Real-Time Query Systems for Complex Data Sources

A dissertation presented
by

Ian Thomas Rose

to
The School of Engineering and Applied Sciences
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts
May 2011

Thesis advisor
Matt Welsh

Author
Ian Thomas Rose

Real-Time Query Systems for Complex Data Sources

Abstract

This dissertation presents techniques for building scalable systems that allow real-time querying of complex data sources. In recent years, networking and sensing advances have dramatically increased the volume of information available to data consumers. However, coping with large scales and high data rates often requires processing data in real time, as it arrives, rather than storing it for later analysis. Our thesis is that by including the data acquisition process in the overall system design, it is possible to build scalable, real-time stream processing systems for complex data sources.

We have built two systems to demonstrate a number of unique design features required for scalable operation in our chosen domains. Cobra is a system that taps online RSS feeds (such as blogs, news articles and websites' user comments) as its data source. Cobra repeatedly crawls a set of RSS feeds, matching the contents to keyword-based user queries, similar to those used in Web search engines. As RSS-based content can change frequently, the design ensures that the latency between crawls is low, while still scaling to a large number of RSS feeds and many concurrent user queries.

Secondly, Argos is a system for widely-distributed, outdoor wireless network mon-

itoring. Capturing 802.11 WiFi traffic across a large urban area, Argos enables a wide range of user queries, such as mobile node tracking, malware detection, and traffic characterization. Use of a wireless mesh network to connect the deployed sniffer nodes introduces additional challenges due to its limited bandwidth capacity. To address this restriction, we designed a novel in-network packet merging process and demonstrate its bandwidth savings. Additionally, Argos provides a variety of channel management schemes; 802.11 defines up to 14 radio channels but each sniffer can only capture from one channel at a time, necessitating policies for when to capture from which channel.

These systems are built around three design principles that aid in the real-time querying of complex data sources: query interfaces tailored to the application's specific data types, optimized data collection processes, and allowing queries to provide feedback to the collection process.

Contents

Acknowledgments

Firstly, I'd like to thank my advisor, Matt Welsh. I've followed a bit of a meandering road in grad school, and Matt deserves much credit for where I am today. I am grateful for his guidance in the various skills of a computer systems graduate student: designing systems, interpreting data, writing papers, and, ultimately, conducting meaningful research.

A debt of gratitude is also owed to Radhika Nagpal, who patiently advised me throughout my first 2 years as I struggled to define my research interests. Radhika provided my first opportunities at finding open research problems, reading and writing papers, and being in a research group.

I also thank Margo Seltzer and Jim Waldo for serving on my thesis committee. Both provided valuable feedback and guidance during the writing process. Additionally, both of their classes were influential in eventually steering me towards this dissertation topic.

I have had the pleasure of working with many other graduate students through the Syrah, CitySense and SSR research groups. Rohan Murty, Ankit Patel and Julius Degesys, in particular, shared many late nights in the lab with me and I am glad to call them friends and colleagues. I am also indebted to Nick Feamster and his research group at Georgia Tech, which has been my academic home-away-from-home for the past two years.

Many thanks go to my parents for supporting me all these years and for instilling in me a love of learning at a young age. To Allison, for moving with me up to Cambridge so I could start this journey; for bravely slogging through many of my paper drafts and talk rehearsals; for letting me go to a conference in Zürich just 3

weeks after our second son was born; and for her love and support every single day. Finally, to our boys, Will and Henry, for lending me perspective, and reminding me of the simple pleasures of the life.

For Allison and our boys, Will and Henry.

Chapter 1

Introduction

In the past decade, dramatic increases in both the amount of data produced by computer systems, as well as the availability of high-speed networks to access to this data, have lead to fundamental changes in how many users interact with data. Now, users frequently demand analysis of data as it arrives, rather than storing the data for later analysis. This need can arise due to application requirements or simply as a means of contending with an otherwise unwieldy volume of data. *Real-time querying* of the data in this fashion requires fundamentally new architectures to effectively support complex analyses over large numbers of diverse, high-volume data sources.

1.1 Stream Processing Engines

One approach taken by many real-time query systems is to model the incoming data as one or more continuous (effectively infinite) *data streams* and perform real-time analysis on this data in the form of one or more long-term *user queries*;

such systems are commonly referred to as *stream processing engines* (SPEs). Recently, SPEs have been successful in a great variety of application areas, including financial market processing [10, 37], battlefield monitoring [37] and network traffic analysis [61, 135]. In some cases, these systems are purpose-built for a single domain (e.g. Tribeca [135] for network traffic analysis), while other systems are designed as general purpose systems (e.g. Aurora [51]).

The benefits of general purpose SPEs are obvious: the effort put into building the system pays off manyfold whenever it is reused in a new area. Nonetheless, system requirements for some applications make it infeasible to use general purpose SPEs, necessitating specialized solutions. The current crop of general purpose SPEs work well when (a) the individual data elements can be naturally represented as tuples consisting of common primitive data types[1] and conforming to a consistent schema, and (b) the data collection process is relatively simple or can be abstracted away into a separate system. For example, a general purpose SPE could be used for a stock tracking application which receives trade notifications of a fixed structure (e.g. $\langle stockPrice, sharePrice, volumeOfShares \rangle$) over a TCP socket.

However, for many data sources one or both of these criteria are not met; for example, a network monitor may operate on packets of many different network protocols (violating the consistent schema requirement), or a low-power wireless sensor network may require custom, or even query-specific, techniques to retrieve data in a power-efficient manner (violating the simple/separable data collection requirement). We refer to these kinds of data sources as *complex data sources*, since they require

[1] e.g. character strings, integers, floating point values

special handling that general purpose SPEs cannot provide.

1.2 Challenges

Both general-purpose and domain-specific SPEs face two universal challenges: providing an effective query interface and achieving high scalability.

- **Effective Query Interfaces:** SPEs must provide some kind of interface for users to express their desired query logic, oftentimes in the form of a query language such as SQL. Designing this interface is difficult due to two competing goals: the interface should be user-friendly and accessible to as wide an audience as possible, but also should be expressive and powerful enough for users to fully express their desired query semantics.

- **Scalability:** SPEs are commonly expected to scale over a variety of dimensions. Early on, systems were primarily concerned with supporting high incoming data rates and large numbers of concurrent queries [51, 56]. Increasingly, however, SPE designs now must support large numbers of data *sources*, possibly distributed over a large geographic area, such as may be the case when reading from a wireless sensor network [54, 96].

Clearly there are no simple, one-size-fits-all solutions to either of these challenges; often, application-specific solutions are required. This is particularly true in query systems for complex data sources. Returning to our prior examples, network monitors may require specialized query interfaces tailored to network traffic analysis, and systems reading from wireless sensor networks may require specialized protocols or

algorithms to extract data in a way that respects the sensor nodes' limited battery power and radio bandwidth. Nonetheless, we have identified three design principles for building *scalable*, *effective* query systems for a variety of complex data sources: query interfaces tailored to the application's specific data types, optimized data collection processes, and allowing queries to provide feedback to the collection process. In this dissertation, we present two systems that demonstrate these techniques in concrete situations and draw some generalizing principles.

Effectively, these techniques delineate the specific areas where general-purpose SPEs can fall short of meeting the needs of systems that query complex data sources. In Chapter 8 we speculate on the future of general- vs. special-purpose SPEs and the degree to which general purpose SPEs may be augmented so as to avoid a proliferation of "one-off" special purpose SPEs.

1.3 Dissertation Summary and Contributions

This dissertation makes the following contributions:

Cobra: A Content-based Filtering and Aggregation System for Blogs and RSS Feeds. Cobra is a query system that allows users to specify keyword-based queries and then receive a personalized RSS feed updated with matching entries (e.g. articles or blog entries) from across thousands of websites. The system design maximizes the number of content sources (crawled URLs) and user queries, while minimizing the update latency (time elapsed between the posting of new content and its delivery to users) and the hardware resources required.

Although Cobra's query interface – a boolean expression over keywords – is somewhat limited in expressive power, it is familiar to web users, since this is the interface used by most search forms (e.g. web search engines, product searches on e-commerce sites). As Cobra's initial goal is to support only filtering and aggregation so as to create personalized RSS feeds, this seems to be a reasonable tradeoff. Additionally, this restricted interface allows Cobra to use optimized text-matching algorithms, which greatly increases the number of concurrent user subscriptions (queries) that Cobra can support.

Cobra detects updated content by repeatedly fetching monitored URLs. Since most page fetches are redundant (the content hasn't been updated since the last fetch), Cobra takes pains to avoid downloading and processing stale content wherever possible, using both HTTP- and RSS-specific techniques; this reduces the load imposed on content servers, as well as Cobra's bandwidth requirements and computational load.

CitySense: An Urban-Scale Wireless Sensor Network and Testbed. CitySense presents users with two opportunities: the ability to evaluate wireless protocols and systems in a complex, "real-world" urban setting (which is near-impossible to replicate fully in simulation or synthetic testbeds), and access to the many sensing opportunities in this environment. A variety of mesh networking projects have given insight to the performance of 802.11 in complex urban settings, but do not allow for external experimentation. CitySense uses a dual-radio design so that experimenters can try out new protocols on one radio without disrupting mesh connectivity to nodes

on the other radio. In many cases, CitySense nodes are accessible solely via the wireless mesh and thus maintaining reliable connectivity is vital.

Additionally, CitySense provides access to a diverse collection of data sources. Environmental scientists, for example, may be interested in air quality or micro-climate data, whereas social scientists might make use of records of human movement patterns. Argos (below) is, in fact, an unorthodox kind of sensing application that uses the experimentation radio of each node as a "sensor".

Argos: An Outdoor Wireless Network Monitor. Argos is a query system for capturing and analyzing ambient wireless traffic over large, outdoor, urban areas. To achieve the geographic distribution required for wide-area, outdoor monitoring, Argos is designed to run over a network of sniffers inter-connected by a wireless mesh network. In many urban settings, this approach enables long-term monitoring of thousands of wireless networks with only a few tens of sniffer nodes. However, this sparseness also introduces a number of new challenges, such as the bandwidth limitations imposed by mesh connectivity and low packet capture rates.

Argos is based on the Click [86] networking toolkit, augmented with various custom operators designed for handling and analyzing wireless packets (especially *merged* packets, a concept specific to wireless monitoring). User queries are structured in a way that allows Argos aggregate packets within the sniffer network, and then perform much of each query's work directly on the sniffer nodes, significantly reducing the bandwidth required for data collection as compared to running all queries centrally.

Queries are also able to change the 802.11 channel of sniffers' capture interfaces.

This is important since radios can tune to only one channel at a time, but most applications are interested in monitoring traffic from all eleven[2] 802.11 channels, and thus some kind of channel-management policy is needed. Although a hard-coded policy could be built into Argos itself, we instead expose this functionality in the query interface to allow *query-specific* policies.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 motivates the need for real-time query systems and reviews prior and related work in this space. Chapters 3 and 4 present the architecture and evaluation of Cobra, a system for real-time querying of large numbers of RSS feeds. Chapter 5 describes our work designing and deploying the CitySense network, and presents some measurements to characterize the overall network performance. Chapters 6 and 7 present the architecture and evaluation of Argos, a system for monitoring the wireless traffic across a large, urban area. Chapter 8 discusses open issues and offers some potential directions for future work. Lastly, Chapter 9 concludes.

[2]The number of 802.11 channels various by specific protocol and country; there are eleven 802.11b/g channels allowed in the U.S.

Chapter 2

Background and Related Work

To motivate the need for real-time query processing in many modern applications, consider that the NASDAQ electronic stock market routinely handles over 70,000 transactions (orders, cancellations and trades) per second[1]. Not only are financial applications growing increasingly sophisticated, but strategies such as high-frequency computational trading (which typically requires very rapid responses to market conditions, sometimes on the order of milliseconds) are leading to formidable performance requirements.

Social networking is another area where real-time query processing is becoming popular. Twitter is one of the current heavyweights in this area, with nearly 200 million registered accounts and 140 million messages (tweets) posted per day[2]. This volume of user-generated content presents an exciting new opportunity for a wide variety of data consumers. Researchers have used Twitter to track flu epidemics [90],

[1] As of December, 2008 [70].

[2] March, 2011. [114].

detect events (e.g., earthquakes) in real-time [125] and monitor the uptime of popular web services [104]. Sales and marketing professionals use Twitter to follow current trends and fads [115]. Social scientists are interested in how people use Twitter and what kinds of interactions take place via it.

As a final example, consider monitoring network traffic on a high-bandwidth Internet backbone link. Caida's[3] real-time Internet traffic monitor for two OC192 links (between Seattle, WA and Chicago, IL and between San Jose, CA and Los Angeles, CA) shows that each averages several Gb/s of continuous traffic. Calculating even simple summary statistics is challenging at these speeds; Caida in fact relies on a sampling method they developed [67].

The common thread between these application scenarios is their analysis of real-time, high-volume streams of data. "Traditional" system designs based on saving the incoming data for later processing often encounter difficulties storing and managing the total volume of data being received and are also unable to provide real-time query results, which are vital in some scenarios (e.g. stock trading).

2.1 A Taxonomy of Streaming Solutions

Before considering specific systems in detail, it is useful to consider how the related work generally fits within a common design space. We find the following dimensions to be useful for this comparison, although these are certainly not the only options:

- *Query flexibility* measures how flexible a system's query interface is, ranging

[3] http://www.caida.org/

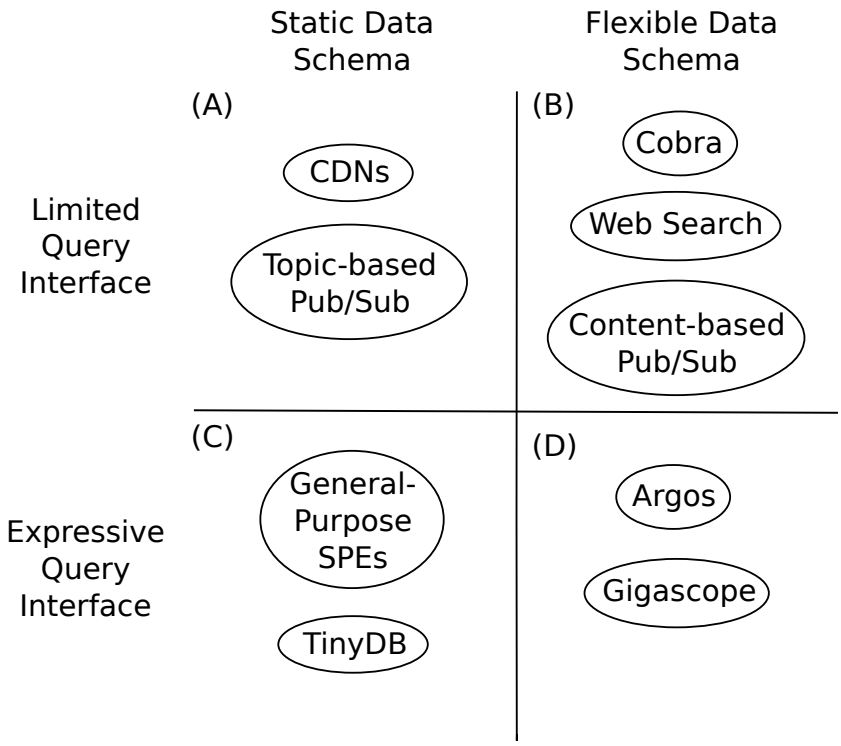


Figure 2.1: **Summary of related streaming query systems.**

from highly expressive, enabling a diverse variety of queries, to highly restricted, allowing only a narrow range of query options.

- *Schema flexibility* measures whether a system requires that input data conform to a static, pre-defined schema, or whether the system can operate on less structured data (such as unstructured text).

Using these criteria, the design space can be partitioned into four general regions, as diagrammed in Figure 2.1. These regions are as follows:

(A) Limited Query Interface & Static Data Schema: Content-delivery net-

works (CDNs) and topic-based publish-subscribe (pub/sub) systems offer narrow access to their content; these two categories of systems require that consumers provide unique content identifiers (e.g., URLs) and the choice of one or more pre-defined topics, respectively. Note that although the content *itself* need not conform to any particular schema, the only fields actually queryable by these systems (i.e., a document's unique identifier or its attached list of topics), *do* conform to specific data schemas. Thus, these systems are properly classified in the "Static Data Schema" region.

(B) Limited Query Interface & Flexible Data Schema: Web search engines and CDNs often handle largely the same kinds of content: web documents. The fundamental difference in their query interfaces is that CDNs can only address documents by identifier (URL), whereas web search engines support querying over the documents' *content* (and possibly other fields, such as author or title). Analogously, content-based pub/sub systems support querying over the content of stored documents, while topic-based pub/sub systems select documents based solely on their topic identifiers. Cobra also falls into this area of the design space since we chose to focus only on content filtering (which requires a fairly simple query interface) instead of more general purpose querying.

(C) Expressive Query Interface & Static Data Schema: General-purpose stream-processing engines (SPEs), as well as some domain-specific SPEs (such as TinyDB), provide highly flexible query interfaces that can be used to implement complex analyses. The tuple-processing operators built into their query interfaces are fundamentally designed to operate on simple, fixed-schema tuples.

(D) Expressive Query Interface & Flexible Data Schema: A number of domains require expressive querying over content that does not easily conform to simple, static data schemas. Although both of the examples cited here (Argos and Gigascope) operate on network packets, other domains may fall under this category as well, such as XML document processing or some scientific computing applications.

In the remainder of this chapter, each of these systems are described in further detail, with particular emphasis on the query interface/model used and the requirements placed on the input data.

2.2 General Purpose SPEs

To address the challenges of real-time querying, one approach has been to design general-purpose *stream processing engines* (SPEs) that can adapt to a variety of application domains. The database community, in particular, has produced a variety of designs, including (but not limited to) Aurora [51, 37], Borealis [54], TelegraphCQ [56], and STREAM [105]. There are also some commercial offerings, such as StreamBase [5] (which grew out of Aurora/Borealis) and System S from IBM [33, 75]. See Babcock at al. [36] for a general survey of stream processing, including a discussion of some of the work cited here.

The typical model for these systems is to accept one or more queries, which are compiled into a *query plan* consisting of a graph of operators. Data streams are assumed to consist of tuples that conform to a fixed schema of common primitive data types. Typical operators manipulate individual tuples (e.g., discard or modify them

according to some criteria), manipulate entire data streams (e.g., split one stream into many or merge many streams into one), or calculate aggregations over some window of data (e.g. every X tuples, or every Y seconds). Each query specifies an output, from which results are returned to the user in some way. Some queries may simply produce a subset of the input data (e.g., "return all temperature readings above 100 degrees"), while others may produce aggregations over the data (e.g., "every 30 seconds, return the max and average of all temperate readings from the past 30 seconds").

For brevity, we discuss only two of the above systems in detail, followed by a summary of the particularly notable features of each of the remaining systems.

2.2.1 Aurora

Query Model: Aurora queries are specified directly as a "boxes and arrows" dataflow graph consisting of operators (boxes) connected by implicit packet queues (arrows). Aurora contains built-in support for seven operator types, such as *filter*, which drops tuples that fail to meet some criterion, *merge*, to combine multiple streams into one, and *map* which applies some user-defined function to each tuple. One unique feature is the ability to specify *connection points* which are bounded storage locations within the network. For example, if a query contains a "1 hour" connection point, then data up to 1 hour old will be continually buffered at that location. Aurora supports adding *ad-hoc* queries to a system while it is running; if an ad-hoc query is attached to a connection point, the query can make use of the buffered data to provide (limited) query results over past data.

Static Optimizations: Before running queries, Aurora performs a number of

static optimizations to improve runtime performance. Firstly, Aurora strips out unneeded attributes as early as possible. For example, if a data stream consists of $\langle time, temperate \rangle$ tuples, but the *time* attribute is never accessed in the query, then transforming this stream into a stream of $\langle temperate \rangle$ tuples as early is possible is advantageous (e.g. any buffered tuples require less storage). Secondly, Aurora combines adjacent operators where possible, since fewer operators makes for fewer tuple transfers and simpler scheduling decisions. Lastly, Aurora examines the estimated cost and selectivity of operators (both assumed to be known a priori) and reorders operators where appropriate. For example, it makes sense to put a low-cost, high-selectivity operator before a high-cost, low-selectivity operator (assuming they commute, which not all operators do).

Runtime Operation: Aurora's runtime duties primarily consist of scheduling operators to run and resolving system overloads by shedding load (dropping tuples). Scheduling consists of selecting which operator should run, and how many tuples it should process before yielding control back to the scheduler. For some operators, processing multiple tuples all in a row (*train scheduling*) is more efficient than processing each tuple in isolation – at the very least, train scheduling reduces the overhead of the scheduler since fewer scheduling decisions are made. Thus, the scheduler tries to take into account the size of each operator's input queue. The scheduler also tries to satisfy each query output's QoS priorities. For example, a query might prioritize low latencies at the expensive of delivery rate (e.g., dropping half of the data so that I can get fast results calculated over the remaining half is better than getting slow results even if they are calculated over *all* of the data). The scheduler attempts to schedule

operators so as to maximize the QoS experienced by each output. Lastly, when a system is overloaded, queues grow unbounded and tuples must be dropped. Aurora sheds load by inserting *drop* operators at locations in the graph that minimize QoS reductions while still relieving the overload. If query outputs specify value-based QoS graphs (e.g., "Temperate values above 100 are particularly important to me"), then the drop operators can use this information to preferentially drop low-value tuples instead of dropping tuples at random.

2.2.2 TelegraphCQ

Query Model: In TelegraphCQ, users can directly specify a dataflow graph for each query (as in Aurora) or can write queries in a SQL-like language, from which a dataflow graph is derived automatically. For tuple processing, TelegraphCQ operators are largely similar to Aurora's, consisting of "standard relational operators such as joins, selections, projections, grouping and aggregation, and duplicate elimination." TelegraphCQ also provides a particularly detailed and flexible *windowing* syntax. Nearly all SPEs provide some notion of time- and tuple-based windows, and most also differentiate between *sliding* windows (which produce an output and advance one step for every tuple processed) and *tumbling* windows (which accumulate a full window of data before producing a single output and then flushing the entire window). TelegraphCQ, however, goes further by supporting essentially arbitrary windowing semantics; users specify via a for-loop-like syntax exactly how the window should evolve for each received tuple.

Static Optimizations: Compared to Aurora, TelegraphCQ places relatively lit-

tle emphasis on static optimizations, instead focusing mostly on runtime operations. The primary exception to this is TelegraphCQ's permanent mapping of queries onto threads in order to support multithreaded operation. When there are more queries than threads, each thread is responsible for multiple queries and it is beneficial to group together "similar" queries to share computation. For example, if two queries both start with the same operator, they can be combined so that the operator is only executed once on each tuple, and then the results are copied and routed to the remainder of each query.

Runtime Operation: Tuples are *adaptively routed* via Eddies [35], which are modules responsible for routing tuples through some number of (at least partially) commutative operators. The main idea is that Eddies, with relatively simple policies, are able to route tuples preferentially through more selective or less costly operators before less selective or more costly operators, *without knowing operator selectivities or costs a priori*. This is obviously beneficial; not only does it mean that profiling of operators is no longer necessary, but it also increases the robustness of the system to unexpected data patterns (e.g., an operator that is low-cost on average may be high-cost on certain tuple values). One danger, however, is that TelegraphCQ explicitly optimizes for overall throughput, and one or more individual queries may starve as a result.

2.2.3 Borealis

Borealis [54] is a strict extension of Aurora, contributing three features. Firstly, Borealis supports dynamic revision of query results. This means that if an input

stream supports *revision records* (which correct or update a prior tuple), Borealis attempts to replay the query using past data, but with the revised tuple in place of the original tuple. In this way, Borealis can create a corrected output value to replace the previous (possibly incorrect) output value. Of course, this is not always possible since Borealis is limited in its ability to buffer data; if a revision record is received for a tuple that was originally received a week ago, there is little chance that the system still has the full prior week's worth of data buffered to replay.

Secondly, Borealis supports dynamic query revision. Queries in Borealis may specify *control lines* from one operator to another, by which new parameters can be sent. For example, if an operator detects a surge in traffic which may lead to system overload (and load shedding), it could use a control line to increase the selectivity of an upstream operator. Alternatively, the operator could provide a new (perhaps less expensive) user function parameter to an upstream *map* operator.

Lastly, Borealis extends the Aurora optimizer to work in distributed settings. Borealis makes use of a hierarchy of optimizers: *local optimizers* run on each node and act locally; *neighborhood optimizers* run on each node but can also load-balance across directly-connected nodes; and a single *global optimizer* operates across the entire network. The idea is simply that any problems (system overload or latency increase) are first dealt with locally. If the situation fails to improve after some time period, the problem is escalated to the neighborhood optimizer, which attempts to load-balance across nearby nodes. Finally, if necessary, the global optimizer is invoked to attempt a solution.

2.2.4 STREAM

STREAM [105] is quite similar to TelegraphCQ is many ways. One primary differentiator of this system is how it handles overload situations; STREAM introduces tuple-dropping operators (like many systems) but also performs *window reduction*. By tightening the bounds of a windowing operator (e.g., aggregating over only the last 10 tuples, instead of the last 30), fewer tuples will need to be stored in memory. In a few special cases, the window should actually be expanded in order to reduce load, such as an operator that is part of a `NOT EXISTS` clause.

2.2.5 System S

IBM's System S [33] focuses heavily on producing highly optimized code during the initial compilation and setup process. Queries are specified in a dataflow language called Spade [75]. The compilation process first produces code for each operator, and then *fuses* operators into *processing elements* (PEs). Tuples are transferred between PEs via queues since (a) each PE runs in its own execution environment (thread or process, depending on the target system), and (b) PEs may even be running on different physical machines. On the other hand, tuples are transferred between operators within the *same* PE by direct function call, which has much lower overhead. Thus, the Spade compiler must strike a balance between creating too few PEs, which may underutilize the hardware resources (e.g., a core will be idle if there is no PE to assign to it), and creating too many PEs, which leads to excessive tuple-transfer overhead. In order to make these decisions, the Spade compiler takes into account both the specific number of machines available (as provided by the user) and the

statistical properties of each operator (e.g. cost, selectivity). The compiler can learn an operator's properties by running it through a special profiling mode.

System S supports more flexible data types than most systems. In particular, *vector* types, which are common in fields such as signal processing and data mining, are supported in addition to the usual scalar types. This is noteworthy because the Spade compiler can utilize Single-Instruction Multiple-Data (SIMD) operations available in most modern processors to accelerate basic arithmetic operations on vector types. Spade also allows queries to forgo static input stream declarations (as is the norm) and instead specify only the input stream's *type*. For example, instead of specifying a particular stock ticker source, a query can state that it will accept any input stream of type `StockTicker`. In this case the system will search for any available, type-compatible (i.e., superset of) streams, which may in fact be the output of other running queries.

2.2.6 Summary and Limitations

These systems are principally concerned with keeping up with the flow of incoming data, and they utilize a wide range of compile-time (e.g., operator reordering) and run-time (e.g., scheduling heuristics) techniques to minimize processing and storage overheads. These systems have proven successful in a variety of application domains. Aurora, for example, has been used to build applications in each of the following domains: financial services, highway traffic monitoring, military battlefield monitoring, and environmental (water toxin) monitoring. System S has been used for astronomical data mining and manufacturing process control and monitoring (amongst others).

Nonetheless, many more applications currently lie out of reach of these systems. In most cases this is due to a failure (within specific domains) to overcome the fundamental challenges we set forth in Section 1.2: *effective query interfaces* and *scalability*.

1. **Effective Query Interfaces:** As noted above, these systems all assume that streams consist of fixed-schema tuples. This is certainly due in part to the fact that many of these researchers come from the database community, and this is the model that the great majority of traditional databases use: each row of a table is effectively just such a tuple, with the table's columns dictating the (fixed) schema of each tuple. However, some applications find this model to be a poor fit; users may have difficulty expressing desired query semantics or the resulting query plan may be unnecessarily inefficient.

   For example, consider a text-processing application for which one desires an operator to parse and return all quotations from each input text block. Its unclear what *type* the output stream should have; e.g., if the input stream is of type $\langle Author, Title, Text \rangle$, an output stream of type $\langle Author, Title, ListOfQuotes \rangle$ seems reasonable, but is not supported since lists are not simple data types. Similarly, the fixed-schema requirement prevents the operator from returning a variable number of fields, as necessary (e.g., $\langle Author, Title, Quote1, Quote2, ..., QuoteN \rangle$). As a final alternative, the operator could implement a 1-to-N transformation, with one tuple (of type $\langle Author, Title, Quote \rangle$) returned for each quotation found in the input text. This schema would be supported, but has the downsides that (a) there is the overhead of repeating the $Author$ and $Title$ fields in every output tuple, and (b) breaking each text blocks' quotations up

into multiple tuples may complicate the implementation of other operators (e.g., counting the number of quotations per text block would require keeping state while tuples are still arriving, and some method of detecting when the last tuple from each text block has been received).

