Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Saptarshi Guha

Entitled
Computing Environment for the Statistical Analysis of Large and Complex Data

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

William S. Cleveland Chair	Bowei Xi
Hao Zhang	Jun Xie

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): William S. Cleveland

Approved by: Jun Xie 08/18/2010
Head of the Graduate Program Date

Graduate School Form 20
(Revised 1/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
Computing Environment for the Statistical Analysis of Large and Complex Data

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Teaching, Research, and Outreach Policy on Research Misconduct (VIII.3.1)*, October 1, 2008.*

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Saptarshi Guha
Printed Name and Signature of Candidate

08/23/2010
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/viii_3_1.html

COMPUTING ENVIRONMENT FOR THE STATISTICAL ANALYSIS OF LARGE AND COMPLEX DATA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Saptarshi Guha

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2010

Purdue University

West Lafayette, Indiana

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

I owe so much. To all of you.

ACKNOWLEDGMENTS

I would not have embarked on this journey had it not been for the encouragement and support received from my friends at Natasha Penta: Rajendran, Srini, Chandan, Timsy, Supratiki, Ashwini, Ashwin, Priya and Ranga.

My friends from the Indian Statistical Institute: Ritwik, Moumita, Nandita, Pritha, Shiv, Pranab and Hemant have been an pillar of support, a refuge of sorts, those rambling email threads, my rope.

My stay in West Lafayette was made memorable with the varied company of Shilpi, Guha Beth,Gautam, Karen, Shraddha, there are so many. And special mention to Namrata, my dear friend and neighbor for six years. The lovely meals, the always edifying conversations, the generosity of spirit made 413 (and a 1/2) a home to come home to.

I must thank Jin, Paul and Ryan for being wonderful and cheerful office mates. It was a pleasure to open the door to see them in and a presence I shall certainly miss.

I am deeply grateful to Revolution Analytics for supporting and financing a part of my research.

I must especially thank the Department of Statistics for being such a heartwarming and diverse group of people. A sincere thanks to Darlene,Becca and Mary, without them I would have been lost in the administrative maze of Purdue University.

Doug and My, this thesis would be blank without you. Every department should have an IT department manned by persons like Doug and My. The best.

Thanks go to my brother Rajarshi, sister-in-law Debarchana and my parents for a home to return to, a home to talk to and a home to remember.

Finally, I cannot state enough my thanks to my advisor, Prof. W.S. Cleveland, the best guide a student could have. Prof. Cleveland shaped the course of my research, introduced me to visualization and the need for large scale computing. I will forever

be in his debt for his support, rigor, advice, demands on perfection and unending guidance. And of course, his elements of style.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Guha, Saptarshi Ph.D., Purdue University, December 2010. Computing Environment for the Statistical Analysis of Large and Complex Data . Major Professor: William S. Cleveland.

Analyzing large data has become very feasible with recent advances in modern technology. Data acquisition has become very fine grained and is available across many scenarios from home power consumption, to network data. Such data sets which can collect data at the level of seconds, quickly become massive. The storage of such data is now possible because of the rapid fall in the hardware prices. It is has become the statisticians' challenge to analyze such massive data sets with the same level of comprehensive detail as is possible for much smaller analyses. Any detailed analysis of such data sets, necessarily creates many subsets and many more data structures. We need approaches to store and compute with them by taking advantage of modern technology such as distributed compute clusters. This forms the backdrop to the three chapters of the thesis: Visualization database for large data sets, a keystroke detection algorithm derived from analyzing hundreds of gigabytes of network data and a merger of the R and Hadoop programming environments that enables topics covered in the first two chapters.

Comprehensive visualization that preserves the information in the data requires a visualization database (VDB): many displays, some with many pages, and with one or more panels per page. A single display results from partitioning the data into subsets, and using the same method to display each subset in a sample of subsets, typically one per panel. The time of the analyst to produce a display is not increased by choosing a large subset over a small one, and every page does not necessarily need to be studied. Some displays might be studied in their entirety; for others, studying only a small fraction of the pages might suffice. On-the-fly computation

without storage does not generally succeed because computation time is too large. The sizes and numbers of displays of VDBs require a rethinking all areas involved in data visualization, including the following: Methods of display design that enhance pattern perception to enable rapid page scanning; Automation algorithms for basic display elements such as the aspect ratio, scales across panels, line types and widths, and symbol types and sizes; Methods for subset sampling; Viewers designed for multi-panel, multi-page displays that scale across different amounts of physical screen area.

