlichen und zeitlichen Diskrepanzen bringt substantielle Vorteile mit sich und ermöglicht eine effiziente Analyse von Netzwerkaktivität unabhängig von Raum und Zeit. Basierend auf den Prinzipien zur Sichtbarkeit in Netzwerken [AKP+08] bildet diese Thesis den ersten praktischen Schritt zur Realisierung einer umfassenden Plattform zur Analyse von Netzwerkaktivität. Dazu präsentieren wird das Design und die Architektur von *Visibility Across Space and Time* (VAST), einer intelligenten Datenbank, die verschiedenste Datenquellen homogen abbilden kann und eine einheitliche Schnittstelle für Anfragen bereit stellt. Ferner implementieren wir einen Prototyp, der Events von unterschiedlichen Quellen archivieren und indizieren kann. Abschließend führen wir eine Performance-Analyse durch um kritische Subsysteme unserer Implementierung zu identifizieren.

Contents

1 Introduction

1.1 Motivation

Security analysis and troubleshooting are key operational networking tasks that face new challenges in large and continuously expanding networks. Incidents that involve multiple entities require investigating data from numerous sources in different formats, yet the ever-increasing volume and heterogeneity of the data renders coherent problem analysis an arduous process. Moreover, wide-scale incidents can span long time periods that exceed the measuring window of today's monitoring infrastructure. Understanding the full scope of extensive attacks with an incomplete context can become a tedious task.

In other words, network operators and security analysts lack a unified view of the network today: descriptions of activity are fragmented in both *space and time*. By fragmented in space, we refer to the heterogeneity of disparate data sources and programmatic interfaces that the network operator has to incorporate to obtain a unified view of the network data. By fragmented in time, we refer to the discrepancy in analysis procedures between past and future activity.

To illustrate fragmentation in time, consider the different techniques a security analyst employs to perform forensic analyses in contrast to the mechanisms used to address potential future activity. While the former task often involves crawling through a large set of log-files that have to be synthesized by scripts in order to gain a global picture, the latter task drives the analyst to use defensive strategies such as configuring firewalls, service ACLs, and intrusion detection system (IDS) policies. In this case, addressing past issues clearly differs from describing possible future actions in terms of the configuration language and as well as the process of analysis itself (e.g. processing plain-text IDS logs versus configuring firewall rules; using scripts to extract information from the logs versus pro-actively configuring alert generation).

In equal measure, fragmentation in space means that the analyst has to manually consolidate heterogeneous sources of information that vary in quality, scope, and expressiveness. For example, consider a DHCP server that does not hand out leases to an erroneous configuration. Several sources in the network record the defective behavior of the DHCP daemon. Creating an integrated view of the available error descriptions becomes a tedious activity: client side error messages vary across different DHCP client implementations, server error messages could be too brief to pin-point the exact error source, and network monitoring tools cannot detect endpoint problems unless they manifest in the communication. When the analyst is faced to debug the erroneous behavior of the DHCP daemon, locating the root of the problem with such a wealth of different data sources is similar to finding a needle in the haystick.

In this thesis, we provide the practical component to previous ideas which set out

principles of comprehensive network visibility [AKP+08]. We devise the architecture and implement the first prototype of *Visibility Across Space and Time* (VAST), a platform that serves as a single vantage point to network activity. VAST is designed to archive descriptions of activity from numerous sources in the network and to query past activity for retrospective analysis. At the same time, operators can codify activity patterns and receive notifications when their pattern matches in the future. Based on a policy-neutral core, the architecture allows a site to customize operations to coalesce, age, sanitize, and delete data.

The realization of such a system requires addressing a number of challenges. First, we need a flexible and expressive data model to unify heterogeneous descriptions of activity. A particularly apt model for this purpose is the *publish/subscribe* scheme, which is based on asynchronous communication using events. In this model, interacting peers publish the availability of an event and subscribe to events published by peers according to local interest.

Second, we need to separate mechanism from policy. This aspect is particularly important when the operation of security devices is concerned. Many devices limit their focus only on real-time detection but ignore the additional step of providing policy-neutral output that can become highly useful at a later point of time. To illustrate, IDS frequently tend to raise an alert that is already coined with a specific policy, such as "host X is an intruder", instead of continuously generating simple descriptions of activity (e.g. "host X accessed n times service S on host Y in the last m minutes"), which can prove invaluable for future forensic analysis or the verification of the IDS's effectiveness.

Third, we have to realize that the temporal dimensions of analysis extend beyond the duality of past and future. By considering an additional *what-if* perspective, we seek to automatically explore the suitability of analysis patterns codifying future behavior if applied to past activity. For instance, false positives can be greatly reduced by testing detection procedures against previous network behavior. In equal measure, the spatial dimensions of analysis do not only cover technical aspects such as log centralization and endpoint management, but also need to engage in policy issues. This is a fundamental requirement for broadening the horizon across local administrative domains to attain a global view. While the subsequent discussions are bounded to enterprise-wide "network visibility", Allman et al. [AKP+08] additionally envision to establish a sweet-spot for information exchange between sites that decided to collaborate to benefit from the synergetic effects.

Finally, we argue to avoid "clean-slate" designs. In operational practice, the very challenge in creating a coherent picture of network activity is often exacerbated by the chaotic details such as heterogeneity and policy frictions. Therefore, we emphasize *incremental deployability* to achieve practical usability. Moreover, the notion of network visibility can serve as a *research enabler* in the context of highly automated forms of analysis and reactive control by giving analysts the opportunity to better understand and gradually increase their trust in such systems.

1.2 Outline

The remainder of the thesis is structured as follows.

Chapter 2. In the second chapter, we provide the necessary background to understand our work. Possible application scenarios are sketched that would significantly benefit from a VAST system. We further introduce the conceptual and technological building blocks of VAST. Finally, we refer to related work in this field and compare our approach to existing ones.

Chapter 3. In the third chapter, we present the design and architecture of VAST. At first, we discuss particularly important guidelines and highlight concrete technical challenges we face. Thereafter, the design is explained in detail: we delve into each component, motivate our decisions and illustrate the system features. Moreover, we present challenging aspects we encountered during the implementation.

Chapter 4. In the fourth chapter, we conduct a preliminary performance evaluation of our prototype implementation. Our results identify the bottleneck of the system and thus indicate which component needs further improvement to handle traffic volumes of operational networks.

Chapter 5. In the fifth chapter, we summarize our work and give perspectives for short-term goals as well as promising directions in the future.

2 Background

To better understand the need for a unified view of network activity across space and time, we outline in §2.1 possible scenarios in which VAST would prove highly useful. In §2.2 we present basic aspects that build the conceptual foundation of the design of VAST. To substantiate this theoretical groundwork, we describe in §2.3 concrete technologies that we employ to implement VAST. Finally, in §2.4 we summarize related work in this field and compare the existing solutions to our approach.

2.1 Fields of Application

Let us have a look at different scenarios in operational networks that face existential problems due to the fragmented nature of descriptions of network activity. In the following application fields proposed by [AKP+08], we point out the practical hurdles and problems and argue why an integrated view of network activity would significantly improve the scenario.

Troubleshooting. Network troubleshooting often requires an operator to incorporate detailed information from scattered sources. Not only does the modular structure of network architectures increase the complexity of analysis, but also the layered design of many protocols makes it hard to locate the errors when the system fails to operate correctly. The more context (e.g. in terms of network activity logs) operators can draw upon, the easier it turns out to be chasing down the error. Therefore, being able to characterize the problem sources from as many perspectives as possible is of high value for the network operator.

Unfortunately, it is a cumbersome process to orchestrate the numerous, heterogeneous data sources. Chasing down the error is similar to finding a needle in a haystack, especially when the amount of available data exceeds the resources to analyze it. Specifically tailored processes can help to cope with the high data volume and complexity, but are limited to *known* analysis procedures.

We argue that there is great practical utility of an infrastructure that does not only help to understand past activity to pin-point error sources, but also allows operators to integrate analogous analysis procedures for future activity. For example, consider the example of a failing automated remote backup caused by wrong permissions of the SSH configuration directory. In addition to solve the problem by correcting the permissions, the operator could as well *codify* the corresponding troubleshooting process and integrate it in a repository of analysis procedures. Future occurrences of similar problem instances could then be detected and handled

with appropriate means. In this particular example, the codification could include automated monitoring of SSH server logs to trigger an automated inspection of the directory permissions on the failing client.

Detecting Intrusions. Another scenario that significantly benefits from detailed information aggregated from many sources is the detection of successful intrusions. Today's intrusion detection systems (IDSs) mostly follow one particular monitoring strategy, that is, their analysis is restricted to process either host or network data. In contrast, a *hybrid* system that synthesizes data from a variety of sources substantially increases the IDS's efficacy. There exist several data sources that can yield synergetic effects when being combined. Examples include data from virus scanners, user keystrokes and interactions with peripheral devices, firewall logs, past DNS lookups, internal NetFlow records, and honeypot data. Not only the sheer availability of additional data enhances the analysis context of the IDS, but also the ability to correlate between the hitherto unconnected data flows can be of great value.

In addition to real-time intrusion detection and prevention, post facto forensics constitute an equally important aspect of comprehensive incident response. The quality of the log archive of past activity greatly determines to which extent forensic analyses succeed. Only a high quality archive allows the security analyst to assess the real depth of an intrusion and quantify the scope of the caused damage. A unified log archive has not only the advantage of making forensic analyses less error-prone and more precise, but can also save a significant amount of time, which is key to defend against an ongoing attack.

Particularly intrusion detection profits from leveraging past information to improve future monitoring. It is highly beneficial for a security analyst being able to codify security breaches in a consistent fashion in order to automatically detect future instances of a certain attack pattern. Moreover, the analyst could extend the temporal dimension to *what-if*, i.e. automated exploration of how analyses intended for future activity would have behaved if applied in the past.

Combating Insider Abuse. A similar problem are framed insider attacks. Although similar to intrusion detection, when dealing with security breaches from inside, it is often difficult to grasp the full scope of the attack at the time of detection because the complete attack can comprise of a chain of actions, each of which the attacker was authorized to do. Yet the abuse manifests only when the undertaken actions occur in a specific order. For example, an employee might first access a sensitive machine, then copy documents to an intermediate machine to conceal the malicious intent, and finally send the sensitive data to a non-authorized party. Once the attack has been discovered, understanding the full implications and the resulting damage requires a readily searchable archive that consists of comprehensive activity logs of all involved entities.

2.2 Concepts

In the following, we discuss two fundamental concepts that we face when turning to the implementation of VAST. First, we need a unified data model to consolidate data from multitudinous disparate sources. In §2.2.1, we argue that asynchronous event-based communication is a particularly apt model for this purpose. Second, the immense volume of the archived data requires an efficient storage mechanism that cannot be addressed with traditional database concepts. We justify our approach in §2.2.2.

2.2.1 Unified Data Model

Today's networks consist of many heterogeneous components that have manifold formats to describe network activity. Particularly large networks exhibit many different types of users, protocols, and applications. From a security and administration point of view, orchestrating the disparate representations of activity is an arduous task. On the one hand, the inhomogeneity introduces redundancies that complicate the process of analysis. On the other hand, the varying expressiveness of representations lowers the chance to pinpoint the desired information in the appropriate level of detail. Consequently, network operators and security analysts strive to minimize an inflation of description formats to keep their analyses tractable.

Thus, we need a both flexible and expressive data model to unify heterogeneous descriptions of activity. A closer look at existing systems (e.g. the Bro NIDS we describe in §2.3.1) that feature a distributed data model shows that asynchronous *event-based* communication proved successful in the past. This model grounds on the *publish/subscribe* [EFGK03] scheme where interacting peers *subscribe* to information according to their interest and *publish* the availability of the data they can provide. Using an event-model has several advantages, as we demonstrate in the following.

Loose Coupling. By plugging an event service between publishers and subscribers through which events are propagated, decoupling can be characterized in three dimensions according to [EFGK03].

1. *Space Decoupling.* The interacting peers do not need to know each other. Publishers send their events through the mediating event service which in turn distributes the events to all subscribers.
2. *Time Decoupling.* Interacting peers do not have to communicate synchronously in time. The event service in-between buffers the notification when subscribers are unreachable and relays the message when the subscriber is back available.
3. *Synchronization Decoupling.* While generating events, publishers are not blocked and can notify subscribers asynchronously through callback handlers when an event has become available. Thus, event generation and processing occurs outside the main control flow in an asynchronous fashion.

Reduced Communication Overhead. Compared to communication models based on repeated polling, the publish/subscribe scheme has a reduced communication over-

head since peers explicitly push the requested information at the correct point of time instead of repeatedly pulling until the desired information is available. This is particularly beneficial in high-volume applications where expensive inter-peer communication can slow down the maximum throughput.

Aggregation. Events already present by themselves an abstraction of activity. In addition, they can be further elevated to higher semantic representations by condensing embodied information to a more succinct form. For example, several observed packets can yield events that report numerous unsuccessful connection attempts. At a later point of time, these events could be packed into summary like "N attempts in the last T seconds".

Sanitization. Having a model that utilizes events to represent information enables well-defined operations on the data. A sanitization logic can alter, sanitize, or completely remove sensitive information from events. Clearing sensitive aspects of the collected data becomes in particular important when establishing cross-site relationships to share data, a topic that goes beyond the scope of this thesis but is further explored in [AKP+08]. For example, before sensitive information leaves the internal site, an anonymization routine could clear IP addresses, timestamps or perform low-level payload cleaning to prevent the disclosure of user credentials.

Policy Neutrality. We understand events as neutral abstractions of network activity at different semantic levels, without imposing policy regulations. However, some systems do not separate analysis from data, i.e. they generate an event stream with pre-conceived judgments. Applications that use such a data stream as input cannot produce neutral output as the analysis is based on a filtered data. Consider the case where an organization deploys the most common and wide-spread open-source NIDS Snort [Roe99]. The system raises an *alert* whenever a signature matches successfully. In general, applying signature sets by definition implies the application of policy on the underlying event stream. Hence, subsequent applications that rely on Snort logs as input source are a priori condemned to produce biased output.

By virtue of these advantages, we base the underlying data model of our framework on events. We envision to instrument all participating entities with the capability to generate events compliant to our data scheme. If directly interfacing turns out to be difficult or impossible, an external converter application can still transform output logs to events which are then forwarded to VAST.

Events are usually elicited by a producer that merely forwards it to subscribed peers, without keeping a permanent copy. Thus, events have an *ephemeral* character, i.e. they exist only for short time period while being processed. Thereafter, they are usually expunged from memory and not available anymore at later times. Without further infrastructure, events are consequently only suitable for real-time analysis. However, a major aspect of VAST is to support analysis of past activity, which naturally requires a mechanism to permanently store events for later inspection and querying.

We describe in §2.4.1 the Time Machine as a similar approach on a smaller scale: a system based on commodity hardware that that can buffer a high-volume network packet stream for several days. Not only does it support historical queries, but it is also possible to subscribe to future occurrences of events that match a specified pattern. By leveraging the heavy-tailed nature of network traffic, the system can store most connections in their entirety, yet skip the bulk of the total volume at a specified limit of bytes per connection. The authors of [AKP+08] distill three fundamental concepts that describe the operation of the Time Machine independent of the type of data being processed: *(i)* archiving a high-volume data stream, *(ii)* applying filter rules that separate more important from less important input to find a balance between available resources and comprehensive storage, and *(iii)* using aging mechanism to expire old data.

From VAST's angle, events instead of packets represent the data. While the Time Machine uses a configurable cutoff to handle the data volume, operators in VAST define filters, implemented as hooks that enable powerful event transformation and aggregation schemes (see §3.2.5). The Time Machine simply discards old packets to reclaim storage space. It is the advantage of an event-based model that old events can be aggregated from detailed, low-level representations into more compact, dense abstractions much smaller in size.

2.2.2 Database Fundamentals

VAST combines both data streaming and archival concepts. Data streaming is concerned to the extent that the system continuously receives and relays streams of events. Data archival is concerned to the extent that events are stored permanently for later querying. We first introduce these different database concepts and highlight the benefits of bitmap indices, a technology that greatly improves the query performance.

Traditional *data base management systems (DBMS)* support queries over data that already exists before the query is issued. These are referred to as *historical queries*. In contrast, *live* or *continuous queries (CQ)* incorporate new arriving data after the query is issued. Database system that answer the latter type of queries are called *data stream management systems (DSMS)* [GO03]. There exist also *hybrid* solutions that combine the concepts from both traditional and streaming databases. In this context, the real-time requirements of live data in combination with expensive I/O operations pose a fundamental challenge.

To answer historical queries, most traditional DBMSs implement a *row-oriented* storage layout where record-like data is appended to the database in form of tuples. This model is particularly optimized for *on-line transaction processing* (OLTP) where write and update operations occur more often than search operations. On the contrary, data warehouse applications primarily process large datasets but seldom add or update existing data. Updates or similar expensive operations are usually postponed to a future point where the system is in an idle state and can handle bulk changes. The performance of such *on-line analytical processing* (OLAP) applications benefits from a *column-oriented* layout of data, where the values are stored contiguously for each column (or attribute). Using a column-oriented structure, only attributes that contain relevant data have to

be processed, thus avoiding to load irrelevant columns into memory. Each column can effectively be seen as a *projection index* [OQ97] that can sometimes even be used to answer queries without consulting indexing structures. *C-Store* [SAB+05] and *MonetDB* [BZN05] are two scientific examples which implement a column-oriented layout.

To support live queries, we need to understand the fundamental differences between traditional and streaming data. Unlike conventional DBMSs that are designed around a bounded, semi-permanent data model, DSMSs deal with a continuous, irregular, and real-time stream of data. Examples include sensor data, network traffic, and stock tickers. The different characteristics impose a number of unique requirements that have been extensively studied in the past [GO03, ScZ05].

Hybrid queries handle both historical and live data simultaneously. An inherent difficulty with this approach is the high I/O cost of delivering historical query results that can prevent the system from processing the incoming stream of data. The system is then forced to forego fractions of the arriving stream. In contrast, traditional DBMS can delay expensive computations at later times when the system experiences less load. This does not apply to systems that steadily grapple with incoming data. The real-time nature of hybrid queries renders concepts from traditional DBMSs and DSMSs inapplicable and demands new solutions. To reduce I/O costs for fetching historical data, it is tempting to retrieve only sampled versions of the historical data [CF04]. In security applications, however, relying solely on sampling techniques bears a high risk of missing critical information in the omitted data. It should therefore be considered only as last resort.