Additionally, in some applications it may be desirable for queries to provide feedback to the data collection process. For example, if the data source is a video camera, users may wish to write queries that execute commands to the camera (e.g., "pan left") in response to the received data. Such semantics are difficult or impossible to express in most of these systems' query interfaces. Systems that support some degree of inter-module control (e.g. feedback loops in System S and control lines in Borealis) focus on dynamically changing the query's *internal* properties, as opposed to affecting external systems.

2. **Scalability:** The process of actually collecting data and marshalling it into a stream format that can be processed by the SPE tends to be particularly domain-specific and difficult to generalize in a useful way. This is in part because data collection is, by definition, the furthest upstream operation in any query, and thus must operate on the greatest volume of data[4]. Thus, scalability is necessarily a primary concern for data collection modules. The above systems generally take the practical approach of providing a few simple adapters for data sources such as local files, web pages, or network sockets, assuming that most applications will require development of custom adapters in order to achieve the

[4]For all practical queries. A query theoretically could actually produce *more* data than it receives, but this is basically never the case.

necessary performance.

Example: Wireless Network Monitoring with Borealis

In order to make these limitations more concrete, we discuss some of the issues that arise when attempting to build a specific system; in this case, an outdoor wireless network monitor built with Borealis. Our choice of Borealis for this exercise is somewhat arbitrary and is not meant to point out specific deficiencies in this system; on the contrary, the difficulties detailed below would all arise (perhaps in a slightly different form) when using *any* of the above SPEs.

Below, we list three requirements of this application (see ch. 6 for details and justifications) that Borealis has difficulty satisfying.

1. Users should be able to easily and efficiently express a wide variety of network monitoring tasks as queries. Tuples in Borealis are collections of name-value pairs. Thus, the straight-forward way of representing a network packet as a tuple would be to assign each field of the packet to an element of the tuple. For example, an Ethernet frame could be represented as $\langle dstAddr, srcAddr, etherType, payload \rangle$ (additional fields could also be included, such as the time that the frame was captured). To do this efficiently, the tuple elements should index directly into the block of memory containing the packet data, which is not currently supported; instead, Borealis must copy out each packet field every time a new protocol header is parsed, which may occur multiple times per packet (e.g., Ethernet, IP, TCP, HTTP). Although this improvement could likely be added to Borealis, this may be non-trivial, and its omission reflects the kinds of applications that guided Borealis' development effort.

2. The system must collect wireless packets captured from a distributed set of sniffer nodes. We assume that the network bandwidth between nodes is sufficient for transferring some portion of the capture packets, but it is impossible for all nodes to forward all captured traffic to a central location for processing. In order to comply with this application constraint, it is necessary to perform some degree of data reduction *within the sniffer network*.

Borealis supports interfacing with, and moving operators into, sensor networks. This provides the basic functionality that we need in order to satisfy this application requirement. However, this complicates the task of writing correct and efficient queries. For example, a naïvely implemented query to count the number of detected 802.11 networks could overestimate the true value since a single network may be detected by multiple sniffers (and thus simply summing the networks detected by each sniffer is incorrect). For this simple query, duplication-insensitive aggregation methods are readily available to yield the correct count [95]. For more complex analyses, however, it can be rather difficult to construct queries that operate correctly and efficiently when moved onto the sniffers.

Argos approaches this problem by aggregating and redistributing packets within the network so that query modules running on a sniffer can operate on a *merged* packet stream, instead of the raw stream of packets as captured locally by the sniffer (for details, see §6.3.3). Although query writers still must take some care to aggregate values correctly, this approach has the advantages that (a) there are no duplicate packets, even across different sniffers, and (b) all traffic from any given 802.11 network is handled on just *one* sniffer (instead of spread across all sniffers that happen to be

nearby). These significantly ease the query writing process.

Argos' approach is technically possible in Borealis; one can simply require that all queries are prefixed by a carefully crafted subquery that implements the packet-merging logic. However, this is a brittle solution, because it relies on two assumptions about the Borealis compiler. Firstly, it assumes that Borealis will notice that every query starts with an identical subquery, and will optimize all N queries to share a single copy of this subquery, instead of each query inefficiently running its own (redundant) copy. Secondly, it assumes that Borealis will not perform any *other* optimizations on the queries (e.g., changing the order of commutable operators) that could modify their subquery prefix, which would in turn prevent the aforementioned N-way subquery merge. Argos avoids these dangers by placing the merging functionality in the core system itself instead of forcing queries to replicate it.

3. The query interface should include the ability to issue commands to individual sniffers to change the sniffer radio to a different 802.11 channel. 802.11 communications span multiple radio channels, only one of which can be monitored at any one time[5]. Thus, unless a system is content with foregoing traffic on all but one channel, some kind of channel *policy* will be necessary. Sniffers could, for example, rotate through each channel, spending a fixed amount of time monitoring each; or sniffers could change the channel in response to certain traffic patterns (such as when nothing "interesting" has been detected on the current channel for a while).

Assuming a custom data stream source (*adapter*) will need to be built to interface with the sniffer radios to actual receive captured packets from the OS and marshal

[5]This ignores the fact that packets from nearby channels can sometimes be "overheard", but this effect is unreliable.

them into a Borealis stream, we can easily add a channel-changing API to this component. However, Borealis provides no reasonable way for queries to access this API. At best, we could hard-code a channel policy into the component itself, but this is certainly less useful than exposing this functionality to the *queries* since different queries may have different preferences.

2.3 Domain-Specific SPEs

We refer to data sources that (a) produce non-tuple-like data, (b) support query feedback, or (c) require optimized data collection as *complex* data sources. This dissertation presents two novel systems for querying over complex data sources, but these are certainly not the first systems to do so. Instead, one of our contributions is to identify common design principles that are found across a wide variety of query systems for complex data sources, and show how these can be applied in two new domains. For illustrative purposes, we detail two existing domain-specific systems built for querying complex data sources.

2.3.1 Example 1: TinyDB

TinyDB [96] is a query system designed for low-power wireless sensor networks. It is described as an *acquisitional query processing* engine – that is, it combines both query processing and data acquisition (i.e. when and where to sample sensors in the network).

1. Query interfaces tailored to the application's specific data types:

TinyDB uses a query language that is similar to SQL, plus numerous extensions

"tailored to the sensor network domain." These include the ability to specify sensing intervals (e.g. "sample and return the temperature every 5 seconds"), event-based queries (e.g. "wait until a `bird-detection` event, then return the average temperature every 5 seconds for 30 seconds"), and lifetime-based queries (e.g. "sample and return the temperature as often as possible such that the nodes' batteries last at least 30 days"). These extensions are all different ways for users to address their scientific needs (collecting enough data from the right places at the right times) while also managing the sensor network's energy constraints.

2. Optimized data collection processes:

TinyDB employs a variety of techniques to prune unnecessary radio communication and sensor sampling, as these are both significant energy consumers in low-power wireless sensor networks. TinyDB attempts to intelligently order sensor samplings so as to skip some if predicates allow. For example, the query `'select accel, mag where accel>100 sample interval 1s'` generally requires two sensor samplings, but the magnetometer sampling is unnecessary if the predicate (`where accel>100`) fails to hold. So by sampling the accelerometer *first*, the query may be aborted early in some cases, avoid the magnetometer sampling. TinyDB also takes into account the actual energy cost of each sensor as well as the expected selectivity of each predicate. As another example, TinyDB handles network congestion by adapting sensing and transmit rates, using semantic query information to drop the least important tuples when transmit queues overflow. Though analogous to the load-shedding that many SPEs perform internally, this is differentiated by occurring *in the sensor network*; a naïve network proxy that collects data from a sensor network without knowledge of

running user queries would not be able to make these optimized decisions.

3. Allow queries to provide feedback to the collection process:

TinyDB supports arbitrary commands; implementation code is pre-compiled in the node binary and can be invoked via the query language. In particular, commands can be set up as *triggers*, to be executed when certain conditions are met. This example sounds a 512ms buzzer whenever high temperatures are detected:

```
SELECT temp FROM sensors
WHERE temp > thresh TRIGGER ACTION SetSnd(512)
```

2.3.2 Example 2: Gigascope

Gigascope [61] is a query system designed for network monitoring applications. It is designed to provide a structured query environment for complex queries in a variety of settings, while maintaining the performance required to monitor high-speed (gigabit and higher) links.

1. Query interfaces tailored to the application's specific data types:

Queries in Gigascope are specified in GSQL, a pure stream query language with SQL-like syntax. Input streams, called *Protocols*, are generated by interpreting (via a library of functions) a sequence of data packets, either from a disk or from live capture; effectively any network protocol can be supported, given an appropriate parsing library. Users can provide their own code to implement special operations, such as IP datagram defragmentation.

2. Optimized data collection processes:

Like many SPEs, Gigascope optimizes queries by rearranging the query plan. A unique aspect, however, is its ability to push some of the query logic into the network interface card (NIC) itself (depending on its capabilities). For example, some NICs support *bpf* (berkeley packet filter) expressions [99] which can be used to filter out unwanted packets as early in the query plan as possible; performing this filtering in the NIC itself significantly reduces system (kernel) load.

3. Allow queries to provide feedback to the collection process:

It does not appear that Gigascope supports any query feedback, although it is admittedly difficult to imagine what kinds of feedback/commands might be desired by users within this domain (wired network monitoring) since packet filtering is already supported by the query language.

2.4 Other Query System Architectures

Stream-based processing is far from the only model used by real-time query systems. Here we survey a few of the most relevant areas of research.

2.4.1 Traditional Distributed Pub-Sub Systems

Publish-Subscribe systems attempt to efficiently deliver content[6] from *publishers* (those that insert content into the system) to *subscribers* (those that query for and consume content out of the system), who are generally unknown to the publishers. Pub-Sub systems can generally be divided into two groups. In *topic-based* systems,

[6]Some systems are described as delivering *events* rather than content, but this is largely a stylistic distinction.

producers publish content to *topic channels*, to which users may subscribe and receive notifications when new content is posted. By contrast, in *content-based* systems subscribers describe content attributes of interest using some query language, and the system filters and matches content generated by the publishers to the subscribers' queries. For a detailed survey of pub-sub middleware literature, see Eugster et al. [68].

Topic-Based Pub-Sub Systems

A variety of topic-based pub-sub systems have been developed. One early example is the tuplespaces model of Linda [52]. Linda effectively implements a publish-subscribe paradigm where all content is in the form of *tuples*, and "topics" take the form of any subset of tuple elements. For example, if we assume `foo` is of type `integer`, then the query

```
in(``a string'', ? foo, ``another string'')
```

will match any existing 3-element tuples that have "a string" and "another string" as the first and third elements, respectively, and whose second element is of type `integer`. One may view this query as a poll for content from the topic channel

```
(``a string'', <integer>, ``another string'')
```

which is an unorthodox, but no less valid, channel identifier. This model has been adopted by a variety of commercial systems, include JavaSpaces [12], T Spaces [143] and TIBCO's event processing software [27].

The ISIS system [76], designed as a toolkit for building fault-tolerant distributed applications, provides *light-weight process groups* (LWGs) as a mechanism for communicating with multiple processes. However, users soon began using LWGs for much

more than just process groups, to the extent that a rearchitecture was necessary to support the much larger numbers of concurrent LWGs. Effectively, LWGs evolved to become arbitrary topic channels as in a dedicated pub-sub system.

A more traditional example, Herald [49], focuses on providing simple topic-based pub-sub functionality while overcoming multiple administrative domains (i.e. lack of trust between some participants), participant failures (including malicious behavior), and network partitions.

The fundamental constraint on all topic-based pub-sub systems is the requirement that publishers and subscribers agree up front on the set of topics (channels). These may not match well with a particular user's specific interests, implying that the user may miss some content of interest and/or receive extra content that they do not want.

Content-Based Pub-Sub Systems

Elvin [126, 127] is one of the earlier pub-sub system to support *content-based* subscriptions. In Elvin, each published item (called a *notification*) consists of a set of named and typed data elements; subscriptions are boolean expressions of operators over these elements. Aside from the typical operators (e.g. equals, greater-than), Elvin provides operators to test for the existence of named elements (`exists()`), check the type of an element (`datatype()`) and perform regular expression matching on a string (`matches()`).

Gryphon [134] differs by (a) distributing the middleware service over a network of servers (Elvin uses a centralized design), and (b) performing transformations of the event stream. For example, stock quote events of type `[ticker, price,`

`volume]` can be transformed into new events of type `[ticker, capital]` where `capital=price*volume`. Subscriptions based on the *price* or *volume* attributes can be matched on event prior to this transformation, whereas those based on the *capital* attribute can can matched after. Although this model of event transformations has some similarities to generalized query processing systems (e.g. the SPEs above), an important distinction is that in Gryphon the graph of transformations are determined statically and *not* by the user queries (subscriptions).

Hermes [118] functions as a hybrid system. Subscriptions first specify an *event type*, which is effectively a topic, and then specify a number of filters over the attributes of that event type. The authors argue that this maintains much of the flexibility of content-based pub-sub while better managing large numbers of different events.

Hermes and Siena [53] are both designed as wide-area services, and thus focus particularly on overlay-based network topologies and routing strategies for both content and subscriptions. Siena in particular supports a collection of different client-server and peer-to-peer topologies, and implements a variety of different routing and replication strategies to ensure efficient communication between servers.

Limitations

Pub-sub systems are limited in their ability to serve as real-time query systems. First and foremost, regardless of whether topic-based or content-based filtering is supported, the query (subscription) languages are quite limited. The subscription interfaces are designed for *identifying* content, not transforming it. There is generally

no way to express even simple aggregates. Of course clients could perform these aggregations themselves, but this comes at a heavy cost. For example, if 10 clients of a stock application each want a count of the number of trades above 100 shares, most SPEs would calculate this aggregate once and then send a copy of the result to each client. A pub-sub system would instead have to send a copy of every trade event above 100 shares to each client, each of which would locally (and redundantly) compute the count.

2.4.2 Web Search Engines

Broadly speaking, the World Wide Web is arguably the largest content provider today. The basic design of the Web differs from query systems by (a) requiring users to request content explicitly (instead of querying for results), and (b) operating in a pull (users request content) rather than a push (servers send content asynchronously) mode. Since the Web's birth, search engines [46] have proliferated as a means by which users can search for content, and the Atom and RSS standards were developed as a convenient way for websites to publish content updates. Although Atom and RSS still require client polling (maintaining the user-pull modality of the Web), client reader applications exist to poll a user-defined list of websites on a regular basis and alert on new updates.

A full treatment of research in web search is well beyond the scope of this dissertation, but it is useful to consider a few specific subareas. Firstly, a number of "blog search engines" have come online in recent years, including Feedster, Blogdigger, Bloglines, IceRocket, and Technorati. Apart from Google and MSN's blog search

services, most of these sites appear to be backed by small startup companies and little is known about how they operate. In particular, their ability to scale to large numbers of feeds and users, use network resources efficiently, and maintain low update latencies is unknown. In Chapter 4 we attempt to measure the update latency of several of these sites. As for MSN and Google, we expect these sites leverage the vast numbers of server resources distributed across many data centers to rapidly index updates to blogs. Although an academic research group cannot hope to garner these kinds of resources, by developing Cobra we hope to shed light on important design considerations and tradeoffs for this interesting application.

Secondly, there is some relation between deep web mining and querying over complex data sources. The "deep web" refers to online content that is not easily crawled and indexed by search engines, such as databases behind HTML web forms. Various studies have estimated that the deep web hides an enormous amount of information, perhaps an order of magnitude more than the currently searchable web [82, 40]. The difficulty in reaching this content lies both in (a) finding the online web forms that access it, and (b) determining (in an automated fashion) what the appropriate range of inputs to the form are. This is challenging since web forms are designed for humans to understand, not software, and there is little in the way of standards or consistency [97, 147].

This problem is not unlike those encountered when designing a query system for any set of diverse, uncoordinated data sources: how does the query system (a) learn about the data sources in the first place, and (b) learn what kind of data is produced and what interactions (e.g. requests or feedback) are supported by the data source.

Currently there are no well-established indices of available online data sources, and few standards for data formatting or sensor interfaces – at best, sites will post human-readable descriptions that are typically difficult for automated processes to find and understand.

2.4.3 Content-Delivery Networks

Content-delivery networks (CDNs) are another method of pairing users with desired content. CDNs replicate content from one or more *origin* websites and distribute it through the network. When a client makes a request, they are directed to the nearest[7] CDN server, which then provides the requested content. This spreads out requests across the CDN network, relieving load on the origin websites, and can improve client performance since content is generally returned more quickly from the CDN – due to network locality – than it would be from the origin websites. Commercial CDNs are currently quite popular; Akamai [2] and Limelight Networks [13] are well-known examples. Coral [73] and CoDeeN [109] are two research CDNs running on Planetlab [117]. CoralCDN functions like a traditional CDN since websites opt-in to having their content served, whereas CoDeeN is viewed most naturally as a web proxy since *users* opt-in to using it. Freedman et al. explain the main difference between these systems thusly: "Although CoDeeN gives *participating* users better performance to *most* web sites, CoralCDN's goal is to gives *most* users better performance to *participating* web sites–namely those whose publishers have 'Coralized' the URLs." Overall, CDNs are relevant by virtue of their focus on efficient data collec-

[7] "Nearest" usually means "lowest round-trip time", but client assignment can take into account other factors as well, such as server load.

tion and/or dissemination, although all major systems require content to be named explicitly (e.g., via URL) instead of providing any kind of query interface.

Corona [121] incorporates aspects of both pub-sub systems and peer-to-peer CDNs. Users specify interest in specific URLs (like a CDN) and Corona periodically polls the URL for changes, which are then sent as an update to users (like a pub-sub system) via instant message notifications. The main goal of Corona is to mitigate the *polling overhead* placed on monitored URLs, which is accomplished by spreading polling load among cooperating peers and amortizing the overhead of crawling across many users interested in the same URL. An informed algorithm determines the optimal assignment of polling tasks to peers to meet system-wide goals such as minimizing update detection time or minimizing load on content servers. Compared to Cobra, Corona provides more flexible trade-offs between the polling load imposed on content sources (web servers) and the average update detection latency (time between when new content is posted and when Corona or Cobra detects it). On the other hand, Corona does not permit an individual user to specify content-based predicates for selecting content; effectively, Corona implements topic-based pub-sub (where URLs are topics).

2.4.4 MapReduce-related Systems

MapReduce [64] is a programming model for distributed processing of large data sets. MapReduce was originally designed to operate in an *offline* manner but its rapid growth in popularity has lead to a number of projects that integrate MapReduce-style processing with streaming data.

DEDUCE [89] is an extension of System S (§2.2.5) that allows users to embed

MapReduce jobs within a real-time query plan. As a motivating example, the authors suggest that automated stock trading applications "usually require periodic analysis of large amounts of stored data to generate a model using MapReduce, which is then used to process a stream of incident updates using a stream processing system."

A different approach is taken by the Cascading [3] project, which acts as a wrapper over the Apache Hadoop [24] implementation of MapReduce. Cascading provides a stream-like API that makes it easier to string together workflows consisting of multiple MapReduce jobs. Thus, Cascading does not fundamentally change the actual *operation* of the jobs in question (in particular, it does not appear to add the ability to process data arriving in real-time), but instead changes the query *interface* from the traditional MapReduce-style to a streaming-data style.

2.5 Network Monitoring and Measurement

Argos (ch. 6-7) is a query system designed for outdoor, wireless network monitoring. In functionality, Argos is related to prior (domain-specific) SPEs built for network monitoring and analysis; Gigascope is presented above (§2.3.2) as a representative example of this group. The primary differentiator is that these systems are optimized for *centralized*, *wired* packet capture, whereas Argos is optimized for *distributed*, *wireless* packet capture. For Argos, this necessitates a variety of new techniques for efficiently processing traffic in a distributed fashion.

We also survey a number of network monitoring systems that are *not* architected as query systems; e.g., systems that are built to capture all the network traffic for offline analysis, without any kind of online querying or filtering functionality. Despite

this, a brief summary of these systems is useful since many of these systems were nonetheless influential in the design of Argos.

2.5.1 Wired Network Monitoring

A variety of systems have been designed for the monitoring of wired networks. Many systems in this class are concerned primarily with ways to manage the high traffic volumes that can be encountered, particularly on core Internet links such as ISP backbones. High traffic volumes can stress a host of components within monitoring systems, including the capturing interface, the CPU (depending on the amount of analyses or number of buffer copies per packet), the available RAM (when saving per-flow statistics), and the local storage.

OC3MON [34], an integrated hardware-and-software solution, saves only per-flow statistics. These are then saved to disk for offline web querying. IPMON [72] saves the first 40 bytes of every packet; the storage requirements of this policy means that continuous operation is infeasible. Instead, IPMON is designed to be run simultaneously at multiple network locations (pains are taken to time-synchronize across locations) for a few hours or (at best) days; these traces are then physically sent to a data repository for offline analyses. Nprobe [102] uses protocol-specific compression to reduce the volume of traffic logged to disk. Each packet is passed through a series of modules that handle each protocol layer (e.g. IP, TCP, HTTP), discarding data that is not considered important (how this determination is made is not discussed).

None of the above systems support any real notion of queries. This is undoubtedly because they are all essentially concerned only with capturing data and storing it

(possibly in a reduced form) for offline analysis. However, many wired monitoring systems *do* incorporate some kind of query processing.

Pandora [112] implements a middle-ground design; although Pandora's end goal remains to store data for later analysis, users are able to specify a series of operators through which packets should flow in order to transform the data prior to storage. These operators perform protocol-related operations, such as stripping a certain header or reconstructing higher-level payloads from multiple packets; there is no notion of the traditional aggregation operators found in SPEs (e.g. `count`, `sum`). For example, given an input stream of Ethernet frames, a user could choose to save all raw IP datagrams, or instead could perform IP defragmentation and TCP reconstruction, thereby saving only the TCP payloads.

Like Gigacope (§2.3.2), Tribeca [135] is a system designed explicitly for analysis of network traffic. These systems can be used to monitor live network traffic or perform offline analyses of network traces (one could, for example, use OC3MON or IPMON to capture a trace, and then later use Tribeca or Gigacope to analyze it). The query languages of these systems are quite different; in Tribeca, users specify an operator graph explicitly, whereas Gigascope uses a SQL-like language. However, both provide a combination of network-traffic-specific operations (e.g., selecting or joining on specific header fields) and general aggregation operations (e.g., `count`, `sum`).

Lastly, there is also a class of systems focused on network intrusion detection. Although some of the above systems are intended to be *suitable* for this purpose, Snort [124] and Bro [113] are two systems designed explicitly for network intrusion

detection (and *prevention* in some cases). At this point, both of these systems are quite mature, with active developer and user communities and "out of the box" support for detecting a large number of common network attacks. The main difference is that Snort is *rule*-based, where each rule specifies some kind of pattern to search for in the network traffic (e.g., a malware signature, or illegal sequence of messages); Bro, on the other hand, is *event*-based, where user-supplied event-handler functions are called whenever important protocol events occur (e.g., TCP connection established, FTP reply sent, Finger request sent).

2.5.2 Indoor WLAN Monitoring

Wireless LAN monitoring, by which we mean capture of real wireless traffic (as opposed to, e.g., analysis of SNMP data, or capture of wireless traffic after it has passed through a gateway into a wired network), was pioneered by Yeo et al. [144, 145] under small-scale conditions (3 sniffers, < 10 access points). An important discovery was that multiple wireless sniffers working in coordination can usually obtain a more accurate picture of the wireless environment than a single sniffer can, due to the spatial variability of wireless transmissions. This is in contrast to wired networks, where monitors on the same LAN nearly always capture identical packet streams (and thus more than one monitor is generally redundant, except for load-balancing or fault-tolerance).

Since then, a number of systems have built on Yeo et al.'s techniques, of which Jigsaw and Wit are representative. Jigsaw [59, 60] is a system for diagnosing MAC-layer behavior by capturing detailed packet traces from a large number of densely-deployed

wireless sniffers. The authors show that by capturing nearly-complete traces of the wireless activity in a building, a number of analyses can be performed, particularly in the area of diagnosing performance problems. Jigsaw was deployed on 39 sensor pods across a single building. A contemporary system, Wit [98], has similar goals to Jigsaw, but emphasizes the use of a formal language for specifying 802.11 behavior, which is then used to perform inference on the captured packet traces. Wit was deployed on five sniffers to monitor traffic during a conference. Antler [120] detects and diagnoses performance problems in real-time, but without the cost of continually capturing all observed network traffic. Instead, each access point initially captures only a few low-bandwidth performance statistics. If a problem appears, Antler progressively "hones in" on the root cause by collecting more traffic that will aid in diagnosis.

The Dartmouth MAP system [128] has a slightly different emphasis; instead of diagnosing performance problems, MAP is designed to detect malicious wireless network activity, such as the presence of rogue access points or a deauthentication attack. In order to monitor all 11 of the 802.11 b/g channels, MAP employs intelligent channel-hopping schemes, some of which we make use of and build upon in Argos.

An important point to note is that all of these systems rely on a dense deployment of sniffers and perform a "microscopic" observation of a single wireless network environment. Argos, in contrast, utilizes a sparse, outdoor sniffer network to perform "macroscopic" observations across an entire city.

2.5.3 Wardriving Studies

The standard approach to studying urban wireless network deployments is *wardriving*, in which a mobile sniffer is used to detect the presence and static properties of deployed wireless LANs [32, 81]. Wardriving studies in numerous cities have revealed an extremely high penetration of wireless networks, and the online Wigle.net database reports over 19 million unique wireless LANs [25]. However, these studies do not typically include any analysis of wireless traffic itself since observations are only made over short periods of time. A related measurement study [48] explored the extent to which wireless clients in moving vehicles can establish Internet connectivity via open access points.

2.5.4 Wireless Sensor Networks

Passive monitoring has also been explored in low-power wireless sensor networks (WSN), including SNIF [123] and LiveNet [57]. Traditionally, debugging or diagnosing network performance anomalies in a distributed system is often accomplished with the use of data recorded by the system itself (e.g., event logs, performance statistics). However, the limited resources of WSN nodes complicates this; the flash storage on a mote may not be sufficient for long-term logging, or the available RAM may be insufficient for debugging code to be included in the executable, for example. Instead, some WSN utilize external wireless monitoring systems to capture data useful for debugging purposes. Thus, these systems focus on just a single sensor network rather than many existing wireless LANs.

Wishbone [107] is system that partitions stream-based applications (specified as

a dataflow graph of operators) such that portions of the application are run on the actual sensor nodes and others are run on the backend server. Wishbone attempts to find a partitioning that minimizes network traffic (e.g., by pushing data-reduction operators into the network) without exceeding the sensor nodes' CPU resources. Argos also distributes queries over both the sniffer network and the backend server, although Argos requires users to hand-partition their dataflow graph. If Wishbone were extended to work with the Click language[8], it would be interesting to explore incorporating it into Argos for automatic query partitioning.

2.5.5 Mesh Network Measurements

A number of studies of the performance and traffic characteristics of urban-scale mesh networks have been published. The RoofNet project has undertaken detailed studies of link-layer and routing performance across a WiFi mesh of 38 nodes across a city [30, 42]. These studies focused on the low-level network performance rather than ambient traffic. A study of the Google WiFi mesh [28] in Mountain View, CA explored the behavior of mesh network users, but did so through capturing data at the wired gateways, and did not look explicitly at the wireless network dynamics.

2.6 Summary

Work related to real-time query systems can largely be divided into three main groups. First are general-purpose stream processing engines (SPEs), which provide

[8]Wishbone is tied to the WaveScript [108] language.

an application-agnostic platform for database-like stream processing operations. Second are domain-specific SPEs, which also use an operators-over-streams style of processing, but are tailored to a particular application domain (typically one for which general-purpose SPEs are poorly suited). Finally, there are a variety of query systems that do not following a streaming model, including pub-sub systems, web search engines, CDNs, and specialized network monitoring tools.

Chapter 3

Cobra Architecture

3.1 Introduction

Weblogs, RSS feeds, and other sources of "live" Internet content have been undergoing a period of explosive growth. The popular blogging site LiveJournal reports nearly 30 million accounts, just over 1 million of which were active over the last month [14]. Technorati, a blog tracking site, reported in 2008[1] that they were tracking 133 million blogs in 81 different languages; collectively these bloggers created nearly 1 million new posts every day [130]. These numbers are staggering and suggest a significant shift in the nature of Web content from mostly static pages to continuously updated conversations.