One such example of a detailed analysis of hundreds of gigabytes of data is the keystroke detection algorithm. This is a streaming algorithm detects SSH client keystroke packets in any TCP connection. Input data are timestamps and TCP-IP header fields of packets in both directions, measured at a monitor on the path between the hosts. The algorithm uses the packet dynamics just preceding and following a client packet with data to classify the packet as a keystroke or non-keystroke. The dynamics are described by classification variables derived from the arrival timestamps and the packet data sizes, sequence numbers, acknowledgement numbers, and flags. The algorithm succeeds because a keystroke creates an identifiable dynamical pattern. One application is identification of any TCP connection as an SSH interactive session, allowing detection of backdoor SSH servers. More generally, the algorithm demonstrates the potential for the use of detailed packet dynamics to classify connections.

The above analysis of network data would be extremely unwieldy (if not impossible) were it not using distributed file systems and computing frameworks. RHIPE is a software system that integrates the R open source project for statistical computing and visualization with the Apache Hadoop Distributed File System (HDFS) and the Apache MapReduce software framework for the distributed processing of massive data sets across a cluster. Distributed programming with massive data sets is by nature complex - issues such as data storage, scheduling and fault tolerance must all be handled. RHIPE uses its tight integration with the HDFS to store data across the cluster. Similarly, it takes advantage of MapReduce to efficiently utilize all the

processing cores of the cluster. Vital, but difficult to implement details, such as task scheduling, bandwidth optimization and recovery from failing computers are handled by Hadoop MapReduce. Most importantly, RHIPE hides these details from the R user, by providing an idiomatic R interface to Hadoop and HDFS cluster. The design of RHIPE strives for a balance between conceptual simplicity, ease of use and flexibility. Algorithms, designed for the MapReduce programming model, can be implemented using the R language, executed from R's REPL (read-eval-print-loop) and the results are directly returned to the user.

1. VISUALIZATION DATABASES

1.1 Introduction

Large, complex datasets have some of the following properties, often all: a large number of records; many variables; complex data structures not readily put into a tabular form of cases by variables; intricate patterns and dependencies in the data that require complex models and methods of analysis. Our goal, despite the complexity, should be comprehensive study that does not lose important information contained in the data.

Nothing serves comprehensive analysis better than data visualization, the only practical way to absorb a large amount of information. This principle has been accepted and practiced for decades. Consider this regression model: the mean of a numeric response is assumed linear in three numeric explanatory variables, and the errors are assumed i.i.d. $N(0,\sigma^2)$. Suppose there are 100 observations of the response and each of the explanatory variables, 400 numeric values altogether. To check the linearity and normality of the model, it is common practice to make, at the very least, a collection of standard displays (Cleveland (1993); Cook and Weisberg (1999)): scatterplot matrix of the four variables (1200 plotted points); three partial residual plots, one for each explanatory variable (300 points); three conditioning plots of the response against and explanatory variable conditional on the other two, one for each explanatory variable (300 points); residuals against fitted values (100 points); absolute residuals against fitted values (100 points); normal quantile plot (100 points); three conditioning plots with residuals in place of the response, one for each explanatory variable (300 points). The number of plotted points is 2400, and each point encodes two numeric values, so 4800 values are displayed. This means the ratio of graphed numeric values is 12 times the number of numeric values in the data.

In an effort to achieve comprehensive analysis of a large dataset with billions or trillions of observations we obviously cannot achieve a factor of 12 in data displays. But we can make a large number of displays to pursue comprehensive analysis, many with a large number of pages, each of which can have many panels. The total number of pages might be measured in thousands or tens of thousands. We call the displays a *visualization database*, or *VDB*. The name applies equally well to displays of the above regression example, so the concept of a visualization database is not new.