Since VAST answers both historical and live queries, it is a *hybrid* system as well. Historical queries are issued by users and return past data. Live queries represent a form of *subscriptions*, where a data that matches a certain condition is continuously delivered to the requesting instance.

Accelerating Historical Queries

Queries aimed at VAST require processing great quantities of historical data that often exceed the size of main memory. At the same time, the system has to continuously archive arriving data. Moreover, historical data is constantly *transformed* by sanitization and aggregation routines of VAST. Due to these high data volumes, continuous updates, and constant transformations, it is imperative to speedup historical queries in order to prevent performance bottlenecks. In the following, we motivate our choice for bitmap indices and explain why they represent an excellent solution to accelerate the types of queries in VAST.

A typical example for a search query would be "Which hosts accessed host **X** in the past **Y** hours and also fetched URL **Z**?". In order to accelerate such queries, *indices* are typically used. The most common indexing method is the *B-Tree* [BM70] together with its variants B^+*-Tree* and B^**-Tree*. These indexing methods exhibit similar computational complexities for searching and updating and are hence a good choice for OLTP applications. Yet in data warehouse applications such as OLAP, where search operations occur at much higher frequency than update operations, it has been shown

that *bitmap indices* perform better than B-Tree variants [WOS04, WOS06, OOW07]. On the flip side, incremental updates on these indices are known to be expensive. The techniques used to increase read performance decrease at the same time their update performance [WOS06]. However, previous work demonstrated that is in fact possible to efficiently update bitmap indices by leveraging the update characteristics of streaming data [RSW+07]. The authors identified two reasons. First, because historical data is never modified, new data can be appended without disturbing existing data. Second, some indices can be built on unsorted data which allows adding new data in time that is a linear function of the number of new records.

Consequently, we can exploit the high performance of bitmap indices and at the same time handle index updates efficiently. This makes bitmap indices an excellent choice for our application. In §2.3.2, we provide insight about the particular bitmap indexing technology we employ in the VAST system. In the following, we explain how bitmap indices work in general.

Bitmap Indices

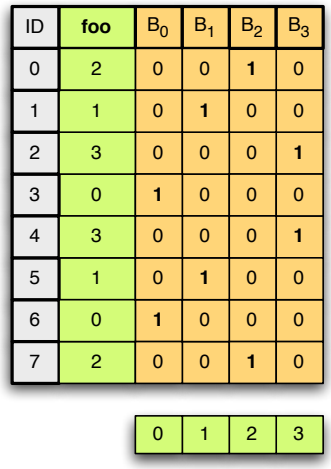
A *bitmap index* [O'N89, WOS06] is an efficient indexing method for mostly read-only data that accelerates multi-dimensional range queries [RSW06]. Because bitmap indices are based on bit arrays (called *bitmaps*), they answer queries by performing bitwise logical operations which are well supported by modern processors. To index a single column in a table, the bitmap index generates c bitmaps, where c is the number of distinct values in the column (also referred to as *column cardinality*). Each bitmap has N bits, with N being the number of records in the table. If the column value in the kth row is equal to the value associated with the bitmap, the kth bit of the bitmap is set to 1, but remains 0 for any other column value.

Figure 2.1 depicts a simple bitmap index for the column **foo** that consists of integers ranging from 0 to 3. Since the column cardinality is 4, the index includes 4 bitmaps, named B_0, B_1, B_2, and B_4. For example, the third and 5th bit of B_3 are set to 1 because the values in the corresponding rows are equal to 3. Answering a query such as "**foo** < 2" translates into bitwise OR (|) operations between successive long-words of B_0 and B_1. The result is a new bitmap that can be used in further operations.

The size of a bitmap index is relatively small for low-cardinality attributes such as "gender" or "month". However, for high-cardinality attributes where each column value is unique in the worst case, the required space grows quadratically with the number of records in the dataset. Clearly, maintaining indices that are much larger than the data itself is not tractable. To reduce the index size, the literature distinguishes three strategies: *encoding*, *compression*, and *binning*. We briefly sketch each technique in the following. A more elaborate discussion can be found in [WK06].

Encoding. To reduce the number of bitmaps or ameliorate query performance, different encoding schemes can be used. The bitmap index from the previous example in Figure 2.1 is *equality-encoded* because each bitmap denotes whether a key equals to a particular attribute value. Hence this encoding scheme proves particularly

Figure 2.1 A bitmap index generated from 4 distinct values.



ID	foo	B_0	B_1	B_2	B_3
0	2	0	0	**1**	0
1	1	0	**1**	0	0
2	3	0	0	0	**1**
3	0	**1**	0	0	0
4	3	0	0	0	**1**
5	1	0	**1**	0	0
6	0	**1**	0	0	0
7	2	0	0	**1**	0

0	1	2	3

apt for *equality queries* such as "host = 192.168.0.1". There exists also encoding schemes that are optimized for *range queries*. While *range encoding* is geared towards *one-sided* range queries such as "port < 1024", *interval encoding* is efficient for *two-sided* range queries such as "1024 < port < 4096".

Figure 2.2 compares equality-encoding to range-encoding for the example query "**foo** < 2". To answer the query for the equality-encoded index in Figure 2.3(a), all bitmaps that correspond to the values less or equal to 2 have to be logically *or*ed, whereas in case of the range-encoded index shown in Figure 2.3(b), only bitmap R_2 has to be accessed. Consequently, range-encoding involves at most one bitmap to be accessed, while equality-encoding involves at worst $c/2$ bitmaps to be accessed, where c is the attribute cardinality. Note that in case of range-encoding, the one bitmap containing only "1"s is usually omitted and only $c-1$ bitmaps are effectively stored.

The *binary encoding* scheme [WLO+85] (also known as *bit-sliced index* [OQ97]) generates only $\lceil log_2\ c \rceil$ bitmaps for c distinct values. For example, an attribute with 12 distinct values requires $\lceil log_2\ 12 \rceil = 4$ bitmaps, where each integer in the range from 0 to 12 corresponds to a binary bitmap from 0000 to 1100. Compared to the interval encoding scheme, binary encoding is much more space-efficient. Yet to answer a query, most of the bitmaps have to be accessed as opposed to interval encoding, where only two bitmaps need to be accessed.

Compression. The second strategy to reduce the bitmap size is compression, a technique that has been extensively studied in the past [Joh99, AYJ00, WOS06]. To compress a bitmap index, each bitmap in the index is typically compressed separately. Two important compression algorithms are *Byte-aligned Bitmap Code* (BBC) [Ant95] and *Word-Aligned Hybrid* (WAH) code [WOS04, WOS06] which employ *run-length*

encoding (RLE) to achieve a reduction in size. A major benefit of BBC and WAH is that they can partake in bitwise operations without decompression, therefore maintaining a good query performance. This is a considerable advantage over generic compression algorithms such as *LZ77* [ZL77] which are slightly smaller but much slower as well.

Theoretical analysis of WAH compressed indices showed that query response time on one-dimensional range queries grows linearly in the number of hits. Since at least the hits have to returned, this result is *optimal* for any search algorithm. While B^+-Trees and B^*-Trees scale optimally as well, they cannot be combined efficiently to answer multi-dimensional ad-hoc range queries.

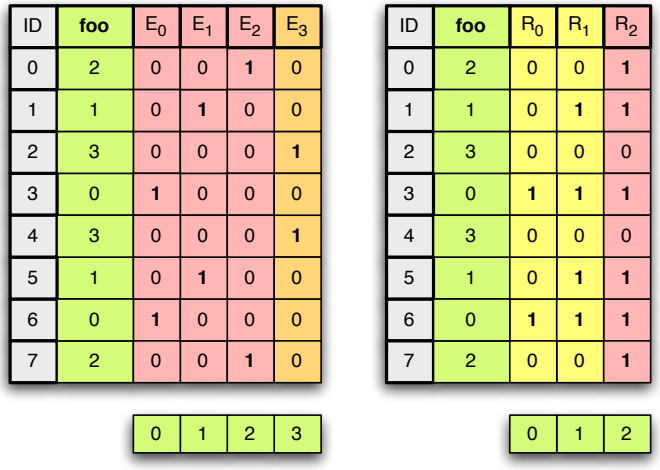
Query response time is determined by both I/O operations and CPU time. For most database operations, CPU time is negligible compared to I/O time. Because BBC compressed indices are smaller in size, it would suggest that they consume less I/O time than WAH compressed bitmaps. However, despite their larger size, WAH compressed bitmap indices were shown to outperform BBC in several measurements [SWS02, WOS01].

Binning. Attributes with very high cardinality are still difficult to cope with, even when employing both compression and encoding schemes. A major problem of dense encoding schemes is that they do not compress well and thus only cost CPU time. In this context, binning represents a fruitful solution as it allows creating indices of pre-determined size, independent of the attribute cardinality. Binned indices generate one bitmap for multiple attribute values instead of producing one bitmap per distinct value. This gives full control over the total number of bitmaps, regardless of the used encoding scheme. However, only a subset of queries can be answered efficiently without examining the original data. Sometimes, returning an exact query answer involves to examine the original records (candidates) to validate that specified condition. Verifying the base data is also referred to as *candidate check*.

For example, consider a bin that covers the values from 42 to 84. A query that asks for values less than 61, all records in the bin are potential hits, even those between 61 and 84. Therefore, the base data has to be examined to perform the candidate check. For high-cardinality attributes, the time required to perform the candidate check could dominate the overall query response time because verifying the base data may cause a high amount of disk page accesses.

There exist a number of strategies to counter the irregular performance of binned indices [Kou00, SWS04, RSW05, RSW06]. One approach is to find the optimal bin size for equality queries [Kou00] and range queries [RSW05]. Since these solutions leverage dynamic programming techniques, they induce a quadratic overhead in the problem size, rendering them intractable for high-cardinality attributes and large datasets. [SWS04] show that binning in fact can accelerate multi-dimensional queries on high-cardinality attributes by reducing the time needed for the candidate check. Further improvements may result from a bin allocation design geared

Figure 2.2 A comparison of *equality-encoded* and *range-encoded* indices.



(a) Equality-encoded. (b) Range-encoded.

towards probabilistic queries with attribute dependencies [RSW06].

Having discussed concepts to realize a unified data model for network activity and to build an efficient scalable storage solution, we now turn to available technologies that implement these concepts.

2.3 Technologies

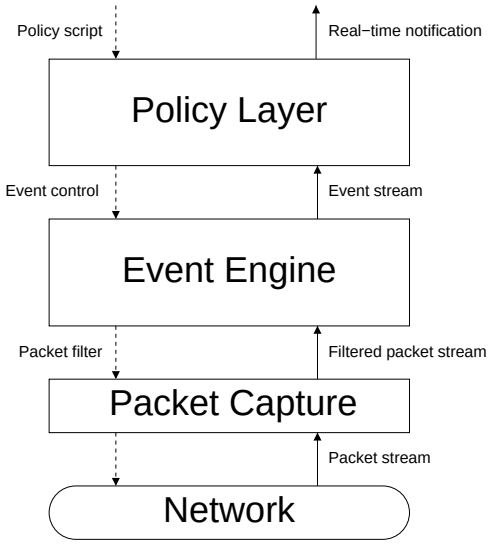
This section presents the concrete technologies that VAST utilizes. In §2.3.1, we illustrate how to realize a unified data model based on policy-neutral NIDS. Thereafter, we introduce VAST's underlying storage engine in §2.3.2, which is capable of archiving high-volume event streams.

2.3.1 Bro

To describe network activity, VAST leverages the event model of the flexible open-source Bro NIDS [Pax99] developed and maintained by Vern Paxson. In the following, we briefly discuss the architecture of Bro and the event generation process. We then turn to the communication framework and client library which gives us a powerful interface to interact with Bro.

Bro has been designed to operate in large-scale networks with high traffic volume. To deal with a copious amount of network data, the system is based on a layered architecture, as shown in Figure 2.3. At the lowest layer, *packet capture*, raw packets from the network interface are received through *libpcap* [Lib]. Not only does employing libpcap enable flexible deployment in various UNIX environments, but also it can greatly

Figure 2.3 Architecture of the Bro NIDS [Som05].



reduce the traffic in the kernel by applying a *BPF* filter expression [MJ93]: rather than copying every packet to the user-space, uninteresting traffic is discarded beforehand in kernel-space, which can significantly reduce the induced load on the system.

The pre-filtered packet stream is then handed up to the next layer, the *event engine*, which re-assembles the packet streams and elicits events. The event engine generates events that represent activity at different semantic levels. Since the resulting events represent a *policy-neutral* snapshot of visible network activity, they perfectly fit into our event model. Some events represent generic protocol-independent activity such as TCP or UDP connection attempts. Others reflect specific protocol activity, e.g. HTTP requests and replies, or high-level notions like a failed user authentication during a login session.

After the event engine has finished the processing, the event stream is passed to the *policy layer*. Using Bro's own rich-typed domain-specific language (DSL), the site's policy is codified in policy scripts that define *event handlers* which are invoked when the corresponding core event is generated. Bro's flexible scripting language allows the definition of custom actions in event handlers. For example, it is possible to maintain state in custom data structures to track activity in the network, raise user-defined events, send out real-time notifications, and execute arbitrary programs (e.g. scanner activity could trigger the execution of a script that dynamically reconfigures firewall ACLs).

Bro is also suited for distributed analysis. In previous work, we built a NIDS cluster [VSL+07] to overcome processing bottlenecks that single-machine installations in large-scale networks inevitably reach. To this end, we leveraged the concept of *independent state* [SP05] to propagate internal fine-grained state from one Bro instance to others. We further investigated hurdles and pitfalls that arose due to the distributed

nature of the system[1]. Yet not only can script-level state be shared, but also core events can propagete among multiple instances, providing a policy-neutral interface to Bro's view of network activity.

To enable third-party applications partaking in the event communication, the lightweight client library *Broccoli*[2][KS05] has been developed, allowing other systems to request, send, and receive Bro events. Using Broccoli, we can instrument VAST with little effort to handle events generated from Bro. Moreover, we can utilize Broccoli to establish a unified event model that can integrate data from a wide range of heterogeneous sources, as we demonstrate in §3.2.2.

2.3.2 FastBit

To archive, index, and query historical data that is sent to VAST, we employ the *FastBit* [Wu05] technology which has been developed by K. John Wu at the Lawrence Berkeley National Laboratory. It is distributed under the open-source LGPL licence since August 2007 and primarily used in scientific applications.

FastBit implements numerous efficient bitmap indexing methods with various encoding, compression, and binning strategies. Most DBMS management systems use B-Tree variants which are efficient for handling typical bank-transaction types of searches where write and update operations prevail. However, B-Tree indices fall short in achieving similar performance for OLAP tasks. For such tasks, bitmap indices outperform conventional indexing methods (see §2.2.2).

A distinctive feature of FastBit is its efficient compression method, the *Word Aligned Hybrid* (WAH) method [WOS04, WOS06]. WAH compression reduces the size of each bitmap in a bitmap index so that the total size of the index remains modest regardless of the type of data. This addresses a fatal limitation of earlier bitmap indices that were only efficient for certain kinds of data fields. Moreover, WAH-compressed bitmaps were shown to have a significantly higher performance than other indices [WOS01, WOS04, OOW07].

Since FastBit has been optimized for data warehousing tasks, it uses a column-oriented table layout in which each table column is stored separately. Hence, FastBit prefers the term *partition* over table. Very big tables are horizontally partitioned as well, each partition containing several million rows. On the filesystem, a partition is represented as a directory that includes a meta-data file and separate files for the data of each column. FastBit can currently only index data types of fixed size (such as integers and floating-point numbers). Arbitrary strings cannot be indexed, however, low-cardinality strings can be converted to integers using a dictionary and thus also benefit from the available indexing schemes.

To minimize I/O operations on index lookups, FastBit stored bitmaps in one index in linear order. This means that a query like "$42 < x < 84$" accesses only bitmaps that represent the values from 43 to 83. The downside of this linear layout is that it does not support efficient update operations. Modifying one bit of a particular index entails

[1] For instance, synchronizing script-level data structures turned out to be a non-trivial task in which race-conditions could distort the distributed analysis.

[2] *Broccoli* is the healthy acronym for "Bro Client Communications Library".

a reorganization of all consecutive bitmaps. For OLAP applications that archive real-time data, this is usually not a problem because once saved, the data is never modified again. VAST does not modify records, but it eventually purges old records to aggregate them into higher levels of abstraction. Based on our suggestions, the FastBit developers recently added the ability to delete records by *deactivating* the records to be removed. Physically removing the data instead would be far too expensive when operating on large datasets and trigger costly index reorganization as well. In contrast, maintaining a special mask that marks affected records as deactivated is a comparatively inexpensive task since it only involves flipping a status bit. This technique also enables support for update operations by first deactivating a specific row and then performing a subsequent append operation, while the actual dataset is still treated as read-only.

2.4 Related Work

In this thesis, we present VAST as an integrated solution to cope with descriptions of network activity that today are both fragmented across space and time. With VAST, network operators and security analysts gain *visibility* of past activity and can formulate future occurrences using a single coherent interface. Based on a policy-neutral data model, the VAST system provides a high level of generecity, allowing the user to customize mechanisms to record, store, condense, and sanitize data.

The term *visibility* has been used in different contexts in the past. Cooke et al. [CMD+06] use the term network-wide visibility to refer to the ability of accessing data in clear at the network-level, which is hardly possible in the presence of encryption and tunneling. To regain visibility, the authors present *Anemone*, a monitoring infrastructure recording per-flow data that is deployed on endpoints.