The problem is that finding interesting content in this burgeoning blogosphere is extremely difficult. It is unclear that conventional Web search technology is well-

[1] 2008 is the last year for which Technorati published an overall blog count in their annual "State of the Blogosphere" posting.

suited to tracking and indexing such rapidly-changing content. Many users make use of RSS feeds, which in conjunction with an appropriate reader, allow users to receive rapid updates to sites of interest. However, existing RSS protocols require each client to periodically poll to receive new updates. In addition, a conventional RSS feed only covers an individual site, such as a blog. The current approach used by many users is to rely on RSS aggregators, such as SharpReader and FeedDemon, that collect stories from multiple sites along thematic lines (e.g., news or sports).

Our vision is to provide users with the ability to perform *content-based filtering and aggregation* across millions of Web feeds, obtaining a *personalized* feed containing only those articles that match the user's interests. Rather than requiring users to keep tabs on a multitude of interesting sites, a user would receive near-real-time updates on their personalized RSS feed when matching articles are posted. Indeed, a number of "blog search" sites have sprung up in recent years, including Feedster, Blogdigger, and Bloglines. However, due to their proprietary architecture, it is unclear how well these sites scale to handle large numbers of feeds, vast numbers of users, and maintain low latency for pushing matching articles to users. Conventional search engines, such as Google, have also added support for searching blogs as well but also without any evaluation.

This chapter describes Cobra (Content-Based RSS Aggregator), a distributed, scalable system that provides users with a personalized view of articles culled from potentially millions of RSS feeds. Cobra consists of a three-tiered network of *crawlers* that pull data from web feeds, *filters* that match articles against user subscriptions, and *reflectors* that serve matching articles on each subscription as an RSS feed, that

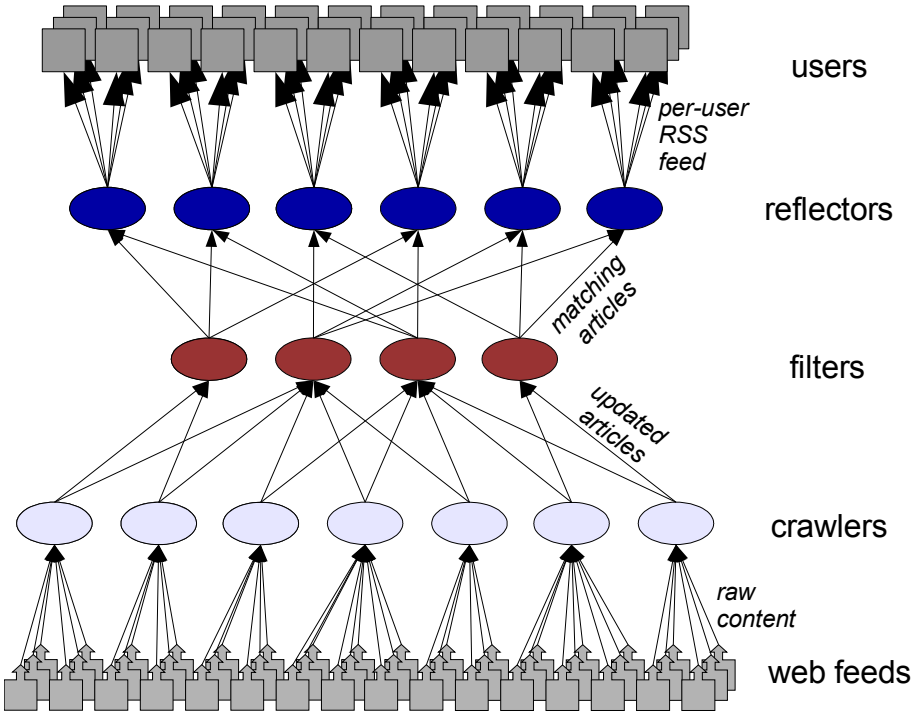


Figure 3.1: **The Cobra content-based RSS aggregation network.**

can be browsed using a standard RSS reader. Each of the three tiers of the Cobra network is distributed over multiple hosts in the Internet, allowing network and computational load to be balanced, and permitting locality optimizations when placing services.

3.2 Cobra System Design

Figure 3.1 shows the overall architecture of Cobra. Cobra consists of a three-tiered network of *crawlers*, *filters*, and *reflectors*. Crawlers are responsible for periodically crawling web feeds, such as blogs, news sites, and other RSS feeds, which we collectively call *source feeds*. A source feed consists of a series of *articles*. The number

of articles provided by a source feed at any time depends on how it is configured; a typical blog or news feed will report only the most recent 10 articles or so. As described below, Cobra crawlers employ various techniques to reduce polling load by checking for updates in the source feeds in a lightweight manner.

Crawlers send new articles to the filters, which match the content of those articles against the set of user *subscriptions*, using a case-insensitive, index-based matching algorithm. Articles matching a given subscription are pushed to the appropriate reflector, which presents to the end user a personalized RSS feed that can be browsed using a standard RSS reader. The reflector caches the last k matching articles for the feed (where k is typically 10), requiring that the user poll the feed periodically to ensure that all matching articles will be detected. This behavior matches that of many existing RSS feeds that limit the number of articles included in the feed. Although the reflector must be polled by the user (as required by current RSS standards), this polling traffic is far less than requiring users to poll many thousands of source feeds. Also, it is possible to replace or augment the reflector with push-based notification mechanisms using email, instant messaging, or SMS; we leave this to future work.

The Cobra architecture uses a simple congestion control scheme that applies back-pressure when a service is unable to keep up with the incoming rate of data from upstream services. Each service maintains a 1MB data buffer for each upstream service. If an upstream service sends data faster than it can be processed, the data buffer will fill and any further send attempts will block until the downstream service can catch up, draining the buffer and allowing incoming data again. This ensures that the crawlers do not send new articles more quickly than the filters can process them, and

likewise that the filters do not pass on articles faster than the reflectors can process them. The provisioner (detailed in Section 3.3) takes this throttling behavior into account when verifying that each crawler will be able to finish crawling its entire list of feeds every 15 minutes (or whatever the *crawl-rate* is specified to be).

3.2.1 Crawler service

The crawler service takes in a list of source feeds (given as URLs) and periodically crawls the list to detect new articles. A naive crawler would periodically download the contents of each source feed and push all articles contained therein to the filters. However, this approach can consume a considerable amount of bandwidth, both for downloading the source data and sending updates to the filters. In a conventional usage of RSS, many users periodically polling a popular feed can have serious network impact [121]. Although Cobra amortizes the cost of crawling each feed across all users, the sheer number of feeds demands that we are careful about the amount of network bandwidth we consume.

The Cobra crawler includes a number of optimizations designed to reduce bandwidth usage. First, crawlers attempt to use the HTTP `Last-Modified` and `ETag` headers to check whether a feed has been updated since the last polling interval. Second, the crawler makes use of HTTP delta encoding for those feeds that support it.

When it is necessary to download the content for a feed (because its HTTP headers indicate it has changed, or if the server does not provide modification information), the crawler filters out articles that have been previously pushed to filters, reducing

bandwidth requirements further and preventing users from seeing duplicate results. We make use of two techniques. First, a whole-document hash using Java's hashCode function is computed; if it matches the previous hash for this feed, the entire document is dropped. Second, each feed that has changed is broken up into its individual articles (or *entries*), which are henceforth processed individually. A hash is computed on each of the individual articles, and those matching a previously-hashed article are filtered out. As we show in Chapter 4, these techniques greatly reduce the amount of traffic between source feeds and crawlers and between crawlers and filters.

3.2.2 Filter service

The filter service receives updated articles from crawlers and matches those articles against a set of *subscriptions*. Each subscription is a tuple consisting of a *subscription ID*, *reflector ID*, and list of *keywords*. The subscription ID uniquely identifies the subscription and the reflector ID is the address of the corresponding reflector for that user. Subscription IDs are allocated by the reflectors when users inject subscriptions into the system. Each subscription has a list of keywords that may be related by either conjunctions (e.g. "law AND internet"), disjunctions (e.g. "copyright OR patent"), or a combination of both (e.g. "(law AND internet) OR (privacy AND internet)"). When an article is matched against a given subscription, each word of the subscription is marked either *true* or *false* based on whether it appears anywhere in the article; if the resulting boolean expression evaluates to *true* then the article is considered to have matched the subscription.

Given a high volume of traffic from crawlers and a large number of users, it is

essential that the filter be able to match articles against subscriptions efficiently. Cobra uses the matching algorithm proposed by Fabret *et al.* [69, 116]. This algorithm operates in two phases. In the first phase, the filter service uses an index to determine the set of all words (across all subscriptions) that are matched by any article. This has the advantage that words that are mentioned in multiple subscriptions are only evaluated once. In the second phase, the filter determines the set of subscriptions in which all words have a match. This is accomplished by ordering subscriptions according to overlap and by ordering words within subscriptions according to selectivity, to test the most selective words first. If a word was not found in the first phase, all subscriptions that include that word can be discarded without further consideration. As a result, only a fraction of the subscriptions are considered if there is much overlap between them.

Due to its sub-linear complexity, the matching algorithm is extremely efficient: matching a single article against 1 million user subscriptions has a 90th percentile latency of just 10 ms (using data from real Web feeds and synthesized subscriptions, as discussed in Chapter 4). In contrast, a naive algorithm (using a linear search across the subscription word lists) requires more than 10 sec across the same 1 million subscriptions, a four order of magnitude difference.

3.2.3 Reflector service

The final component of the Cobra design is the reflector service, which receives matching articles from filters and reflects them as a personalized RSS feed for each user. In designing the reflector, several questions arose. First, should the filter service

send the complete article body, a summary of the article, or only a link to the article? Clearly, this has implications for bandwidth usage. Second, how should filters inform each reflector of the set of matching subscriptions for each article? As the number of matching subscriptions increases, sending a list of subscription IDs could consume far more bandwidth than the article contents themselves.

In our initial design, for each matching article, the filter would send the reflector a summary consisting of the title, URL, and first 1 KB of the article body, along with a list of matching subscription IDs. This simplifies the reflector's design as it must simply link the received article summary to the personalized RSS feed of each of the matching subscription IDs. Article summaries are shared across subscriptions, meaning if one article matches multiple subscriptions, only one copy is kept in memory on the reflector.

However, with many active subscriptions, the user list could grow to be very large: with 100,000 matching subscriptions on an article and 32-bit subscription IDs, this translates into 400KB of overhead *per article* being sent to the reflectors. One alternative is to use a bloom filter to represent the set of matching users; we estimate that a 12KB filter could capture a list of 100,000 user IDs with a false positive rate of 0.08%. However, this would require the reflector to test each user ID against the filter on reception, involving a large amount of additional computation.

In our final design, the filter sends the complete article body to the reflector without a user list, and the reflector *re-runs* the matching algorithm against the list of active subscriptions it stores for the users it is serving. Since the matching algorithm is so efficient (taking 10ms for 1 million subscriptions), this appears to be

the right trade-off between bandwidth consumption and CPU overhead. Instead of sending the complete article, we could instead send only the union of matching words across all matching subscriptions, which in the worst case reduces to sending the full text.

For each subscription, Cobra caches the last k matching articles, providing a personalized feed that users can access using a standard RSS reader. The value of k must be chosen to bound memory usage while providing enough content that a user is satisfied with the "hits" using infrequent polling; typical RSS readers poll every 15-60 minutes [94]. In our current design, we set $k = 10$, a value that is typical for many popular RSS feeds (see Figure 4.1). Another approach might be to dynamically set the value of k based on the user's individual polling rate or the expected popularity of a given subscription. We leave these extensions to future work.

In the worst case, this model of user feeds leads to a memory usage of $k * subscriptions * 1KB$ (assuming articles are capped at 1KB of size). However, in practice the memory usage is generally much lower since articles can be shared across multiple subscriptions. In the event that memory does become scarce, a reflector will begin dropping the content of new articles that are received, saving to users' feeds only the articles' titles and URLs. This greatly slows the rate of memory consumption, but if memory continues to dwindle then reflectors will begin dropping *all* incoming articles (while logging a warning that this is happening). This process ensures a graceful degradation in service quality when required.

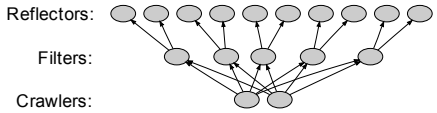
A user subscribes to Cobra by visiting a web site that allows the user to establish an account and submit subscription requests in the form of keywords. The web server

coordinates with the reflectors and filters to *instantiate* a subscription, by performing two actions: (1) associating the user with a specific reflector node; and (2) injecting the subscription details into the reflector node and the filter node(s) that feed data into that reflector. The response to the user's subscription request is a URL for a private RSS feed hosted by the chosen reflector node. In our current prototype, reflector nodes are assigned randomly to users by the Web server, but a locality-aware mechanism such as Meridian [142] or OASIS [74] could easily be used instead.

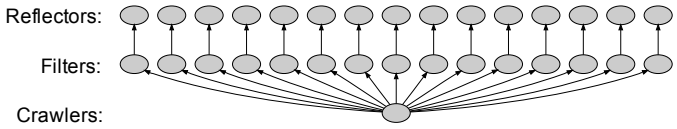
3.2.4 Hosting model

We designed Cobra to take advantage of the features offered in today's cloud offerings, including easy replication of services, redundancy across geographic and administrative regions, and highly available elastic resources. Distribution also allows the placement of Cobra services to take advantage of improved locality when crawling blogs or pushing updates to users.

This hosting model allows us to make certain assumptions to simplify Cobra's design. First, we assume that physical resources in a hosting center can be dedicated to running Cobra services, or at least that hosting centers can provide adequate virtualization [23, 39] and resource containment [38] to provide this illusion. Second, we assume that Cobra services can be replicated within a hosting center for increased reliability. Third, we assume that hosting centers are generally well-maintained and that catastrophic outages of an entire hosting center will be rare. Cobra can tolerate outages of entire hosting centers, albeit with reduced harvest (fraction of content servers actively crawled) and yield (fraction of results feeds accessible to users) [71].



(a) Configuration for 4x CPU and 25 Mbps bandwidth



(b) Configuration for 1x CPU and 100 Mbps bandwidth

Figure 3.2: **Operation of the Cobra network provisioner.** These figures show how provisioner results can vary for different constraint combinations; in both of these cases the network is provisioned for 800,000 feeds, 8 million subscriptions, and 1 GB of memory, but the CPU and bandwidth constraints differ. (a) Shows the resulting configuration when the CPU constraint is 4x the default value (see text) and the bandwidth constraint is 25 Mbps. (b) Shows the resulting configuration when the CPU constraint is the default value (1x) and the bandwidth constraint is 100 Mbps.

Finally, we assume that allocating resources to Cobra services and monitoring their performance at runtime can be performed centrally. These assumptions strongly influence our approach to service provisioning as we are less concerned with tolerating unexpected variations in CPU and network load and intermittent link and node failures, as is commonly seen on open experimental testbeds such as PlanetLab [117].

3.3 Service provisioning

As the number of source feeds and users grows, there is a significant challenge in how to provision the service in terms of computational horsepower and network bandwidth. Server and network resources cost money; additionally, a system may have limitations on the amount of physical resources available. Our goal is to determine the minimal amount of physical resources required to host a Cobra network capable of supporting a given number of source feeds and users. For this purpose, we make use of an offline *service provisioning* technique that determines the configuration of the Cobra network in terms of the number of crawlers, filters, and reflectors, as well as the interconnectivity between these services.

The provisioner takes as inputs the target number of source feeds and users, a model of the memory, CPU and bandwidth requirements for each service, as well as other parameters such as distribution of feed sizes and the per-user polling rate. The provisioner also takes as input a set of node constraints, consisting of limits on inbound and outbound bandwidth, maximum memory available, and CPU processing power. Note that this last value is difficult to measure directly and thus we model it simply as a dimensionless parameter relative to the processing performance observed on Emulab's pc3000 machines[2]. For example, a CPU constraint of 0.75 implies that the provisioner should assume that nodes will process messages only 75% as fast as the pc3000s. The provisioner's output is a graph representing the topology of the Cobra network graph, including the number of feeds assigned to each crawler and the number of subscriptions assigned to each reflector and each filter.

[2] 3.0 GHz 64-bit Xeon processors

Our provisioner makes the simplifying assumption that, with regards to resource consumption, all source feeds are equivalent, as are all user subscriptions. This is certainly not true in reality; source feeds can vary widely in update rate and content size [94], and users will submit queries of varying length (which affects CPU load) and will poll their results feeds at different rates. However, we assume that, for typical parameter ranges, the number of feeds assigned to each crawler and the number of subscriptions assigned to each filter and reflector will be large enough that the properties of individual feeds and subscriptions will be insignificant compared to their average behavior.

Additionally, the provisioner models each Cobra service as running on a separate physical host with independent memory, CPU and bandwidth constraints. This results in a conservative estimate of resource requirements as it does not permit multiple services within a hosting center to share resources (e.g., bandwidth). A more sophisticated algorithm could take such resource sharing into account.

The provisioner attempts to configure the network to meet the target number of source feeds and users while minimizing the number of services. The algorithm operates as follows. It starts with a simple 3-node topology with one crawler, one filter, and one reflector. In each iteration, the algorithm identifies any constraint violations in the current configuration, and greedily resolves them by *decomposing* services as described below. When no more violations exist, the algorithm terminates and reports success, or if a violation is found that cannot be resolved, the algorithm terminates and reports failure.

An *out-decomposition* resolves violations by replacing a single service with n repli-

cas such that all incoming links from the original service are replicated across the replicas, whereas the outgoing links from the original services are load balanced across the replicas. An *in-decomposition* does the opposite: a single service is replaced by n replicas such that all outgoing links from the original service are replicated across the replicas, whereas the incoming links from the original services are load balanced across the replicas.

In resolving a violation on a service, the choice of decomposition type (in- or out-) depends both on the type of violation (in-bandwidth, out-bandwidth, CPU, or memory) and the type of service (crawler, filter or reflector). Figure 3.3 shows which decomposition is used in each situation.

When faced with multiple violations, the algorithm uses a few simple heuristics to choose the order in which to resolve them. Some violations have the potential to be resolved indirectly in the course of resolving other violations. For example, if a crawler service has both in-bandwidth and out-bandwidth violations, resolving the in-bandwidth violation is likely to also resolve the out-bandwidth violation (by reducing the number of feeds crawled, we also implicitly reduce the number of feed updates that are found and output by the crawler). Thus it is preferable in this case to resolve the in-bandwidth violation first as it may solve both violations with one decomposition. In general, when choosing which of multiple violations to resolve first, the algorithm will choose the violations with the least potential to be resolved indirectly, thus saving the violations with higher potential until as late as possible (in the hopes that they will "happen to be resolved" in the mean time).

Although this greedy approach might lead to local minima and may in fact fail

Service	Violation	Decomposition & Reason
Crawler	in-bw	*in* – reduces # of feeds crawled
	out-bw	*in* – reduces rate of updates (fewer sent to filters)
	cpu	*none* – crawler cpu is not modeled
	memory	*none* – crawler memory is not modeled
Filter	in-bw	*in* – reduces # of crawlers sending updates to each filter
	out-bw	*in* – reduces rate that articles are received, reducing rate that articles are output to reflectors
	cpu	*out* – reduces # of subscriptions that articles match against
	memory	*out* – reduces # of subscriptions stored on the filter
Reflector	in-bw	*none* – not resolvable; reflectors need updates from all feeds (so users can receive all matching articles)
	out-bw	*out* – reduces # of subscriptions, reducing web request rate (of match results) by users
	cpu	*out* – reduces # of subscriptions that articles match against and # of article-queues that are updated
	memory	*out* – reduces # of stored subscriptions and article lists

Figure 3.3: **Provisioner choice of decomposition for each service/violation combination.**

to find a topology that satisfies the input constraints when such a configuration does exist, in practice the algorithm produces network topologies with a modest number of nodes to handle large loads. We chose this greedy iterative approach because it was conceptually simple and easy to implement. Figure 3.2 shows two provisioner topologies produced for different input constraints.

3.3.1 Service instantiation and monitoring

The output of the provisioner is a *virtual graph* (see Figure 3.2) representing the number and connectivity of the services in the Cobra network. Of course, these services must be instantiated on physical hosts. A wide range of instantiation policies could be used, depending on the physical resources available. For example, a small startup might use a single hosting center for all of the services, while a larger company might distribute services across multiple hosting centers to achieve locality gains. Both approaches permit incremental scalability by growing the number of machines dedicated to the service.

The Cobra design is largely independent of the mechanism used for service instantiation. In our experiments described in Chapter 4, we use different strategies based on the nature of the testbed environment. In our dedicated cluster and Emulab experiments, services are mapped one-to-one with physical hosts in a round-robin fashion. In our PlanetLab experiments, services are distributed randomly to achieve good coverage in terms of locality gains for crawling and reflecting (described below). An alternate mechanism could make use of previous work on network-aware service placement to minimize bandwidth usage [31, 119].

After deployment, it is essential that the performance of the Cobra network be monitored to validate that it is meeting targets in terms of user-perceived latency as well as bandwidth and memory constraints. Also, as the user population and number of source feeds grow it will be essential to re-provision Cobra over time. We envision this process occurring over fairly coarse-grained time periods, such as once a month or quarter. Each Cobra node is instrumented to collect statistics on memory usage,

CPU load, and inbound and outbound bandwidth consumption. These statistics can be collected periodically to ascertain whether re-provisioning is necessary.

3.3.2 Source feed mapping

Once crawler services have been instantiated, the final step in running the Cobra network is to assign source feeds to crawlers. In choosing this assignment, we are concerned not only with spreading load across multiple crawlers, but also reducing the total *network load* that the crawlers will induce on the network. A good way to reduce this load is to optimize the locality of crawlers and their corresponding source feeds. Apart from being good network citizens, improving locality also reduces the latency for crawling operations, thereby reducing the update detection latency as perceived by users. Because the crawlers use fairly aggressive timeouts (5 sec) to avoid stalling on slow feeds, reducing crawler-feed latency also increases the overall yield of a crawling cycle.

We assign source feeds to crawlers in a latency-aware fashion. One approach is to have each crawler measure the latency to all of the source feeds and use this information to perform a coordinated allocation of the source feed list across the crawlers. Alternately, we could make use of network coordinate systems, such as Vivaldi [63], which greatly reduces ping load by mapping each node into a low-dimensional coordinate space, allowing an estimate of the latency between any two hosts to be measured as the Euclidean distance in the coordinate space. However, such schemes require end hosts to run the network coordinate software, which is not possible in the case of oblivious source feeds.

Instead, we perform an offline measurement of the latency between each of the source feeds and crawler nodes using King [79]. King estimates the latency between any two Internet hosts by performing an external measurement of the latency between their corresponding DNS servers; King has been reported to have a 75th percentile error of 20% of the true latency value. It is worth noting that many source feeds are hosted by the same IP address, so we achieve a significant reduction in the measurement overhead by limiting probes to those nodes with unique IP addresses. In our sample of 102,446 RSS feeds, there are only 591 unique IP addresses.

Given the latency matrix between feeds and crawlers, we perform assignment using a simple first-fit bin-packing algorithm. The algorithm iterates through each crawler C_j and calculates $i^\star = \arg\min l(F_i, C_j)$, where $l(\cdot)$ is the latency between F_i and C_j. $F_{i^\star}$ is then assigned to C_j. Given F feeds and C crawlers, we assign F/C feeds to each crawler (assuming $F > C$). We have considered assigning varying number of feeds to crawlers, for example, based on the posting activity of each feed, but have not yet implemented this technique.

Figure 3.4 shows an example of the source feed mapping from one of our experiments. To reduce clutter in the map we show only 3 crawlers (one in the US, one in Switzerland, and one in Japan) and the 5 nearest crawlers, according to estimated latency, for each. The mapping process is clearly effective at achieving good locality and naturally minimizes traffic over transoceanic links.

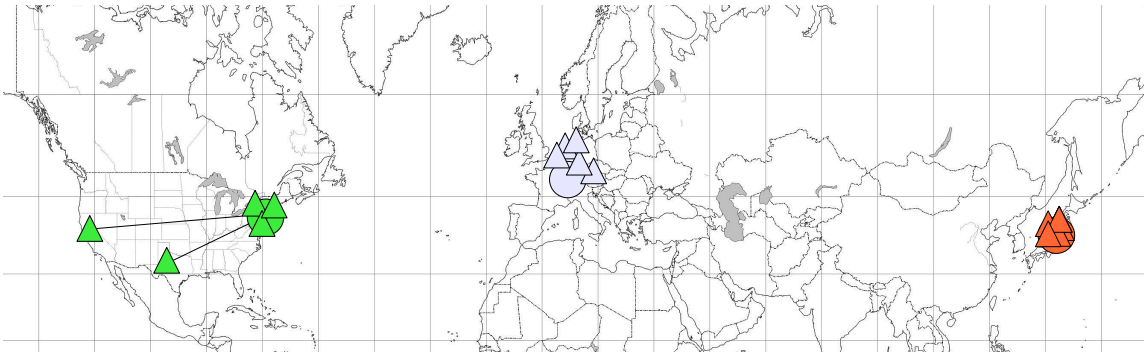


Figure 3.4: **An example of locality-aware source feed mapping.** Three crawlers are shown as circles and the 5 nearest source feeds, according to estimated latency, are shown as triangles. Colors indicate the mapping from feeds to crawlers, which is also evident from the geographic layout.

3.4 Implementation

Our prototype of Cobra is implemented in Java and uses the SBON (*stream-based overlay networks*) [119] substrate for setting up and managing data flows between services. Note, however, that the placement of Cobra services onto physical hosts is determined statically, at instantiation time, rather than dynamically as described in previous work [119]. A central *controller node* handles provisioning and instantiation of the Cobra network. The provisioner outputs a logical graph, which is then instantiated on physical hosts using a (currently random) allocation of services to hosts. The instantiation mechanism depends on the specific deployment environment.

Our implementation of Cobra consists of 29178 lines of Java code in total. The crawler service is 2445 lines, the filter service is 1258 lines, and the reflector is 622 lines. The controller code is 377 lines, while the remaining 24476 consists of our underlying

SBON substrate for managing the overlay network.

3.5 Summary

We have designed Cobra as a distributed, scalable system to aggregate and filter millions of RSS feeds into results feeds personalized to each subscriber. Cobra consists of a three-tiered network of crawlers that periodically crawl web feeds, filters that match crawled articles to user subscriptions, and reflectors that provide recently-matching articles on each subscription as an RSS feed. Each of the three tiers of a Cobra network is distributed over multiple hosts in the Internet, allowing computational and network load to be balanced across servers, and allowing for locality optimizations when placing services.

Chapter 4

Cobra System Evaluation

We have several goals in our evaluation of Cobra. First, we show that Cobra can scale well to handle a large number of source feeds and user subscriptions. Scalability is limited by service resource requirements (CPU and memory usage) as well as network bandwidth requirements. However, a modestly-sized Cobra network (Figure 3.2) can handle 8M users and 800,000 source feeds. Second, we show that Cobra offers low latencies for discovering matching articles and pushing those updates to users. The limiting factor for update latency is the rate at which source feeds can be crawled, as well as the user's own polling interval. We also present data comparing these update latencies with three existing blog search engines: Google Blog Search, Feedster, and Blogdigger.

We present results from experiments on three platforms: a local cluster, the Utah Emulab testbed [140], and PlanetLab. The local cluster allows us to measure service-level performance in a controlled setting, although scalability is limited. Our Emulab results allow us to scale out to larger configurations. The PlanetLab experiments are

intended to highlight the value of source feed clustering and the impact of improved locality.

We use a combination of real and synthesized web feeds to measure Cobra's performance. The real feeds consist of a list of 102,446 RSS feeds from syndic8.com, an RSS directory site. The properties of these feeds were studied in detail by Liu et al. [94]. To scale up to larger numbers, we implemented an artificial *feed generator*. Each generated feed consists of 10 articles with words chosen randomly from a distribution of English words based on popularity rank from the Brown corpus [17]. Generated feed content changes dynamically, with update frequencies similar to those observed in real feeds [94]. The feed generator is integrated into the crawler service and is enabled by a runtime flag.

Simulated user subscriptions are similarly generated with a keyword list consisting of the same distribution as that used to generate feeds. We exclude the top 29 most popular words, which are considered excessively general and would match essentially any article. (We assume that these words would normally be ignored by the subscription web portal when a user initially submits a subscription request.) The number of words in each query is chosen from a distribution based on a Yahoo study [43] of the number of words used in web searches; the median subscription length is 3 words with a maximum of 8. All simulated user subscriptions contain only conjunctions between words (no disjunctions). In Cobra, we expect that users will typically submit subscription requests with many keywords to ensure that the subscription is as specific as possible and does not return a large number of irrelevant articles. Given the large number of simulated users, we do not actively poll Cobra reflectors, but

	median	90th percentile
Size of feed (bytes)	7606	22890
Size of feed (articles)	10	17
Size of article (bytes)	768	2426
Size of article (words)	61	637

Figure 4.1: **Properties of Web feeds used in our study.**

rather estimate the additional network load that this process would generate.