The difference between the large and the small datasets is that we must partition the data into subsets, sample the subsets, and apply a visualization method to each sample. We subject each sample subset to the same comprehensive analysis as the small dataset. Of course, the sampling frame must be chosen in a way that characterizes the data. Backing us up are numeric methods that can often be run on all subsets, or on a sample much larger than that for making displays, that help in making good sampling frame choices.

We are very optimistic because we have had substantial success for many data analysis and methodological projects. invoking just a few basic concepts for VDBs, a new distributed computing environment described later, new methods for display design and sampling, and with off-the-shelf initial solutions many VDB components, VDB performance can increase by a rethinking of all areas involved in data visualization, including the following: Methods of display design that enhance pattern perception to enable rapid page scanning; Automation algorithms for basic display elements such as the aspect ratio, scales across panels, line types and widths, and symbol types and sizes; Methods for subset sampling; Viewers designed for multi-panel, multi-page displays that scale across different amounts of physical screen area.

This article discusses the basic concepts for a VDB, its hardware and software components, and the development of methods and algorithms that can improve the performance of VDBs. Readers are encouraged to look at a Web site prepared in coordination with this article that houses a number of VDB managers, some evolving because they involve current projects (vdb). We will make reference in this article to

three of the VDBs — named *surveillance*, *joke*, and *connection* — that involve the analysis of three data sets.

The surveillance data are the daily counts of chief complaints from emergency departments (EDs) of the Indiana Public Health Emergency Surveillance System (PHESS) (Grannis et al. (2006)). The complaints are divided into eight classifications; one is respiratory. Data for the first EDs go back to November 2004, and new EDs have come online continually since then. There are now 76 EDs in the system. Respiratory complaints for the 30 EDs with the most data, are analyzed in (Hafen et al. (2009, to appear)), and surveillance is the VDB for this analysis.

The Jester project (jes; Goldberg et al. (2001)) has collected ratings on a scale of -10 to 10 of 100 jokes from 73,421 raters from April 1999 to May 2003. Our joke dataset has the 14,116 raters who evaluated all jokes. Visualization methods are revealing the properties of the data as a guide to building a statistical model that will allow prediction of the ratings of an individual from a set of 10 gauge jokes.

Much Internet communication consists of connections between two hosts who send packets back and forth, each 1500 bytes or less. The TCP protocol manages the connections for many applications such as web page delivery (http), email (smtp), and encrypted remote login (ssh). Our connection dataset is packet traces for TCP collected on a subnet of the Purdue Statistics Department and organized by connection. The traffic monitor sees all traffic between two VLANS that make up the subnet, and between the subnet and the outside. The data for each packet are the arrival timestamp and certain information from the TCP and IP headers: source-destination IP addresses (anonymized), ports, and sequence numbers; size of payload in packet; values of size flags — SYN, FIN, PSH, RST, and ACK; and ACKed sequence number. Trace collection has been carried out on four separate days for a total of 96 hr. There are 749,128 connections; the binary version of the raw data is 146 gigabytes; this is converted to a distributed linux flatfile database of 190 gigabytes with 1.49 billion rows, where each row corresponds to a packet. The packets are organized by connec-

tion because the research topic is an approach to network security based on analysis of connection properties as a function of time and the logical network topology.

1.2 Partitions, Analysts, Computing

1.2.1 Partitions

An important strategy of comprehensive analysis for a large complex dataset is to partition it into small subsets in one or more ways, and apply numeric methods and visualization methods to each of a sample of subsets. Section 1.1 discussed the use of a large number of plotted points per observation that is commonly carried out for small datasets. Achieving comprehensive analysis of a large dataset requires preserving this for the subsets, analyzing each in detail. The sampling method can vary by method; it is common to have numeric methods applied to more subsets than visualization methods. In some cases, sampling can be exhaustive: all subsets. Two non-exhaustive sampling methods, representative and importance, are discussed in Section 1.3.

Partitioning can be carried out in many different ways. Often, we start with a *core partition* that arises naturally from the structure of the raw data. This is a soft concept but useful nevertheless. The subsets of the core then are often further partitioned by variables other than those that defined the core.