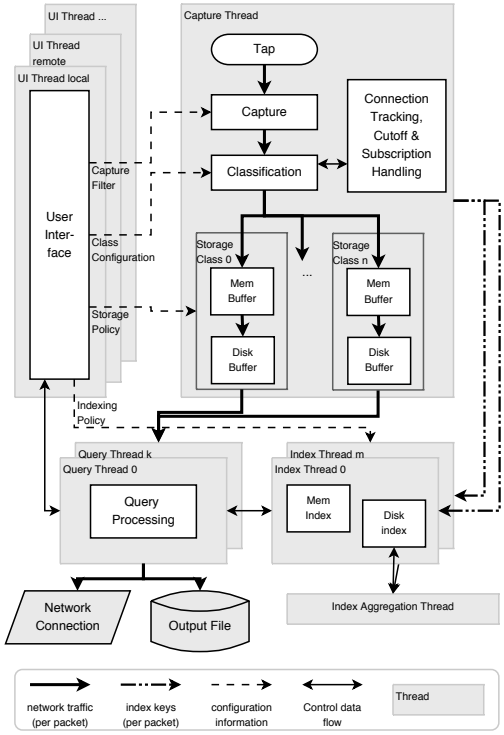
Previous research suggested many different approaches, ranging from large-volume data recording and indexing, real-time data streaming, and network traffic analysis. However, we are not aware of a comparable approach with a similar focus. To our knowledge, VAST constitutes a unique architecture which holds great promise to significantly improve key operational networking tasks such as troubleshooting, detecting intrusions, and mitigating insider attacks. We devote §2.4.1 to the *Time Machine* which has several intersections with our work. In §2.4.2, we summarize various existing approaches in the literature.

2.4.1 Time Machine

The *Time Machine*[3] *(TM)* efficiently records network traffic streams to support later inspection of activity that turns out to be interesting only in retrospect [MSD+08]. Similar to VAST, the TM aims at improving key operational networking tasks — such as troubleshooting and forensic analysis. Not only does it include a user-driven interface for interactive queries, but it also features an automated interface coupled to a NIDS. The latter capability makes it possible to conveniently "travel back in time".

[3]The authors came up with this name well before its use by Apple for their backup system [KPD+05].

Figure 2.4 Architecture of the Time Machine [MSD+08].



By exploiting the heavy-tailed nature of network traffic [PF94], the TM captures nearly all connections in their entirety but foregoes the bulk (heavy tail) of the total volume, effectively storing only a fraction of the full traffic. To this end, a customizable *cutoff* can be specified that causes only the first N bytes of each connection to be recorded. As a result, large connections still contain the most interesting part (e.g. header information, protocol handshakes, and authentication dialogs) but occupy much less space than the full version.

The architecture of the TM is heavily based on a multi-threaded design which allows multi-core CPUs to exploit the separation of archival, indexing, and external interactions. As illustrated in Figure 2.4, users or remote applications like a NIDS spawn a dedicated *User Interface Thread* to handle interactions. Incoming network traffic is captured off a network tap, classified, and then indexed on disk to accelerate user queries. To this end, the *Capture Thread* first records and classifies packets, then assigns each packet to the corresponding *storage class*, which allows the user to define different storage options. As an example, traffic associated with suspicious hosts can be configured with a larger cutoff and retained longer. Each issued query runs in a dedicated *Query Thread*. To accelerate query answers, *Index Threads* maintain index structures to quickly locate and return buffered packets from either memory or disk. The *Index Aggregation Thread* performs additional index housekeeping, such as merging smaller index files into larger ones.

A prototype of the TM is already in operational use at the Lawrence Berkeley National Laboratory (LBNL), a security-conscious research lab with approximately 10,000 hosts and 10 Gbps uplink connectivity. The security staff hitherto employed `tcpdump` to bulk-record traffic for later forensic analysis in case of a security incident. Yet the high traffic volume (up to 1,5 TB/day) cannot be stored in its entirety, forcing the operators to constrain their recordings on a tractable subset. The analysis hence excludes HTTP traffic and FTP data transfers among 10 key services that each constitute a blind-spot in forensic analysis of security incidents. Clearly any volume reduction heuristic faces this fundamental risk for evasion. Confronted with the trade-off of storing a sampled version of the full traffic stream versus recording the beginning of *all* connections, the latter addresses practical problems more adequately. A prototype deployment of the TM at LBNL has proven already an invaluable tool in investigations of numerous attacks and turned out to be particularly helpful with chasing down the ever-increasing number of attacks conducted over HTTP.

Already a preliminary version of the TM showed that high-volume packet recording can be accomplished efficiently [KPD+05]. The new reimplementation provides major performance improvements, a richer user-interface, and coupling with a NIDS that can query the TM for past traffic on demand. In our work, we aspire to elevate these ideas to a more generic level. While the TM provides high-performance network traffic capture, indexing and retrieval, VAST extends these concepts to arbitrary events that can stem from a wide range source. Similar to the TM's architecture, VAST is parallelized from scratch to exploit the performance benefits of modern multi-core CPUs.

2.4.2 Data Management

One major aspect of the design of VAST is the storage component which continuously archives and indexes incoming events for later retrieval. On the storage side, there exist some approaches geared to off-line network traffic analysis [CYBR06, SBUK+05] but these systems cannot handle live streaming data. Similar to applications like electronic trading, sensor networks, and inventory tracking, our work handles real-time streaming data arriving at a constant rate. This type of data is explicitly supported by streaming databases whose requirements have been studied extensively in the past [GO03, ScZ05]. In the following, we briefly sketch some of the existing approaches.

Gigascope [CJSS03] makes use of the fact that packet headers can be well expressed with a relational database scheme and implements a SQL-like query interface that operates on raw network packet streams. Heavy low-level optimizations enable line-rate monitoring with moderate hardware requirements. However, this approach is tightly bound to a small set of custom network monitoring applications, rendering the integration of new queries a tedious task.

MIND [LBZ+05] is a distributed query processing component for wide-area networks that creates distributed multi-dimensional indices over flow data from passive network sensors. *PIER* [HCH+05] follows a similar approach. Yet both systems rely on DHT data structures that cannot keep pace with high insertion rates. Moreover, they operate only on aggregated flow information, rather than on arbitrary data.

The *CoMo project* [IDM04] provides a network monitoring infrastructure featuring both live and retrospective queries on streaming data. In a distributed setup, multiple CoMo systems can propagate queries to pinpoint relevant data locations in the network. Despite its high-speed layout, network traffic is stored in a circular on-disk buffer which imposes functional limitations.

TelegraphCQ [CCD+03] builds an adaptive stream processing engine for high-volume data streams based on the open-source database PostgreSQL. Recently, the system has been extended to allow querying both historical and real-time data simultaneously [RSW+07, CF04]. To handle the expensive I/O operations for fetching historical data, the approach retrieves only a sampled version of the incoming data stream.

An interesting hybrid approach follows *Hyperion* [DS07], a system for archiving, indexing, and on-line retrieval of high-volume data streams. Their write-optimized streaming file system, StreamFS, poses an alternative to the traditional relational database storage mechanisms. Alas, the low-level programming and file system implementation is only available for the Linux platform.

It is the main problem of all existing approaches that they do not feature *graceful* data reduction techniques. Portions of relevant data or network traffic is completely discarded instead of kept partially at a semantic higher abstraction. Moreover, VAST *by design* supports generic event data in contrast to approaches that operate only on a specific data type such as network traffic and thus greatly reduce their applicability.

Timely answering queries is a non-trivial task due to the intricate nature of both queries and data. Bitmap indices are a powerful method to accelerate queries, but only few DBMSs provide implementations because there exists no definitive design for this index type [OOW07]. While ORACLE [ORA] includes support for bitmap indices, other major DBMS like *IBM DB2* [IBM] and *Microsoft SQL* [MSQ] do not ship with an implementation. They only create bitmaps as an intermediate result during hash joins and do not use them as general means to index data. *Sybase IQ* [syb] offers a competitive bitmap indexing technology for commercial OLAP applications. In our work, we employ FastBit (see §2.3.2), an open-source bitmap indexing technology that proved efficient in several measurements [SWS02, WOS01]. *RIDBit* [Rin02] is another bitmap index technology that was initially developed as an exercise for a database course. It uses N-ary storage to structure table rows and implements a graceful switching between B-tree and bitmap indices. Although RIDBit indices are smaller in space, FastBit can answer more queries in the same time requiring fewer I/O operations [OOW07].

Previous work assessed the performance of more traditional indices such as B-Trees and hash indices for streaming applications, but the update performance turned out to be too poorly to use them in real-time scenarios [CF04]. In order to remain sorted, B-Tree indices and their variants require to re-examine existing data in addition to the new data. The required time to update indices is hence a super-linear function of the new data to be added. While hash indices induce less CPU load for incremental updates, they produce a substantial quantum of random I/O. Further, hash-based indices exhibit low performance on grouped aggregation and range queries, operations that many applications need to handle efficiently.

3 The VAST System

Having understood the benefits of unifying heterogeneous descriptions of network activity across space and time, in this chapter we present the architecture and implementation of *Visibility Across Space and Time* (VAST), an intelligent database that processes network activity logs in a comprehensive, coherent, and scalable fashion. VAST accumulates data from disparate sources and provides a central vantage point to facilitate global analyses in an integrated manner.

After framing objectives in §3.1 that serve as important guidelines during design and implementation, we present the architecture of VAST and its components in §3.2. We close this chapter by pointing out interesting aspects of the implementation in §3.3.

3.1 Objectives

Our work is fundamentally based on the principles of comprehensive network visibility developed by Allman et al. [AKP+08]. In §3.1.1, we present some of these aspects which are relevant in the scope of this thesis. Thereafter, we outline in §3.1.2 the practical challenges we face with our proof-of-principle implementation.

3.1.1 Principle Guidelines

To build a unified network monitoring infrastructure, the authors of [AKP+08] identified several design guidelines from which we illustrate a relevant subset in the following.

Data Breadth. The clear benefit of unifying activity draws from the ability to incorporate *any* existing data source into the monitoring infrastructure. It must be able to seamlessly integrate from numerous systems and devices, such as syslog daemons, intrusion detection systems, web servers, firewalls, routers, and embedded devices.

Large Measurement Window. Since attacks can span over long time periods, the effectiveness of the monitoring infrastructure is greatly determined by its ability to search past data. A crucial factor for successful forensic analysis is the richness of available information. The quality as well as the quantity of accessible data have a high impact on the results of the investigation. Consider insider attacks that can manifest over long time intervals. When trying to understand the full extent of a breach, a comprehensive archive that reaches back for a long time could reveal initial attempts to hide the abuse or uncover similar non-authorized activities.

Not only does a large measurement window help to identify attacks that span a long time period, but it also provides a means to uncover any kind of attack

conducted a very long time ago. Nearly all intrusion detection systems operate in real-time today and discard the incoming data after processing it. However, an extensive archive allows re-inspecting activity that became only interesting in retrospect, long after the incident happened.

Storage Management. The never-ending stream of data will eventually consume all available storage capacities. In order to continue the archiving process, old data must yield space for the new arriving data. At the same time, the measurement window should remain as large as possible. To resolve these conflicting goals, operators need the ability to configure smart data retention policies. For example, the Time Machine (see §2.4.1) can buffer network traffic for several days and allows operators to control the stored data volume by specifying a cutoff limit to skip the heavy tail of large connections, yet record most connections in their entirety.

Graceful Degradation. Instead of completely expunging old data, degradation mechanisms should gracefully transform data into more compact forms that still contain useful information. This type of aggregation allows powerful *aging* schemes that can help to cope with a high-volume event-stream over a long period of time. For example, old packets can be rolled into connection summaries. When the next aging step is due, these summaries can further be elevated to traffic matrices that describe the connection paths along with the total data volume exchanged in the communication. Despite the loss of details over time, the abstracted information can constitute the missing link in a chain of previous activities.

Common Analysis Procedure. The monitoring infrastructure does not only provide insight into past activity, but also allows issuing prospective queries that manifest as *event feeds* which constantly deliver qualifying results to their subscribers. Both aspects should be approached with the same procedures to overcome the fragmentation in time of analysis methods.

From a practical perspective, integrating both historical and live queries in a single mechanism has a tremendous advantage. On the one hand, patterns of past activity can indicate what to search for in the future. On the other, false positives can be greatly reduced by testing newly proposed policy rules against previous network behavior.

3.1.2 Technical Challenges

Our approach can be regarded as a fusion of conventional DBMSs and DSMSs, thus forming a *hybrid* system. Consequently we face challenges and limitations of both technologies. It is an ambitious task to merge the functionality of both approaches because the approaches are optimized for different application domains.

Real-Time Nature. All components that interface with VAST provide their information in form of a continuous stream of events. As a single sink for incoming data, the system must provide enough resources to archive the arriving events quickly and entirely, without foregoing new data.

High Concurrency. A daunting challenge is to adequately process the incoming event stream and at the same time offer enough resources to perform parallel tasks, such as answering retrospective queries, dispatching event feeds, reorganizing index structures, and constantly aggregating old events to optimize available storage capacities. Consequently, we emphasize the need for a *concurrent* and *asynchronous* system architecture. Exploiting modern multi-core architectures to a maximum extent should not only involve the use of threads, but also consider task-based parallelization strategies.

Compatibility. Components that provide their information to the VAST system may a priori not support the used event model. But since the fundamental benefit of the system is based on the integration of inhomogeneous data sources, a mechanism is required to augment external components with the ability to partake in the communication of our event model. When direct manipulation of the data sources is impossible, a conversion utility should still be able to gather the generated output and transform it into events without losing semantic content.

Usability. VAST differs from traditional DBMSs or DSMSs in that it suffices to support a custom query language which represents only a subset of full SQL. Both Network operators and remote applications interact with VAST. With regard to human interaction, the user interface must be amenable to convenient interactive analysis. Interaction with remote applications requires a delivery mechanism over the network. Ultimately, the infrastructure should support exchange of arbitrary events between peers that support communication in the underlying data model.

Commodity Hardware. Finally, the system should not require expensive custom hardware to achieve the desired performance. Instead we prefer to employ commodity hardware to enable cost-effective deployment. Ideally, performance bottlenecks can be countered by adding a new machine that overtakes a dedicated task to unburden a component that reached its resource limits.

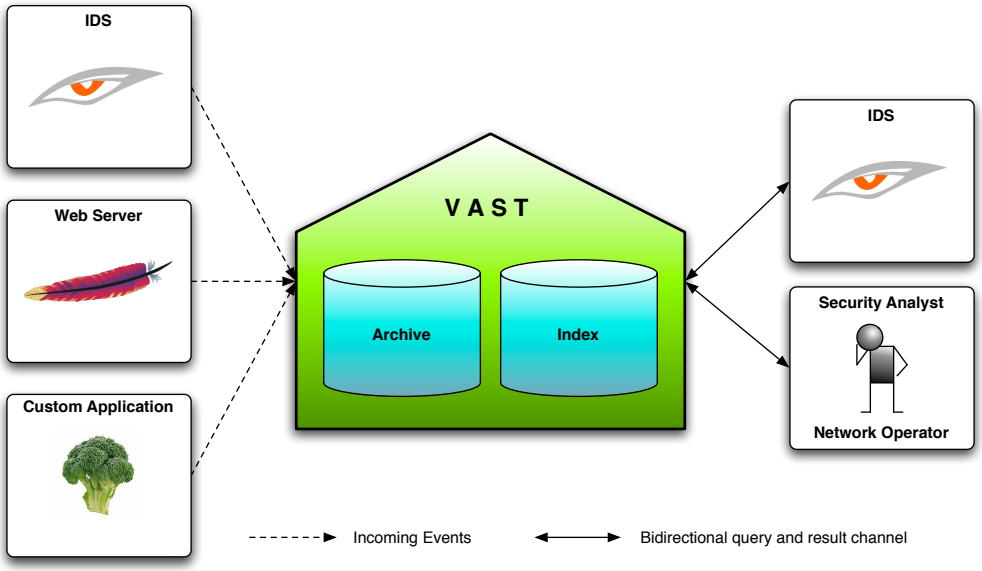
3.2 Architecture

In this section we present the architecture of VAST. First, we give a high-level overview of the system in §3.2.1. We then discuss in §3.2.2 the unified data model used to describe network activity. In this context, we present the *event specification* as mechanism to express heterogeneous types of data. Thereafter, we describe how VAST archives events in §3.2.3. After illustrating the data retrieval process in §3.2.4, we finally turn in §3.2.5 to aggregation and bookkeeping tasks which are constantly executed in the background.

3.2.1 Overview

VAST is a scalable, distributed system that archives events from a wide range of sources and offers a query interface to extract a stream of events. As illustrated in Figure 3.1,

Figure 3.1 Architecture of VAST.



an event source could be a NIDS, web server, logging daemon, or an arbitrary *Broccoli-enabled* application. VAST archives the incoming event streams by storing a raw copy in the *archive* and indexing the event details in a separate *index*.

An interactive query interface allows users to extract historic activity in form of events. For example, the interface supports retrospective queries like "which events include activity of IP address X today" or "which events in the last month included access of URL Y". The query result consists of a list of events that match the given condition. In a further step, the user can inspect and parse the event to extract the desired details.

Further, users should have the ability to install *live queries* that remain active. By subscribing to a specific event pattern, such as "all URLs of this form", VAST instantiates a filter for all future events matching the given condition. The corresponding query is registered as an *event feed* which continuously relays the qualifying events to the user or a remote application.

3.2.2 Data Representation

In order to deal with inhomogeneous components, we need a generic data model that can represent various types of information. As presented in §2.2.1, a particularly apt model for this purpose is the *publish/subscribe* scheme which is based on asynchronous communication using *events*. In this model, interacting peers publish the availability of an event and subscribe to events published by peers according to local interest.

The open-source NIDS *Bro* implements this event model (see §2.3.1). Our motivation to employ this particular model in VAST is threefold:

Unified Data Model. Bro ships with *Broccoli*, a flexible client-library to outfit third-party

applications with the capability of communicating in the same event model. The ability to transform internal state of arbitrary devices into events that can be sent to VAST yields a powerful mechanism to unify the communication of heterogeneous components.

To illustrate, consider the event `http_request` raised by a NIDS watching network traffic. The ability to instrument a web server in addition to generating the syntactically same event allows operators to write generic event handlers that can process multiple events from previously incompatible sources.

In-depth Network Visibility. By passively analyzing network traffic, Bro already provides a policy-neutral snapshot of network activity. Broccoli augments this network-level view with an additional host-level context. Since VAST can receive events from both Bro and Broccoli-enabled application, we can elevate the spatial dimension of network visibility to a very detailed extent.