4.1 Properties of Web feeds

Liu et al. [94] present a detailed evaluation of the properties of RSS feeds, using the same list of 102,446 RSS feeds as used in our study. Figure 4.1 summarizes the size of the feeds and individual articles observed in a typical crawl of this set of feeds between October 1–5, 2006. The median feed size is under 8 KB and the median number of articles per feed is 10.

Figure 4.2 shows a scatterplot of the size of each feed compared to its crawl time from a PlanetLab node running at Princeton. The figure shows a wide variation in the size and crawl time of each feed, with no clear relationship between the two. The large spike around size 8000 bytes represents a batch of 36,321 RSS feeds hosted by *topix.net*. It turns out these are not static feeds but dynamically-generated aggregation feeds across a wide range of topics, which explains the large variation in the crawl time.

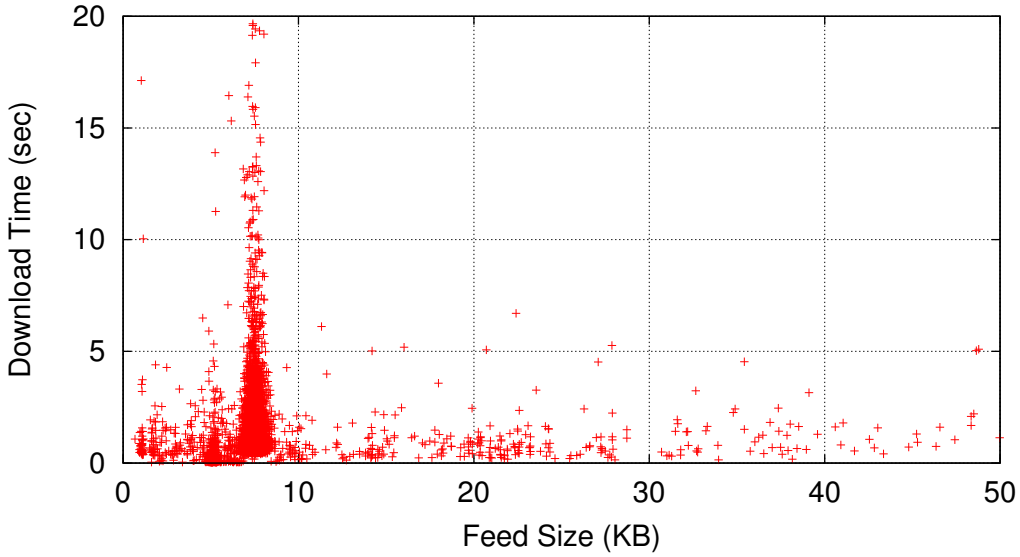


Figure 4.2: **Relationship between feed size and crawl time.**

4.2 Microbenchmarks

Our first set of experiments measure the performance of the individual Cobra services.

4.2.1 Memory usage

Figure 4.3 shows the memory usage of a single Reflector service running on Emulab, as articles are received over time. In each case, the usage follows a logarithmic trend. However, the curves' obvious offsets make it clear that the number of subscriptions stored on each reflector strongly influences its memory usage. For example, with a half-million subscriptions, the memory usage reaches ∼310 MB after receiving 60,000 articles, whereas with 1 million subscriptions the memory usage reaches nearly

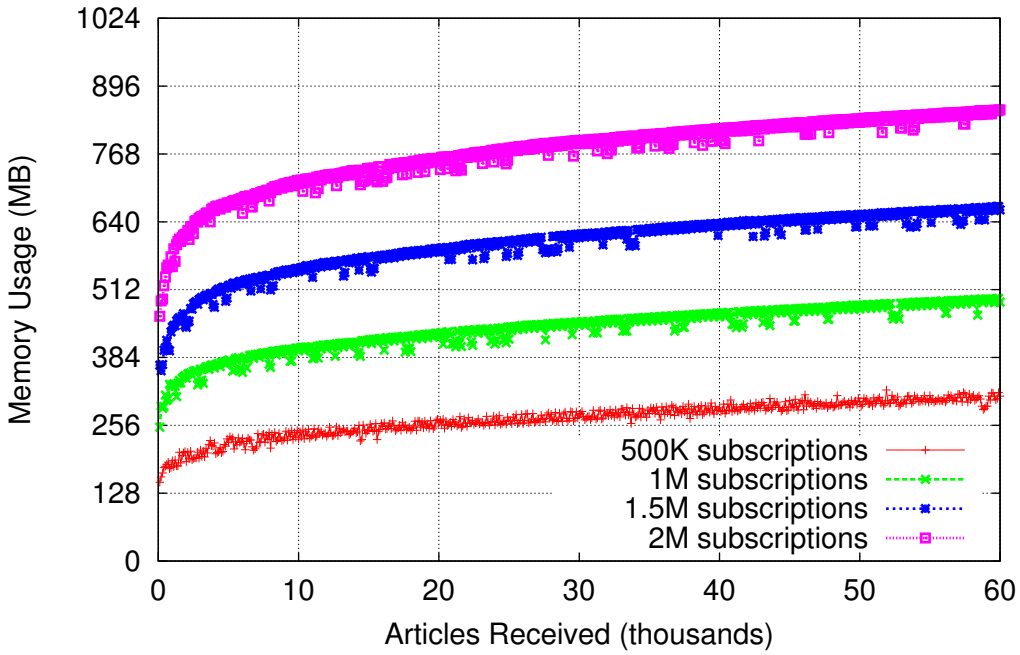


Figure 4.3: **Memory usage of the reflector service over time.** The x-axis of this figure is the total number of articles received by the reflector. For context, we estimate that a set of 1 million feeds can be expected to produce an average of ∼48.5 updated articles every second, or ∼2910 each minute.

500 MB. This is not surprising; not only must reflectors store each actual subscription (for matching), but also each user's list of articles.

However, after the initial burst of article storage, the rate of memory consumption slows dramatically due to the cap (of $k = 10$) on each user's list of stored articles. This cap prevents users with particularly general subscriptions (that frequently match articles) from continually using up memory. Note that in this experiment no articles (or article contents) were dropped by the reflectors' scarce memory handling logic (as described in Section 3.2.3). The only time that articles were dropped was when a

user's list of stored articles exceeded the size cap.

This experiment assumes that articles are never expired from memory (except when a user's feed grows beyond length k). It is easy to envision an alternative design in which a user's article list is cleared whenever it is polled (by the user's RSS reader) from a reflector. Depending on the frequency of user polling, this may decrease overall memory usage on reflectors but an analysis of the precise benefits is left to future work.

In contrast, the memory usage of the crawler and filter services does not change as articles are processed. For crawlers, the memory usage while running is essentially constant since crawlers are unaffected by the number of subscriptions. For filters, the memory usage was found to vary linearly with the number of subscriptions ($\sim$0.16 MB per 1000 subscriptions held) and thus changes only when subscriptions are added or removed.

4.2.2 Crawler performance

Figure 4.4 shows the bandwidth reduction resulting from optimizations in the crawler to avoid crawling feeds that have not been updated. As the figure shows, using last-modified checks for reading data from feeds reduces the inbound bandwidth by 57%. The combination of techniques for avoiding pushing updates to the filters results in a 99.8% reduction in the bandwidth generated by the crawlers, a total of 2.2 KB/sec for 102,446 feeds. We note that none of the feeds in our study supported the use of HTTP delta encoding, so while this technique is implemented in Cobra it does not yield any additional bandwidth savings.

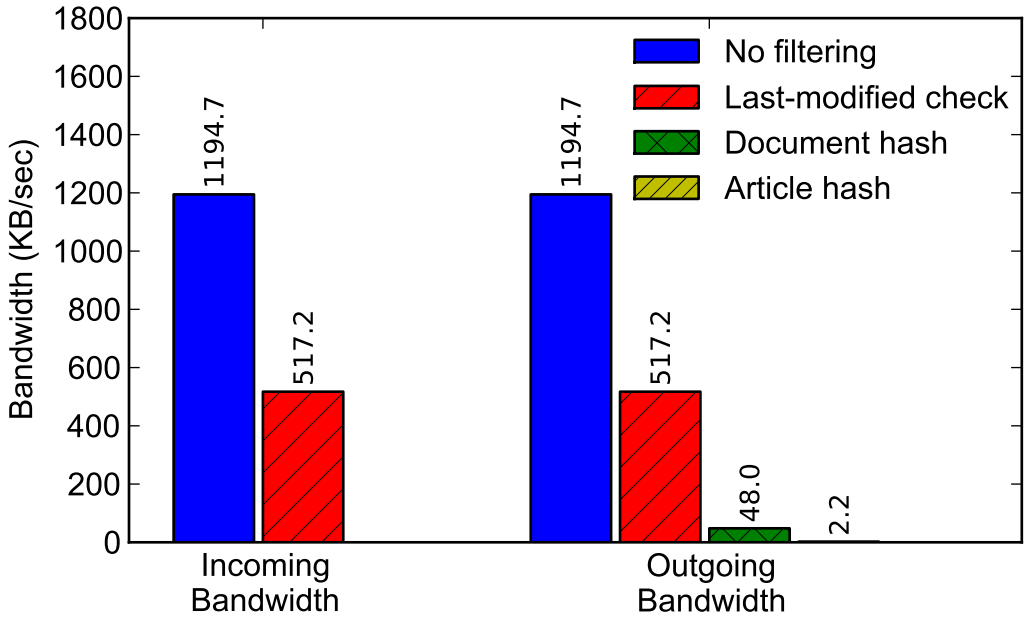


Figure 4.4: **Bandwidth reduction due to intelligent crawling.** This graph shows the amount of data generated by the crawler using different techniques: (a) crawl all feeds; (b) filter based on last-modified header; (c) filter based on whole-document hash; and (d) filter based on per-article hashes.

The use of locality-aware clustering should reduce the time to crawl a set of source feeds, as well as reduce overall network load. From our initial set of 102,446 feeds, we filtered out those that appeared to be down as well as feeds from two aggregator sites, *topix.net* and *izynews.de*, that together constituted 50,953 feeds. These two sites host a large number of dynamically-generated feeds that exhibit a wide variation in crawl times, making it difficult to differentiate network effects.

Figure 4.5 shows the time to crawl the remaining 34,092 RSS feeds distributed

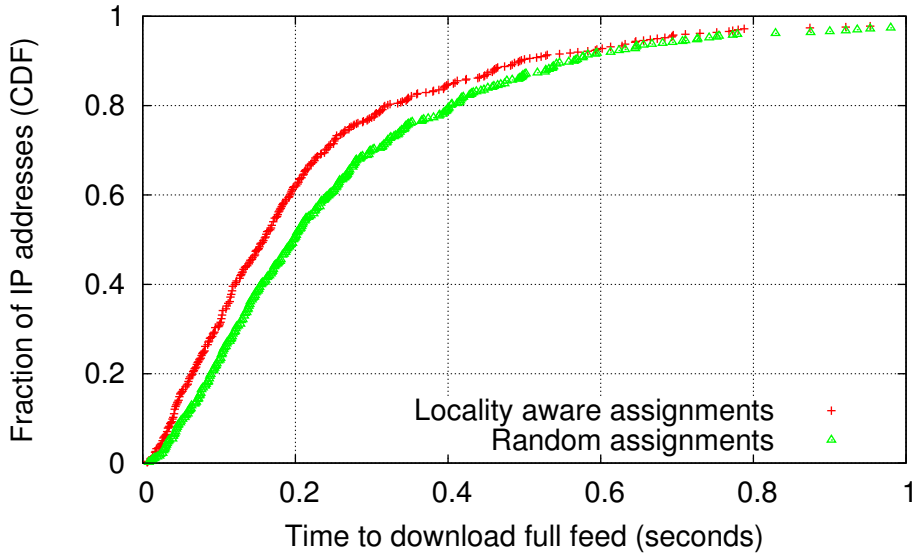


Figure 4.5: **Effect of locality-aware clustering.** This is a CDF of the time to crawl 34092 RSS feeds across 481 separate IP addresses from 11 PlanetLab hosts, with and without the locality aware clustering.

across 481 unique IP addresses. 11 crawlers were run on PlanetLab distributed across North America, Europe, and Asia. With locality aware mapping, the median crawl time per feed drops from 197 ms to 160 ms, a reduction of 18%.

4.2.3 Filter performance

Figure 4.6 shows the median time for the filter's matching algorithm to compare a single article against an increasing number of user subscriptions. The matching algorithm is fast, requiring less than 20 ms to match an article of 2000 words against 1 million user subscriptions. Recall that according to Figure 4.1, the median article

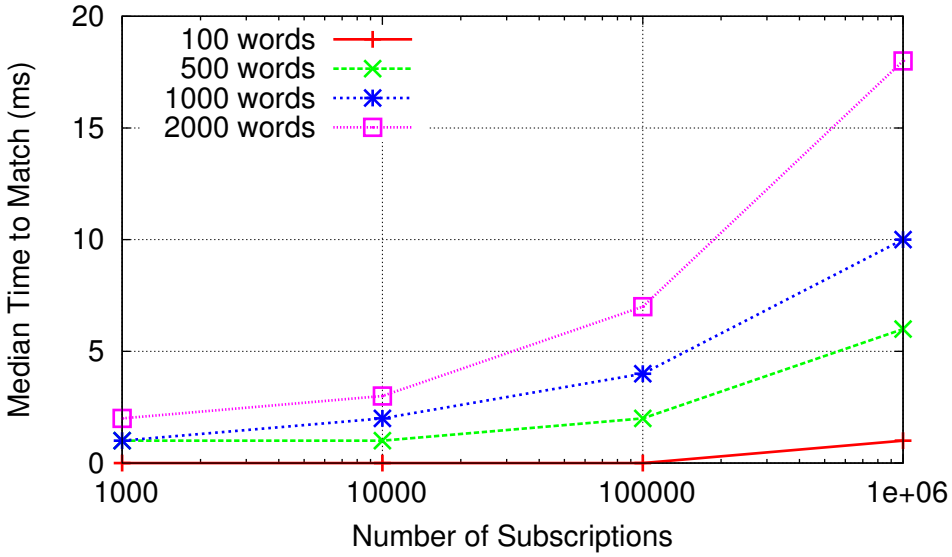


Figure 4.6: **Article match time versus number of subscriptions and number of words per article.** The median time to match an article is a function of the number of subscriptions and the number of words per article.

size is just 61 words, so in practice the matching time is much faster: we see a 90th percentile of just 2 ms per article against 1 million subscriptions. Of course, as the number of incoming articles increases, the overall matching time may become a performance bottleneck, although this process is readily distributed across multiple filters.

4.3 Scalability measurements

To demonstrate the scalability of Cobra with a large number of feeds and user subscriptions, we ran additional experiments using the Utah Emulab testbed. Here,

Subs	Feeds	Crawlers	Filters	Reflectors
10M	1M	1	28	28
20M	500,000	1	25	25
40M	250,000	1	28	28
1M	100,000	1	1	1

Figure 4.7: **Topologies used in scalability measurements.** The last topology (100K feeds, 1M subscriptions) is meant to emulate a topology using the live set of 102,446 feeds.

we are interested in two key metrics: (1) The *bandwidth consumption* of each tier of the Cobra network and (2) The *latency* for an updated article from a source feed to propagate through the three tiers of the network. In total, we evaluated four different topologies, summarized in Figure 4.7.

Each topology was generated by the provisioner with a bandwidth constraint of 100 Mbps[1], a memory constraint of 1024 MB, and a CPU constraint of the default value (1x). In addition, we explicitly over-provisioned by 10% as a guard against bursty traffic or unanticipated bottlenecks when scaling up, but it appears that this was largely unnecessary. We ran each topology for four crawling intervals of 15 minutes each, and checked the logs at the end of every experiment to confirm that none of the reflectors dropped any articles (or article contents) to save memory (a mechanism invoked when available memory runs low, as discussed in Section 3.2.3).

Figure 4.8 shows the total bandwidth consumption of each tier of the Cobra net-

[1]We feel that the 100 Mbps bandwidth figure is not unreasonable; bandwidth measurements from PlanetLab indicate that the median inter-node bandwidth across the Internet is at least this large [92].

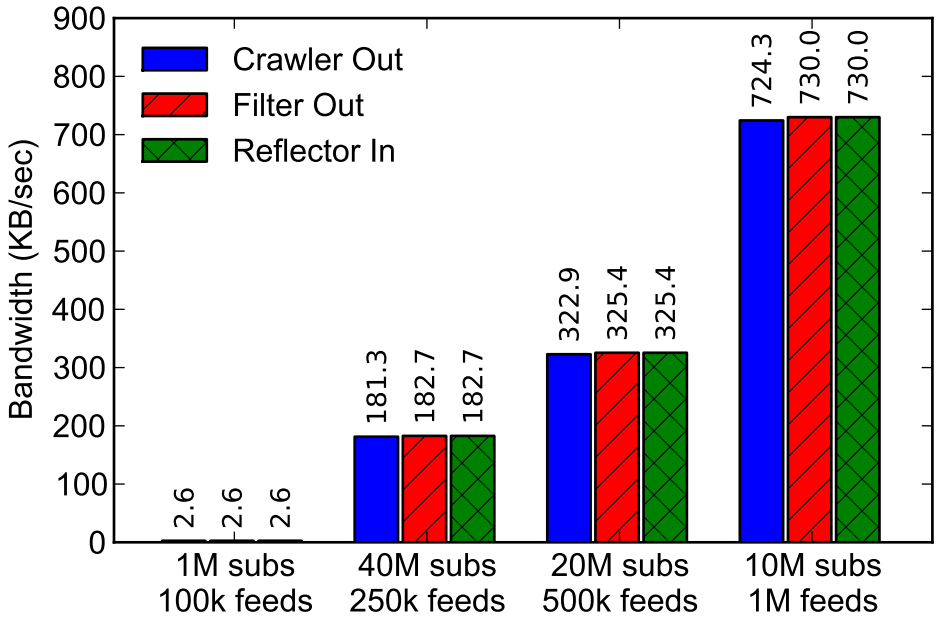


Figure 4.8: **Bandwidth consumption of each tier.** The bandwidth of each tier is a function both of the number of feeds that are crawled and of the fan-out from each crawler to the filters.

work for each of the four topologies evaluated. As the figure shows, total bandwidth consumption remains fairly low despite the large number of users and feeds, owing mainly to the effective use of intelligent crawling. Note that due to the relatively large number of subscriptions in each topology, the *selectivity* of the filter tier is nearly 1; every article will match some user subscription, so there is no noticeable reduction in bandwidth from the filter tier (the very slight increase in bandwidth is due to the addition of header fields to each article). One potential area for future work is finding ways to reduce the selectivity of the filter tier. If the filters' selectivity can be reduced,

that will reduce not only the filters' bandwidth consumption, but also the number of reflectors needed to process and store the (fewer) articles sent from the filters. One way to lower filter selectivity may be to assign subscriptions to filters based on similarity (rather than the current random assignment); if all of the subscriptions on a filter tend towards a single, related set of topics, then more articles may fail to match any those subscriptions.

We are also interested in the *intra-network latency* for an updated article passing through the three tiers of the Cobra network. To gather this data, we instrumented the crawler, filter, and reflector to send a packet to a central logging host every time a given article was (1) generated by the crawler, (2) received at a filter, (2) matched by the filter, and (3) delivered to the reflector. Although network latency between the logging host and the Cobra nodes can affect these results, we believe these latencies to be small compared to the Cobra overhead.

Figure 4.9 shows a CDF of the latency for each of the four topologies. As the figure shows, the fastest update times were observed on the 1M feeds / 10M subs topology, with a median latency of 5.06 sec, whereas the slowest update times were exhibited by the 250K feeds / 40M subs topology, with a median latency of 34.22 sec. However, the relationship is not simply that intra-network latency increases with the number of users; the median latency of the 100K feeds / 1M subs topology was 30.81 sec - nearly as slow as the 250K feeds / 40M subs topology. Instead, latency appears more closely related to the number of subscriptions stored *per node* (rather than in total), as shown in Figure 4.10.

As mentioned at the end of Section 3.2.2, nodes are able to throttle the rate at

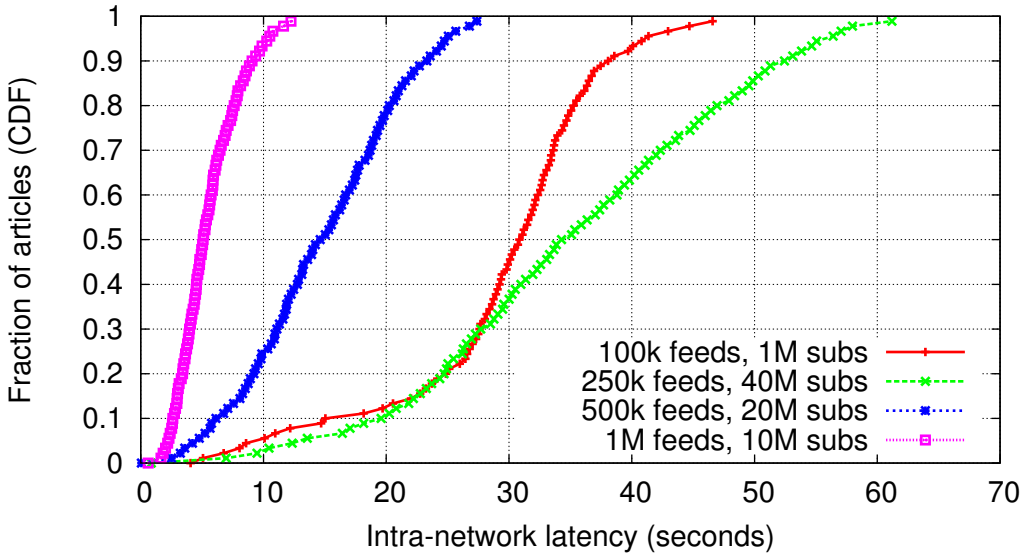


Figure 4.9: **CDF of intra-network latency for various topologies.** This experiment shows that the intra-network latency is largely a factor of the processing load on filter and reflectors.

which they are passed data from other nodes. This is the primary source of intra-network latency; article updates detected by crawlers are delayed in reaching reflectors because of processing congestion on filters and/or reflectors. Since the time for a filter (or reflector) to process an article is related to the number of subscriptions that must be checked (see figure 4.6), topologies with larger numbers of subscriptions *per node* exhibit longer processing times, leading to rate-throttling of upstream services and thus larger intra-network latencies. Figure 4.10 shows a clear relationship between the number of subscriptions per node and the intra-network latencies. However, even in the worst of these cases, the latencies are still fairly low overall. As the system is scaled to handle more subscriptions and more users, Cobra will naturally load-balance

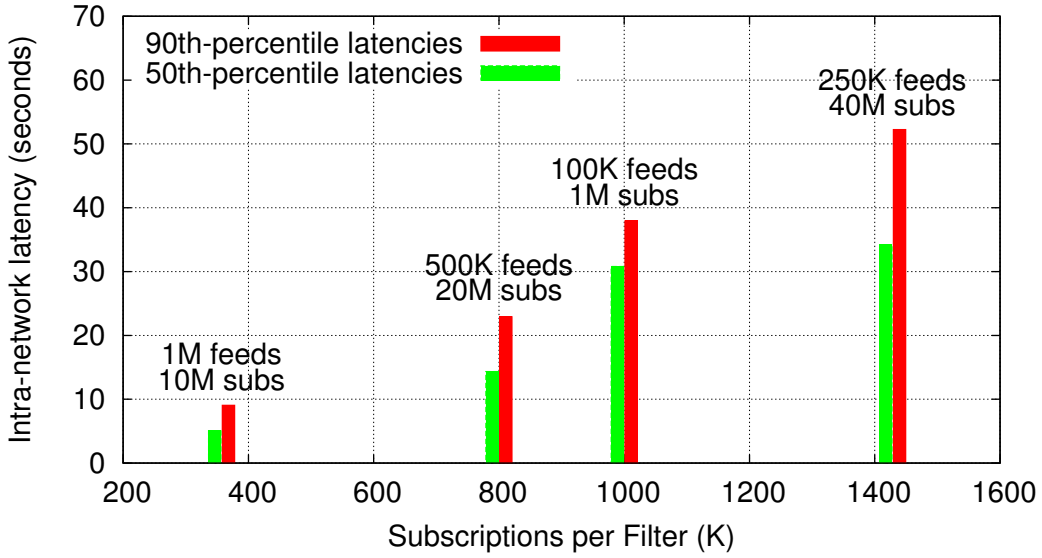


Figure 4.10: **Intra-network latency as a function of subscriptions per Filter.** This figure shows the relationship between intra-network latency and the number of subscriptions stored on each Filter (note that in each of these topologies, the number of filters equals the number of reflectors, and thus the x-axis is equivalent to "Subscriptions per Reflector (K)").

across multiple hosts in each tier, keeping latencies low.

Note that the user's *perceived update latency* is bounded by the sum of the *intra-network latency* once an article is crawled by Cobra, and the *crawling interval*, that is, the rate at which source feeds are crawled. In our current system, we set the crawling interval to 15 minutes, which dominates the intra-network latencies shown in Figure 4.9. The intra-network latency is in effect the minimum latency that Cobra can support, if updates to feeds could be detected instantaneously.

4.4 Comparison to other search engines

Given the number of other blog search engines on the Internet, we were curious to determine how well Cobra's update latency compared to these sites. We created blogs on two popular blogging sites, LiveJournal and Blogger.com, and posted articles containing a sentence of several randomly-chosen words to each of these blogs.[2] We then searched for our blog postings on three sites: Feedster, Blogdigger, and Google Blog Search, polling each site at 5 sec intervals.

We created our blogs at least 24 hours prior to posting, to give the search engines enough time to index them. Neither Feedster nor Blogdigger detected *any* of our postings to these blogs, even after a period of over four months (from the initial paper submission to the final camera-ready). We surmise that our blog was not indexed by these engines or that our artificial postings were screened out by spam filters used by these sites.

Google Blog Search performed incredibly well, with a detection latency as low as 83 seconds. In two out of five cases, however, the latency was 87 minutes and 6.6 hours, respectively, suggesting that the performance may not be predictable. The low update latencies are likely the result of Google using a *ping service*, which receives updates from the blog site whenever a blog is updated [9]. The variability in update times could be due to crawler throttling: Google's blog indexing engine attempts to throttle its crawl rate to avoid overloading [62]. As part of future work, Cobra could be extended to provide support for a ping service and to tune the crawl rate on a

[2]An example posting was "certified venezuela gribble spork." Unsurprisingly, no extant blog entries matched a query for these terms.

per-site basis.

We also uncovered what appears to be a bug in Google's blog indexer: setting our unique search term as the title of the blog posting with no article body would cause Google's site to return a bogus results page (with no link to the matching blog), although it appears to have indexed the search term. Our latency figures ignore this bug, giving Google the benefit of the doubt although the correct result was not returned.

In contrast, Cobra's average update latency is a function of the crawler period, which we set at 15 minutes. With a larger number of crawler daemons operating in parallel, we believe that we could bring this interval down to match Google's performance. To our knowledge, there are no published details on how Google's blog search is implemented, such as whether it simply leverages Google's static web page indexer.

4.5 Comparison to Prior Work

Its useful at this point to identify some of the ways that Cobra differs from pub-sub systems (§2.4.1). These are important to recognize because the Cobra query interface – a simple boolean expression over content keywords – can be supported by each of the content-based pub-sub systems previously discussed. Thus, at first glance it may appear that Cobra's functionality could be supported by most or all of these systems. However, Cobra differentiates itself from these systems in two other ways.

First, distributed content-based pub-sub systems such as Siena leave it up to the network administrator to choose an appropriate overlay topology of filtering nodes.

As a result, the selected topology and the number of servers may or may not perform well with a given workload and distribution of publishers and subscribers in the network. Cobra provides a separate provisioning component that outputs a custom-tailored topology of processing services, ensuring that Cobra can support a targeted work load. Our approach to system provisioning is independent from our application domain of RSS filtering and likely could be used to provision a general-purpose pub-sub system like Siena, as long as appropriate processing and I/O models are added to the service provisioner.

Second, Cobra integrates directly with existing protocols for delivering real-time streams on the Web — namely, HTTP and RSS. Most other pub-sub systems such as Siena do not inter-operate well with the current Web infrastructure, for example, requiring publishers to change the way they generate and serve content, and requiring subscribers to register interest using private subscription formats.

4.6 Summary

Our evaluation of Cobra shows that, with modest hardware resources, our design scales to large numbers of monitored feeds and user subscriptions. For example, 51 Emulab nodes are able to support 20 million subscriptions while crawling 500,000 feeds every 15 minutes. Through intelligent crawling and content hashing, crawlers are able to reduce incoming bandwidth by 57% and outgoing bandwidth by 99.8%. Our locality-aware placement reduces the average crawl time by 18%. Finally, the intra-network latency is on the order of 10s of seconds (dictated primarily by the number of subscriptions per filter), and thus dominated by typical crawling intervals.