1.2.2 The Data Analyst

The time for a data analyst to create a display for a single subset by writing commands in the environment used for computing with the data can vary from very small to large. But once the the commands are written, a reasonable programming environment results in a negligible additional command-time difference between small and large samples. So in this regard, a large visualization database is not significantly more costly than a small one.

The data analyst spends more time looking at a large sample than at a small sample. To understand the data as a whole, there needs to be a requisite number of subsets, and this can be quite large. But studying displays and thinking about the data, and not the programming language, is time well spent. It does not have to be an undue burden. The implication of a very large number of displayed subsets is that the display resulting from an application of a visualization method will take up a very large amount of virtual screen space, far larger than the available physical screen space, and so must be viewed sequentially. Some displays might be scanned entirely; for others, a small fraction of the pages might suffice. In work described briefly in Section 1.3, we are developing methods of display design that invoke principles of visual perception to enhance pattern perception and enable rapid page scanning. We are also developing viewers for the sequential task that can reduce the analyst time substantially.

1.2.3 Computing

Partitioning, because it leads to embarrassingly parallel computation, can benefit immensely from distributed computing environments such as RHIPE (RHIPE (b)), a recent merging of the R interactive environment for data analysis (R Development Core Team (2005)) and the Hadoop distributed file system and compute engine (Hadoop). This benefits all methods used in the analysis project, both numeric and visual.

1.3 Sampling, Trellis, Three VDBs

In representative sampling, survey variables are defined that measure properties of the subsets. A subset sampling frame is chosen to encompass the multidimensional space of the survey variables and to spread out the points in the space in a uniform way by some definition. In importance sampling, samples have values of the variables that lie in a specific region of importance of the multidimensional space of the variables.

Partitioning for data visualization has been going on for small data sets for some time and is the stimulus for trellis display, a framework for visualization that provides conditioning plots: displays of a set of variables conditional on the variables of others (Becker et al. (1996)). The trellis framework works well for the partitioning of large complex datasets, creating display documents with pages, and with panels on each page in a rectangular array. Often, each panel shows a core subset, but in some cases a single core subset can spread out across many panels when additional variables partition the core further.

1.3.1 Joke Dataset

Two core partitions have been used — by joke and by rater — resulting in 100 and 73,421 subsets respectively. For the by-joke partition, the sampling for all numeric and visualization methods is exhaustive. For the by-rater partition, sampling for numeric methods is exhaustive. For the by-rater partition, sampling for visualization methods is representative. All models explored for the data have a rater location effect because there are hard graders and easy graders; the estimates of the location effect are one of our survey variables. One diagnostic display of any model is a normal quantile plot of residuals for each rater. Our representative sampling frame for this method is 4000 raters selected so that the rater-effect estimates are as close to uniformly spaced as possible from the minimum to the maximum estimate.

Figure 1.1 is the first page of a 100-page, 4000-panel trellis display of the representative sample for the normal quantile method. Each panel is a normal quantile plot of the residuals for one rater and a model fitted to a logistic transformation of the ratings rescaled to the interval $[0, 1]$. The line on the plot goes through the two quartile points of the display. As we go left to right, bottom to top, and through the pages of the display, there is an increase in the estimates of the rater location effect, shown in the lower right of each panel of the display. The strip labels, which could have shown these values, have been eliminated to save space. The model in this case

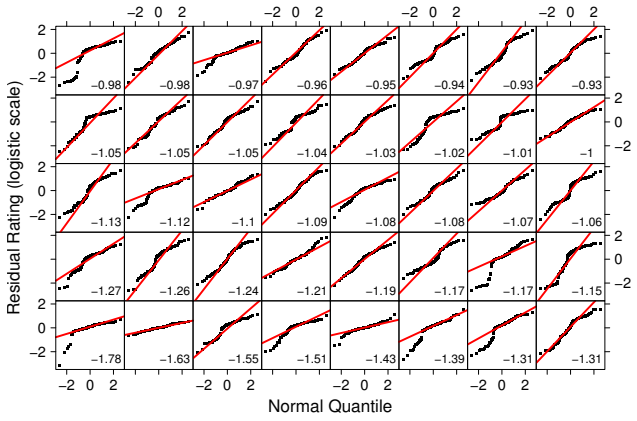


Figure 1.1.: First page of 100-page trellis display of normal quantile plots by rater. There are 8 columns and 4 rows of panels per page.

is very simple, additive main effects of jokes and raters, and error terms with identical normal distributions with mean zero.