Transparent Integration. VAST acts both as sink and source of events, that is, it represents a *transparent* actor in the network since events can be sent to and received from the system in a uniform fashion. This flexibility is particularly beneficial when integrating VAST into larger setups that span across the boundaries of the local administrative domain.

Event Specification

Operating on events presupposes that we understand their full semantics. The Bro event model features events that consist of a name and zero or more typed *arguments*. To minimize the space-overhead when sending events across the network, only the event name, argument data, and type information are transferred, but not the argument names. We refer to these compact representations that do not contain the argument names as *raw events*.

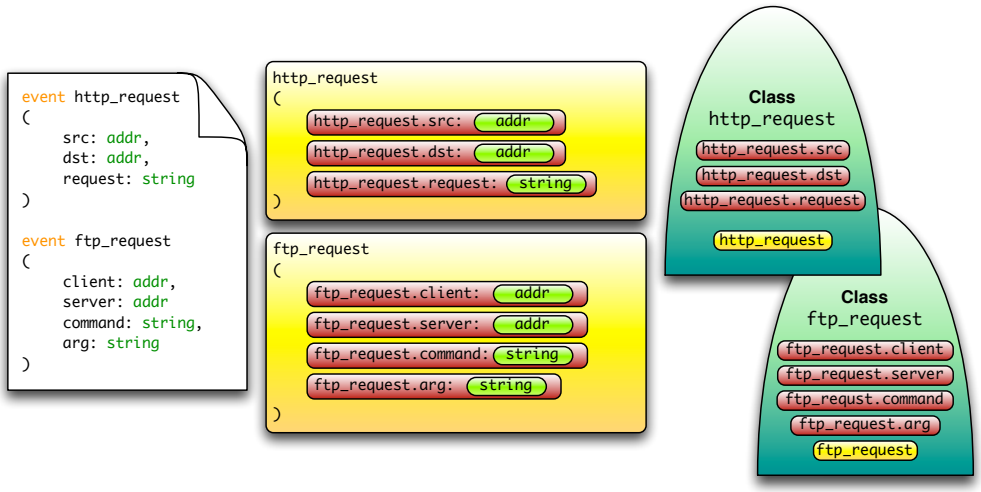
The missing semantic context is codified in the *event specification*, a document that VAST maintains to manage event meta data. Separating the event structure from its semantics gives us a flexible mechanism to change the meaning and behavior of events when they are processed. For example, it is possible to rename arguments, change the way they are indexed, or link together similar arguments of different events.

The VAST specification language is similar to the Bro scripting language and offers *(i)* primitives types such as `addr`, `count`, `string`, etc. *(ii)* compound types such as records, tables and sets, and *(iii)* argument *attributes* to customize the structure and behavior of events ad libidum, e.g. to rename an argument, assign it to a different class or discard it completely.

To illustrate, consider the synthetic event

```
event activity
(
    sender: addr,
```

Figure 3.2 Example event specification.




```
    receiver: addr
)
```

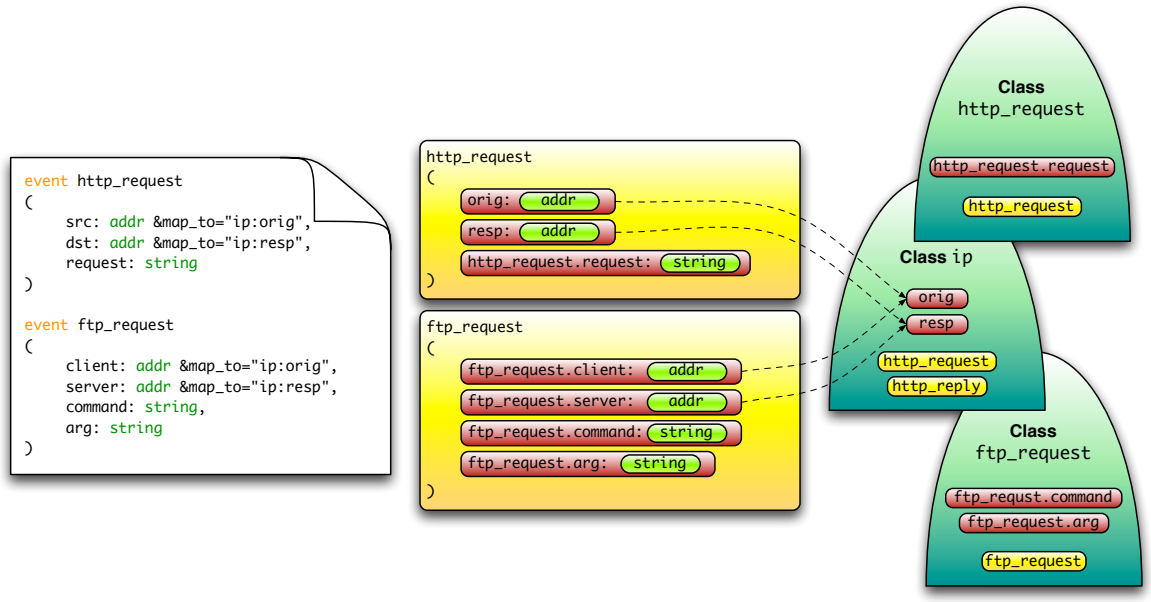
which could be raised by a NIDS whenever a certain type of activity is detected. The event contains two arguments, `sender` and `receiver`, that are both of type `addr`. When Bro sends an `activity` event to VAST, the arriving raw event only allows deducing that both arguments have type `addr`. Due to the lacking meta information, it is impossible to interpret event arguments semantically. But exactly this meta information is required to construct a detailed archive of activity and motivates the event specification.

To group related or similar semantic activity, an event is assigned to zero or more *classes*. A class consist of a list of arguments and the contributing events. By default, every event generates a separate class because a computer cannot deduce semantic relationships between events reliably.

Figure 3.2 exemplifies this concept. The document on the left is the textual representation of the event specification. To declare an event, the `event` keyword is used followed by the event name. Thereafter, a comma-separated list of zero or more arguments can be specified, enclosed by braces to denote the event context. Each argument consists of a name and a type which are separated by a colon. The two rectangles in the middle of Figure 3.2 abstract the internal representation of the events. The right hand side depicts the generated classes, which illustrate the default case where the classes have the same structure as the events.

The example has one problem, though. From a user perspective, the query "which events from the last hour involve communication between host A and host B?" should ideally consider both event `http_request` and `ftp_request`. The corresponding query that connects (or *joins*) the two events would be similar to:

Figure 3.3 Changing argument semantics using the `&map_to` attribute.



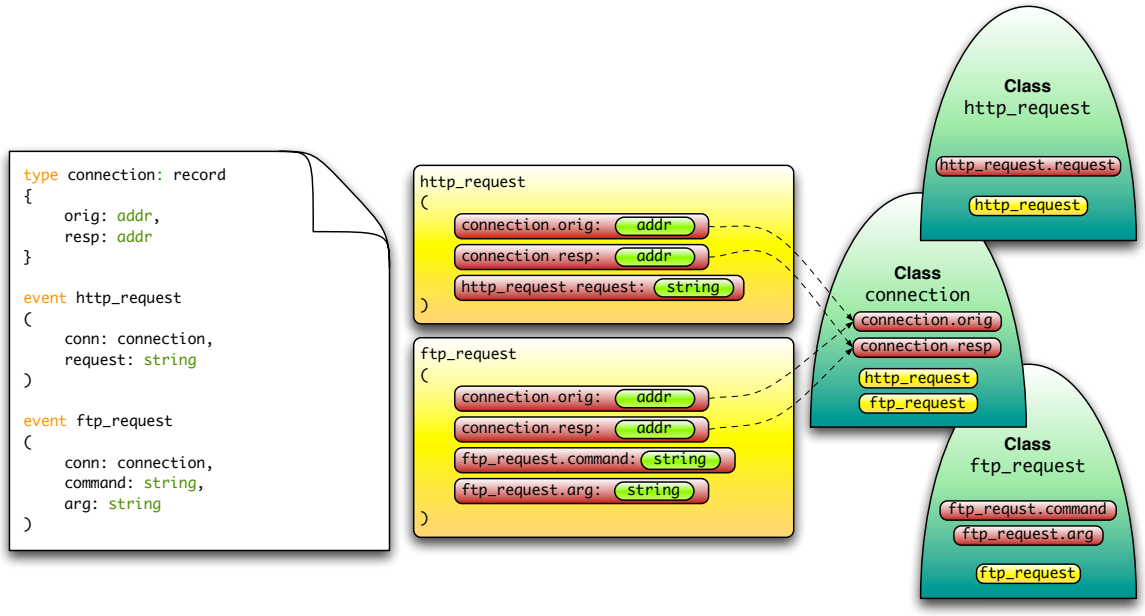

```
(http_request.src = A OR ftp_request.client = A)
                        AND
(http_request.dst = B OR ftp_request.server = B)
```

With hundreds of events, this explicit connection *at query time* not only introduces a significant overhead for the query issuer, but is also quite error-prone. Therefore, the event specification allows factoring out related information. The following change in Figure 3.3 uses the `&map_to` attribute in the specification to create a separate class `ip` which holds IP endpoint information. In general, attributes are prefixed with an ampersand (`&`) and can be appended after the argument type declaration. In our second example shown in Figure 3.3, we appended `&map_to="ip:orig"` to the `src` argument of the event `http_request`. The `&map_to` attribute consists of two parts. The part before the colon indicates the new class, whereas the part after the colon represents the new argument name.[1] Although not explicitly listed as such in the specification, the class `ip` has the following internal structure based on the above mappings:

```
class ip
(
    orig: addr,
    resp: addr
)
```

[1] Either class or name are optional, but at least one of them, including the colon, must be given.

Figure 3.4 Using records to group shared data.



Our initial query now looks much simpler and expresses the intent of the user in a more natural fashion:

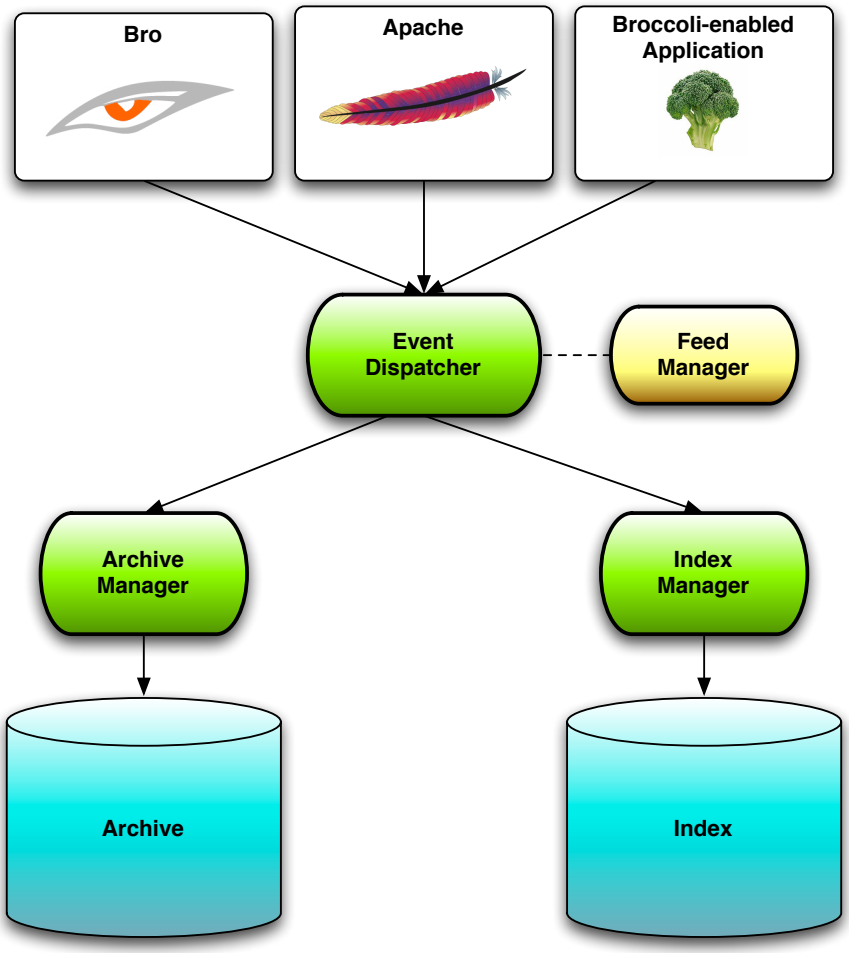
```
ip.orig = A AND ip.resp = B
```

While the `&map_to` mechanism is particularly useful to fine-tune the semantic behavior of individual arguments, *global* types offer a more general means to group and refer to shared data across events. A slight variation of the previous example is illustrated in Figure 3.4. Here, we introduced a new global record `connection`. The record consists of two arguments, `orig` and `resp`, that represent communicating IP endpoints. Ultimately, the effect in this example is the same as with the `&map_to` attribute. In contrast to `&map_to` which does not tamper with the structure of the involved events, defining a global record changes the argument type and thus the event format. Therefore, such a change is only possible when having control over the event generating application.

3.2.3 Event Archival

A high-level overview of the event archival process is visualized in Figure 3.5. All three components involved in the archival process are designed to operate in a distributed fashion. That is, each component can be deployed on a separate machine, communicating over a network connection. Because each component is self-contained and has an associated event queue, scaling components across multiple machines is naturally enabled by the system design.

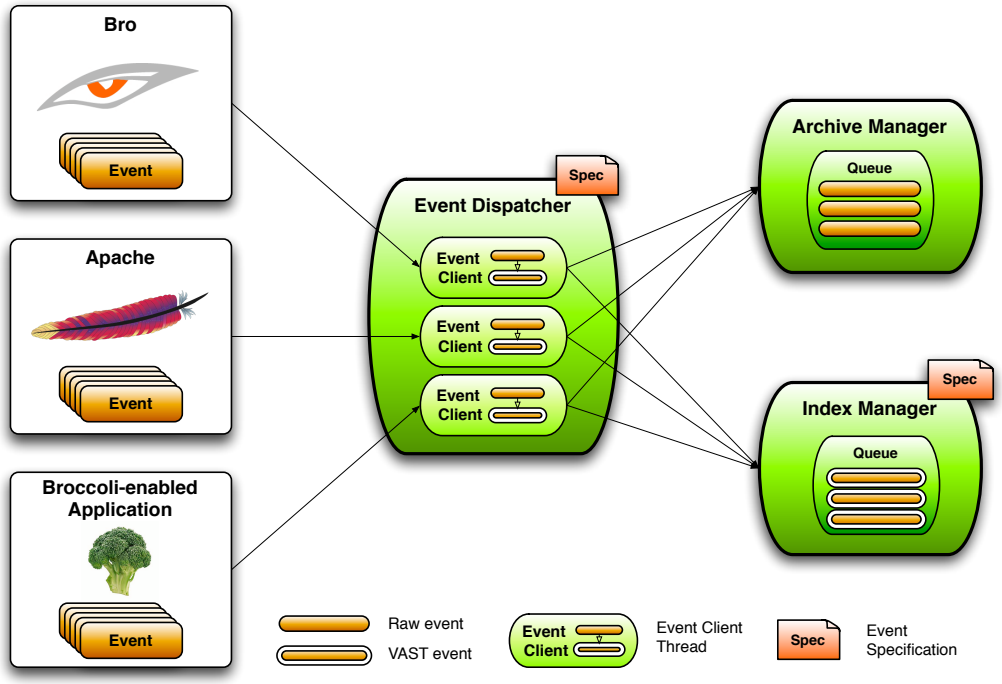
Figure 3.5 High-level view of the event archival process.



In order to interact with VAST, event producers connect to the *event dispatcher* and send their events through a Broccoli communication channel. The dispatcher assigns to each incoming event a unique monotone 64-bit identifier and creates two copies of the event. The first copy is the *raw event*, an unmodified version of the serialized Broccoli event augmented with the obtained 64-bit ID that is compressed and forwarded to the *archive manager*. The second copy is the *VAST event* which is converted from a raw event into an internal data structure. It is forwarded to the *index manager* which accesses the event arguments through a visitor interface. We will discuss the *feed manager* later in the context of live queries (see §3.2.4).

A more detailed picture of the dispatcher is show in Figure 3.6. Every incoming connection spawns an *event client* which is a dedicated thread per client in the event dispatcher. Upon instantiation, an event client requests event delivery for all events that appear in the event specification. To this end, the event dispatcher needs access to the

Figure 3.6 Parsing and dispatching arriving events.



specification. For each incoming event, the client executes a handler in which it *(i)* requests the event ID form the dispatcher, *(ii)* creates the raw event, and *(iii)* creates the VAST event. Finally, the events are inserted into the queues of the archive manager and index manager. In a distributed setup with components on different machines, enqueuing events corresponds to sending both, the raw and VAST event, over the network.

The archive represents a comprehensive record of all events that have ever been sent to VAST. It stores events unmodified, i.e., in their serialized form as obtained from Broccoli. Archiving raw events rather than VAST events has the advantage that query results can directly be sent over the network using Broccoli. The archive consists of a collection of *containers* offering a table-like interface to append data. Figure 3.7 visualizes an archive and index container. The former has two columns to store evens associated with their ID, the latter has an ID column and one column for each argument of the class that this container represents.

Slicing: Horizontal Data Partitioning

Common among archive and index is a specific *slicing* scheme. A slice is an abstraction that maps an object with an identifier to a specific bucket. The archive uses *event ids* as identifier and the objects are *containers*. The index uses *timestamps* as identifier that map a *set of containers* (also referred to as *index slice*). Slicing is similar to the *binning* strategy employed by FastBit, only at a higher level of abstraction.

Figure 3.7 Archive and index containers.