Chapter 5

Designing and Measuring an Outdoor Wireless Testbed

5.1 Motivation

Another application area of interest is gathering data from a city environment. A sampling of existing projects demonstrates the variety and richness of data sources found in urban settings:

- *London Congestion Pricing*: using cameras to recognize license plates, additional fees are charged to those driving in central London during peak hours [15]
- *Skyhook Wireless*: by querying a database containing the locations of over 100 million wireless networks, mobile devices can estimate their geographic location using whatever wireless networks are observed nearby [21]
- *Fade To Black*: a network of upturned web-cams record pollutant depositions

(causing the camera images to quite literally fade to black over time) [7]

- *Sniper Localization*: networks of acoustic sensors detect and localize in 3-dimensions the source of gunshots in complex urban environments [131]

The overall goal of the CitySense [106] project is to support the development and evaluation of novel sensor networks on an urban scale. Substantial cost and effort are required to design, deploy and evaluate wireless sensor systems at scale; many research groups must be content with simulations or small-scale, homegrown test deployments. By serving as a *multi-purpose, programmable* platform, CitySense "lowers the bar" for the deployment and evaluation of new systems for urban sensing. Unlike the mesh networks mentioned previously, CitySense explicitly is *not* intended to serve as a platform for community wireless Internet access; such a goal would greatly limit the kinds of research that users could perform on the network. In other words, our goal is to support disruptive research that would interfere with the network's ability to serve as a reliable provider of Internet connectivity.

To our knowledge, CitySense is the first general-purpose urban-scale sensor network. Currently, selected CitySense nodes feature a mix of multi-variate weather (measuring precipitation, wind speed/direction, etc.) and CO_2 sensors. Additionally, nodes' 802.11 radios can serve as a surprisingly useful "sensor" of sorts. The Argos system (ch. 6-7) uses the radio in a sniffer role to collect and analyze ambient WiFi traffic. However, one could also imagine sniffing wireless traffic as a way to gain insight into other phenomena. For example, tracking mobile devices could give an idea of people's movement patterns, or tracking vehicle-borne networks (WiFi is increasingly available in taxis and buses) could reveal traffic conditions. Finally, one

can imagine many other interesting data streams within the urban environment, such as noise levels and air pollution.

In order to achieve the geographic coverage required by many sensing applications, CitySense nodes are widely deployed. This enables rich data collection, but also means that many nodes are installed in hard-to-reach locations, such as the tops of streetlights. Thus, many nodes were deployed with (1) wireless-only network connectivity (due to a lack of wired network access), and (2) greatly limited physical access. These challenges informed our approaches to designing and operating the CitySense network, as reliability, transparency (e.g., for remote debugging), and network efficiency were of paramount importance.

The impetus for CitySense arose from recent work on city-wide wireless mesh networking, including the RoofNet [30], CUWin [4], and TFA [50] projects to provide connectivity to communities using inexpensive wireless equipment. While those projects were focused on the networking layer for providing general-purpose Internet connectivity, the key realization is that mesh router nodes (often based on embedded Linux PCs) could be augmented with a range of sensors as well as opened up to remote programming by end users.

In the following sections we present the overall architecture of CitySense, followed by measurement results characterizing the performance and stability of the network.

5.2 Architecture

CitySense consists of the following primary components:

1. a collection of 802.11-equipped single-board computers ("nodes"), deployed

Figure 5.1: **Map of the CitySense network in Cambridge, MA. The area shown is just under 7x5 km.**

throughout an urban environment

2. a wireless mesh backhaul network used for communication between nodes

3. backend support servers (web server, database, etc.) for managing nodes' installed software, stored collected data and the like

CitySense currently consists of 27 wireless nodes deployed on streetlights and rooftops around Cambridge and Somerville, MA in four (disconnected) clusters together spanning over 9.7 km^2 (see Figure 5.1). Each node consists of either a Soekris net4826 (233 MHz CPU, 128 MB of RAM) or ALIX 2c2 (500 MHz CPU, 256 MB of RAM) single-board computer, powered either directly from the streetlight mounting

Figure 5.2: **CitySense node deployed on a street lamp.** The thick antenna pointing downwards on the right side is connected to the 900MHz backhaul radio, and the thinner antenna on the left side is connected to the 802.11 sniffer radio. The node is powered from the streetlight.

or from rooftop electric sockets. Both configurations use a Wistron CM9 802.11a/b/g mini-PCI radio with an 8 dBi omnidirectional antenna and a Ubiquiti XR9 900 MHz mini-PCI radio with a 6 dBi omnidirectional antenna. The XR9 is a high-powered radio designed for long-range operation, with a peak transmission power of 28 dBm. It uses the standard 802.11 MAC and PHY, albeit in a nonstandard frequency band, with PHY rates up to 54 Mbps. Figure 5.2 shows one of our nodes deployed on a

streetlight.

A small number of the deployed nodes have wired Ethernet access, but for the remainder the only available network access is via the nodes' equipped radios. To enable connectivity, each node runs the OLSR-NG [18] mesh routing protocol on the high-power XR9 radio. This enables IP-level routing throughout the network, allowing nodes to communicate with each-other, as well as with the backend servers (by routing via a nearby Ethernet-equipped node). The lower-power CM9 radio does not participate in the wireless mesh and instead is provided for use by experimenters.

Each node runs a custom FreeBSD image as its operating system. A series of backend applications were developed for managing the creation of new OS images and their distribution, over the wireless mesh, to deployed nodes (since it would be infeasible to retrieve each node for every OS update). Although much more can be said about this sophisticated system and the steps it takes to avoid errors (retrieving a "bricked" node can be quite time consuming), its development was primarily the responsibility of other CitySense team members and thus a more detailed description is out of place in this dissertation.

5.3 Urban Deployment Considerations

The outdoor urban environment presents several challenges to the design of a wireless network of sensors. These include: physical environment factors, network coverage, and network security.

5.3.1 Physical environment

Extreme weather, theft and vandalism, and malfunctioning hardware are realities that we expected to face when deploying in an outdoor urban environment. Each node is housed in a NEMA 6x-rated weatherproof enclosure, which provides both protection from the elements as well as a small degree of insulation during winter. We have not encountered any cases of theft or vandalism, although on occasion nodes that draw power from indoors (e.g., via power cables snaked down ventilation shafts to the top floor) have been unplugged by unwitting occupants. Hardware malfunctions have been observed in almost every possible component, including fried Soekris/ALIX boards, wedged network hubs, and a variety of bad cables and loose connections.

The physical environment also directly affects nodes' wireless communications. Urban settings feature dense collections of buildings, oftentimes of various heights and sizes, which greatly complicate signal propagation patterns. Additional environmental complications include various signal absorbers (e.g., humans, trees), signal reflectors (e.g., metal roofs, fences) and sources of wireless interference (e.g., WiFi devices, cordless phones). A variety of studies have examined these effects on 802.11 radios [30, 50, 45], showing that urban environments severely limit the effective communication range.

It is illustrative to consider how existing urban and rural mesh networks differ; Table 5.1 provides a sampling of each. The urban networks are all much more densely deployed, with links roughly 100 times shorter than those supported in the rural networks. There are two main reasons why urban networks require this density:

1. **Obstructions** - Urban nodes frequently do not have line-of-sight connections

Type	Network	Nodes	Link Lengths
urban	CitySense	27	50-250 meters
urban	Roofnet [42]	38	50-400 meters
urban	Google WiFi [77]	482	100-200 meters
rural	WiLDNet (Aravind) [136]	10	5-15 km
rural	WiLDNet (AirJaldi) [136]	10	10-41 km
rural	Digital Gangetic Plains [41]	13	5-39 km

Table 5.1: **Comparison of existing urban and rural wireless mesh networks. "Link Lengths" describes the majority of the wireless links in that network.**

to neighboring nodes, which reduces link quality. Rural settings are relatively free of obstructions, enabling long distance line-of-sight links.

2. **Omnidirectional Antennas** - Relative to directional antennas, omnis sacrifice range for the ability to communicate in any direction[1]. This is important in urban settings where it's often difficult to know a priori (i.e., when first deploying nodes) which links will be strongest. The reason for this is that urban environments often affect wireless signals *unpredictably*. For example, in an open area (e.g., most rural settings), shorter links are nearly always stronger than longer ones. In complex, urban settings, however, this is often not the case; rarely is distance strongly correlated with link quality [30, 50, 88]. Using omnidirectional antennas may reduce nodes' communication ranges, but allows them a diversity of neighbors so that the strongest links can be used.[2]

[1] Any direction in the xy-plane, that is; communicate up or down is very limited

[2] An added benefit of the neighbor diversity that omnis provide is greater robustness to link failures.

5.3.2 Network Coverage

Streetlight mounting benefits network connectivity between CitySense nodes in two ways. First, it leverages the natural line-of-sight paths provided on the straight parts of roadways. Second, the uniform mounting height, roughly 10 meters above the street level, helps RF reception range by reducing the strength of the ground reflected signal as well as the probability of interference from non-CitySense ground-based RF emitters such as WiFi radios in private laptops and access points. Rooftop mounts, of course, do not realize these benefits (although rooftop-mounted nodes *are* far from street-level reflections, they instead need to contend with reflections off the roof itself).

To ensure that mesh connectivity is maintained in the face of unplanned node outages, we deployed nodes such that the inter-node spacing would be roughly less than half the range for the radios. This ensures RF overlapping such that two nodes will not lose connectivity should a single intermediate node fail. Actual placement locations, of course, are heavily influenced by the availability of deployment sites. This is particularly true for clusters of rooftop-mounted nodes, as some entire buildings are off-limits to us due to technical (e.g., no power sources available) or social (e.g. we could not secure appropriate approvals) reasons. On average, each CitySense node reports 4.5 neighbors[3], which leads to a good amount of redundancy in the face of node or link failures.

[3]Here we define a node's *neighbors* simply as all nodes that are 1-hop away in IP routing, as determined by the OLSR mesh routing protocol. In most cases, a node is also able to communicate (to some degree) with some other nodes, but these links are weak enough that OLSR chooses to route to those nodes via better-connected intermediaries.

5.3.3 Network Security

Security and interference are critical concerns in urban environments where many private 802.11 networks may exist. CitySense uses a two-layer approach to security. At the link layer, we employ WEP encryption to deter snooping by passive listeners. Although WEP keys can be cracked fairly easily by knowledgeable attackers [138], WEP encryption still helps deter the casual snooper. Additionally, the mesh radio operates in a nonstandard 900 MHz frequency band, which both eliminates all interference from ambient WiFi devices[4], as well as further frustrating the casual snooper. As a second, more robust layer of defense, applications typically use a secure transport layer protocol, e.g., SSL or SSH, for inter-node communication.

5.4 Network Measurements

This section evaluates the CitySense testbed's network performance using a dataset of point-to-point TCP and Ping measurements. The TCP measurements were obtained using the `Iperf` bandwidth measurement tool [11] (v2.0.4), while Ping measurements were obtained using the `ping` utility of FreeBSD 7.1-STABLE. Measurements were performed nightly between all unique ordered pairs of nodes (one at a time, to avoid interference between simultaneous tests). For TCP, we recorded the mean throughput achieved over a 60-second `Iperf` measurement; if no connection could be established, a value of 0 was recorded (this occurred in $< 5\%$ of cases). For

[4]Other devices, such as cordless phones or 802.15.4 radios, may operate in the 900 MHz band and thus cause interferences, but these are unlikely to be numerous or powerful enough to cause an appreciable effect on the network.

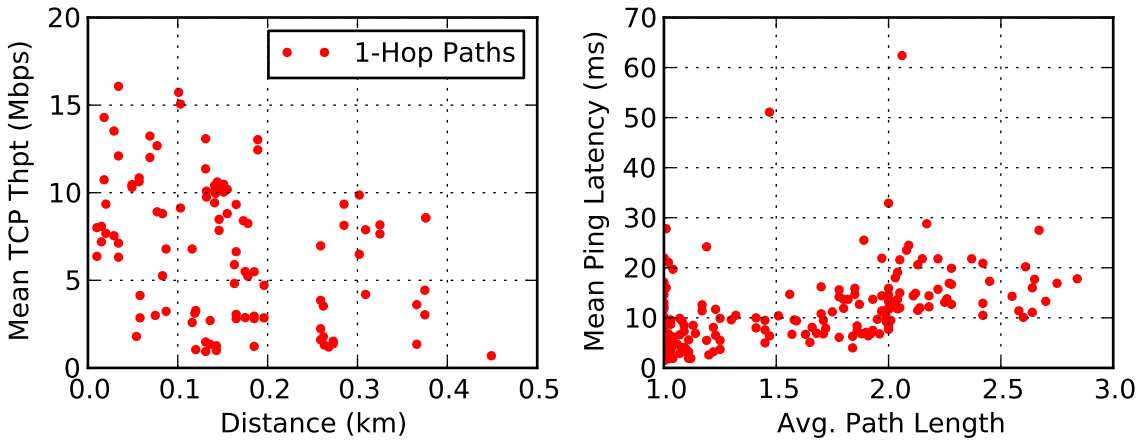
Ping, we recorded the mean round-trip time over 10 packets; overall, 25% of packets we lost (these did not contribute to the mean RTTs recorded). For each path's TCP and Ping measurements, we removed the extreme values (largest and smallest 5%) to ensure that spurious measurements were not included (such as those obtained during short-lived routing misconfigurations or transient hardware failures).

Network measurements have run continually since June, 2008. Since that time, the network has undergone many changes, including deployments of many more nodes, occasional node retrievals (e.g., for repairs), numerous software updates, and various changes to networking and routing parameters. Any of these can affect measured networking performance and thus care must be taken when comparing measurement data over long timescales. For consistency of analysis, the data presented here was collected over 5 months in 2009 during which time no major changes to the network occurred.

Figure 5.3 shows that physical distance is generally not a good predictor of either throughput or latency; this is likely due to the urban environment's affect on wireless signals (as discussed above) which can have a greater impact on signals than simple free-space path loss. This result is in line with previous studies of urban wireless communications, including the TFA-Wireless mesh [50], the Roofnet mesh [30], and the Dartmouth College infrastructure network [88].

5.4.1 TCP Throughput

Figure 5.4(a) shows the wide range of TCP performance that we observe. Most multi-hop paths are concentrated at the low end ($\leq$ 3 Mbps) whereas single-hop



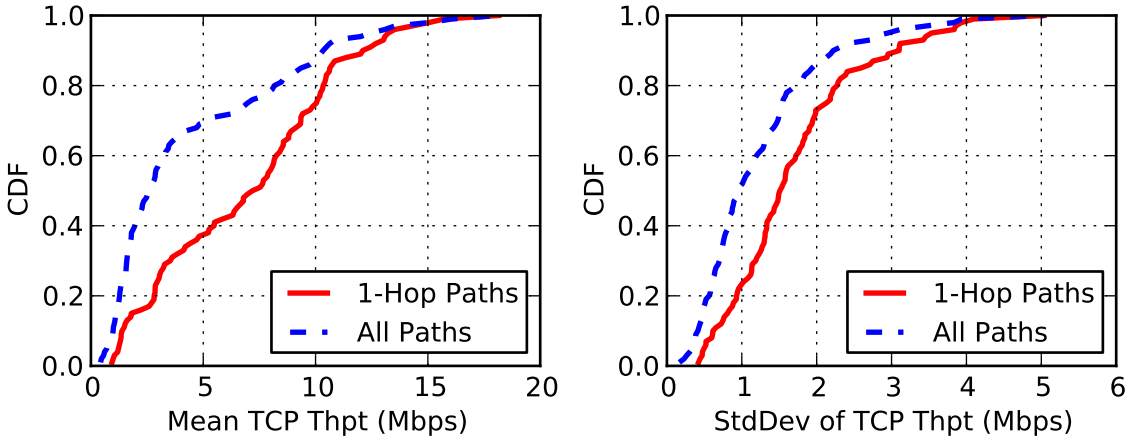
(a) (b)

Figure 5.3: **Scatter plot showing the relationship of distance versus mean TCP throughput (left) and Ping latency (right) across all single-hop network links.**

paths are distributed more evenly throughout the full range. This trend of most paths concentrating at the lower bandwidths was also observed by the Bicket et al. in the Roofnet network [42].

The *stability* of each link's TCP performance is also of interest. As shown in Figure 5.4(b), the standard deviations of CitySense paths are mostly in the range of 0.5 Mbps to 3 Mbps, with a small number of particularly unstable paths up to 5 Mbps. 1-hop paths are roughly 0.5 Mbps more stable than all paths considered together.

Finally, it is also useful to consider the coefficient of variance (the ratio of standard deviation to mean) for each path. A standard deviation of 1 Mbps, for example, is



(a) (b)

Figure 5.4: **CDF plots showing the distributions of the TCP means (left) and standard deviations (right) over all paths in the CitySense network.**

much more significant on a path that averages only 1.5 Mbps versus on a path that averages 20 Mbps. The coefficient of variance provides a way of comparing TCP stability across both high- and low-bandwidth paths.

Figure 5.5 shows a CDF of the TCP coefficient of variance across all paths. Most paths show significant variability; 66% of paths have a coefficient between 0.28 and 0.53 and only 3% of paths have a coefficient less than 0.1. Overall, we conclude that the typical TCP performance of CitySense is comparable or slightly better than that reported for other deployed wireless mesh networks, including Roofnet and TFA-Wireless. However, very few paths show strongly consistent TCP performance, suggesting that long-lived applications should be prepared for significant throughput variability over time. Section 5.4.3 explores some implications of this.

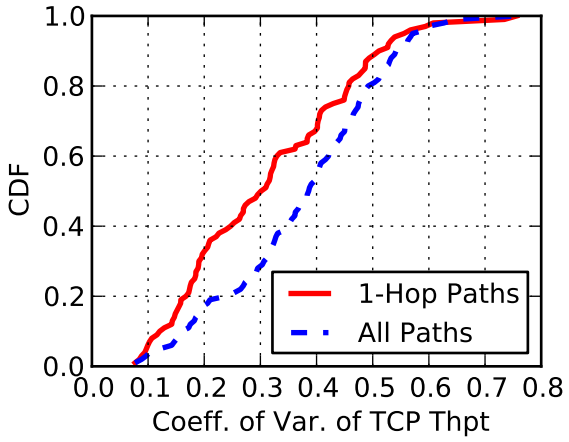


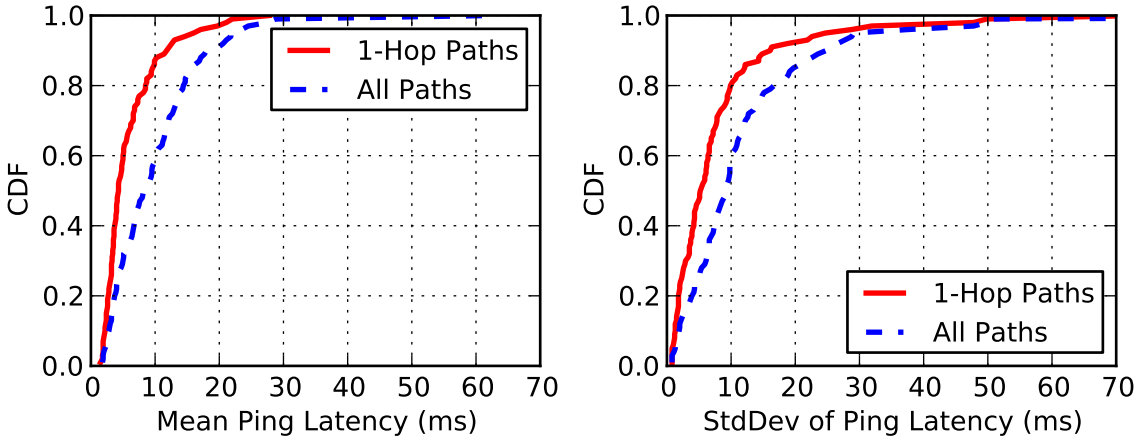
Figure 5.5: **CDF plot showing the distribution of the TCP coefficient of variance over all paths in the CitySense network.**

5.4.2 Ping Latencies

Figure 5.6 shows that the latencies of most paths ($> 80\%$) have means and standard deviations both under 20 ms. These means are comparable to those reported for Roofnet[5]. A handful of paths show poor performance and stability (e.g. one path averages 63.7 ms latency with a standard deviation of 128.3 ms).

Figure 5.5 shows a CDF of the Ping coefficient of variance across all paths. Overall, paths show greater Ping variability than TCP variability; 80% of paths have a coefficient between 0.5 and 1.5.

[5]Latencies are not reported for TFA-Wireless.



(a) (b)

Figure 5.6: **CDF plots showing the distributions of the Ping means (left) and standard deviations (right) over all paths in the CitySense network. For legibility, the x-axis of (b) is truncated; both CDFs on that plot extend to a maximum of 174.8 ms.**

5.4.3 Implications for Applications

The implications of the CitySense network's performance is, by necessity, application-dependant, but we can nonetheless draw some general conclusions. Generally, applications must contend with two factors: (i) bandwidth and latency vary significantly *across paths* (spatial variance; figures 5.4(a) & 5.6(a)), and (ii) in many cases, the bandwidth and latency of a single path vary significantly *over time* (temporal variance; figures 5.5 & 5.7).

Spatial Variance

The "average path" in CitySense supports a bandwidth of 4.4 Mbps with a la-

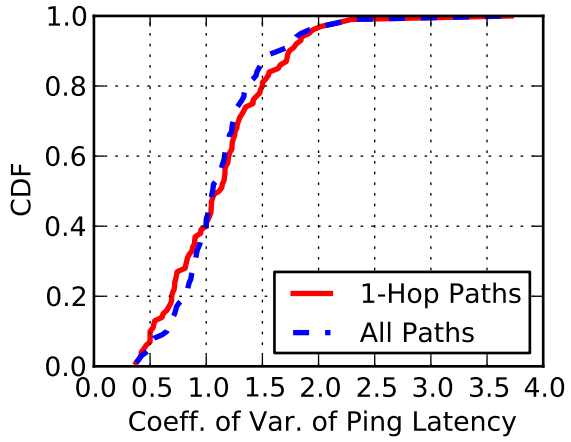


Figure 5.7: **CDF plot showing the distribution of the Ping coefficient of variance over all paths in the CitySense network.**

tency of only 10 ms, but applications clearly cannot expect this level of performance between all nodes pairs since paths vary significantly. At the same time, provisioning for the least-capable path would lead to inefficient use of the network's resources. Instead, applications should expect to adapt to a range of network conditions. For example, a sensor network application could tune its data aggregation or sampling rate independently for each node, according to the node's observed network capacity. In this way, well-connected nodes will return richer (i.e. more frequent and/or larger) data samples whereas poorly connected nodes will return less frequent and/or smaller data samples to avoid saturating their network connection.

Temporal Variance

Applications, particularly long-running ones, also must handle changes in network performance due to temporal variations. As an example, consider an application fea-

turing real-time, cross-network (i.e. server to node) user interactions. In such applications, there is a limit to the amount of latency that a user will tolerate; e.g., 100 milliseconds is an oft-cited latency threshold for whether or not users can perceive a delay after issuing a command [101]. Although all paths averaged < 100 ms on our ping test (Figure 5.6(a)), it was not rare for individual ping *packets* exceed this latency. Recall that each ping measurement consisted of 10 ping packets; in 3% of all ping measurements, the mean RTT (across all 10 packets) was ≥ 100 ms and in 8% of all ping measurements, the *maximum* RTT was ≥ 100 ms. Even more dramatically, 0.4% and 1.3% of measurements had a mean RTT and max RTT ≥ 1 *second*, respectively. Its worth noting that these slow measurements were *not* concentrated in just a few paths, as one might expect. On the contrary, over half of paths (58%) recorded at least one measurement with a mean RTT ≥ 100 ms, and no single path accounted for more than 3% of the 100+ ms measurements.

To summarize, although all paths have fairly low latencies (< 100 ms) *on average*, most paths occasionally observe individual packet RTTs high enough to be of concern in certain applications. Interactive applications therefore must be able to handle occasional latency spikes noticeable or even disruptive to users. Our experiences using SSH [146] support this conclusion. SSH is our primary method of accessing CitySense nodes[6] and is frequently used to query/configure node state and start/stop/manage background processes. We have found that SSH works well for these kinds of short-duration sessions, but can be unreliable for long-duration sessions due to timeouts.

[6]When necessary (e.g. if network connectivity is lost), nodes can be accessed via serial connection, but this requires physical access and thus is a last resort.

5.5 Summary

CitySense represents a unique combination of urban-sensing platform and outdoor wireless networking testbed. As a sensor network, CitySense offers potential access to a variety of urban data sources, including microclimate conditions, air quality, noise levels, and ambient wireless network traffic. As a networking testbed, CitySense offers researchers the ability to deploy and test new protocols and distributed systems in a setting with authentic urban effects (reflections, multipath fading, etc.) that are notoriously difficult to model accurately in simulation or synthetic testbeds.

Chapter 6

Argos Architecture

6.1 Introduction

Wireless networks are becoming the *de facto* access network for many Internet users, stemming from the tremendous growth in the use of laptops, smartphones, and other mobile devices. Even many desktop systems come equipped with 802.11 interfaces and can be used without a wired Ethernet connection, and devices such as Apple's Time Capsule permit automated backup over a wireless connection. As a result, the performance and behavior of wireless networks are critical for supporting the Internet. At the same time, growth in wireless network deployments, especially in densely populated urban settings, has been tremendous. A city block might have hundreds of separate access points and thousands of wireless clients. Of course, all of these devices must share limited ISM spectrum, leading to potentially high congestion.

Yet, we have little understanding of what wireless network traffic looks like "in the wild" at urban scales, with many access points and clients interacting. There are

many open questions that we would like to answer: What are the characteristics of wireless network traffic? How does traffic vary over space and time? Is there evidence of malware or other malicious traffic traversing the airwaves? Can we characterize user mobility patterns, and what can we learn about individual clients?

This chapter describes *Argos*, a city-wide wireless sensor network designed to study wireless network traffic and dynamics over urban scales. Argos collects data from multiple WiFi sniffers mounted on streetlights and rooftops around a city and performs decentralized trace merging and filtering to support multiple user queries processing the captured traffic. Argos sensor nodes are themselves connected with a backhaul wireless mesh network (in an orthogonal frequency band) to enable scalability and large spatial coverage. Using 26 nodes of the CitySense testbed (see ch. 5) deployed outdoors around Cambridge, MA, we present the first city-wide study of wireless network traffic dynamics, comprising 2630 access points and 65,073 unique clients spanning an area of 9.7 km^2.

Within the overall context of this dissertation, the design of Argos builds upon our experience developing Cobra in multiple ways. Both systems implement query interfaces tailored to their particular application needs, although, for Cobra, this meant a relatively simple, keyword-based interface. In contrast, Argos queries require a much richer interface, including mechanisms for requesting various different kinds of packet streams and for changing the sniffer radio channel according to different policies. In both Cobra and Argos, efficient data collection is vital for achieving good scalability, although for different reasons; Cobra seeks to minimize the physical resources required to support a given workload (number of data sources and consumers) whereas Argos

seeks to maximize the number of data sources that it can capture from, subject to the sniffer network's bandwidth capacity.

We recognize that passive monitoring of wireless networks raises numerous privacy concerns. Indeed, Google has recently found itself under fire for inadvertent packet captures from their Street View cars [129]. Argos anonymizes captured packet traces to protect against obvious user privacy violations, but we recognize that this does not eliminate all risks, and that reconciling the conflicting goals of data collection and user privacy will be an ongoing effort.

6.2 Background and Motivation

The vast growth of wireless LANs has led to new challenges for characterizing traffic and user behavior, as well as understanding the complex dynamics underlying the interaction between multiple access points and clients. Up until now, studies have focused on either microscopic analyses of individual networks (say, in an office building [60, 98]), wardriving studies that take a static snapshot of network deployments [32, 81], or macroscopic analyses of isolated mesh networks [28, 30]. The performance, behavior, and variation of wireless networks "in the wild" has never been studied at the urban scale.

This trend has created new difficulties for wireless networks. Increasing density can induce severe performance problems when multiple users share the same limited radio spectrum. Given the cooperative nature of 802.11 MAC protocols, it is possible for buggy implementations or malicious users to hog the spectrum. Likewise, the shared nature of the RF medium means that the efficiency of spectrum utilization is

a real concern. Malware that blasts a large number of packets or bandwidth-hogging file-sharing applications can cause performance problems for many nearby users. New applications and mobile devices are putting increasing pressure on wireless networks and leading to new user behaviors, such as free-riding on open networks.