The resultant-vector banking aspect ratio (see Section 1.5) of a quantile plot is 1, a quantile plot can fit in a relatively small space, and the aspect ratios of our screens are 0.625, so we make 4000 plots: 100 pages, each with 8 columns and 5 rows of panels, and each page nearly fills our smallest physical screen since $5/8 = 0.625$ (see Section 1.4). The first page of the display is shown in Figure 1.1.

For this plot, the 100 pages can be visually scanned in a few minutes because our visual systems can effortlessly detect departures of the plotted points on a panel from the line. Across the plot, as on the first page, the line follows the pattern of the points which means the normal is a good approximation of errors.

1.3.2 Surveillance Dataset

For the surveillance data, the core partitioning is by emergency department (ED). Because the dataset size is moderate, the partition sampling is exhaustive for both numeric and visualization methods. Developing models and testing results depended

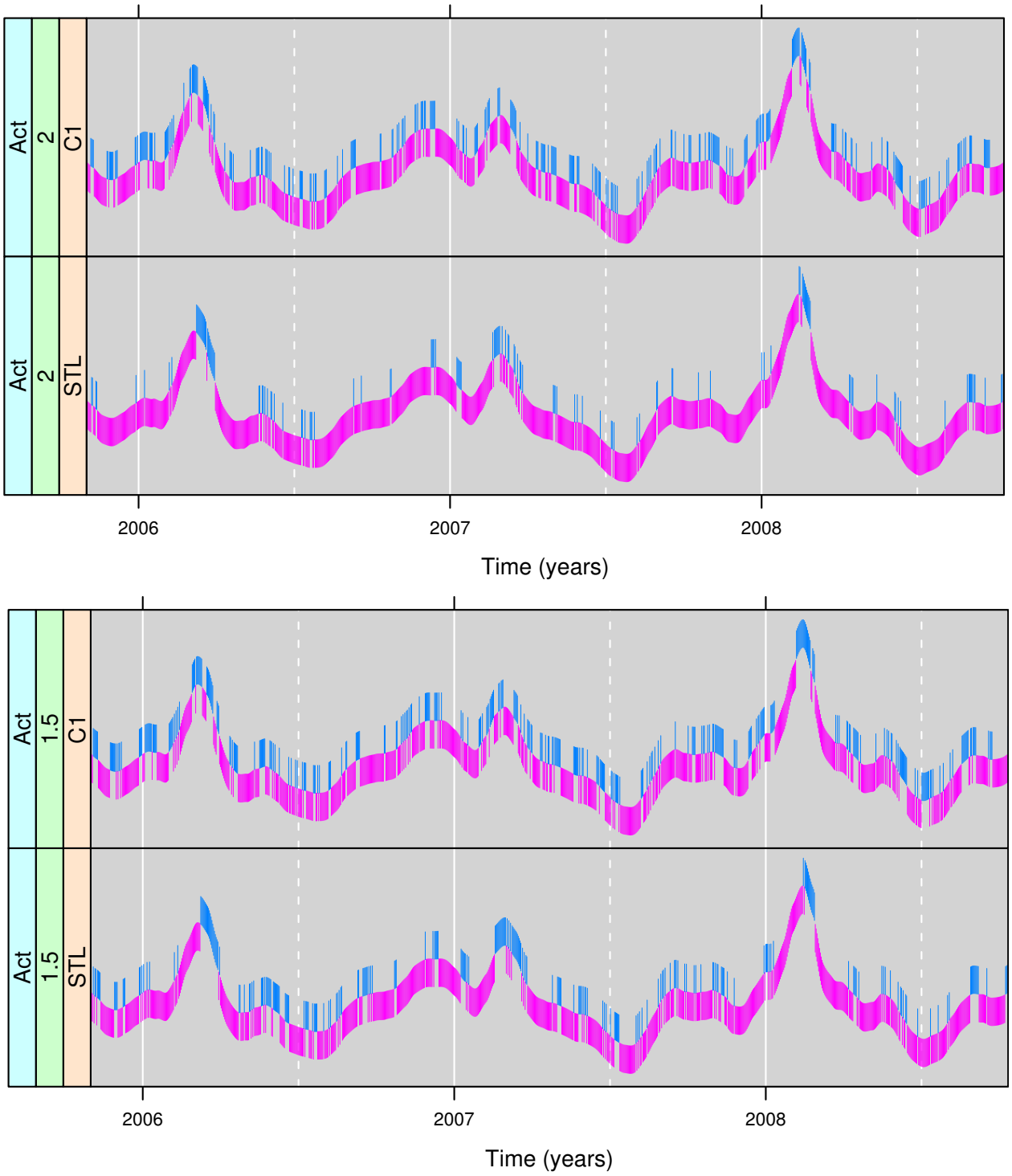


Figure 1.3.: Top two panels of six of Page 2 of a 90-page, 540-panel trellis display.

heavily on the visualization methods that populated the VDB, many applied to each ED separately, but also some showing results on different panels across EDs.

Figure 1.2 shows the top two panels of the Page 1 of a 90-page, 540-panel trellis display; in the full display there is 1 column and 6 rows on each page. Figure 1.3 shows the top two panels of Page2. In the project, new methods for modeling respiratory counts for each ED are based on STL, the nonparametric seasonal-trend numeric decomposition procedure. Square root counts of each are decomposed into inter-annual, yearly-seasonal, day-of-the-week, and random-error components. Using this decomposition method, a new synoptic-scale (days to weeks) numeric outbreak detection method was developed. The STL detection method, and a version using a lesser amount of data, STL(90), were tested along with four existing and widely know methods: GLM, C1, C2, C3. Each method was tested on each ED count series. An outbreak occurrence was added on a particulary day, the outbreak methods applied, and detect or non-detect within 14 days recorded. This was done for each ED on each day separtately starting with the 366th day of data. There are 3 different outbreak magnitudes (2, 1.5, and 1), and 30 EDs (with anonymized names such as Act). With 6 detection methods, 3 outbreak magnitudes, and 30 EDs, there are $540 = 6 \times 3 \times 30$ outbreak test sequences across time. Each panel shows one test sequence for one method, one magnitude and one ED.