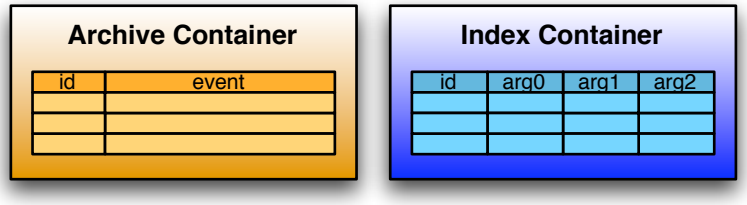
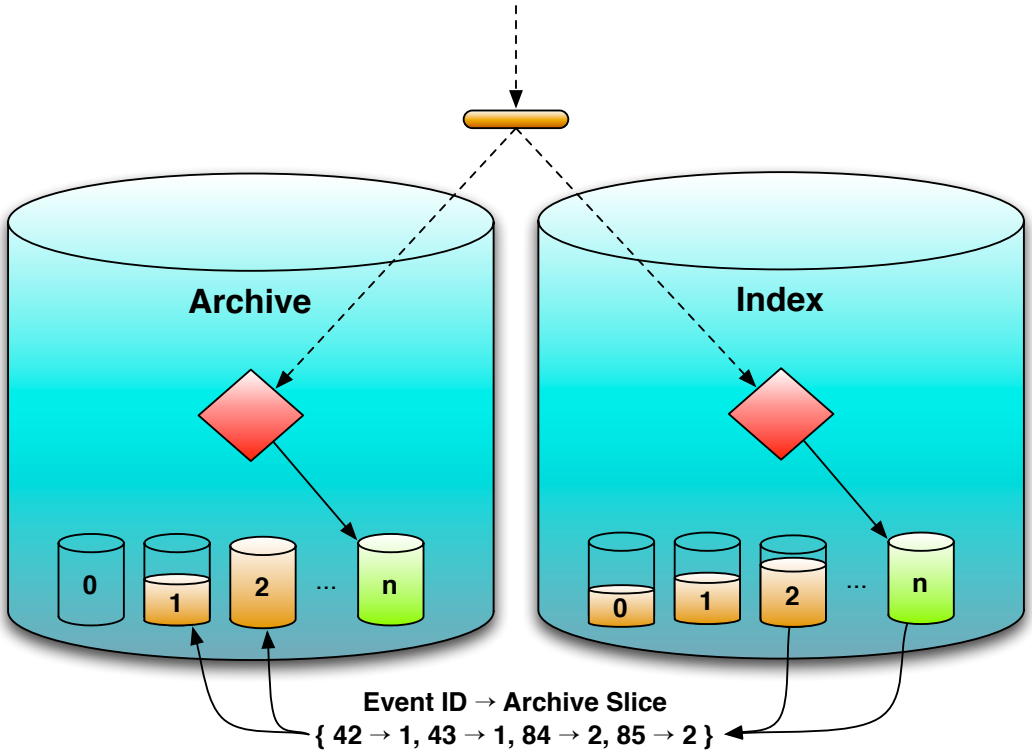


Figure 3.8 Slicing of archive and index.



This technique partitions data horizontally. As illustrated in Figure 3.8, an incoming event is both sent to the archive and index, where the corresponding slice is determined. Because object identifiers are monotone, the event is usually appended to the last slice.[2] When looking at the index to find specific events, the list of returned event ids allows us to efficiently locate the corresponding events in the particular archive slice (see §3.2.4).

Slicing helps only to bypass FastBit's maximum container size, but we also expect to use it in conjunction with VAST's aging and aggregation framework to expire slices

[2] Due to multi-threading or components distributed across multiple machines, it is possible that *out-of-order* events arrive that have to be sent to a previous slice.

with no more valid entries.[3] The slice interval is customizable by the user and denotes the maximum number of events to be stored per container for the archive, and a time interval in seconds for the index. For example, if the slice intervals were 20,000 and 60, an archive slice would hold 20,000 events and all events arriving in the same 60 seconds interval would belong to the same index slice.

Pluckers: Indexing Events Concurrently

When issuing queries like "which events include hosts that accessed IP address X more than n times in the last hour", we usually formulate a query condition that refers to a particular event argument. In the above example, we are interested in events that have an argument that represents a destination IP address. One approach to extract the relevant events would be to iterate over the entire archive and inspect each event. This is clearly an inefficient strategy for large event archive. To improve the query performance, we therefore index each argument value by associating it with the corresponding event ID.

Continuing the above example, consider the fictitious event `transfer` with the following structure:

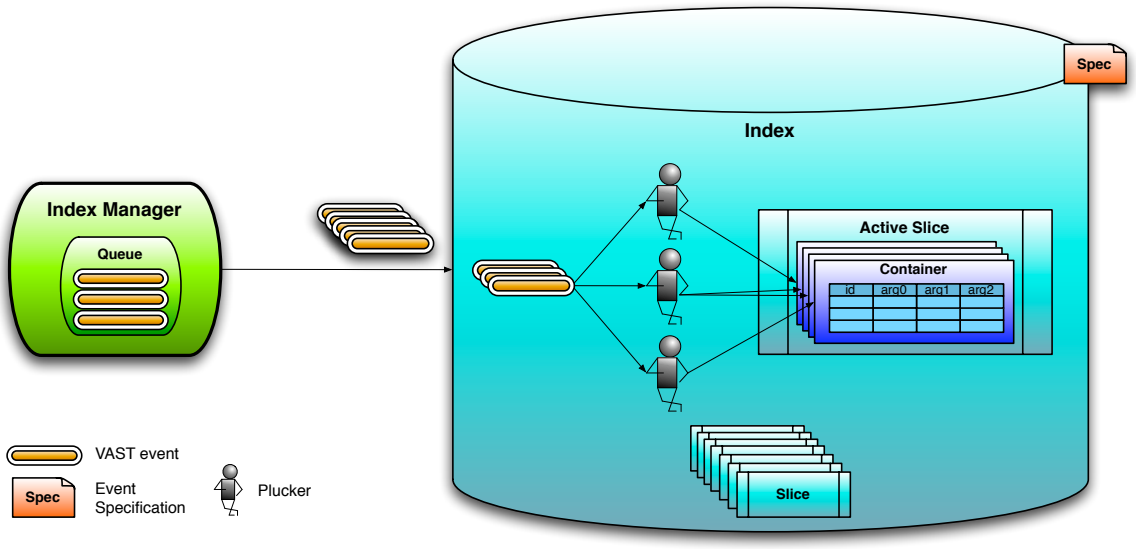
```
event transfer (
    from: addr,
    to: addr,
    data: string
)
```

An event with such a simple structure is translated to a class that consists of the same arguments. The corresponding container would contain columns for the three arguments, plus one column for the event ID. That is, the column names would be `id`, `from`, `to`, `data`. When indexing an instance of this event, the ID and each argument are written in one container row. Unfortunately, we encounter also more complex scenarios in practice: events can include arguments which belong to several classes (e.g manually remapped arguments).

To counter these intricacies, we introduce the *plucker*, a dedicated entity to distribute all arguments of one event to the corresponding containers. Upon slice creation, the plucker consults the event specification to generate a list of containers that have to be accessed for a particular event. When an event arrives, the plucker performs the following steps, as illustrated in Figure 3.9. First, it sends the event ID to all containers that have to be accessed for the given event. Second, the plucker iterates over the event arguments and writes the data of each argument to the corresponding container. Similar to a database transaction, the plucker finally commits the involved containers when all arguments have been processed successfully.

[3] At the time of this writing, aggregation and aging only exists conceptually. In this context, slicing helps us to overcome the append-only nature of FastBit partitions by deleting a partition entirely after all its entries are marked as invalid through an aggregating process.

Figure 3.9 Indexing events.



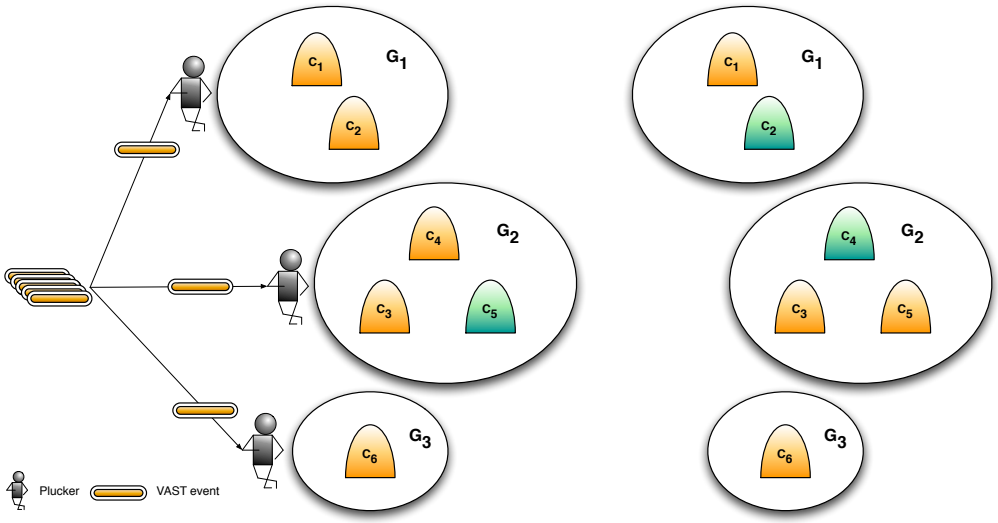
Another benefit of the indirection introduced by the plucker is the ability to distribute event indexing in a *lock-free* fashion. In high-volume environments, the queue of arriving events can grow very fast. To cope with the high volume, we parallelize the indexing process by indexing multiple events simultaneously. Alas, one event can be scattered across multiple containers and while being indexed, all involved containers are locked until the plucker commits the event. Thus, employing multiple consumers is dependent on the *fan-out* of the event specification: the more shared classes exist among events, the smaller is the potential for parallelization in a lock-free fashion.

Groups

Upon loading the specification, we know which containers need to be accessed for which events. We divide classes into *groups* that bundle events with "physical" relationships. That is, events which are part of multiple classes need synchronized disk access to avoid race conditions that could compromise the container consistency. Consequently the number of groups indicates the potential degree of parallelization when indexing events. Consequently, the number of groups represents the degree of concurrency we get based on a given event specification.

Generally, one plucker serves one group and at least one plucker must always exist. Dependent on the number of groups and available system resources, further pluckers can be instantiated, but at most as many pluckers as groups exist. When using multiple pluckers, the list of events known to arrive is partitioned: each event is assigned to a group. Based on the event name, the index then routes the event to the appropriate plucker.

Figure 3.10 Lock-free indexing.

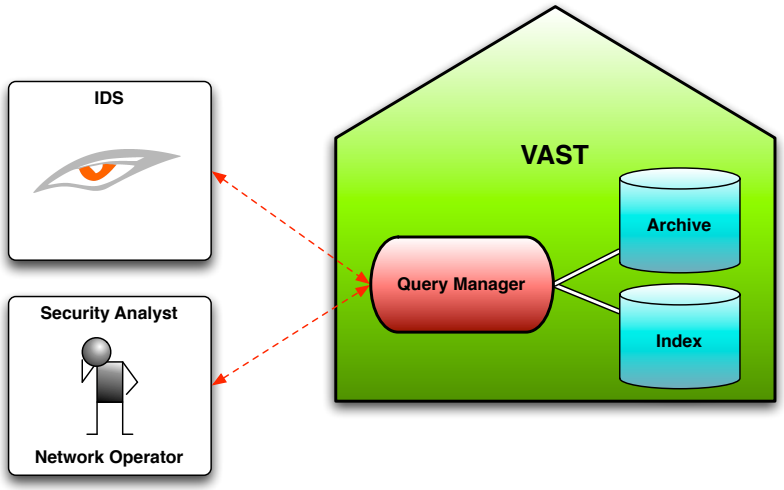


(a) First view. (b) Second view.

Figure 3.10 illustrates this aspect. In this example, three groups, G_1, G_2, and G_3 contain the disjunct sets of classes $\{C_1, C_2\}$, $\{C_3, C_4, C_5\}$, and $\{C_6\}$. Depending on the arriving event, different classes are accessed in each group. Figure 3.11(a) shows the access patterns for one arbitrary event and Figure 3.11(b) for another. In both Figures, optimal group *utilization* is achieved in group G_3 which includes only one class. Events that belong to this group fall in one single class. In contrast, group G_1 includes two classes because at least one event in this group is scattered across C_1 and C_2. While an event of G_1 event is being processed, no other events that belong to either C_1 or C_2 can be indexed. In Figure 3.11(a), an incoming event assigned to group G_1 is part of both C_1 and C_2. A different event, arriving at a later point of time, might however only be part of C_1, as sketched in Figure 3.11(b). Class C_2 is then in an idle state, i.e. not accessed. Group G_1 does not achieve optimal parallelization because some events lock only a subset of all classes. Finally, group G_2 includes an example where one event touches two of three classes (C_3 and C_4) and another uses two different classes (C_3 and C_5). This would mean that one class remains always idle, yielding a sub-optimal utilization of the group.

As a result, the potential degree of lock-free parallelization is heavily dependent on the structure of the events. For instance, most of the events that arrive via Bro contain a connection record as first argument. All events that share this argument are effectively assigned to one group and thus cannot benefit from this optimization. Note that VAST is designed to incorporate events from numerous sources. In a heterogeneous environment with multiple event producers, we expect to gain significant advantages using this scheme.

Figure 3.11 Schematic view of the query architecture.



3.2.4 Event Retrieval

VAST offers operators and security analysts a centralized vantage point where they can retrieve information about past and future activity. Moreover, remote applications such as a NIDS can interact with VAST to obtain a policy-neutral event stream to analyze or subscribe to a specific set of events that are forwarded when they enter the system. To issue a query, a client connects to the *query manager*. This component provides a well-defined interface to access archive and index. We will cover the query manager in-depth later in this section.

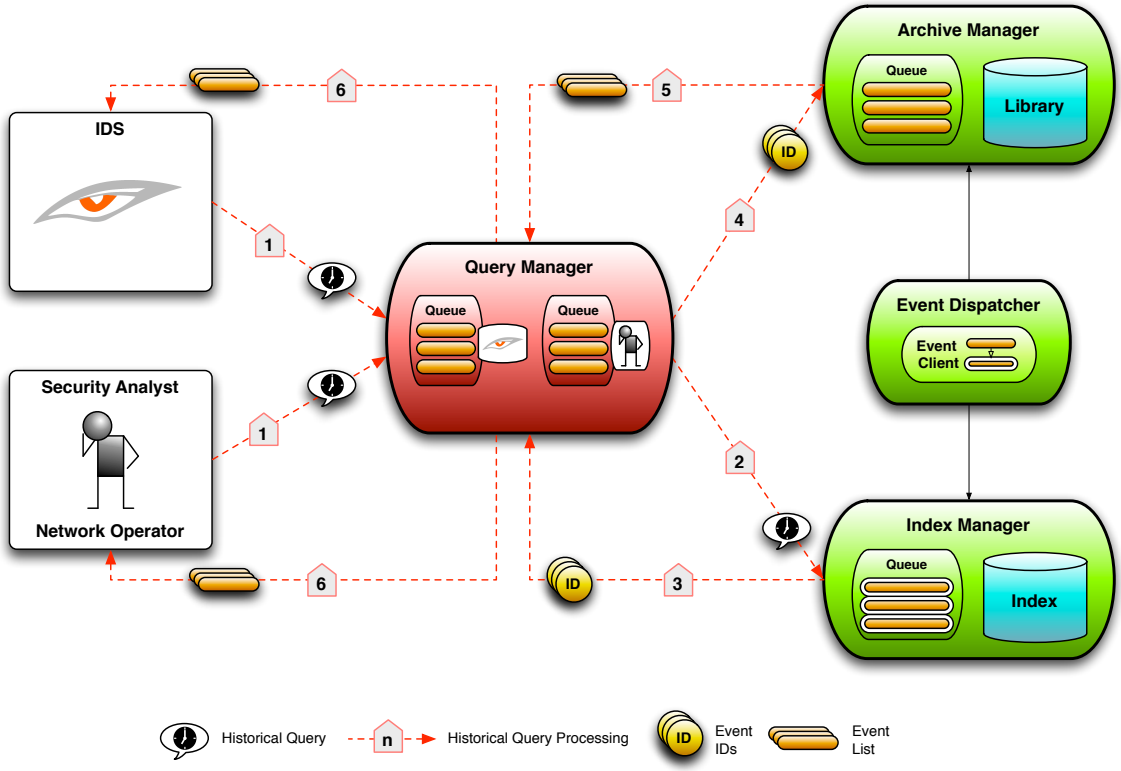
Queries sent to VAST are represented as events as well. This approach has the advantage that we can employ Broccoli to create an asynchronous, bi-directional communication channel. While the client uses this channel to issue queries, VAST uses the same channel to send the qualifying results to the client.[4]

In general, we distinguish two types of queries: *historical* and *live* queries. The former type consults the event archive to return a set of existing events. The latter type does not access the event archive, but instead creates an *event feed* which automatically forwards new events to the user if they match the query condition. It is also possible to issue a combined query that first delivers historic results and then stays active to deliver future events that match the condition.

Furthermore, a client can specify a query target different from itself to redirect the query result to a different machine. For example, this proves beneficial when an analyst issues a manual query to feed a stream of events into an intrusion detection system. In the following, we first discuss the design of historical queries and thereafter turn to live

[4] Query events take a different data path into the system than events that are archived and indexed. Rather than connecting to the event dispatcher, clients that issue queries connect to the query manager which listens on different port.

Figure 3.12 Historical query processing.



queries.

Historical Queries

The schematic process of a historical query is sketched in Figure 3.12 and involves the six following steps.

1. Both users and remote application can issue queries by creating a *query event* locally and sending it to the query manager. Each incoming query is associated with a dedicated *query queue* that is processed asynchronously to deliver matching events back to the client.

2. Executing the query instructs the query manager to parse the event, extract the query condition and access the index to obtain a list of event ids that match the condition.

3. The index evaluates the condition and sends a list of matching ids back to the index manager.

4. If the list was empty the query manager signals the client that the query did not yield a result and terminates the connection. Otherwise, it consults the archive to extract the events with the given list of ids.

5. The archive manager then sends the extracted events back to the query manager which inserts them into the corresponding query queue.

6. As soon as events are inserted into the client queue, the query manager forwards the events over the existing Broccoli connection to the client.

Live Queries

In contrast to historical queries, *live queries* do not consult the event archive. Instead, they install an *event feed* which continuously relays matching events to the query issuer.

On the client side, the only difference to historical queries is a flag indicating the query type. On VAST side, live queries differ significantly from their historical counterpart. While historical queries can simply be converted into one or more index lookups, live queries exhibit a different nature. Instead of "pulling" data from disk, a constant stream of events is "pushed" into the system. This subtlety has significant implications for the design of the live query architecture. In particular, a very high or bursty event arrival rate can induce an unpredictable workload and thus reduce the system resources required for processing the event stream.