There is also a serious concern about privacy and security of wireless LANs. The existing encryption standards, such as WEP and WPA2 [83], have all been shown to have weaknesses [80, 138], and it is possible to break WEP keys in a matter of minutes [44]. Moreover, many networks are unencrypted [48], so absent some form of end-to-end encryption, user traffic travels over the airwaves in the clear. Even in encrypted networks, it is possible to fingerprint individual wireless users based on leaked information, such as 802.11 probe requests [110]. A variety of end-user tools exist to capture wireless traffic, break encryption keys, and perform spoofing, denial-of-service and other attacks [1, 16, 26]. One of the goals of this chapter is to highlight the privacy risks that arise with large-scale, coordinated traffic snooping.

6.2.1 Why Argos?

The goal of Argos is to enable urban-scale monitoring of wireless networks, permitting multiple users to execute queries against the captured traffic. To achieve high spatial coverage, this requires multiple sensor nodes deployed throughout a city that can capture ambient wireless network traffic and perform filtering, aggregation, and trace merging with other sensors. To protect the privacy of users, sensor nodes should anonymize their raw traffic traces to remove sensitive information. To ensure scalability, sensor nodes should not require a wired back-channel, and preferentially

use a scalable wireless mesh to communicate with each other and the Internet.

A wide range of potential users would find value in a citywide wireless monitoring network. Wireless networking researchers can conduct detailed studies of traffic dynamics and network performance in a complex urban setting. Captured packet traces can be used to drive new protocol and system designs rather than relying on synthetic traffic that may not exhibit realistic behavior. Security researchers can use Argos to understand privacy and security risks inherent in wireless networks, such as the presence of malicious traffic and privacy leakage through side-channels. Finally, researchers in fields such as sociology and anthropology have an increasing interest in understanding wireless network traffic and user behaviors, such as characterizing the popularity of online media sources or Web search patterns across different parts of a city.

6.2.2 Challenges

There are many challenges associated with deploying a city-wide wireless monitoring network. Firstly, individual sniffer nodes often exhibit relatively poor packet capture rates. In a sparse, outdoor deployment, it is unlikely that we will be able to capture packets from a given source (client or access point) with high fidelity. Unlike dense indoor monitoring studies, where 99% or more of the traffic can be captured, in an outdoor sniffer network the typical capture rates are on the order of 5-10%. This requires that sniffers merge their packet traces to increase coverage, and new techniques that we develop to infer missing data and recover useful information from an extremely lossy data stream. Although many Argos queries focus only on high-

level, aggregate observations of wireless network activity, when "drilling down" into a particular source or packet stream is called for, we can improve capture coverage by coordinating between multiple sniffer nodes. For example, multiple sniffers in an area can synchronize their channel hopping schedules to maximize the joint probability of collecting packets from a given source.

Secondly, the system should scale up to many sniffers deployed across a large urban area. It is unreasonable to expect every sniffer node to have a wired connection to the Internet; this would substantially limit the locations at which sniffers could be installed and increase deployment cost and management overhead. For this reason, Argos makes use of wireless mesh networking to enable connectivity between sniffers and the Internet. The mesh backhaul uses an orthogonal frequency band (900 MHz) to avoid interference with the traffic sources being monitored. However, this gives rise to a related challenge in that wireless mesh networks are typically quite bandwidth-constrained, and the total amount of traffic being captured by a set of sniffers may exceed the capacity of the mesh. In our deployment, the backhaul mesh links range from a few Mbps to just a few hundred Kbps. For this reason, it is generally not possible to send the raw packet traces back to a central server for processing; as in many conventional sensor networks, filtering and aggregation must be performed within the sniffer network itself.

Thirdly, there is substantial diversity in the ambient wireless traffic present throughout a city, ranging from home networks with a small amount of traffic to large, enterprise-wide wireless LANs supporting thousands of users. The spatial distribution of wireless LANs is highly irregular and temporal diversity is seen across multiple

timescales. In addition, increasing diversity of the client device population, including (mostly) static laptops to highly mobile smartphones, makes it challenging to obtain a clear picture of traffic patterns.

6.3 Argos Architecture

Argos utilizes a two-tier architecture consisting of multiple *sniffer nodes* and a single *central server*. Sniffer nodes are interconnected via a backhaul network, such as a wireless mesh, allowing sniffers to communicate with each other as well as the central server. Sniffers capture wireless traffic and perform local filtering and processing on the raw packet streams. In addition, sniffers can perform *in-network traffic merging* to merge their individual streams, increasing capture fidelity. Results from queries are delivered to the central server where they are conveyed to the user. Figure 6.1 shows the architecture of a single sniffer node; the remainder of this section details each component.

6.3.1 User Queries

In Argos, wireless packet streams are processed by *user queries*. Multiple concurrent queries can execute on the Argos network at one time. Relatively few constraints are placed on the structure of queries, allowing for a wide variety of applications. Simple examples include generation of summary statistics over a sliding time window or filtering packets according to header fields.

Each query consists of two dataflow graphs composed of packet-processing operators: a *sniffer* dataflow, which is replicated on each of the sniffer nodes, and a *server*

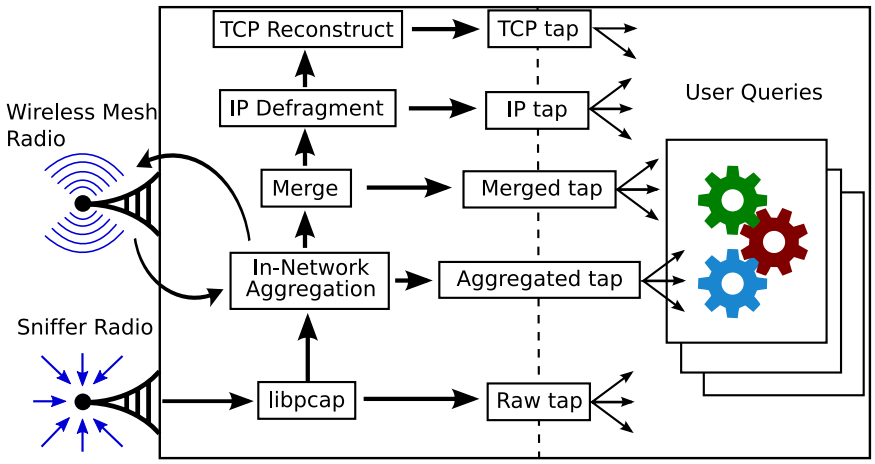


Figure 6.1: **This block diagram shows how packets flow through a single Argos sniffer.** Starting with packet capture from the sniffer radio in the lower left, packets are passed up the chain of processing elements. At each step, a *tap* provides access to the packet stream at that point for any interested user queries.

dataflow which is executed only on the central server. Each sniffer dataflow instance receives input packets from the Argos node on which it is running via one or more *taps* and sends results (via the backhaul network) to the server dataflow. The server dataflow may perform additional processing or aggregation on the results received from the sniffers, as well as log the output of the query and report the results to the user.

Each Argos sniffer provides five taps by which sniffer dataflows can receive packets, based on what degree of preliminary processing is desired. The *raw* tap provides packets as soon as they are captured by the local interface. The *aggregated* and *merged* taps provide packets after any duplicate captures are aggregated and then

merged from multiple sniffer nodes, as described below. The *IP* and *TCP* taps provide defragmented IP datagrams and reconstructed TCP sessions, respectively. Many queries make use of only a single tap, but in some cases combinations of multiple taps are useful.

User queries are implemented using the Click software router [86], which provides a rich set of interfaces for network packet processing. A query's sniffer and server dataflows are written directly in the Click language. Each query also specifies a small number of configuration and performance-related parameters, including the list of network taps that the sniffer dataflow uses. For each specified tap, the dataflow must have one input over which Argos will pass the appropriate packets. Any packets output by the dataflow are automatically transferred over the backhaul network to the central server and passed as input to the query's server dataflow. This allows easy aggregation of results from the entire sniffer network without having to deal with explicit network connections between sniffers and the central server.

6.3.2 Sniffer Nodes

Argos sniffer nodes consist of single-board PCs coupled with one or more 802.11 radios for packet capture. Our prototype runs on the CitySense network (detailed in Section 5.2), with each node using a Wistron CM9 802.11a/b/g radio with an 8 dBi omnidirectional antenna for packet capture. Each node runs an instance of the Argos sniffer process, which accepts raw captured packets from the operating system, performs in-network processing (as described below), and emulates a TCP/IP network stack (for IP defragmentation and TCP stream reconstruction). Packet capture is

performed using the standard *libpcap* [22] interface provided by the OS. The wireless device driver prepends each captured packet with a *radiotap* pseudo-header [20], which includes fields such as the received signal strength and type of physical layer modulation used.

Anonymization: Argos sniffers support a range of data anonymization mechanisms to prevent private information from leaking into user queries. At present, use of these mechanisms is optional, since we have not yet opened our prototype up to external users. Argos implements widely-accepted techniques for masking source and destination MAC addresses and IP addresses through hashing [111]. In addition, Argos can optionally drop arbitrary portions of a captured packet including individual header fields and the entire payload. Of course, the impact of these techniques depends very much on the semantics of the query, and there is no one-size-fits-all approach that will work for all queries. We are currently investigating the use of a stronger differential privacy guarantee that will prevent Argos queries from leaking information about specific users (§8).

Wireless mesh backhaul: Sniffers are interconnected via a backhaul network that provides them with connectivity to each other and the central server. Depending on the deployment scenario, a wide range of backhaul network options are possible. Direct Ethernet connectivity to sniffers may be possible in some situations, although the cost can become prohibitive, especially for sniffers deployed on rooftops and streetlights in a city, which is our focus. Cellular or WiMax connections may also be possible, although they incur additional subscription costs and would not enable direct connectivity *between* sniffers. Also, most current commercial offerings have

fairly limited uplink capacity (< 1 Mbps) [85, 141].

In Argos, we focus on the use of wireless mesh as a cost-effective and scalable backhaul network solution. Wireless mesh has been widely studied and deployed in a number of research and commercial settings [4, 8, 29, 42]. Given the reasonably close spatial proximity of most Argos sniffers, mesh is a good choice for interconnectivity, and can be deployed at low cost with no recurring fees. Mesh also allows sniffer nodes to communicate directly with each other to enable cross-sniffer collaboration, such as in-network merging of packet streams captured from multiple sniffers.

Of course, it is important that the wireless mesh backhaul does not interfere with the ambient traffic being monitored by Argos. In our prototype, the wireless mesh uses a secondary radio operating in a different frequency band (900 MHz) than the sniffer radio, to ensure that there is no crosstalk. The mesh itself is also an 802.11 network but operates in the non-standard frequency band and provides up to a few Mbps of throughput on each link.

6.3.3 In-network Traffic Processing

Given that an individual sniffer will only receive a partial view of the traffic from a given source, merging the packet streams from multiple sniffers can enable higher packet coverage. Merging also removes duplicates, which is important for the accuracy of some queries. Yeo et al. [145] were the first to use this technique and it has since been used by other wireless network monitors [60, 98]. In each of these systems, every sniffer sends its raw packet stream to a central server for merging, which is only viable for indoor environments with ample backhaul network capacity.

However, such a centralized approach is unsuitable for large-scale outdoor sniffer networks with constrained backhaul network capacity. Even with compression, the volume of captured traffic can overload the backhaul network, leading to substantial packet loss. Likewise, the centralized approach does not scale as the number of sniffers and network diameter increase.

To address this problem, we develop an approach to *in-network traffic processing* that attempts to balance the traffic load on each backhaul network link to minimize saturation and packet loss. The idea is to logically partition the global packet stream so that each sniffer node is responsible for some disjoint fraction of the stream. Sniffers forward each captured packet to its designated *aggregator* node, which (a) merges the partial streams it receives, and (b) processes the merged packets by executing the user queries.

Formally, we define the partitioning function $p : S \rightarrow N$ where S is the set of traffic sources (i.e., a wireless client or access point being monitored) and N is the set of aggregator nodes. Each sniffer capturing packets from a source $s \in S$ applies the partitioning function $p(s)$ to determine the aggregator $a \in N$ that should receive packets sent by s. Each aggregator merges the packets that it receives, before passing them to a locally-running instance of each user query. This approach is inspired by the *MapReduce* paradigm used for large-scale distributed processing in datacenters [64].

Traffic partitioning: Argos must take into account the cost of transferring each packet from the capturing node to the aggregator node. In a wireless mesh environment, it is desirable to avoid sending captured packets over multiple network hops, as this increases overall network load. Also, the choice of partitioning method will

determine the set of packets that are observed by each aggregator node after merging. This affects user queries, since each sniffer dataflow will only be able to observe its locally-merged portion of the global packet stream. Ideally, the partitioning method should not restrict the query logic.

Our approach to traffic partitioning is based on assigning all traffic captured for a given *basic service set* (BSS) to a single aggregator node. The BSS represents a single access point and all of its associated clients. The set of sniffer nodes that capture traffic from a given BSS tend to be within close physical proximity to each other, so this choice of partitioning function should minimize backhaul traffic. Also, many queries naturally wish to observe properties of individual clients or access points, so partitioning according to BSS allows queries running on each aggregator node to observe the full merged packet stream from each source.

To implement this approach, each sniffer maintains a table mapping each BSSID to its aggregator node. For each packet captured, the sniffer looks up the appropriate aggregator and forwards the packet. In cases where an aggregator has not yet been assigned, this table is constructed as follows. Each sniffer maintains a count of the number of packets captured from each BSSID. These counts are periodically reported to the central server. The central server assigns aggregation duties to the node with the largest number of received packets for each BSSID. This approach ensures that the fewest number of packets will need to be sent from *other* sniffer nodes to the aggregator, since the aggregator has already captured the largest number of packets for this BSS.

Although most 802.11 frames specify a BSSID, this is not the case for control

frames (e.g., ACK and RTS frames), which specify only their destination. To properly associate these packets with their BSSID, sniffers maintain a cache of MAC address to BSSID mappings, which is updated whenever a non-control frame is captured. The BSSID can then be determined for a control frame by consulting the cache. In addition, we need a special case for frames sent to the broadcast address. Instead of attempting to aggregate and merge these frames, we simply pass all broadcast frames through for local processing by the capturing sniffer.

Stream merging: On each aggregator node, stream merging is performed in a manner similar to Jigsaw [60]. Received packets' timestamps are adjusted to account for clock skew differences between sniffers. As suggested by Yeo et al. [145], we use beacon frames to update the current timeskew estimates between sniffers; unlike other wireless frames, beacons provide unambiguous synchronization points by virtue of their 64-bit timestamp field, which ensures that every beacon is unique. Next, the packet stream is merged by buffering packets for a window of time and searching for duplicate packets (from different sniffers) with similar timestamps; these are assumed to be copies of the same original packet transmission and are merged into a single packet. When a packet is output by the merger, it is annotated with a list of all of the sniffers that captured the original transmission, along with each sniffer's local capture time and received signal strength. This information is important for a variety of spatially-aware queries as we will discuss below.

In conventional approaches to in-network aggregation, nodes route sensor values up a collection tree to the root, merging values at each hop along the way. In comparison, Argos merges packets only once (at an aggregator node). This leads to two

efficiency improvements: first, routing a packet to its aggregator node is generally cheaper than routing it all the way to the root; second, execution of user queries can occur in-network on aggregator nodes instead of only at the root. Conventional approaches generally cannot push queries into the network because merging of values can happen at every node and thus the final value is not obtained until the root is reached. In Argos, however, a packet is "final" after its one and only merge (on its aggregator node) and thus we can immediately apply user queries.

Channel identification: After performing traffic merging, it is necessary to determine the original 802.11 transmission channel of each packet. When packets are first captured, Argos annotates them with the current channel of the capturing interface. However, this is not necessarily the correct channel, since 802.11 channels overlap and packets transmitted on one channel can be received on nearby channels. For example, in one 10-minute packet trace from an outdoor sniffer tuned to channel 2, over 75% of the received packets were actually transmitted on channel 1 (a much busier channel, leading to frequent packet bleed-over).

To handle this situation, each sniffer node maintains a cache of which channel each AP is operating on, which is announced in the AP's beacon frames. This information is used to determine the transmission channel for most captured packets. In cases where packets are captured from a BSSID for which no beacons have yet been received, we fall back to using the capturing interface's channel as a guess.

6.3.4 Protocol Stack Emulation

Merged traces are subject to protocol stack emulation, allowing for higher-level analysis of the network traffic by user queries: IP fragments are reassembled and TCP flows are reconstructed. Due to sniffers' limited capture abilities, many TCP flows are expected to be only partially reconstructed, leaving "holes" in the stream where one or more TCP packets were missed. Rather than rejecting these flows, we annotate each with a listing of which sequence-number ranges were and were not captured. We use timeouts to determine when to "close" and output each TCP flow, as there is no guarantee of capturing the FIN packet to mark its completion.

We expect that many classes of queries will be able to make good use of partially-reconstructed TCP flows. For example, any query examining application-level headers (e.g., HTTP requests or peer-to-peer traffic) may need to capture only the first 1-2 packets in a TCP stream. Rule-based intrusion detection systems, such as Snort [124], can detect many attacks with only a small number of captured bytes if they happen to line up with the appropriate attack signature.

Finally, we note that, in theory, queries could receive packets from the *merged* tap and then perform IP and/or TCP processing themselves, and this would actually simplify Argos itself, because it would reduce the number of network taps that it must support. However, IP fragmentation and TCP reconstruction can be quite memory intensive (due to large amounts of fragment buffering) and thus, when running multiple user queries, it is much more efficient to perform this processing once and share the results with all queries that are interested rather forcing each query to repeat the work.

6.3.5 Sniffer Channel Management

A unique challenge in Argos is that of simultaneously monitoring multiple radio channels. There are 14 total 802.11b/g channels (of which 11 are permitted for use in the US). One approach is to equip each sniffer node with multiple radios. Previous indoor WiFi monitoring systems, such as Jigsaw [60], used 3 radios on each sniffer node tuned to the most common 802.11 channels (1, 6, and 11); however, in a large, urban-scale setting we expect to see a substantial amount of traffic on nonstandard channels. For example, we estimate (by observing captured beacons) that 20% of the APs located in the vicinity of our current sniffer deployment utilize channels other than 1, 6 and 11. Clearly, if we want to include all nearby wireless networks in our monitoring, it would be prohibitive to equip sniffers with as many radios as there are available channels.

Our approach is to perform intelligent channel hopping on the sniffers to maximize packet capture coverage for user queries. The simplest approach would entail a simple static schedule with fixed dwell times, although this is unlikely to achieve the best results given the variations in channel occupancy. A better approach is to dynamically weight dwell times based on channel occupancy [65]. Since query requirements may vary, Argos does not stipulate any single channel hopping policy, but instead provides mechanisms for queries to specify their own policy.

Given that traffic of interest to a user query may be picked up by multiple sniffers, it is also desirable to *coordinate* channel hopping across nearby sniffers in order to maximize capture rates. Argos provides a *channel focusing* mechanism, whereby a sniffer can request that other nearby sniffers switch to a given channel so as to

improve the overall chances of picking up interesting traffic. For example, a query running on a given sniffer may detect traffic of interest (say, TCP packets destined for a certain IP address) and instruct other nearby sniffers to switch to the same channel to improve capture fidelity for the remainder of the traffic stream. When channel focusing is activated by a query running on some sniffer, the sniffer sends its current radio channel to all "nearby" sniffers. Currently we define sniffer "nearness" as geographic distance, although other options exist, such as the degree of overlap in captured traffic. Each of these neighboring sniffers will immediately change to the advertised channel. For a fixed period of time, the original sniffer and all recruited sniffers will ignore all channel-change requests, including those from local queries' channel policies as well as any other channel focusing messages.

With multiple concurrent user queries, each query may wish to listen on a different channel at the same time. In Argos, we address this through the use of *channel leases* and *prioritization*. A query can request that the sniffer radio be changed to a given channel c for duration d with priority p. The channel manager will grant the lease to the query with the highest priority. If the priority of the request is greater than the priority of the current lease, the current lease will be preempted. When the current lease expires, pending lease requests are processed in decreasing priority order. Often, each query is assigned a static priority at configuration time (e.g., according to user ID), although many policies are possible.

As an example, the default weighted channel policy will iterate through the available channels according to an estimate of the amount of traffic being captured on each channel. A query can temporarily override this default policy by requesting a

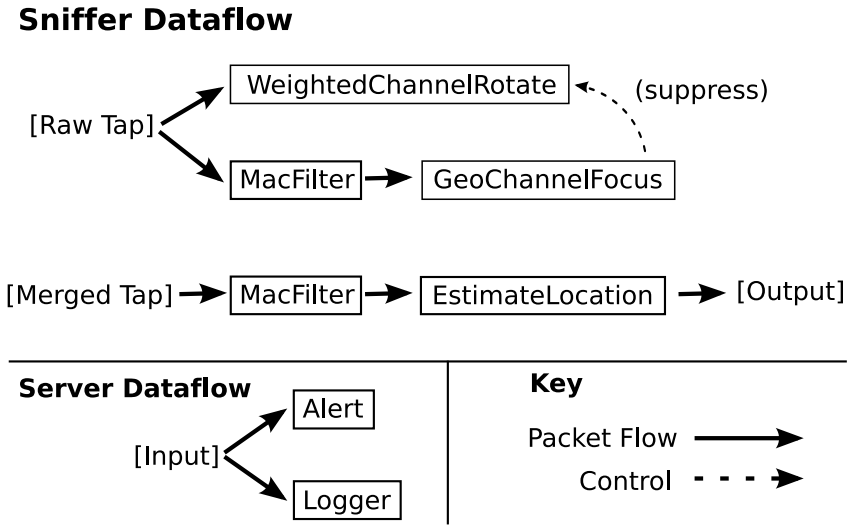


Figure 6.2: **The sniffer and server dataflows of a "stolen laptop finder" example query.** Packets are passed to or from the query at bracketed elements (e.g. `[Raw Tap]`).

coordinated channel focusing across a set of sniffers, by making a lease request as described above. Once the lease expires, the sniffer will return to the default policy.

6.4 Example Query: Stolen Laptop Finder

An example Argos query is illustrated in Figure 6.2, which implements a simple stolen laptop finder. The query maintains a list of target MAC addresses representing stolen laptops that should be tracked (e.g., after being reported stolen by the owner). On each sniffer, locally captured packets are passed (via the *raw* tap) to a *WeightedChannelRotate* element that periodically adjusts the channel-hopping schedule according to the number of packets observed on each channel. Packets are also

passed to a *MACFilter* element that drops all packets whose source MAC address is not in the list of targets. Matching packets are passed on to trigger the *GeoChannelFocus* element, which recruits nearby sniffers to switch to the same channel. This will increase the network's overall ability to recover packets from the stolen laptop (which may aid in identifying its location or current user). The *WeightedChannelRotate* element uses a low priority when obtaining leases for channel-hopping. This ensures that the *GeoChannelFocus* element, using a higher priority, can to suppress channel-hopping when a laptop is detected.

After in-network merging, the merged packet stream is passed to each query via the *merged* tap, where a second filtering for target MAC addresses is performed. Any matching packets are passed to an *EstimateLocation* element that uses each merged packet's annotation of which sniffers captured that packet and the associated received signal strength to estimate the transmitting laptop's location. Finally, lists of detected laptops and estimated locations are pushed to the central server, which alerts the end user.

6.5 Implementation and Deployment

We have implemented and deployed a complete prototype of the Argos system using the CitySense network (§5.2) as the hardware platform. It is important to note that the nodes in our deployment were not located to maximize packet capture. Instead, the siting of sniffers was dominated by physical and logistical factors; where we are allowed access, there are suitable mounting areas, and electric power is available). Hence, the geographic distribution of the sniffers is nonuniform and not intended to

be ideal for ambient wireless monitoring.

As noted previously, Argos is implemented using Click [86]. In addition to reusing a number of existing Click elements, we implemented a total of 46 new elements, which perform operations such as packet partitioning (by BSSID), packet-stream merging, and channel hopping control. We also make use of the QuickLZ [19] compression library for compressing network traffic.

6.6 Summary

Argos is an distributed wireless network monitor, designed for urban-scale sensing. Argos captures ambient wireless traffic and runs custom user queries to process or analyze the network packets. To scale to a large number of sniffers connected solely through a bandwidth-constrained wireless mesh network, Argos performs in-network aggregation of packets, which enables user queries to be run directly on the sniffer nodes. As compared to a centralized query model, this enables a high degree of data reduction and a corresponding reduction in mesh network traffic. To provide monitoring coverage of all 802.11 channels, Argos provides customizable modules that queries can incorporate to build arbitrarily complex channel policies. Finally, Argos provides *channel focusing* as a technique to increase the network's aggregate capture fidelity when needs arise (e.g. when particularly important traffic is detected).

Chapter 7

Argos System Evaluation

7.1 Performance Evaluation

In this section, we evaluate the performance of Argos' approaches to in-network traffic processing and coordinated channel focusing. For traffic processing, the primary metric we are concerned with is the amount of backhaul bandwidth required to send packet traces between sniffer nodes. We demonstrate that Argos' dynamic aggregator assignment substantially reduces the bandwidth requirements compared to sending all traffic to the central server for merging.

For coordinated channel focusing, the goal is to improve the sniffer network's ability to detect packet streams of interest by focusing multiple sniffer nodes on the same channel once a sniffer detects the target. We show that Argos' triggered channel focusing improves capture of event traffic compared to simpler channel hopping approaches.

Class	Frequency	Offered Traffic Load
APs	18%	5514 Bytes/sec
Active Clients	14.5%	287 Bytes/sec
Idle Clients	65%	49 Bytes/sec
Ad-hoc Stations	2.5%	931 Bytes/sec

Table 7.1: **Model for distribution and traffic load of transmitters, based on live deployment data.**

7.1.1 In-network traffic processing

First, we consider the impact of Argos' approach to in-network traffic processing on backhaul bandwidth usage. The benefits depend on several factors, including the volume of traffic being captured, the backhaul network topology, and the backhaul capacity. Given that our deployment of Argos represents a single point in a large parameter space, to study these effects we make use of a simplified analytical model that allows us to vary each parameter separately.

The model is based on a square grid of 25 sniffers, with each sniffer having direct mesh backhaul links to the four adjacent sniffers. The central server is placed in the center of the grid. There are 4000 traffic sources uniformly randomly distributed throughout the space; each source is randomly chosen from the four classes in Table 7.1, based on observations from the real Argos deployment described in Section 7.2. Each source i generates CBR traffic according to the rate shown in the table, denoted r_i. A sniffer s captures a fraction of the traffic from each source i according to the function $F_s(i) = \texttt{dist}(i, s)^{-\alpha}$, where `dist` is the distance from the source to the sniffer. We chose a pathloss exponent of $\alpha = 3$ which is a good estimate for urban

settings [122]. Hence, the total volume of traffic captured by s is $\sum_i F_s(i) \cdot r_i$.

Though intended to be general, this setup is based on parameters from our live deployment (§7.2), including the number of sniffers (25), the average number of traffic sources typically detected during the day (4000), and the offered traffic that we infer per transmitter (table 7.1). The constant factors of the packet capture function were scaled such that the average capture rate was 8%, to match capture performance inferred from our deployment. Some aspects of this model are undoubtedly unique to our deployment, although prior campus-based studies have reported comparable values for client and AP traffic rates [87], as well as for the ratio of the number of active clients to active APs [87, 137].

Given the capture rates for each sniffer, we can calculate the backhaul network load that arises with (a) a centralized policy in which each sniffer transfers all of its captured traffic directly to the sink, (b) a randomized assignment of traffic streams to aggregator nodes; or (c) Argos' in-network processing. In each case, we calculate the load imposed on each link of the backhaul network. Traffic is compressed by each sniffer prior to transmission, using an experimentally-determined compression ratio of 65%. For the centralized policy, we calculate a shortest path from each sniffer to the sink. The total load on each backhaul link is just the sum of the load induced by each sniffer whose traffic traverses that link. For the two in-network processing cases ((b) and (c)), aggregator nodes reduce the traffic stream by a percentage and transmit the remainder to the central sink node.