The diagnostic display shown partially in Figures 1.2 and 1.3 reveals the effect of the seasonal component on detection performance. The strip labels to the left of each panel show the outbreak method, magnitude and ED. Outbreak method changes the fastest, each page has 6 panels showing the 6 methods for one combination of magnitude and ED. Magnitude changes next fastest; on page one at the top of the figure, the value is 2; on page two at the bottom of the figure, it is 1.5, and on page three (not shown), it is 1. ED changes the slowest, so pages one to three are Act, pages four to six are the next ED, and so forth.

The curve formed by the bottoms of the blue lines and the tops of the red lines on each panel is the all-data STL seasonal component; this component is used because

C1, C2, and C3 do not involve a decomposition, and of the three remaining methods, which use a decomposition, the all-data STL component does the best job of tracking the yearly seasonal pattern. Each vertical line emanating from the curve shows the results of starting an outbreak on the day at which the line is drawn and determining if detection occurred on or before the 14th day of the outbreak; red is detected, and blue is not detected.

Much can be seen from this display about detection performance and how it changes with the behavior of the yearly seasonal component. All methods, as expected, detect more frequently for magnitude 2 than 1.5. STL is the best performer. STL failure to detect occurs most frequently during periods of decline in the seasonal component because STL seasonal on the last day of data does not keep up with the decline which reduces the apparent ramping up of the outbreak.

1.3.3 Connection Dataset

The core partitioning is by connection so there are 749,128 subsets. Analysis associated with the connection VDB focuses on a numeric statistical method: a rules-based statistical algorithm (RBSA) that classifies each packet of a connection as a client keystroke from an ssh connection or not. The algorithm uses the timestamps, packet payload sizes, and flags in both directions to carry out the classification. The goal for network security is to classify the whole connection as interactive ssh or not. The algorithm is very accurate at the packet level but does have some misclassifications; we found that classifying the connection as interactive ssh if there are 5 or more keystrokes, has very few damaging misclassifications.