In historical query architectures, queries are brought to the event data. Live queries have diametrically opposed characteristics: data is brought to the queries. This twist necessitates a different architecture for live queries. Our design is depicted by Figure 3.13. Issuing a live query consists of the following steps:

1. Both users and remote applications can issue queries by creating a *query event* and sending it to the query manager.

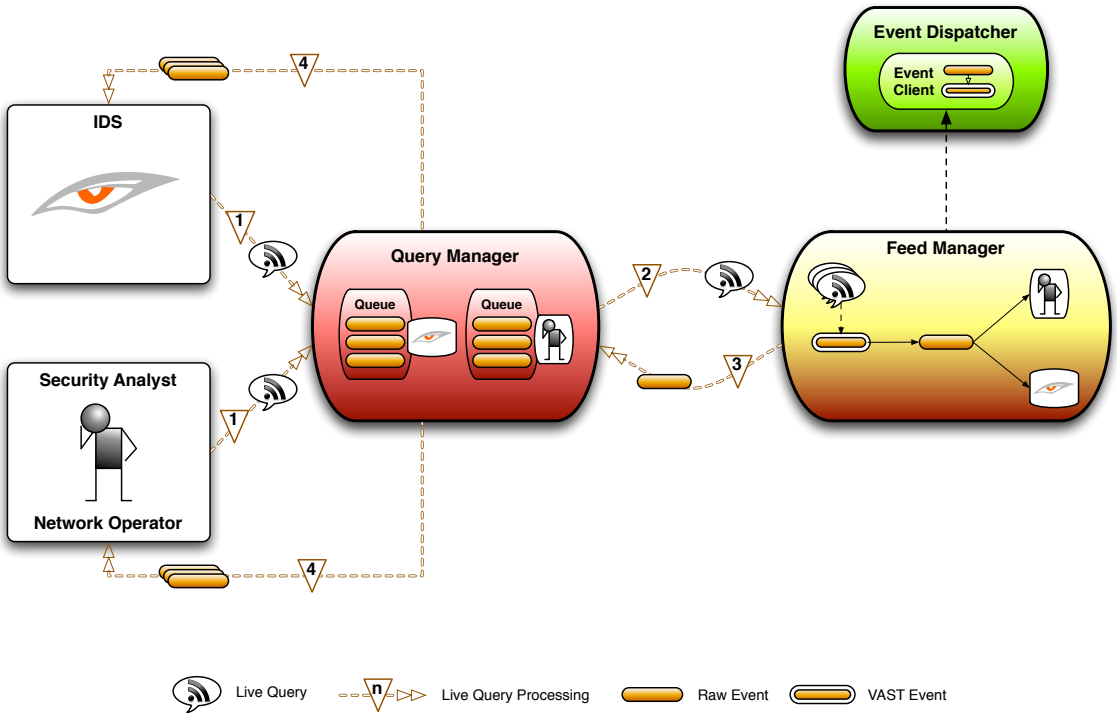
2. The query manager parses the event and forwards it to the feed manager which creates a live query based on the given condition. Each VAST event is "pushed" into the feed manager and compared against all live queries.

3. If a match was successful, the feed manager requests the corresponding raw event from the dispatcher and sends it to the associated query queue of the query manager.

4. As soon as events are inserted into the client queue, the query manager forwards the events over the existing Broccoli connection to the client.

A key challenge in the design of the live query architecture is to avoid significant performance penalties. Because each incoming event has to be matched against all live queries, the processing overhead can be non-negligible when facing large event volumes. To cope with a very high event load, concurrent query execution could substantially ameliorate the scalability of the system.

Figure 3.13 Live query processing.



Another issue represents the retention strategy for raw events. Because the feed manager asks for any raw event after encountering a positive match, the dispatcher would have to keep the raw events in memory until the corresponding VAST event is processed. One possible solution would be to forward the raw event to the feed manager as well. While it would remove the dependency to the dispatcher, it greatly increases the data sent across the network in a distributed setup.

Note that the live query architecture is still in a very fledgling stage. We have not yet found an ideal solution but would highly appreciate feedback to come up with a tractable approach.

3.2.5 Event Aggregation

A key challenge in networks with a high traffic volume is managing the space capacities that are required to maintain a complete archive of activity. A common strategy to deal with this challenge is to operate on a sampled version of the traffic [CF04]. This solution is clearly sub-optimal for the application scenarios we outlined in §2.1. Particularly the efficacy of intrusion detection and forensics is determined by the breadth of available data that can be drawn upon. Sampling opens new avenues for evasion and also increases the risk to miss critical information included in the skipped data.

Another strategy is to discard old data completely in order to make room for fresh data. However, this approach conflicts with maintaining a large measurement window (see §3.1.1). Environments with high arrival rates and large volume could induce short expiration intervals that do not keep data long enough to be beneficial. For example, the Time Machine which we discussed in §2.4.1 uses this technique to record most of the connections in their entirety, yet skip the data-intensive bulk of the total volume.

. In contrast to network packets, events represent an abstraction of activity and are particularly apt to be elevated incrementally to more succinct and higher forms of abstraction. For instance, a packet stream could yield events reflecting unsuccessful connection attempts. After a certain amount of time, these events are aggregated into summaries like "N connection attempts in the last t seconds". Later on these summaries escalate into alarms when a certain threshold is transgressed.

What we have just described is a mechanism based on *graceful degradation* that successively transform bulky low-level data into more space-efficient but still useful representations. To this end, the architecture of VAST features an *aging* component which does not exist for its own sake but represents an experience-driven compromise: rather than sampling or discarding old data entirely, aggregating events into more compact forms still includes information that is often useful in retrospect. Despite the loss of granularity over time, abstracted information can still carry security-relevant information or constitute the missing link in a chain of actions.

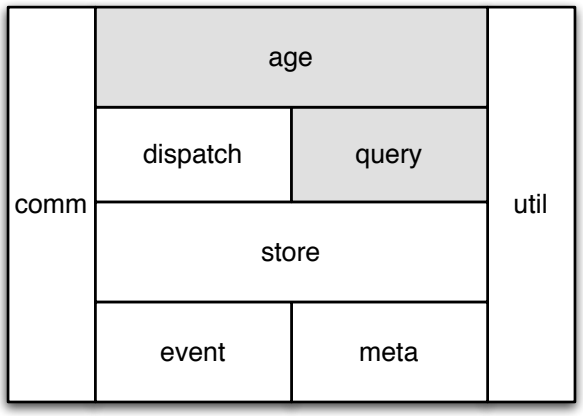
3.3 Implementation

VAST is implemented entirely in C++ and currently consists of 7,130 lines of code of which 673 are comments. Since our long-term goal is not only to create a prototype but also an application to be employed by large institutions, we emphasized a careful development process and distilled separate modules that have small interfaces. Our implementation has a layered architecture with lower layers not depending on higher layers.

To benefit from the latest language features that will be part of the next C++ language standard [C++], we make judicious use of the *Boost* [Boo] libraries, a set of peer-reviewed STL extensions to facilitate modern software engineering practices. For example, we use the *Boost.Thread* library extensively and leverage the parser framework *Boost.Spirit* to describe the grammar of the event specification.

As shown in Figure 3.14, the low-level building blocks of the system are the `meta` and `event` layer. While the former implements the *event specification* with event and type meta information, the latter contains data structures used during event processing, such as raw events and VAST events. On top resides the `store` layer which provides storage abstractions for archive and index, the slicing logic, and an enhanced FastBit interface for data streaming. Both the `dispatch` and `query` layer make use of the storage engine. Finally, the aging and aggregation layer (`age`) is placed on top as it acts like a client performing queries and inserting new events. The surrounding layers, `comm` and `util`, provide abstractions for network communication and useful templates for singleton

Figure 3.14 Modular implementation: higher layers only depend on lower layers.



objects, factories, and thread-safe data structures. The shaded layers (`query` and `age`) are not yet implemented in our current prototype.

3.3.1 Event Compression

When archiving a high event volume over a very long time window, already small efforts of keeping data growth under control have a substantial effect in the long run.

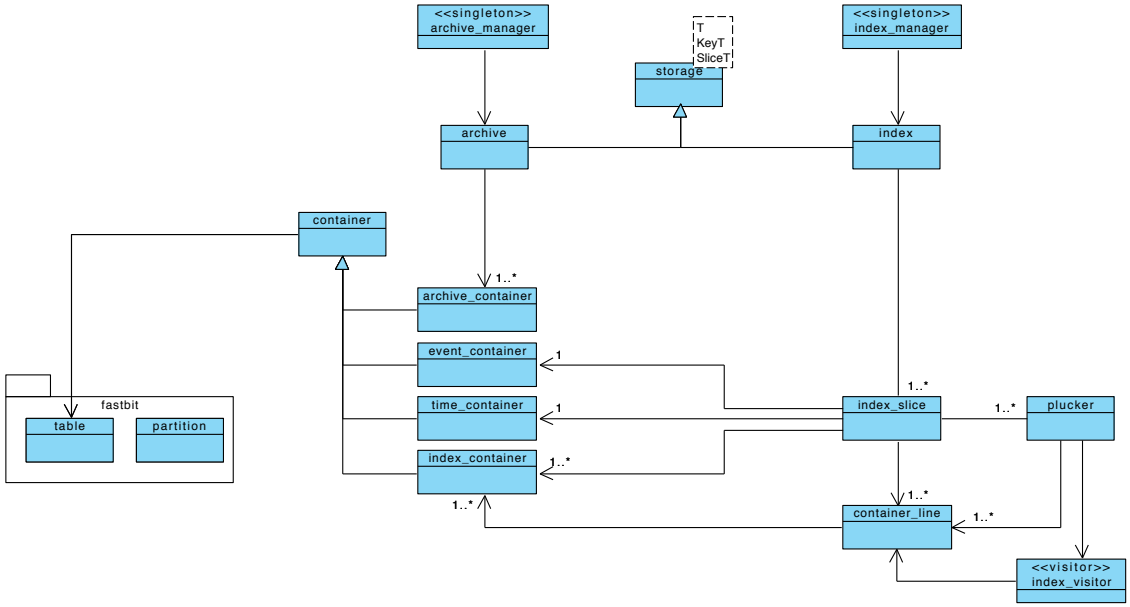
Recall that the event dispatcher creates a raw byte copy of the incoming Broccoli event. Storing raw data (also referred to as *blob* data) in a FastBit is only supported by means of NULL-terminated C-strings. However, the contents of the raw Broccoli event can include NULL bytes and thus cannot be stored in its pristine form. In order to avoid the NULL bytes, we encode the raw bytes constituting the Broccoli event. To this end, we created a custom run-length encoding of NULL bytes that achieves a compression rate of approximately 30%.[5]

Our byte-stuffing algorithm is simple and straight-forward: each non-NULL character is copied without modification. NULL bytes are replaced with a special *encoding symbol* followed by a *null symbol*. The byte after the null symbol holds the number of consecutive zeros in the raw byte stream. If this number becomes 256, we write the value 255 (or `0xFF`), reset the NULL byte counter to 1, and advance to the next byte. We continue this procedure until the number of zeros remains less than 256. If we encounter the encoding symbol in the raw byte stream, we simply duplicate it in the encoded byte stream.

Although we could use more sophisticated algorithms with higher compression rates, we explicitly chose a simple encoding scheme, given that disk space is cheap compared to CPU cycles.

[5]Raw Broccoli events contain apparently enough consecutive NULL bytes to yield such a high compression rate.

Figure 3.15 Class diagram of the `store` layer.



3.3.2 Storage Layer

The storage layer is directly used by the `dispatch` and `query` layers, as shown in Figure 3.14. It implements archive and index through a flexible storage abstraction which is based on the slicing concept we discussed in §3.2.3. Moreover, the storage layer implements containers as an adapter for FastBit tables and partitions.

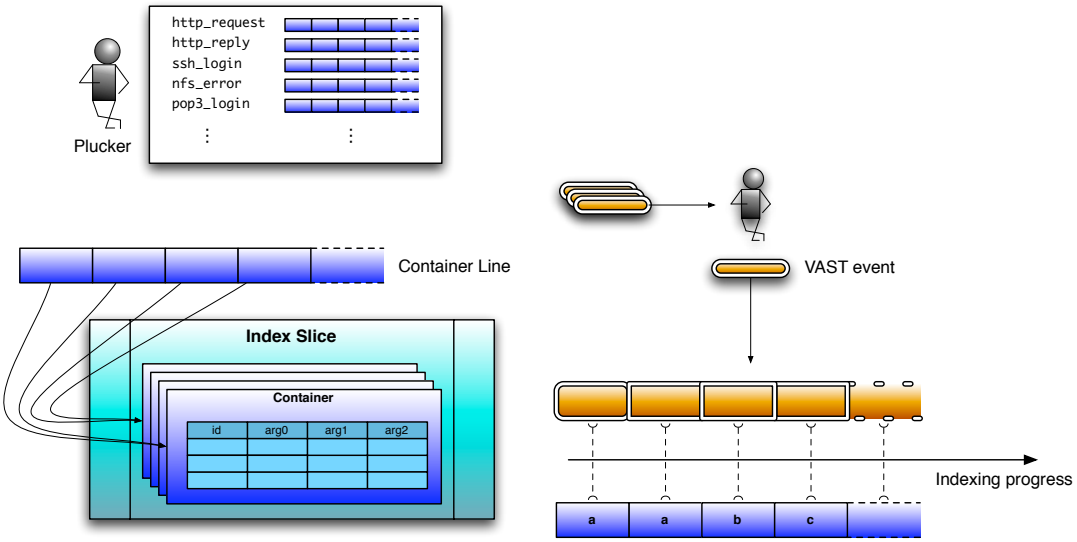
A class diagram of the `store` layer is depicted by Figure 3.15. The external interface is provided by two singleton *boundary objects*: the `archive_manager` and `index_manager`. These managers coordinate access to `archive` and `index`, which both derive from the `storage` class. Because slicing differs only syntactically between archive and index, we can factor out the logic into a separate class with three template parameters, `T`, `KeyT`, and `SliceT`:

`T`. This template parameter forces deriving classes to adhere to the interface provided by `storage`. In this case, we implement *static polymorphism* by means of the *curiously recurring template pattern (CRTP)* [Cop95].[6] For example, the children of type `archive` and `index` have to implement the `sync()` member function which writes in-memory contents to disk.

`KeyT`. The key type of the associative container to identify a particular slice. While the archive uses a 64-bit unsigned integer as event IDs (`uint64_t`) to locate a slice,

[6] Traditional software-engineering practice makes extensive use of dynamic polymorphism through more expensive virtual functions, even though many polymorphic aspects can be decided at compile-time.

Figure 3.16 Pluckers in charge of container lines.



(a) One container line per event.

(b) Processing incoming VAST events.

the index uses standard UNIX timestamps (`std::time_t`).

`SliceT`. The slice type. The archive uses an `archive_container` for this type. For the index, this type corresponds to `index_slice`, a class that accommodates a collection of `index_container` objects (one per class) on which a set of pluckers operate.

Container lines

When indexing event arguments, there is usually more than one container involved. In other words, an event is often part of multiple classes in the event specification. For example, the majority of Bro events have a `connection` record as first argument. This record alone spans over multiple classes because it contains further records, such as `conn_id` and `endpoint`.

To prevent unnecessary container lookups for each argument, we use a per-slice container cache which associates each event argument with its corresponding container. We call this cache a *container line*. It consists of a vector of container pointers that can be accessed through an iterator-like interface, that is, calling `advance()` on a container line object yields the next container. As visualized in Figure 3.17(a), a plucker has container lines for each event it is responsible for. Each time the index is sliced, new pluckers are created and container lines are regenerated for the current set of containers.

Figure 3.17(b) illustrates the indexing process using container lines. The plucker goes through each event argument and writes its value to the corresponding container. Since different argument types need to be treated differently, the plucker uses internally an

event *visitor* as helper object to handle these variations. For example, writing a boolean value in a container differs from storing a table with multiple entries.

Out-of-sequence Events

The multi-threaded design of the event archival process as described in §3.2.3 can cause occasional *out-of-sequence* events. An out-of-sequence event in the archive has a smaller id than the current slice id. Likewise, an out-of-sequence event in the index has a smaller *timestamp* than the current slice. In the following discussion, we use the archive to illustrate this issue although it equally applies to the index.

Recall that the event dispatcher monotonically assigns increasing event ids. Since events have different sizes, their construction time can vary. Hence the queue insertion order does not necessarily equal to the id assignment order. As a result, events that arrive at the archive are not guaranteed to be in order which can trigger a new slicing process while the old slice has not yet been filled entirely. In addition, multiple threads consume events to parallelize the archival process which further influences the archival order of events.

These architectural characteristics occasionally lead to out-of-sequence events. Such events cannot be stored in the current slice because they belong to an older slice from the past. To illustrate, assume the maximum slice size is 100. An new event e_{100} with id 100 triggers the creation of a new slice. Later, an out-of-sequence event e_{99} with id 99 arrives.

To handle this artefact, we calculate the correct slice for each arriving event using the function shown in Figure 3.17. In this code fragment, the template parameter `KeyT` corresponds to a `uint64_t` which represents event ids in the archive, `obj_id` is the current event id for which we perform slice lookup, and `max_objects_` holds the number of maximum objects per slice. The first line calculates the *slice base* for the current event, i.e. the event id that triggered the creation of a new slice. The slice base is the key that maps to a particular slice in memory.

Directory Layout

On disk, slices are stored in directories and the slice base is used as directory name. For example, the disk layout for the archive is as follows

```
vast/archive/1
vast/archive/1000
vast/archive/2000
```

with `max_objects_` set to 1000. The index layout, which uses timestamps as `KeyT`, would be similar to

```
vast/index/1224355288/http_request
vast/index/1224355288/http_reply
vast/index/1224355288/...
```

Figure 3.17 Slice selection algorithm for arriving events.

```
slice_ptr select_slice(KeyT obj_id)
{
    KeyT base = obj_id - (obj_id % max_objects_);

    if (base == current_slice_id_)
        return current_slice_;

    typename storage_map::const_iterator i = slices_.find(base);
    if (i != slices_.end())
        return i->second;

    return create_slice(base);
}
```

```
vast/index/1224358888/http_request
vast/index/1224358888/http_reply
vast/index/1224358888/...
```

with `max_objects_` set to 3600. Note that although the index uses the same slicing algorithm, an index slice corresponds to a directory which itself contains multiple containers. An archive slice, however, consists only of one container per slice.