The amount by which in-network processing reduces a traffic stream depends on two independent factors. First is packet merging, which reduces traffic at a rate

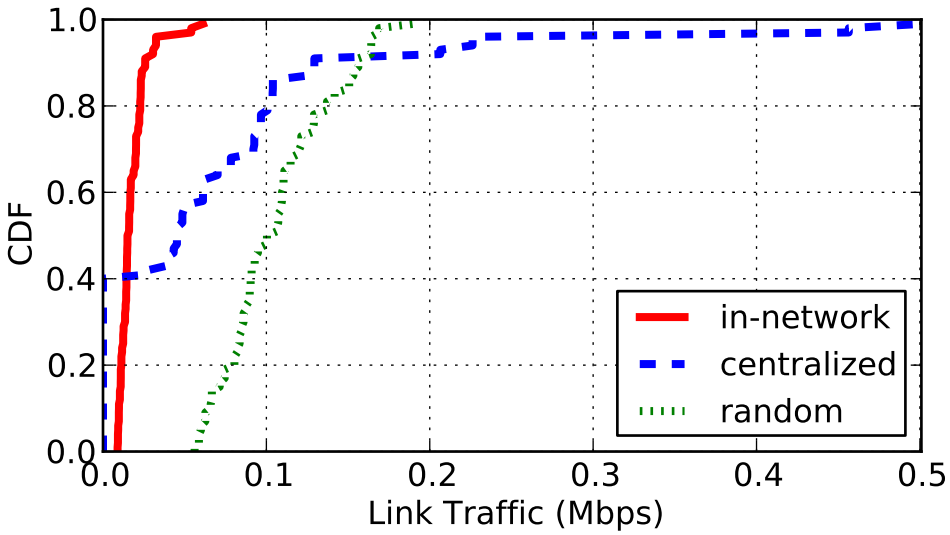


Figure 7.1: **Distribution of traffic load on backhaul network links with centralized and in-network traffic processing, using 25 sniffers and 4000 transmitters, resulting in a total offered load of 34 Mbps.**

proportional to the frequency of duplicates in the captured stream. This depends on the sniffer network's density and the channel selection pattern (since nearby sniffers can only capture duplicate packets if they are tuned to the same channel). In our network we see an average merge rate of only 7%.

Secondly, in-network processing also reduces traffic streams by executing the user queries. The queries' summed output data rate depends on the number and types of the queries, but in our case the 10 queries that performed all of the logging and data collection for sections 7.2 and 7.3 resulted in a 90% data reduction (this includes the 7% reduction just from packet merging). In other words, the combined output of this set of queries was one-tenth the data capture rate. From this, we assume in our

model that in-network processing results in a 90% data reduction on each aggregator node.

Figure 7.1 shows the distribution of load on backhaul links for each policy. In this case, the total offered load from the traffic sources is 34 Mbps. As the figure shows, the centralized approach induces a wide variation of traffic loads on each backhaul link, since the links closest to the sink must carry much more traffic than nodes near the leaves. In-network processing spreads the load more evenly across the mesh, with a max link usage of 0.06 Mbps, 8 times less than with the centralized policy. The randomized scheme only partially realizes the benefits of in-network processing, resulting in a max link usage of 0.19 Mbps, and incurs the highest mean and median link usages (0.1 Mbps each). Figure 7.2 shows how the max link usage varies as the total offered load from traffic sources increases.

7.1.2 Coordinated channel focusing

Next we evaluate our technique of channel focusing in order to allow multiple sniffers to capture traffic from a given source. The objective is to capture as many packets as possible from an "event of interest," such as an anomalous traffic pattern or transmissions from a specific wireless client. Since an interesting event can occur on any 802.11 channel at any time, an uncoordinated approach to channel hopping is unlikely to capture many packets from an event. In the channel focusing strategy, if any one of the sniffers detects a packet representing an interesting event, it recruits the 3 (geographically) nearest sniffers to switch to the same channel and begin capturing packets. Our metric is the number of packets captured for each event.

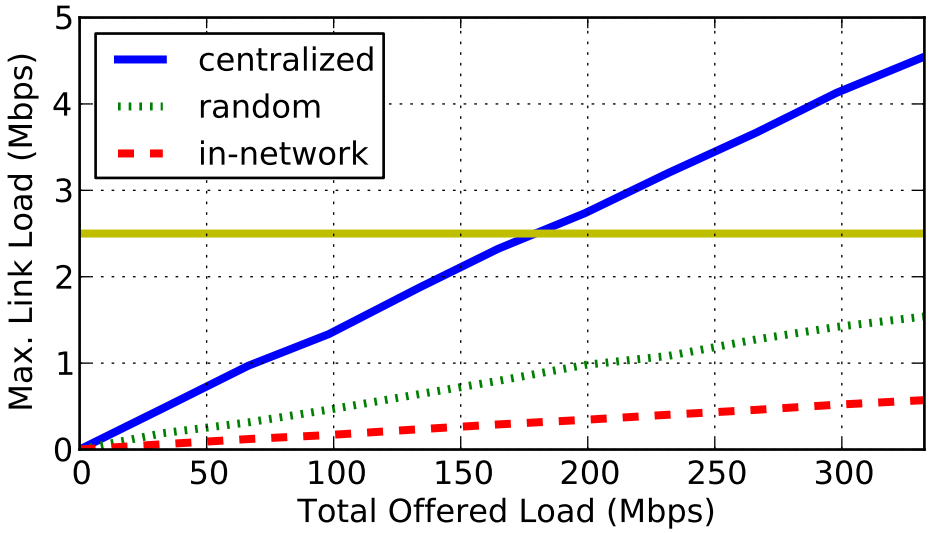


Figure 7.2: **Scalability of in-network processing with increasing load.** The number of sniffers was fixed at 25 while the number of transmitters was scaled up. For context, 2.5 Mbps is a (generous) estimate for current wireless mesh capacity limits; we expect that sniffer networks with any links loaded beyond this amount will suffer from network congestion.

In order to have a fair comparison, an ideal experiment should use the same source traffic for each channel hopping policy. However, given that each sniffer can only listen to a single channel at a time, we have to emulate this setting with a packet trace captured offline. We captured a set of 5-minute packet traces from 9 nearby sniffers in our Argos deployment. There are 11 traces in total, one for each 802.11 channel. For the experiments, sniffer nodes read from the traces instead of capturing packets from the radio. We emulate a situation where a sniffer can "change channels" by reading data from the appropriate trace; that is, the traces virtually overlap in

time, although they were initially captured at different times.

We randomly chose 5 "interesting events" within the captured traces. Each event is defined as a packet chosen at random followed by all packets transmitted from the same station over the next 10 seconds. Each event is comprised of at least 100 packets. The performance metric is the fraction of "interesting event" packets that are successfully captured, in aggregate, by the entire sniffer network.

We evaluate three different channel hopping schemes. *Rotation Only* uses weighted channel hopping in which each sniffer independently rotates through channels with dwell times proportional to the amount of traffic observed on each channel. *Detect and Hold* causes a sniffer to remain on the same radio channel for 10 sec after detecting a single packet from an interesting event, thereby increasing its chance of picking up more packets from the event. Finally, *Channel Focusing* has the first node that detects an event recruit nearby sniffers to switch to the same channel for 10 sec.

Figure 7.3 shows the fraction of packets captured for each of the 5 events (labeled A–E) for each of the three policies. In each case, channel focusing greatly improves the capture fidelity compared to the *Rotation Only* policy. Indeed, a single sniffer holding on the same channel (*Detect and Hold*) results in a substantial improvement to event capture, with improvements ranging from 41% to 77%. Channel focusing with neighbor recruitment further improves capture rates up to 34% above the *Detect and Hold* policy. This shows that there is clear benefit from coordination across sniffer nodes.

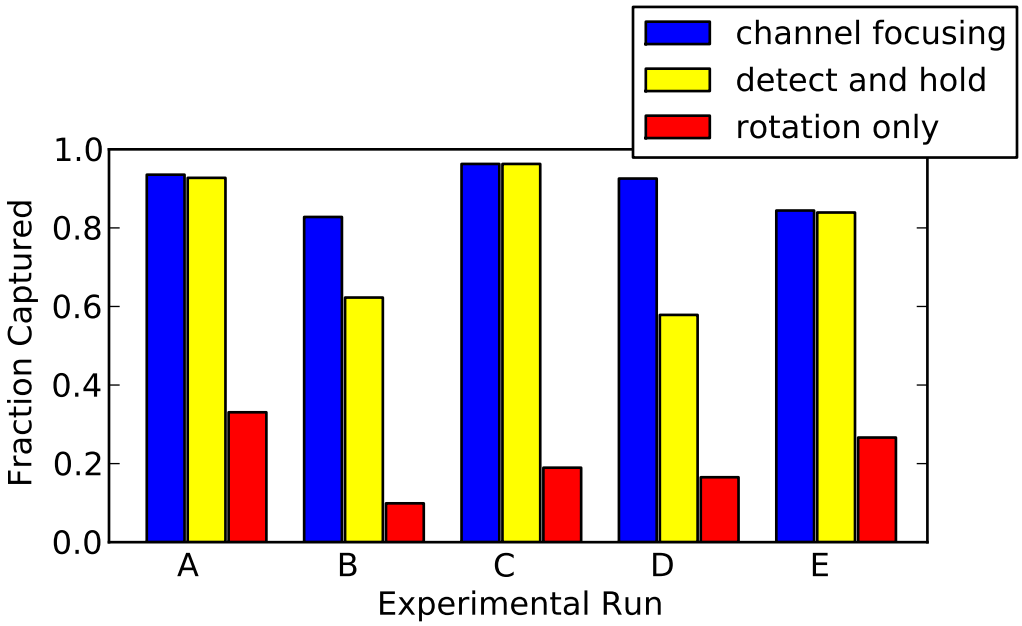


Figure 7.3: **Event traffic capture rates with and without channel focusing.**

7.2 Urban Wireless Traffic Characterization

In this section, we leverage the Argos deployment to perform a detailed characterization of wireless network usage across the city. Unlike previous wardriving studies [32, 81], we have the benefit of continuously monitoring wireless traffic from multiple vantage points, rather than taking a single snapshot view of wireless traffic from a single mobile sensor. This yields a much richer picture of the urban wireless landscape than previous studies have revealed.

7.2.1 Overall network population

We conducted a number of different measurement studies on Argos that spanned a total of 6 months. This section presents data from a 12 day period over which detailed traffic measurements were recorded. In total, we detected 2630 access points and 65,073 wireless clients, although our rate of capture varied widely across this population. By way of comparison, these counts are each over 25 times larger than those reported for a 24-hour trace taken by the largest indoor wireless sniffer deployment [60]. A usage study of Google's 500+ node outdoor mesh network [28] recorded 30k clients over a 28 day trace, but this includes only the mesh traffic, as opposed to all nearby wireless networks.

Overall, we captured a total of 1.1 TB of traffic across 2.4 billion packets. It turns out that a single wireless client in the vicinity was responsible for 81% of all the captured bytes and 30% of all captured packets; this "spammer" node appears to be continuously transmitting max-sized 802.11 frames. Unfortunately these frames are encrypted so there is not much we can say about them – this traffic is excluded from the remaining analyses in this paper. The amount of non-spammer traffic captured per sniffer varied greatly, from just 3 MB to nearly 23 GB, with an average and median of 8.5 GB and 9.6 GB, respectively. This variance is not surprising, due to differences in the sniffers' locations as well as the densities and activity levels of nearby wireless networks. Overall, 61% of the packets and 44% of the networks utilized some form of encryption.

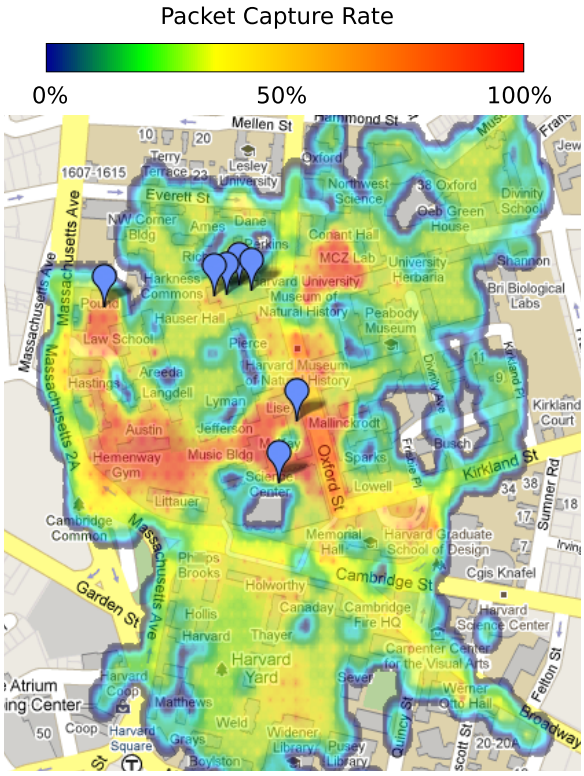


Figure 7.4: **Spatial coverage of the Argos sniffer network, in terms of the fraction of packets captured from a mobile laptop in each location. Sniffers are shown as blue markers.**

7.2.2 Spatial coverage

The first major question is, how much spatial coverage is afforded by the Argos sniffer network? That is, over what physical area can the sniffers observe ambient wireless traffic? We conducted an experiment in which a user with a laptop roamed around the Harvard University portion of our deployment area. This area includes a number of multi-story buildings as well as several open areas, typical of a university

campus. The laptop was equipped with a GPS receiver and continuously broadcasted UDP datagrams containing the current GPS coordinates. The sniffers observed this traffic and, for each position of the laptop, we determined the fraction of packets captured by Argos.

Figure 7.4 shows the results; the (extremely noisy) packet reception rates from all 7 sniffers are combined and the data is smoothed to yield a clearer picture of the spatial coverage. The figure shows that although our capture ability is related to distance (as expected), it's a very rough correlation. On the one hand we were able to capture packets from a laptop up to 430 m from the nearest sniffer, which is surprising given the presence of several tall buildings obstructing the line-of-sight path between the laptop and sniffers. On the other hand, there are also areas quite close to multiple sniffers that had low packet capture rates.

7.2.3 Traffic capture coverage

The second question is, how much of the total ambient traffic was Argos able to detect? This is a difficult question to answer, as we do not have ground truth as to how much ambient traffic there actually is. As an estimate, we compute packet reception coverage for each sniffer by counting 802.11 beacons received from access points. Access points broadcast beacons at a fixed interval (typically 10 Hz), so it is straightforward to calculate the percent of beacons captured from only the total number of beacons captured and the elapsed time. As noted by others [84], beacon capture rates can be used as a rough proxy for overall packet capture rates, although their capture rates tend to be somewhat higher than other traffic as they are transmitted at low

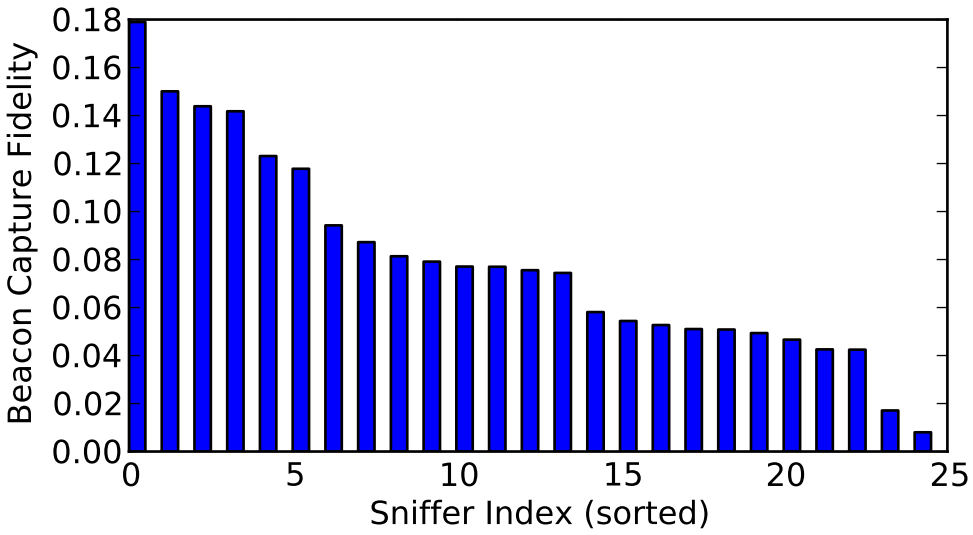


Figure 7.5: **Capture fidelity as fraction of beacons captured by each sniffer.**

PHY rates.

As shown in Figure 7.5, sniffer coverages range from 18% to just under 1%; the overall coverage of the entire network was 8%. These values are much lower than what is typically seen with indoor monitoring networks, where coverage can exceed 95% [60]. We discuss the implications of this (and possible ways to improvement traffic capture) later in this section. Also note that this figure omits APs for which we captured only a small number of beacons (< 10) as fidelity estimates in these cases are likely to be inaccurate; 21% of detected APs fell into this category.

The next question we can ask relates to the variation in traffic over time. Figure 7.6 shows the types of TCP traffic (classified by port number) captured over a representative weekday. Unsurprisingly, HTTP and HTTPS are the dominant traffic classes, with Email (POP3, IMAP), FTP and NetBIOS (not shown) making up most

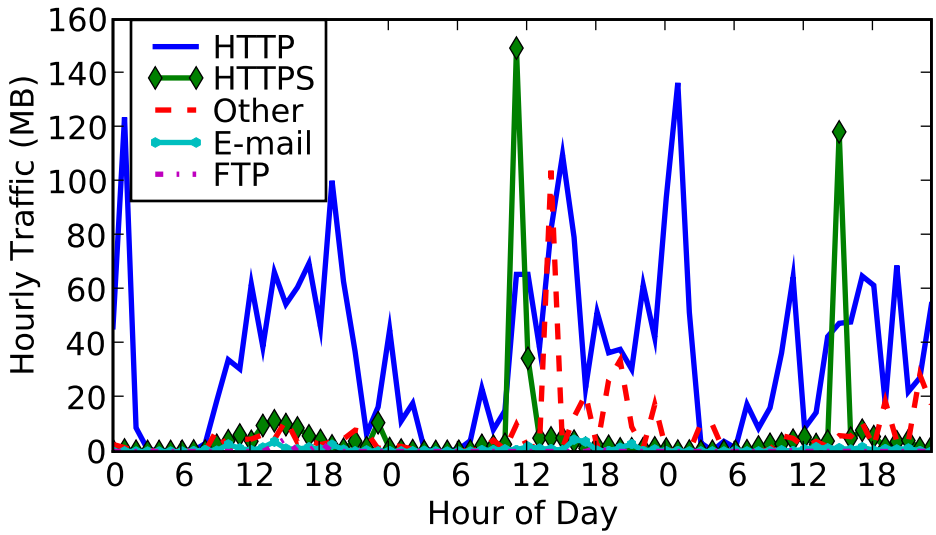


Figure 7.6: **TCP traffic, by class, over 3 consecutive weekdays.**

of the identifiable remainder. Although the expected diurnal patterns are clearly present, the captured traffic was quite bursty with hourly per-class traffic spikes up to 147 MB. Each of the two prominent HTTPS traffic spikes represents traffic from a single AP. However, in both cases multiple clients were associated with the AP so we could not unambiguously determine which specific client was the source.

Similarly, the two HTTP traffic spikes at 1:00am on days 1 and 3 were each traced to a single AP. Since the traffic events occurred in the middle of the night, only 2-3 clients were observed associated with the AP. One client in particular is the likely source for both of the traffic spikes; this client was associated throughout both events, during which we observed the client make hundreds of web requests. Many of these requests were to video-hosting sites (e.g. `youtube.com`, `netflix.com`) further explaining the sudden traffic usage.

7.2.4 Discussion

From these measurements, it is clear that Argos' ability to monitor large populations over significant geographic areas comes at the cost of missing a significant percentage of the ambient traffic. Although this somewhat constrains the applications suitable for Argos, our current performance turns out be adequate to perform many interesting analyses (§7.3). Nonetheless, its useful to briefly consider two sources of packet loss that we have identified, with an eye towards future system improvements.

Firstly, one must consider the urban environment, with its frequent line-of-sight path obstructions by nearby buildings, plus the extremely dense penetration of extant WiFi networks. Although these difficulties are largely innate to urban settings, we expect that one could reduce their impact as our sniffers were not optimally located for packet capture and simply use 8 dBi omnidirectional antennas, rather than dish or patch antennas that would have aided packet capture.

We also note that Argos' long-range packet capture may be biased towards packets sent at low PHY rates, as these packets can be received at lower signal strengths than those sent at higher PHY rates. Since we have no way to know the true distribution of PHY rates used in our measurement area, we instead utilize packet traces captured in other settings as an imperfect comparison. We examined the distribution of PHY rates present in packet traces from a conference in 2006 [55] and an academic building in 2007 [58]. In both cases, the sniffers were densely deployed and thus captured nearly 100% of the traffic, yielding a complete picture of the PHY rates in use. The indoor traces show 20% and 6% (respectively) of packets were sent at 36 Mbps or above, whereas this is true for only 0.5% of the packets captured by Argos. We tentatively

conclude from this that Argos is indeed biased against high data rate packets. At PHY rates of 24 Mbps and below, there was no clear trend.

Secondly, many packet transmissions are missed simply because all nearby sniffers are tuned to other channels at the time. The above measure of fidelity estimates Argos' capture rate across *all* traffic, but this is not optimal for many queries. If a query is interested in a subset of the traffic, the rate of capture of *just that traffic* can be increased if the query can predict on which channels the traffic is most likely to occur. For example, our queries use the default channel policy, which weights channels by the total traffic volume on each, whereas a query interested only in TCP traffic might instead weight channels by just the 802.11 *data* traffic. This would deprioritize channels with lots of 802.11 management traffic (e.g. beacons) but little higher layer traffic.

We tested a simplistic scenario with a query interested solely in traffic from channel 1, which lead to a network-wide fidelity of 25%. This is a significant improvement over the 8% reported earlier, although this is somewhat of a best-case scenario because the channel prediction was trivial (all nodes statically assigned to channel 1) – we predict that most queries can realistically expect a fidelity somewhere between 8% and 25%, depending on the pattern of traffic they are interested in and how accurately one can predict when and on which channels the traffic will occur. Developing channel policies to that can forecast different classes of traffic is a challenging area of future work.

Rank	*Requests*	*Clients*	*Site*
1	1856	301	www.facebook.com
2	736	239	www.google.com
3	267	5	www.huffingtonpost.com
4	249	5	www.craigslist.org
5	239	1	www.repubblica.it
6	232	143	www.apple.com
7	162	22	images.apple.com
8	139	17	my.lesley.edu
9	133	46	www.youtube.com
10	131	7	www.seas.harvard.edu

Table 7.2: **Top ten websites visited by users, ranked by number of captured requests.**

7.3 Case Studies

In this section, we showcase Argos' capabilities for enabling complex user queries against the rich data stream of ambient wireless traffic. We present four case studies to highlight potential use cases for Argos: tracking of popular Web sites and Google searches from different parts of the city; detection of malicious traffic and malware over the airwaves; tracking of public transport services; and fingerprinting individual clients based on their wireless network behavior. These case studies are intended to demonstrate Argos' capabilities to distill complex wireless network traffic to yield a high-level view.

7.3.1 Popular Websites and search patterns

The first case study deals with exploring the popularity of Web sites visited by wireless users around the city. Determining which website a user is visiting is non-trivial: simply looking at packets destined for port 80 reveals IP addresses, but many of these are for auxiliary servers (such as ad sites and CDN servers) that are not being directly visited by the user. We implemented a query that captures the HTTP request headers from users and looks for a `Host:` field in the request, which most browsers add to indicate which host is being visited. We also look for HTTP responses where a `Set-Cookie` header is used. Using this technique, we captured a total of 523,818 HTTP requests, for which we could determine the site name for 62,292 of them. 6143 unique websites were visited, and Table 7.2 shows the top 10 after we exclude ad sites and CDN servers.

We captured Web search queries by looking for HTTP requests to URLs containing appropriate strings (e.g., Google searches include `&q=` followed by the query phrase). We captured 1,725 web searches, 686 of which were to Google. The most popular search terms are not particularly surprising, dominated by words such as *in*, *of*, and *boston*.

7.3.2 Malicious traffic

Most studies of malware on the Internet [103, 93] are based on traces captured from wired networks or border routers. One notable exception is Stone-Gross et al.'s study of user traffic at a conference [132]; the authors demonstrate that identifying malware through wireless sniffing is possible, although they also utilized packet traces

Count	*Snort Rule*
1229	SQL ping attempt
794	x86 shell code exploit
204	SQL probe response overflow attempt
121	Web server buffer overrun attempt
87	FTP traffic encrypted

Table 7.3: **Top five Snort alerts.**

taken from the wired network. Additionally, they estimated a capture fidelity of more than 90% owing to their compact, indoor setting.

We are interested in whether Argos can detect malicious behavior over the airwaves. We set up an Argos query to feed merged traces to the Snort [124] network intrusion detection system, configured with a standard set of 5303 detection rules. We added two additional rules [6] to detect traffic from the Conficker botnet. Over 11 days, Snort raised 2979 alerts across 37 rules; 141 unique client devices were implicated by the alerts. Table 7.3 shows the top 5 triggered alerts. The Conficker shellcode botnet rules triggered 5 times on one client, which was subsequently observed engaging in what appeared to be typical scanning behavior, attempting to connect to TCP ports 80 and 445 across a large number of geographically-dispersed IP addresses.

Note that Snort, designed primarily for wired networks, assumes that it can observe complete packet traces – given the limited fidelity of wireless packet capture in Argos, it is promising that a wide range of malicious activity could still be detected. These results demonstrate that there is substantial promise for the use of passive monitoring for detecting and tracking malicious behavior in urban scale settings.

7.3.3 Tracking public transport services

Another interesting use for Argos involves detecting and tracking public transport vehicles and their passengers. Soon after deploying Argos, we noticed the occasional appearance of SSIDs such as `Coach0228_Box-078` and `MBTA_WiFi_Coach1601_Box-139`. A bit of research, and the observation that these SSIDs were only ever captured by the sniffers in the western Cambridge cluster near a commuter rail line, lead us to the realization that these were access points installed on public commuter trains.

To test the sniffers' ability to track moving vehicles, we wrote a query to detect the passage of trains and infer their direction of travel. Sniffers perform weighted channel hopping to search for packets from any known train BSSIDs; when a train packet is detected, channel focusing is initiated to maximize the network's capture potential while the train passes. To infer the direction of travel, we rank-order the 3 sniffers located alongside the railway tracks according to the mean timestamp of the packets captured at each.

Table 7.4 shows one typical day's worth of data. Argos captured 2801 packets from 42 unique access points, corresponding to 2–4 APs per train. Trains were detected passing by on 34 occasions, with directionality inferred in 29 of the cases (4 of which were incorrect). In total, we have observed 456 unique users associated with the train networks and have captured 804 Web requests from passing trains.

We have also detected WiFi networks related to buses in the area, with names such as `boston_bus_600-798`, `Dartmouth Coach 0801`, and `Concord Coach 921 Left Side`. 47 bus-mounted networks have been discovered by the Argos node that is mounted near Interstate 93, a major highway leading into and out of Boston. We have detected

Observed		**Scheduled**		**Observed**		**Scheduled**	
Time	Direction	Time	Direction	Time	Direction	Time	Direction
7:10	In	7:03	In	16:13	Out	16:12	Out
7:34	**Out***	7:30	In	16:24	In	16:19	In
7:46	Out	7:39	Out	16:52	(none)	16:52	Out
7:48	In	7:47	In	16:58	In	16:56	In
8:18	(none)	8:11	In	17:05	**In***	17:02	Out
8:33	Out	8:30	Out	17:35	Out	17:32	Out
8:57	In	8:40	In	17:53	(none)	17:51	In
9:08	Out	9:07	Out	17:56	**In***	17:52	Out
9:22	In	9:18	In	18:33	Out	18:32	Out
9:55	Out	9:52	Out	19:48	Out	19:48	Out
10:11	(none)	10:04	In	19:51	In	19:48	In
11:32	Out	11:32	Out	20:36	In	20:36	In
11:43	In	11:41	In	20:58	Out	20:57	Out
12:41	In	12:36	In	21:37	In	21:36	In
13:36	Out	13:32	Out	22:55	(none)	22:52	Out
14:20	**Out***	14:19	In	23:37	In	23:34	In
15:14	Out	15:12	Out	00:25	Out	00:22	Out

Table 7.4: **Train schedule (time and inbound/outbound direction) inferred from captured traffic ("observed"), compared to the true schedule as published on the MBTA website ("scheduled") for Dec 9, 2009.**

144 unique users and 176 web requests on these networks.

7.3.4 Wireless client fingerprinting

Our final case study asks the question: how much information can we infer about individual users based on their wireless network traffic? Clearly, a great deal can be learned by watching an individual user's Web surfing or email activity, a concern which is readily addressed through encryption.

However, *even if a user associates with an encrypted network*, it is possible to glean a great deal of information based on data that the 802.11 protocol transmits in the clear. An example is the contents of 802.11 probe request packets. To discover networks in its vicinity, an 802.11 client broadcasts probe requests containing the plain-text SSIDs of networks it wishes to discover; if any of these networks receive the request they send a corresponding probe response. Many operating systems remember the full set of networks that a user has previously associated with, and transmit this set of SSIDs in probe request messages. As a result, by capturing 802.11 probe requests, we can learn the locations that a user has previously visited.

To explore this, we wrote a query to capture 802.11 probe requests and the SSIDs contained within them. We detected 21,546 unique clients with a mean of 470 probe requests sent per client. Looking at the number of *unique* SSIDs observed per client, the distribution is heavy-tailed, with only a single SSID observed for the majority of clients. Nonetheless, a number of clients probed for enough unique SSIDs, to reveal significant information about their past behavior.

We manually inspected the top five clients by number of unique SSIDs; they are

Rank	*Total probe reqs*	*Unique SSIDs*	*Locatable SSIDs*
1	7431	49	28
Locations: *MBTA trains, Portland OR, Acton MA, Austin TX*			
2	87	48	11
Locations: *Harvard, BU, MIT*			
3	370	46	10
Locations: *Manchester UK, Belgium, Tulsa OK, Chicago IL, ...*			
4	632	47	10
Locations: *Little Rock AR, Sacramento CA, Atlanta GA, ...*			
5	120	47	0
Locations: *(none identified)*			

Table 7.5: **Locations previously visited by users, inferred from probe requests.**

summarized in Table 7.5. For each SSID, we used the `wigle.net` wardriving database to attempt to locate each network, and found for 4 of the 5 clients that many of the SSIDs had a unique geographic location. The table lists some of the locations that each user is inferred to have traveled based on their probe requests.

Clearly, this is a case where the 802.11 protocol is revealing potentially sensitive information about a user's whereabouts, well beyond the sensing range of the Argos network itself. Most WiFi users are probably unaware of this feature of 802.11. It is worth underscoring that this information is revealed even if the user only ever associates with encrypted networks. The use of probe requests to track users has been previously proposed, although in a different context [110]; instead of considering clients' past behavior, as we do, the researchers instead used the set of SSIDs that each user probes for as a way to uniquely identify that user even if they spoof their MAC address.

The tracking techniques described here can become more powerful when used in conjunction. For example, user #1 above seems to be a frequent MBTA train user. We could conceivably perform (i) fine-grained spatial and temporal tracking over short distances (by mapping networks they are observed to associate with), (ii) inference of spatial and temporal behavior over moderate distances (by combining their past behavior with MBTA train schedules), and (iii) coarse-grained spatial (but not temporal) tracking over long distances (via mapping of probe requests' SSIDs). There are severe implications for user privacy if this technique were employed on a wide scale.

7.4 Summary

Our evaluation of in-network processing shows that this technique is highly effective at reducing the *peak* mesh traffic (as measured by the most heavily loaded link). We find that the actual merging of packets results in relatively modest savings (7%); instead, the great majority of gains are obtained by data reduction performed by the queries themselves (which are dependant on in-network processing to be able to run within the sniffer network). Although this means that the practical benefits depend on the specific queries being run, our (unoptimised) queries achieved 90% data reduction. For coordinated channel focusing, our evaluation shows that capture of targeted traffic can increase by up to 34% above a *detect-and-hold* policy.

To evaluate our deployment, we present data obtained from a detailed, 12-day period of traffic measurements. During this time we ran four simple case study queries to test the utility of the system and showcase some preliminary insights that

Argos enables: popular websites and search patterns, detection of malicious traffic, tracking commuter trains, and probe-request-based user fingerprinting and tracking.

Chapter 8

Discussion and Future Work

In this chapter we offer thoughts on future trends within the space of real-time query systems, discussing how Cobra and Argos fit into the bigger picture. This leads naturally to a brief overview of potential directions for future work.

8.1 Generalizing Cobra and Argos

In Chapter 1 we argued that current general purpose SPEs fall short of meeting the needs of applications in certain application domains, leading to a proliferation of specialized systems in those areas. With Cobra and Argos, we have presented two new query systems designed to operate in two such domains; filtering and aggregation of RSS feeds, and wireless network monitoring, respectively. However, this dissertation's primary goal is to identify *general* principles for real-time querying, of which Cobra and Argos are simply two possible instantiations. Thus it is important to consider generalizations of Cobra and Argos.

It is somewhat tempting to suggest that, ideally, the functionality of existing SPEs (both general purpose and domain-specific) should eventually be merged so that a single system can handle the needs of a huge variety of application domains. However, we find this somewhat unrealistic, as the diversity of functionality would simply overwhelm one system. This is not to say that its not technically possible, only that managing the development of such a system would be unwieldy. For any one use (say, financial monitoring), a great majority of the system's functionality would be useless (dedicated to other domains). Such a scenario would likely lead to a splintering of the development effort into smaller, more focused areas.

Instead, we believe it is more realistic to target query systems that each focus on a single, large application area. This is a balanced approach; the domain should be broad enough to enable significant reuse of the system, and yet small enough that little of the system's functionality would be irrelevant to any particular application.

Cobra, for example, could be folded into a generalized query system over streaming text. The rise of "Web 2.0" and user-generated content on the Internet has yielded a wealth of textual data, of which blogs and other RSS feeds are merely one class. One can imagine a system that pairs a generalized text-processing engine with optimized data collection modules for RSS content (already implemented in Cobra), Twitter feeds, Facebook updates, or even SMS text messages. Many systems that currently use only one of these data sources (e.g. [90, 125]) could seamlessly integrate these other sources (social and legal issues of access to the data notwithstanding).

Argos is built using Click [86], and thus is already part of an ecosystem designed around networking systems. Click is a toolkit (language, compiler, and runtime) orig-

inally designed for building modular software routers, but has since adapted for many diverse networking-related purposes. Click has been used to build a honeypot [139], a wireless protocol simulation [91] and an Ethernet "watchdog" device [66], to name but a few examples. The common thread between these systems (including Argos) is that their basic operation is to operate on a stream of packets. Any new applications that are centered around packet-based operations are typically able to reuse many existing Click operators. In our work developing Argos we used many existing Click operators while also introducing a variety of new operators, mostly related to the analysis of wireless packets, which is an area that has received less attention from the Click community.

This general idea is quite similar to one put forth by Stonebraker and Çentintemel in 2005 [133]; namely, that the commercial database market "will fracture into a collection of independent database engines." The authors primarily cite stream-processing, with its many deep, architectural differences from traditional DBMSs, as driving this fissure, but they also speculate on other possible database specializations: data warehousing, sensor networks, text search, scientific databases, and XML databases. The "vision" of this work mirrors our own.

8.2 Future Work

Working to expand and integrate Cobra and Argos into more generalized platforms, as above, is one potential direction for the future, although a number of more targeted opportunities also exist.

For Cobra, it would be beneficial to provide users with tools for focusing their

queries, in the event that they receive too many results. Our current matching algorithm does not rank results by relevance, only by date. Likewise, the algorithm is unconcerned with positional characteristics of matched keywords; as long as all keywords match an article, it is delivered to the user. Much work has been done on ordering web search results (e.g. PageRank [78]), although it is unclear how successful these techniques would be at ranking RSS entries.

Another open question is how to rapidly discover new Web feeds and include them into the crawling cycle. According to one report [130], over 176,000 blogs were created *every day* in July 2006. Finding new blogs on popular sites such as Blogger and LiveJournal may be easier than more generally across the Internet. While the crawler could collect lists of RSS and Atom URLs seen on crawled pages, incorporating these into the crawling process may require frequent rebalancing of crawler load. Finally, exploiting the wide distribution of update rates across Web feeds offers new opportunities for optimization. If the crawler services could learn which feeds are likely to be updated frequently, the crawling rate could be tuned on a per-feed basis. This is particularly important considering the high rate of blog abandonment [14].

Argos would benefit from a more rigorous notion of privacy in the user query interface. We are exploring the use of *differential privacy* [100] which provides a very strong guarantee: namely, query results are formally indistinguishable when run with and without any one record. Effectively, this can make it impossible for an Argos user to know definitively whether or not a specific person (wireless client) is being monitored by Argos. This will require that Argos respond only to statistical queries, rather than yielding specific details of individual packets or users, but we believe this

could be sufficient for many users.

Secondly, Argos' capture coverage could be improved by incorporating mobile sniffer nodes into the system (e.g. mounted on cars or buses [47]), or allowing individual users with laptops or WiFi enabled mobile phones to run Argos sniffers on their own devices; this raises the substantial challenge of managing the population of participatory sensors and integrating their data with that captured from static Argos nodes.

Chapter 9

Conclusions

In recent years, advances in networking technology have lead to remarkable increases in both bandwidth and ubiquity. Thus, not only are existing systems now producing more data than ever before, but now more *kinds* of systems are networked and able to produce data. As a way of contending with this growing flood of data, this dissertation considers the use of real-time query systems and, particularly, their ability to operate on *complex data sources*.

We presented three design principles for building *scalable*, *effective* query systems for complex data sources: query interfaces tailored to the application's specific data types, optimized data collection processes, and allowing queries to provide feedback to the collection process. To demonstrate these principles, we presented the design and evaluation of the Cobra and Argos systems.

Cobra is a system for real-time content-based search and aggregation on Web feeds. Cobra is designed to be incrementally scalable, as well as to make careful use of network resources through a combination of offline provisioning, intelligent crawling

and content filtering, and network-aware clustering of services. Our prototype of Cobra scales well with modest resource requirements and exhibits low latencies for detecting and pushing updates to users.

Argos is an urban-scale sensor network that combines the use of in-network traffic processing, intelligent channel management, and a rich user query interface to allow users to access this complex source of ambient data. We have shown that Argos' approach to in-network traffic processing substantially reduces backhaul network load and that our dynamic channel hopping strategy improves capture coverage. Through an extensive characterization of citywide WiFi traffic and several case studies, we have demonstrated Argos' ability to support detailed analysis of the network behavior.

In addition, we have described our architectural vision for the CitySense testbed, along with an analysis of selected network performance metrics. CitySense serves as a novel platform for urban-sensing and mesh networking (with real-world physical effects) applications; the Argos project serves as an example of the new kinds of research that CitySense enables.

The research presented here is meant to broaden the range of data sources for which real-time query systems can be built, but there are certainly many more data sources that will require specialize approaches not considered here. And we should expect that in the near future, the need will likely arise for systems that can query over entirely new kinds of data sources.

Bibliography

[1] Aircrack-ng. http://www.aircrack-ng.org/.

[2] Akamai. http://www.akamai.com/.

[3] Cascading. http://www.cascading.org.

[4] Champaign-urbana community wireless network. http://www.cuwin.net/.

[5] Complex event processing, event stream processing, streambase streaming platform. http://www.streambase.com/.

[6] Detecting conficker — the honeynet project. http://www.honeynet.org/node/388.

[7] Fade to black. http://www.bureauit.org/ftb/.

[8] Freifunk. http://www.freifunk.net/.

[9] Google blog search faq. http://www.google.com/help/blogsearch/about_pinging.html.

[10] Ibm research esps pilots. http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.pilots.html.

[11] Iperf. http://iperf.sourceforge.net/.

[12] Js-javaspaces service specification. http://river.apache.org/doc/specs/html/js-spec.html.

[13] Limelight networks. http://www.limelightnetworks.com/.

[14] Livejournal. http://www.livejournal.com/stats.bml.

[15] London congestion pricing. www.vtpi.org/london.pdf.

[16] Lorcon. http://802.11ninja.net/lorcon.

[17] Net dictionary index – brown corpus frequent word lising. `http://www.edict.com.hk/lexiconindex/`.

[18] Olsr. `http://www.olsr.org/`.

[19] Quicklz. `http://www.quicklz.com/`.

[20] Radiotap.org. `http://www.radiotap.org/`.

[21] Skyhook wireless. `http://www.skyhookwireless.com`.

[22] Tcpdump/libpcap public repository. `http://www.tcpdump.org/`.

[23] Vmware esx server. `http://www.vmware.com/products/vi/esx/`.

[24] Welcome to apache hadoop! `http://hadoop.apache.org`.

[25] Wigle.net. `http://www.wigle.net/`.

[26] Wireshark. `http://www.wireshark.org/`.

[27] Tibco publish-subscribe. `http://www.tibco.com`, 2005.

[28] Mikhail Afanasyev, Tsuwei Chen, Geoffrey M. Voelker, and Alex C. Snoeren. Analysis of a mixed-use urban wifi network: when metropolitan becomes neapolitan. In *IMC '08: Proc. of the 8th ACM SIGCOMM conference on Internet measurement*, 2008.

[29] Mikhail Afanasyev, Tsuwei Chen, Geoffrey M. Voelker, and Alex C. Snoeren. Analysis of a mixed-use urban wifi network: when metropolitan becomes neapolitan. In *IMC '08: Proc. of the 8th ACM SIGCOMM conference on Internet measurement*, 2008.

[30] Daniel Aguayo, John Bicket, Sanjit Biswas, Glenn Judd, and Robert Morris. Link-level measurements from an 802.11b mesh network. In *SIGCOMM '04: Proc. of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, 2004.

[31] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proc. of VLDB'04*, 2004.

[32] Aditya Akella, Glenn Judd, Srinivasan Seshan, and Peter Steenkiste. Self-management in chaotic wireless deployments. In *MobiCom '05*, 2005.

[33] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: a distributed, scalable platform for data mining. In *DMSSP '06: Proc. of the 4th international workshop on Data mining standards, services and platforms*, 2006.

[34] Joel Apisdorf, K. Claffy, Kevin Thompson, and Rick Wilder. Oc3mon: Flexible, affordable, high performance statistics collection. In *LISA '96: Proc. of the 10th USENIX conference on System administration*, 1996.

[35] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD '00: Proc. of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[36] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2002.

[37] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13:370–383, 2004.

[38] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI '99: Proc. of the 3rd symposium on Operating Systems Design and Implementation*, 1999.

[39] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proc. of the 19th ACM symposium on Operating systems principles*, 2003.

[40] Michael K. Bergman. The deep web: Surfacing hidden value. *The Journal of Electronic Publishing*, 7(1), 2001.

[41] Pravin Bhagwat, Bhaskaran Raman, and Dheeraj Sanghi. Turning 802.11 inside-out. *SIGCOMM Comput. Commun. Rev.*, 34:33–38, January 2004.

[42] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *Mobicom '02: Proc. of the 11th annual international conference on Mobile computing and networking*, 2005.

[43] Donna Bogatin. Yahoo searches more sophisticated and specific. `http://blogs.zdnet.com/micro-markets/index.php?p=27`, may 2006.

[44] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: the insecurity of 802.11. In *Mobicom '02: Proc. of the 7th annual international conference on Mobile computing and networking*, 2001.

[45] Vladimir Brik, Shravan Rayanchu, Sharad Saha, Sayandeep Sen, Vivek Shrivastava, and Suman Banerjee. A measurement study of a commercial-grade urban wifi mesh. In *IMC '08: Proc. of the 8th ACM SIGCOMM conference on Internet measurement*, 2008.

[46] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7: Proc. of the seventh international conference on World Wide Web 7*, 1998.

[47] John Burgess, Brian Gallagher, David Jensen, and Brian Neil Levine. Maxprop: Routing for vehicle-based disruption-tolerant networks. In *IEEE INFOCOM*, 2006.

[48] Vladimir Bychkovsky, Bret Hull, Allen K. Miu, Hari Balakrishnan, and Samuel Madden. A Measurement Study of Vehicular Internet Access Using In Situ Wi-Fi Networks. In *12th ACM MOBICOM Conf.*, 2006.

[49] L. F. Cabera, M. B. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *Workshop on Hot Topics in Operating Systems*, 2001.

[50] Joseph Camp, Joshua Robinson, Christopher Steger, and Edward Knightly. Measurement driven deployment of a two-tier urban mesh access network. In *MobiSys '06: Proc. of the 4th international conference on Mobile systems, applications and services*, 2006.

[51] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02: Proc. of the 28th international conference on Very Large Data Bases*, 2002.

[52] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[53] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[54] Uğur Çentintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proc. of CIDR*, 2005.

[55] Ranveer Chandra, Ratul Mahajan, Venkat Padmanabhan, and Ming Zhang. CRAWDAD data set microsoft/osdi2006 (v. 2007-05-23). Downloaded from `http://crawdad.cs.dartmouth.edu/microsoft/osdi2006`, 2007.

[56] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR '03*, 2003.

[57] Bor-Rong Chen, Geoffrey Peterson, Geoff Mainland, and Matt Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *DCOSS '08*, 2008.

[58] Yu-Chung Cheng. CRAWDAD data set ucsd/cse (v. 2008-08-25). Downloaded from `http://crawdad.cs.dartmouth.edu/ucsd/cse`, August 2008.

[59] Yu-Chung Cheng, Mikhail Afanasyev, Patrick Verkaik, Péter Benkö, Jennifer Chiang, Alex C. Snoeren, Stefan Savage, and Geoffrey M. Voelker. Automating cross-layer diagnosis of enterprise wireless networks. In *SIGCOMM '07: Proc. of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007.

[60] Yu-Chung Cheng, John Bellardo, Péter Benkö, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *SIGCOMM '06: Proc. of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006.

[61] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proc. of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[62] Matt Cutts. More webmaster console goodness. Matt Cutts: Gadgets, Google, and SEO (blog), `http://www.mattcutts.com/blog/more-webmaster-console-goodness/`, October 2006.

[63] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM '04: Proc. of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, 2004.

[64] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Proc. of the 6th symposium on Operating Systems Design and Implementation*, 2004.

[65] Udayan Deshpande, Tristan Henderson, and David Kotz. Channel sampling strategies for monitoring wireless networks. In *WiNMee '06*, 2006.

[66] Khaled Elmeleegy, Alan L. Cox, and T. S. Eugene Ng. Etherfuse: an ethernet watchdog. In *SIGCOMM '07: Proc. of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007.

[67] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better netflow. In *SIGCOMM '04: Proc. of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, 2004.

[68] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35, June 2003.

[69] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, and K. A. Ross. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proc. of the 2001 ACM SIGMOD international conference on Management of data*, 2001.

[70] Daniel Fisher. Company of the year: Nasdaq - forbes.com. `http://www.forbes.com/forbes/2009/0112/056.html`.

[71] Armando Fox and Eric A. Brewer. Harvest, yield and scalable tolerant systems. In *Proc. of the 1999 Workshop on Hot Topics in Operating Systems*, 1999.

[72] Chuck Fraleigh, Sue Moon, Bryan Lyles, Chase Cotton, Mujahid Khan, Deb Moll, Rob Rockell, Ted Seely, and Christophe Diot. Packet-level traffic measurements from the sprint ip backbone. *IEEE Network*, 17, 2003.

[73] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *NSDI '04: Proc. of the 1st USENIX symposium on Networked Systems Design and Implementation*, 2004.

[74] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazieres. Oasis: Anycast for any service. In *NSDI '06: Proc. of the 3rd USENIX symposium on Networked Systems Design and Implementation*, 2006.

[75] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proc. of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[76] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-weight process groups in the isis system. *Distributed Systems Engineering*, 1(1):29–36, September 1993.

[77] Google. Mountain view coverage map. `http://wifi.google.com/city/mv/apmap.html`.

[78] S. Grin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7*, 1998.

[79] K. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Second Usenix/ACM SIGCOMM Internet Measurement Workshop*, 2002.

[80] Finn Michael Halvorsen, Olav Haugen, Martin Eian, and Stig F. Mjlsnes. An improved attack on tkip. In *NordSec*, 2009.

[81] Dongsu Han, Aditiya Agarwala, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, and Srinivasan Seshan. Mark-and-sweep: getting the "inside" scoop on neighborhood networks. In *IMC '08: Proc. of the 8th ACM SIGCOMM conference on Internet measurement*, 2008.

[82] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep web. *Communications of the ACM*, 50, May 2007.

[83] IEEE. Part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE 802.11-2007*, 2007.

[84] Amit P. Jardosh, Krishna N. Ramachandran, Kevin C. Almeroth, and Elizabeth M. Belding-Royer. Understanding link-layer behavior in highly congested ieee 802.11b wireless networks. In *E-WIND '05*, 2005.

[85] Dongmyoung Kim, Hua Cai, Minsoo Na, and Sunghyun Choi. Performance measurement over mobile wimax/ieee 802.16e network. In *WOWMOM '08*, 2008.

[86] Eddie Kohler. *The Click modular router*. PhD thesis, Massachusetts Institute of Technology, 2000.

[87] David Kotz and Kobby Essien. Analysis of a campus-wide wireless network. In *Mobicom '02: Proc. of the 8th annual international conference on Mobile computing and networking*, 2002.

[88] David Kotz, Calvin Newport, and Chip Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dept. of Computer Science, Dartmouth College, July 2003.

[89] Vibhore Kumar, Henrique Andrade, Buğra Gedik, and Kun-Lung Wu. Deduce: at the intersection of mapreduce and stream processing. In *EDBT '10: Proc. of the 13th International Conference on Extending Database Technology*, 2010.

[90] Vasileios Lampos, Tijl De Bie, and Nello Cristianini. Flu detector - tracking epidemics on twitter. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2010)*, pages 599–602, 2010.

[91] B. Latré, B. Braem, I. Moerman, C. Blondia, E. Reusens, W. Joseph, and P. Demeester. A low-delay protocol for multihop wireless body area networks. In *MOBIQUITOUS '07: Proc. of the 2007 Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, 2007.

[92] S.-J Lee, P. Sharma, S. Banerjee, S. Basu, and R. Fonseca. Measuring bandwidth between planetlab nodes. In *PAM '05: Proc. of Passive and Active Measurement Workshop*, 2005.

[93] Zhichun Li, Anup Goyal, Yan Chen, and Vern Paxson. Automating analysis of large-scale botnet probing events. In *ASIACCS '09*, 2009.

[94] H. Liu, V. Ramasubramanian, and E.G. Sirer. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *IMC '05: Proc. of the 5th ACM SIGCOMM conference on Internet measurement*, 2005.

[95] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI '02: Proc. of the 5th symposium on Operating Systems Design and Implementation*, 2002.

[96] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), 2005.

[97] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep web crawl. *Proc. of the VLDB Endowment*, 1, 2008.

[98] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Analyzing the mac-level behavior of wireless networks in the wild. In *SIGCOMM '06: Proc. of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006.

[99] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proc. of the USENIX Winter 1993 Conference Proceedings*, 1993.

[100] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD '09: Proc. of the 35th SIGMOD international conference on Management of data*, 2009.

[101] Robert B. Miller. Response time in man-computer conversational transactions. In *Proc. of the December 9-11, 1968, fall joint computer conference, part I*, 1968.

[102] Andrew Moore, James Hall, Christian Kreibich, Euan Harris, and Ian Pratt. Architecture of a network monitor. In *PAM '03: Proc. of Passive and Active Measurement Workshop*, 2003.

[103] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2), 2006.

[104] Marti Motoyama, Brendan Meeder, Kirill Levchenko, Geoffrey M. Voelker, and Stefan Savage. Measuring online service availability using twitter. In *WOSN'10: Proc. of the 3rd conference on Online social networks*, 2010.

[105] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR '03*, 2003.

[106] Rohan Murty, Geoffrey Mainland, Ian Rose, Atanu Roy Chowdhury, Abhimanyu Gosain, Josh Bers, and Matt Welsh. Citysense: An urban-scale wireless sensor network and testbed. In *2008 IEEE International Conference on Technologies for Homeland Security*, 2008.

[107] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. Wishbone: Profile-based partitioning for sensornet applications. In *NSDI '09: Proc. of the 6th USENIX symposium on Networked Systems Design and Implementation*, 2009.

[108] Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett. Design and evaluation of a compiler for embedded stream programs. In *LCTES '08: Proc. of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, 2008.

[109] Vivek S. Pai, Limin Wang, KyoungSoo Park, Ruoming Pang, and Larry Peterson. The dark side of the web: an open proxy's view. *SIGCOMM Comput. Commun. Rev.*, 34, January 2004.

[110] Jeffrey Pang, Ben Greenstein, Ramakrishna Gummadi, Srinivasan Seshan, and David Wetherall. 802.11 user fingerprinting. In *Mobicom '02: Proc. of the 13th annual international conference on Mobile computing and networking*, 2007.

[111] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The devil and packet trace anonymization. *SIGCOMM Comput. Commun. Rev.*, 36(1), 2006.

[112] Simon Patarin and Mesaac Makpangou. Pandora: a flexible network monitoring platform. In *Proc. of the USENIX 2000 Annual Technical Conference*, 2000.

[113] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), 1999.

[114] Carolyn Penner. Twitter blog: #numbers. `http://blog.twitter.com/2011/03/numbers.html`.

[115] David Perdew. How to track trends with twitter tools. `http://goarticles.com/article/How-To-Track-Trends-With-Twitter-Tools/4141128/`.

[116] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish-subscribe systems. In *Proc. of the 7th International Conference on Cooperative Information Systems (CoopIS)*, 2000.

[117] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *OSDI '06: Proc. of the 7th symposium on Operating Systems Design and Implementation*, 2006.

[118] P. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. In *1st Workshop on Distributed Event-Based Systems*, 2002.

[119] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulosand M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[120] R. Raghavendra, P. A. K. Acharya, E. M. Belding, and K. C. Almeroth. Antler: A multi-tiered approach to automated wireless network management. In *IEEE Conference on Computer Communications Workshops, 2008. INFOCOM*, 2008.

[121] V. Ramasubramanian, R. Peterson, and G. Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *NSDI '06: Proc. of the 3rd USENIX symposium on Networked Systems Design and Implementation*, 2006.

[122] Theodore S. Rappaport. *Wireless Communications: Principles and Practice (2nd Edition)*. Prentice Hall PTR, 2 edition, 2002.

[123] Matthias Ringwald and Kay Rmer. Snif: A comprehensive tool for passive inspection of sensor networks. *6. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze". Aachen, Germany*, 2007.

[124] Martin Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99*, 1999.

[125] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *WWW '10: Proc. of the 19th international conference on World wide web*, 2010.

[126] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. of Queensland AUUG Summer Technical Conference*, 1997.

[127] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proc. of AUUG2K*, 2000.

[128] Yong Sheng, Guanling Chen, Hongda Yin, Keren Tan, Udayan Deshpande, Bennet Vance, David Kotz, Andrew Campbell, Chris McDonald, Tristan Henderson, and Joshua Wright. MAP: A scalable monitoring system for dependable 802.11 wireless networks. *IEEE Wireless Communications*, 15(5):10–18, 2008.

[129] Maggie Shiels. Google admits wi-fi data collection blunder. `http://news.bbc.co.uk/2/hi/technology/8684110.stm`.

[130] David Sifry. State of the blogosphere, august 2006. `http://www.sifry.com/alerts/archives/000436.html`.

[131] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04: Proc. of the 2nd international conference on Embedded networked sensor systems*, 2004.

[132] Brett Stone-Gross, Christo Wilson, Kevin C. Almeroth, Elizabeth M. Belding, Heather Zheng, and Konstantina Papagiannaki. Malware in ieee 802.11 wireless networks. In *PAM '08: Proc. of Passive and Active Measurement Workshop*, 2008.

[133] Michael Stonebraker and Uğur Çentintemel. "one size fits all": An idea whose time has come and gone. In *ICDE '05: Proc. of the 21st International Conference on Data Engineering*, 2005.

[134] R. Strom, G. Banavar, T. Chandra, M. Kaplan, and K. Miller. Gryphon: An information flow based approach to message brokering. In *Proc. of the International Symposium on Software Reliability Engineering*, 1998.

[135] Mark Sullivan and Andrew Heybey. Tribeca: a system for managing large databases of network traffic. In *Proc. of the USENIX 1998 Annual Technical Conference*, 1998.

[136] Sonesh Surana, Rabin Patra, Sergiu Nedevschi, Manuel Ramos, Lakshminarayanan Subramanian, Yahel Ben-David, and Eric Brewer. Beyond pilots: Keeping rural wireless networks alive. In *NSDI '08: Proc. of the 5th USENIX symposium on Networked Systems Design and Implementation*, 2008.

[137] Diane Tang and Mary Baker. Analysis of a local-area wireless network. In *Mobicom '02: Proc. of the 6th annual international conference on Mobile computing and networking*, 2000.

[138] Erik Tews and Martin Beck. Practical attacks against wep and wpa. In *WiSec '09*, 2009.

[139] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP '05: Proc. of the 20th ACM symposium on Operating systems principles*, 2005.

[140] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI '02: Proc. of the 5th symposium on Operating Systems Design and Implementation*, 2002.

[141] Mike P. Wittie, Brett Stone-Gross, Kevin C. Almeroth, and Elizabeth M. Belding. Mist: Cellular data network measurement for mobile applications. In *BROADNETS*, 2007.

[142] B. Wong, A. Slivkins, and G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *SIGCOMM '05: Proc. of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 2005.

[143] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37, July 1998.

[144] Jihwang Yeo, Moustafa Youssef, and Ashok Agrawala. A framework for wireless lan monitoring and its applications. In *WiSe '04*, 2004.

[145] Jihwang Yeo, Moustafa Youssef, Tristan Henderson, and Ashok Agrawala. An accurate technique for measuring the wireless side of wireless networks. In *WiTMeMo '05*, 2005.

[146] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), January 2006.

[147] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Understanding web query interfaces: best-effort parsing with hidden syntax. In *SIGMOD '04: Proc. of the 2004 ACM SIGMOD international conference on Management of data*, 2004.