Malformed events

There is another factor which affects the size of archive slices. Malformed events that cannot be parsed are ignored. Nonetheless, they consume an event id from the dispatcher and thus "poke a hole" in the monotone list of incrementing event ids. As opposed to an index slice which accommodates all events that fall into a particular time interval, an archive slice has a fixed size because event ids are unique. The solution to this problem is easy: the dispatcher has to assign event ids after the event has been created.

We have not encountered this problem in our setup because we know the structure of all arriving events: our test event specification is automatically generated by a small converter application which takes both Bro `bif` and Bro policy files as input to generate a valid VAST specification from the collected types and events. Such an automated mechanism can prevent malformed events in advance.

Lazy Container Expiration

Our slice-based storage architecture, as described in §3.2.3, is particularly apt for implementing flexible aging and aggregation strategies.

Aggregation means expunging old events and converting them into a more space-efficient representation. Because a FastBit partition — and consequently our container abstraction — stores data consecutively in linear order, deleting an entry is an expensive operation (see §2.3.2). Therefore, deleting an entry does not physically remove it but

rather marks it as invalid by flipping a bit. This property motivated us to attach a *validity counter* to each container to keep track of the number of valid entries. Appending an entry to a container increments the counter, invalidating (or deactivating) an entry decrements it.

For example, several aging steps delete old events and hence decrement the validity counter of a container until it is empty (i.e., it validity counter reached zero). Since an empty container is not needed anymore, we can now remove it completely. In case of the archive where a slice corresponds to a single container, we can immediately remove empty containers. The index requires more careful treatment. Because an index slice consists of multiple containers, we can only delete the slice when all of its containers are empty. Thus, an index slice maintains another validity counter that keeps track of non-empty containers. When it reaches zero, the slice can be safely removed from disk. Figure 3.8 illustrates validity counters by displaying slices with different "fill levels".

As we do not delete events immediately, we say container expiration is performed in a *lazy* fashion. This technique is crucial to avoid costly I/O operations that would otherwise re-order both container data and the corresponding indices for each delete operation.

4 Evaluation

This chapter evaluates the performance characteristics of VAST. In the scope of this thesis we implement a proof-of-principle prototype that is capable of archiving and indexing a stream of events. Therefore, our preliminary evaluation focuses on the event and storage layer: high performance at these layers is vital to deploy the system in environments that exhibit a large-volume event stream.

At first, we outline our methodology in §4.1 and then describe our testing environment in §4.2. Thereafter we evaluate the archival performance and analyze the storage space in §4.3.

4.1 Methodology

In order to generate reproducible measurements we resort to trace-based offline analysis. As event source we run Bro on a `pcap` trace, with a large fraction of its analyzers enabled to induce a high event load. VAST listens on a TCP socket at port 42000, waiting for event producers to connect. As soon as Bro connects, VAST establishes a Broccoli connection and subscribes for all events listed in the event specification. That is, events generated by Bro are forwarded to VAST only if a subscription for this particular event name exists.

Although VAST can handle multiple event producers in parallel, we only use one connecting Bro instance in our measurements. Moreover, we observed that the event load generated by a single event producer is high enough to yield insightful results.

Generally, trace-based analysis entails a "compressed" notion of time because the trace is processed as fast as possible, without respecting the inter-packet arrival gaps. To generate a more realistic workload, we use Bro's *pseudo-realtime* mode which inserts processing delays while receiving packet input from a trace. These delays equal to the natural inter-packet arrival gaps. Internally, Bro defers packet processing until the time difference to the next packet timestamp has elapsed. All our measurements are conducted with a pseudo-realtime factor of one, thus reflecting real-time trace replay.

To quantify the performance of the storage engine, we use the following metrics in our analysis:

Event Rate. The number of events VAST processes per second. This figure reflects the number of event handler invocations across all event clients. Of particular interest is the maximum (or *peak*) event rate that can be achieved: the more events can be processed per second, the higher the chance that VAST can handle the dynamics of large-scale operational networks with a highly variant number of arriving events.

Event Queue Size. While the event rate focuses on event consumption, the event queue size represents the number of events that are deferred on the event producer side. If VAST's underlying Broccoli communication channel is saturated (i.e., the Broccoli event buffer is full), event senders accumulate events in a local queue until VAST consumes events from the buffer and to make room for following events.

In general, a growing sender queue indicates that events arrive faster than VAST can process them. For example, when events arrive in bursts, the queue size can jump up quickly. It is then imperative to quickly consume events to prevent the queue from overflowing that causes the event producer to terminate the Broccoli connection.

4.2 Testbed

We conduct our measurements with a packet trace of traffic from the *International Computer Science Institute (ICSI)* in Berkeley. The trace has been recorded on November 6 and spans 24 hours, ranging from 01:45:02 AM to 01:45:04 the next day. It contains 32,923,230 packets from 447,893 IP flows (avg. 73.51 pkts/flow) that incorporate 155,394 distinct source addresses. The observed average rate is 1.81 Mbps (standard deviation 3.97 M) and the peak rate amounts up to 92.45 Mbps. 95.15% bytes are TCP traffic, whereas 4.56% constitute UDP traffic. The most prevalent protocols in terms of bytes are HTTP (66%) and HTTPS (2.81%).

We conduct our experiments on a machine with four Intel Xeon CPUs (each at 1.6 GHz) with 3.2 GB of RAM. The operating system is a Linux 2.6.18 SMP kernel provided by Fedora release 8 (Werewolf).

The *slicing interval* of the archive was set to 20,000, meaning that the size of an archive container holds at most 20,000 events. Similarly, the slicing interval of the index was set to 60 seconds, that is a new a new set of index containers is created each minute. Moreover, archive and index enforced that maximum three slices exist in-memory. When creating a new slice, VAST evicts the oldest slice in memory in order to enforce the in-memory slice limit.

In general, slice access is handled in a *lazy* fashion. If access to a slice not in memory is requested (e.g., during query processing), it can be easily re-instantiated from its disk representation.

4.3 Archival Performance

In the following, we investigate the event archival performance. Archiving a stream of events involves the `dispatch`, `store`, and `event` layer. Thus, scrutinizing where VAST spends its CPU cycles helps us to understand which expensive components of the system and spot potential bottlenecks. Also the memory footprint of VAST gives us insightful information. A high footprint means that a lot of internal state in dynamic data structures is maintained. If the memory usage steadily grows, we likely encounter a leak. Contrariwise, a low and constant footprint witnesses an efficient use of main memory.

4.3.1 Resource Utilization

To get a detailed picture of the resource usage, we employ Goggle's perftools [Gpe] to both understand memory footprint and CPU usage. The perftools heap profiler indicates that VAST has memory footprint of 29.7 MB.

Turning to CPU utilization, our expectation was that most of the CPU time would be spent in the `store` layer. However, it turned out that 52% of time was actually spent in the `event` layer. We then conducted new measurements to analyze this layer in isolation. To this end, we turned off the storage engine by uncommenting the relevant code in the Broccoli event handler which is executed for each arriving event. With an empty event handler, CPU measurements only account for the underlying Broccoli event reassembly: complete events are ignored and freed immediately.

Evaluating Broccoli alone, the perftools CPU profiler reports that the function `__bro_ht_free()` and its callees spent 51.7% of CPU time. This function is called when a Bro hash table object is deallocated. It iterates over a fixed number of slots (default: 128) and frees each table slot. Even if the hash table remains empty, slots are allocated and freed. We assumed that the deallocation of numerous empty tables caused this high figure and worked with the Bro developers to prevent slot allocation for empty tables. Indeed, after patching `__bro_ht_free()` to prevent slot allocation for empty hash tables, the CPU utilization of this particular function could be reduced to to 6.3%. As a result, the maximum processing rate of events doubled from roughly 1,500 to 3,000 events per second.

4.3.2 Storage Layer Performance

We now turn to the actual event processing overhead of VAST's storage layer. After discussing event rate and sender event queue size, we present a more detailed analysis of the CPU utilization.

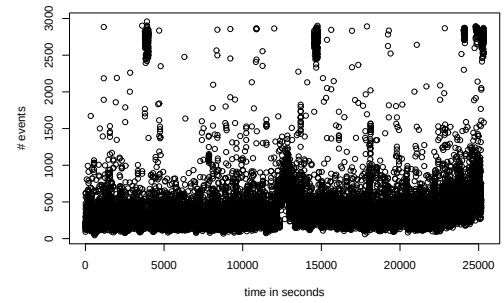
Event Rate and Queue Size

The peak event rate is an adequate means to analyze throughput and contention in the `event` layer. Ultimately, a high peak event rate is crucial to cope with bursty event arrivals: it indicates that the system can handle a high-volume event stream.

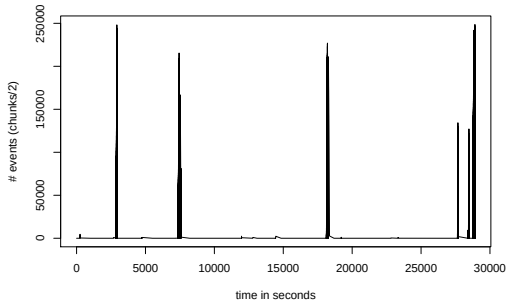
As baseline, we run VAST with empty event handlers, thereby disabling the storage component. The event processing rate is displayed in Figure 4.1(a). The corresponding event queue size on the Bro side is shown on the right hand side in Figure 4.1(b).[1] The average event arrival rate per second is 420.3 with a standard deviation of 460.9, and a peak rate of 2,959. Then, we enabled the storage component of VAST and observed event rates and queue sizes as shown as in Figure 4.1(c) and Figure 4.1(d). In this case, the average event arrival rate was 307.8, the standard deviation 243.3, and the maximum number of events per second 1707. When running with empty event handlers, we see in Figure 4.1(a) that the high event rates after 4,000, 15,000, and 25,000 seconds directly

[1] A Bro event consists of two chunks which is why the y-axis of this plot is labeled with "chunks/2".

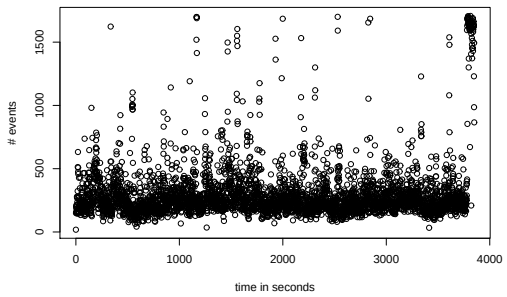
Figure 4.1 Event processing rates with corresponding Bro queue size.



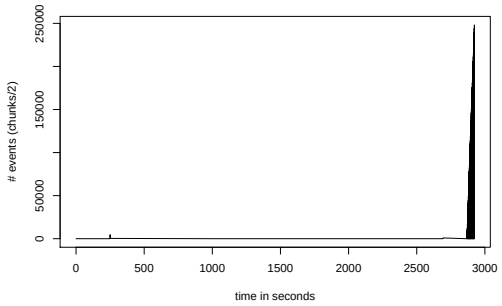
(a) Event processing rate (empty handler).



(b) Bro queue size (empty handler).



(c) Event processing rate.



(d) Bro queue size.

correlate with the peaks in Figure 4.1(b) which represent an explosive growth of the event queue size.on the Bro side.

These figures clearly show that events arrive faster than they can be processed by Broccoli. Unfortunately, we could not run our measurements to completion. The sheer volume of the event stream caused Bro to terminate the network connection, because events on the VAST side arrived faster than they could be processed by Broccoli. After Bro's event queue reaches a threshold of 250,000 events, it closes the network connection. Therefore, our measurement with enabled storage engine reflects only the first 10% of the measurement with empty event handlers. That is, Bro terminates the connection after the first spike in Figure 4.1(b), which is the one single spike at the end of Figure 4.2(d).

It is important to note that the queue size almost reached its maximum in the run with empty handlers. Therefore, the slightest amount of additional work would have caused a termination of the connection anyway. Enabling the storage engine naturally causes the peak event rates to drop. In our case, where Broccoli utilizes roughly half of the CPU, the peak event rate also drops by a factor of two. We emphasize that this unfortunate problem is not spurred by our VAST implementation, but rather Broccoli, a third-party component out of our direct control. In fact, our prototype works well as

expected: it correctly archives and indexes a stream of events.

CPU utilization

Having discovered that the `event` layer cannot handle the arriving event stream rates during peak intervals, we further investigate this observation by analyzing the CPU utilization. As mentioned earlier, the perftools CPU profiler reports that 52% of CPU time is spent in the `event` layer (i.e., in Broccoli code), when running with an enabled storage engine. However, this figure is an arithmetic mean gathered from a number of profiler samples, and thus only helps to understand the relative performance of internal components. The perftools profiler does not help to analyze the absolute CPU performance and to see what happens during the peaks.

Therefore, we implemented a custom profiler that measures the CPU utilization of the VAST process over time. Our profiler runs in a dedicated thread. Each second, it invokes the `getrusage` system call to measure the time elapsed since the previous measurement, and then sleeps for 1,000 milli seconds until the next measurement. To illustrate, if a process runs 1 second with a CPU utilization of 100%, `getrusage` returns 1 to denote the CPU spent the entire second in this process. Note that `getrusage` can return values greater than 1 for multi-threaded applications like VAST running on machines with multiple CPU cores. Also, when we refer to CPU time, we mean the sum of user and system time spent in the process.

Figure 4.2 visualizes the CPU utilization recorded by our profiler for two different runs: one time with disabled storage engine (circles) and the other time with enabled storage engine (triangles). The CPU utilization over time is shown in Figure 4.2(a), where the x-axis denotes the elapsed time in seconds since program start and the y-axis the CPU time spent in the entire VAST process. Analyzing the run with disabled storage engine (empty handler), we recognize that the three spikes where CPU utilization reaches 1.0 directly correspond to the spikes with peak event rates in Figure 4.1(a). This observation confirms that Broccoli is fully saturated when facing peak event rates. Turning to the run with enabled storage engine (triangles), we see that the measurement stops after roughly 4,000 seconds. This is the point where Bro terminates the connection due to its full event queue, as mentioned above. The CPU utilization is greater than 1 due to the multi-threaded nature of our implementation.Both the `store` layer and the `event` layer launch several threads.

To analyze at the storage layer in isolation, we subtract the time spent in Broccoli from the total CPU time. At the point of termination, we then end up with utilizations ranging from 0.0 and 0.5 (with a few negligible outliers). While Broccoli fully utilizes one core at that point, our implementation would have more capacities available for higher event rates.

A complementing illustration of the CPU utilization is shown in Figure 4.2(b). The line with circles represents the run with disabled storage engine (empty handler) and the line with triangles represents the run with enabled storage engine. In addition, the line with crosses (storage) shows the storage engine in isolation. As above, the time spent in Broccoli is subtracted from the entire time spent in VAST. When looking at

the empty handler run, we see two thin spikes: one large one (median 0.05) and a small one near by 1.0 that represent the three spikes in Figure 4.2(a). With enabled storage engine, the average CPU utilization is naturally higher (median 0.11) and the curve is also wider. The small spike at 1.0 is not existent here because the full Bro queue caused early termination (as above). Turning to the storage engine in isolation, we observe that the median shifts from 0.11 to 0.06 when ignoring Broccoli, confirming that the storage engine itself does not impose a significant overhead.

To summarize, the performance analysis shows that our implementation is efficient and works as expected. VAST archives a stream of events with low CPU overhead. Our preliminary evaluation also revealed limitations in one of the third-party libraries that we are relying on. As this component could become VAST's bottleneck, we will further explore its performance characteristics in the future.

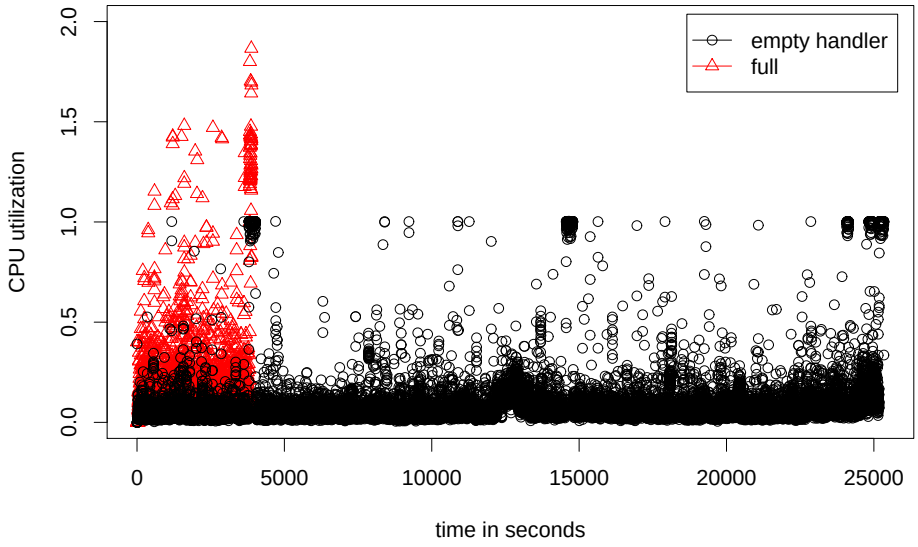
4.3.3 Storage Layer Space Overhead

Finally, we investigate the space overhead of raw events on disk. VAST could archive and index 1,180,000 events until Bro closed the network connection. The size of the archive amounts to 4.2 GB and the index occupies 377 MB (11.48%) of disk space. Archive meta data (64-bit IDs and container meta data) constitutes only 0.01% of the entire archive volume.

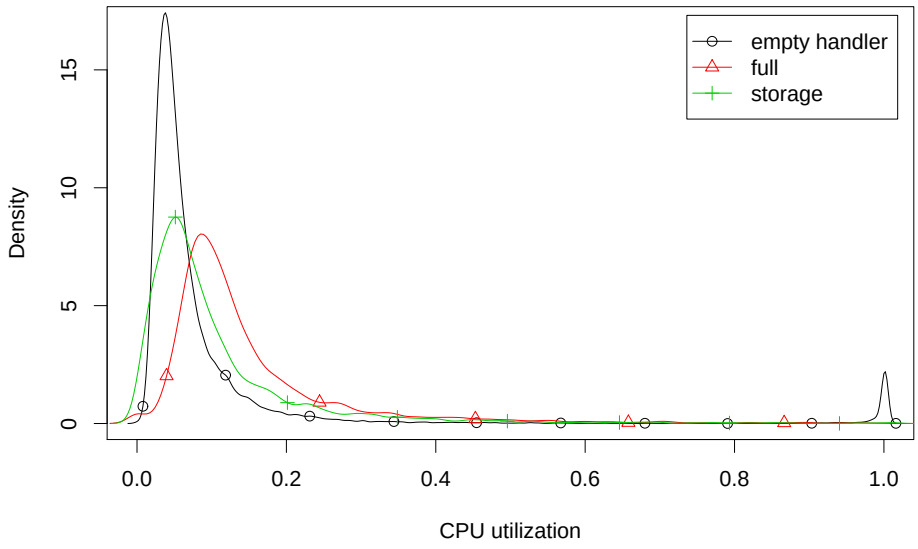
Figure 4.3 characterizes the types of events that have been archived. The event with the highest frequency was `conn_weird` (428,778). Bro generates this event whenever it encounters a "weird" connection, such as an unexpected TCP state change. The second highest frequency has `packet_contents` (249,013) which contains the raw packet contents of a particular connection.

Understanding the event mix gives the operator a fine-grained tuning knob to trade-off resources. If an operator is not interested in `conn_weird` events, the event volume could in this case be reduced by 36.3%. Further, more system resources would be available to sustain higher event peaks.

Figure 4.2 CPU utilization for enabled and disabled storage engine.

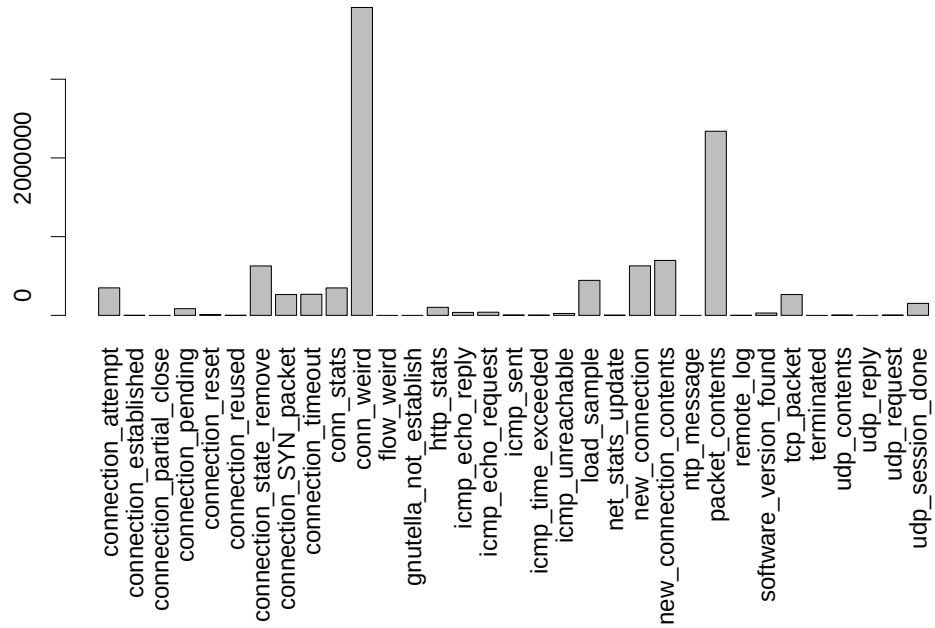


(a) CPU utilization (over time).



(b) CPU utilization (density).

Figure 4.3 Distribution of different event types.



5 Conclusion

5.1 Summary

Key administrative networking tasks, such as security analysis and troubleshooting, face new problems in large-scale networks. With incidents involving numerous systems and devices, their investigation requires examining data from multiple sources in disparate formats. However, the sheer volume and heterogeneity of data renders the analysis an arduous process. Further, wide-scale incidents can manifest over long time periods that exceed the measuring window of today's monitoring infrastructure. Understanding the full scope of problems that extend over a long span of time, with only incomplete information available, is clearly ineffective and error-prone.

Therefore, these operational tasks would significantly benefit from a unified view of space and time. Yet descriptions of activity are fragmented across space and time today: multiple incompatible data formats have to be merged in order to obtain a coherent data archive, and analysis procedures applied to past activity differ from analyzing future instances.

In this thesis, we present the design and architecture of a *Visibility Across Space and Time* (VAST) system, an intelligent distributed database that processes network activity logs in a comprehensive and coherent fashion. Based on previous ideas which formulated principles of comprehensive network visibility [AKP+08], our work continues in this direction by providing a first proof-of-principle implementation. VAST accumulates data from disparate sources and provides a central vantage point to facilitate global analysis of network activity. Our architecture supports two types of queries: on the one hand, historical queries crawl the event archive to extract past activity. On the other hand, live queries allow users to subscribe to future events. These event feeds send events to the user when the query condition matches. A significant advantage of this design is the uniform treatment of information, whether examining past or future activity.

Furthermore, the design of VAST allows recording events over a large time window using aging and aggregation schemes that gracefully transform old data into more space-efficient representations. Rather than sampling or deleting old data completely, aggregating events into more compact forms still includes information that proves often useful in retrospect. Although losing granularity over time, high-level information still can still contain valuable information or constitute the missing link in a chain of actions.

Building such a system requires addressing a number of challenges. To deal with heterogeneity of different data formats, VAST uses the rich-typed event model of the Bro *network intrusion detection system* (NIDS). In this model, events represents a generic abstraction to describe activity, thus separating mechanism from policy. This unified data model has enough expressiveness to incorporate arbitrary data in a homogeneous

fashion. To adapt the structure of events or change their archival and indexing characteristics, VAST uses a document called *event specification* which centralizes event meta data management.

A key component of VAST is the high-performance storage engine that archives and indexes events, while at the same time delivers query results to users or remote applications. To answer high-dimensional queries efficiently, we use bitmap indices implemented by FastBit. On top, we carefully designed and implemented a concurrent data streaming engine to fully benefit from modern many-core architectures. The real-time nature of a continuously arriving event stream requires the system to save enough resources to handle the dynamic load. Due to its distributed design, VAST can be scaled across multiple machines to balance the work load: event archiving, indexing, and query management can execute as independent components on separate machines.

Our preliminary performance evaluation shows that our prototype is efficient and works as expected: it archives and indexes a high-volume event stream with a small overhead. Our evaluation also discovers limitations in a third-party library that we are relying on. Because this component could become the bottleneck of the entire system, we will further explore its performance characteristics in the future.

Overall, our work is the step towards a comprehensive system facilitating a unified approach to the analysis of network activity. We expect it to enable further fruitful work in the field of operational network visibility.

5.2 Outlook

During our conceptual and programming work we discovered several interesting avenues for future work that lay beyond the scope of this thesis. Below we sketch the most important approaches and next steps.

Our next near-term goal is to implement the query engine. As described, it will support SQL-like queries and allow users to install event feeds. In the long run, we envision to provide graphical user interface to improve the usability of the system.

To improve event throughput in Broccoli, we currently experiment with different tuning parameters of the Broccoli protocol that reads data from a network socket in rounds, in each of which a fixed number of bytes is read. Changing the number of rounds or the number of bytes read affects the event throughput. Initial experiments with different parameters did not yield significant improvement. However, we have not yet explored the entire spectrum and continue investigating issue in the future.

In future versions of VAST, we aim at improving concurrent event archival and avoid random I/O whenever possible. Our current implementation uses the interface provided by FastBit to write the data to disk. However, FastBit is geared towards OLAP applications and not the archival of streaming data. Consequently, the implementation is not optimized to handle a constant input data stream and continuous index updates. Nonetheless, recent work showed that it is possible to marry these approaches (see §2.2.2). We therefore plan to replace the streaming data input path with a custom module that *(i)* supports adaptive memory retention strategies, *(ii)* performs sequential I/O

in the presence of multiple archival threads (e.g. by maintaining meta data about the locality of event data), and *(iii)* consistently uses asynchronous I/O operations to retrieve data from the network instead of spawning a dedicated thread per connection. To this end, we plan to migrate our network handling code to the *Boost.Asio* library [Boo] which offers a platform-independent interface for asynchronous I/O.[1] On the one hand, we achieve better portability. On the other hand, the asynchronous nature of the library fits well into the event-driven architecture of VAST.

We envision also to improve the degree of concurrency during event indexing. In §3.2.3 we introduced the notion of groups to parallelize the indexing process. Environments that yield only a few groups unfortunately do not benefit from this mechanism. A small number of groups usually stems from many *inter-event* dependencies. For example, the `connection` class contains 253 of 288 events. In other words, the `connection` argument is shared across 253 events. The total number of groups we get from a default Bro configuration is 23, yielding ample space for improvement.

To remove inter-event dependencies, we wish to allow the specification maintainer to *inline* particular event arguments by changing their scope from global to local. Another promising direction to reduce such dependencies is to keep the shared arguments global, but offer more fine-grained control for *intra-class* load-balancing. One way to accomplish this would be to introduce a special attribute for shared types in the specification that denotes a load-balancing factor (e.g. `&balance = 10`). VAST would then re-partition the group into 10 distinct groups. Internally, the shared event arguments are then distributed across 10 distinct classes to reflect this change.

We further plan to support storage versioning. In the current version of VAST, a modification in the event specification (e.g. due to a changing event structure) requires to start over with a new database. However, the aim of VAST is to provide a large archive of activity that reaches far back in time. Changes of the event structure naturally occur as systems evolve and VAST should be capable of adapting to these changes. To this end, we picture a *snapshot* approach which allows to associate a subset of the database with a particular specification. Whenever the specification changes, a new snapshot is created. A challenging task will then be the creation of valid queries. VAST users should be able to query old snapshots as well as the current. In addition, it should be possible to create a merged view of all specifications to query consistent data across snapshots.

The global nature of network attacks makes them difficult to counter with only a strictly local vantage point. We envision VAST to act as a platform to facilitate operationally viable, cross-institutional information sharing of attack details. Sharing details in today's operational practices is highly inefficient, cumbersome, and often requires human intervention. Therefore we intend to automate significant elements of cross-organizational security analyses via event-oriented analysis scripts that reduce human involvement.

[1] We currently use self-written, generic wrapper-classes to provide an object-oriented abstraction for socket communication.

Acknowledgements

We would like to thank Robin Sommer for the numerous productive discussions about the system architecture and for the detailed comments on this thesis. Christian Kreibich has also been a great help with his immediate response to the Broccoli issues we raised. We would like to thank K. John Wu for the prompt implementation of the FastBit enhancements we suggested. We also want to express our gratitude to Vern Paxson for his invaluable feedback on the entire project.

Bibliography

[AKP+08] Mark Allman, Christian Kreibich, Vern Paxson, Robin Sommer, and Nicholas Weaver. Principles for developing comprehensive network visibility. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, July 2008.

[Ant95] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression*, page 476. IEEE Computer Society, Washington, DC, USA, 1995.

[AYJ00] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 329–338. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-715-3.

[BM70] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141. ACM, Rice University, Houston, Texas, USA, November 1970.

[Boo] Boost C++ Libraries. `http://www.boost.org`.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 225–237. Asilomar, CA, USA, January 2005.

[C++] The c++ standards committee. `http://www.open-std.org/jtc1/sc22/wg21/`.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*. Asilomar, CA, January 2003.

[CF04] Sirish Chandrasekaran and Michael J. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB*, pages 348–359, 2004.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651. ACM Press, New York, NY, USA, 2003. ISBN 1-58113-634-X.

[CMD+06] Evan Cooke, Richard Mortier, Austin Donnelly, Paul Barham, and Rebecca Isaacs. Reclaiming network-wide visibility using ubiquitous endsystem monitors. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, page 32. USENIX Association, Berkeley, CA, USA, 2006.

[Cop95] James O. Coplien. Curiously recurring template patterns. *C++ Report*, 7 (2):24–27, 1995. ISSN 1040-6042.

[CYBR06] Bee-Chung Chen, Vinod Yegneswaran, Paul Barford, and Raghu Ramakrishnan. Toward a query language for network attack data. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)*, page 28. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2571-7.

[DS07] Peter J. Desnoyers and Prashant Shenoy. Hyperion: High Volume Stream Archival for Retrospective Querying. In *Proceedings of the 2007 USENIX Annual Technical Conference*. Santa Clara, CA, June 2007.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. ISSN 0360-0300.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003. ISSN 0163-5808.

[Gpe] Google perftools. `http://code.google.com/p/google-perftools`.

[HCH+05] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, , and Aydan R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *Proceedings of the 2nd biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA, USA, January 2005.

[IBM] IBM Database 2. `http://www.ibm.com/db2`.

[IDM04] Gianluca Iannaccone, Christophe Diot, and Derek McAuley. The CoMo white paper. Technical Report IRC-TR-04-017, Intel Research, September 2004.

[Joh99] Theodore Johnson. Performance measurements of compressed bitmap indices. In *VLDB '99: Proceedings of the 25th International Conference on*

Very Large Data Bases, pages 278–289. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-615-7.

[Kou00] Nick Koudas. Space efficient bitmap indexing. In *CIKM '00: Proceedings of the ninth international conference on Information and knowledge management*, pages 194–201. ACM, New York, NY, USA, 2000. ISBN 1-58113-320-0.

[KPD+05] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *IMC'05: Proceedings of the Internet Measurement Conference 2005 on Internet Measurement Conference*, page 23. USENIX Association, Berkeley, CA, USA, 2005.

[KS05] Christian Kreibich and Robin Sommer. Policy-controlled event management for distributed intrusion detection. In *ICDCSW '05: Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05)*, pages 385–391. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2328-5-04.

[LBZ+05] Xin Li, Fang Bian, Hui Zhang, Christophe Diot, Rah Govindan, Wei Hong Hong, and Gianluca Lannaccone. Advanced indexing techniques for wide-area network monitoring. In *ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops*, page 1184. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2657-8.

[Lib] libpcap. `http://www.tcpdump.org`.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Winter USENIX Conference*, 1993.

[MSD+08] Gregor Maier, Robin Sommer, Holger Dreger, Anja Feldmann, Vern Paxson, and Fabian Schneider. Enriching network security analysis with time travel. In *SIGCOMM '08: Proceedings of the 2008 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, New York, NY, USA, August 2008.

[MSQ] Microsoft SQL Server. `http://www.microsoft.com/sql/default.mspx`.

[O'N89] Patrick E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59. Springer-Verlag, London, UK, 1989. ISBN 3-540-51085-0.

[OOW07] Elizabeth O'Neil, Patrick O'Neil, and Kesheng Wu. Bitmap index design choices and their performance implications. In *IDEAS '07: Proceedings of*

the 11th International Database Engineering and Applications Symposium, pages 72–84. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2947-X.

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49. ACM, New York, NY, USA, 1997. ISBN 0-89791-911-4.

[ORA] ORACLE. `http://www.oracle.com`.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[PF94] Vern Paxson and Sally Floyd. Wide-area traffic: the failure of poisson modeling. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 257–268. ACM, New York, NY, USA, 1994. ISBN 0-89791-682-4.

[Rin02] Denis Rinfert. *Term Matching and Bit-Sliced Index Arithmetic*. PhD thesis, University of Massachusetts Boston, 2002. Director-Patrick O'Neil.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[RSW05] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing candidate check costs for bitmap indices. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 648–655. ACM, New York, NY, USA, 2005. ISBN 1-59593-140-6.

[RSW06] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Minimizing i/o costs of multi-dimensional queries with bitmap indices. In *SSDBM '06: Proceedings of the 18th International Conference on Scientific and Statistical Database Management*, pages 33–44. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2590-3.

[RSW+07] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein. Enabling real-time querying of live and historical stream data. In *SSDBM '07: Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, page 28. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2868-6.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005. ISBN 1-59593-154-6.

[SBUK+05] M. Siekkinen, E. W. Biersack, G. Urvoy-Keller, V. Gol, and T. Plagemann. Intrabase: Integrated traffic analysis based on a database management system. In *E2EMON '05: Proceedings of the End-to-End Monitoring Tecques and Services on 2005. Workshop*, pages 32–46. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7803-9249-3.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005. ISSN 0163-5808.

[Som05] Robin Sommer. *Viable Network Intrusion Detection in High-Performance Environments*. PhD thesis, Technical University Munich, 2005.

[SP05] Robin Sommer and Vern Paxson. Exploiting independent state for network intrusion detection. In *Proc. Computer Security Applications Conference*, 2005.

[SWS02] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. Strategies for processing ad hoc queries on large data warehouses. In *DOLAP '02: Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 72–79. ACM, New York, NY, USA, 2002. ISBN 1-58113-590-4.

[SWS04] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. Evaluation strategies for bitmap indices with binning. In *DEXA*, pages 120–129, 2004.

[syb] Sybase IQ. `http://www.sybase.com/products/datawarehousing/sybaseiq`.

[VSL+07] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS Cluster: Scalably Stateful Network Intrusion Detection on Commodity Hardware. In *RAID '07: Recent Advances in Intrusion Detection, 10th International Symposium*, Lecture Notes in Computer Science, pages 107–126. Springer, September 2007.

[WK06] Robert Wrembel and Christian Koncilia. *Data Warehouses And Olap: Concepts, Architectures And Solutions*. IRM Press, 2006. ISBN 1599043645.

[WLO+85] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 448–457. VLDB Endowment, 1985.

[WOS01] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. A performance comparison of bitmap indexes. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 559–561. ACM, New York, NY, USA, 2001. ISBN 1-58113-436-3.

[WOS04] Kesheng Wu, Ekow Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 24–35. VLDB Endowment, 2004. ISBN 0-12-088469-0.

[WOS06] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006. ISSN 0362-5915.

[Wu05] Kesheng Wu. Fastbit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16:556–560, 2005. ISSN 1742-6596.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.