Our sampling for the RBSA numeric method is exhaustive. The ssh well-known port is 22, but the algorithm is applied to all connections since backdoor login services can be created by intruders. Even though the algorithm is computationally intensive, distributed computing with RHIPE makes computation for all subsets feasible. One of our importance sampling frames for data visualization is all connections classified

as interactive login, having a server port that is not port 22, and having one host not on the Purdue campus. One of our visual displays used in this importance sampling is shown in Section 1.6.

1.4 Screen Space, Viewers

To consume large VDBs we want as much physical screen space as possible. We have been experimenting with two 30 inch monitors, each 2560x1600 in resolution. This is equivalent to ten 1024x768 monitors, but very omporantly, it locates the monitors so that the resulting screen space does not greatly exceed the visual field of the analyst and can be viewed with small movements of the head.

Designing the visual displays and a viewer to look at them must consider visual processes. However, on a typical data analysis project, as a practical matter, displays also need to be viewable on devices with much smaller amounts of physical screen space such as laptops. So the viewer and display design must accommodate viewing for varying amounts of screen space. To make the displays scale across screen space, we design them so that each page is viewable on a 15 inch screen; mouse clicks take us through a document with one page at a time, this is, one page in the visual field at a time. For larger amounts of screen space, the viewer allows multiple pages to be displayed, which is very beneficial. If we display a block of 12 pages, 6 on each of the two monitors, then a mouse click takes us to a new block of 12 pages, so the click process is reduced by a factor of 12. Within a block, eye movement over the 12 displays provides even more rapid and effective visual decoding than 12 sequential views on a small screen.

1.5 Element Automation Algorithms

There are many levels of display design. The first and fundamental one is the method: qualitative and quantitative information of a specific type are shown by a display whose overall visual design is of a certain type. Examples are scatterplots

with a smooth curve superimposed, normal quantile plots with a line through the quartile points, and box plots.

For each display method there are many design decisions that must be made additionally about basic elements such as the aspect ratio; values of tick marks; line widths and colors; text sizes and fonts for labels; plotting-symbol sizes and colors; taking the log of a variable or not; when there are two more panels with the same variables displayed, choosing the horizontal or vertical scales across panels to have the same range, to have the same number of units/cm, or to be free ranging; and many more.

A data analyst is completely responsible for the decision about the choice of method, but is very well served by a system that can assist in the decisions about basic elements. Systems today already engage in this to some extent — for example, choosing the values of tick marks — but much more can be done to develop automation algorithms for basic elements and to study their mathematical and statistical properties. It is especially critical for VDBs because computation is sometimes not immediate.

The aspect ratio of a display, the height of a rectangle just enclosing the data divided by the width, has an immense impact on our ability to judge the rate of change of one variable as function of another, usually conveyed by the slopes of line segments that make up a curve. Figure 1.4 shows an example: the STL seasonal component for one ED from the surveillance data. The segments provide information about the component including the local rate of change with time. A change in the aspect ratio changes the physical slopes of the segments which in turn changes our ability to visually decode slopes to judge rate of change. In the 1980s, it was demonstrated that *banking to 45°*, which means choosing the aspect ratio to center the absolute values of the slopes on 1, greatly enhances the judgment of rate of change. If we take this general principle and add to it a specific definition of the meaning of centering on 45°, then the result is a banking automation algorithm for the aspect ratio.

We are studying a new banking algorithm that proceeds conceptually as follows: each segment with a negative slope is replaced with a segment of the same length but with the sign of the slope dropped to make it positive; the new segments, as vectors, are added to form a resultant vector; the segments and their resultant are displayed in a hypothetical display with an aspect ratio that makes the resultant have an orientation of 1. The aspect ratio of the original display is chosen so that the absolute orientations are the same as those of the hypothetical display. This *resultant-vector automation algorithm* was used in Figure 1.4; the aspect ratio is 0.12. Figure 1.5 graphs the seasonal component again but with the aspect ratio equal to 0.04. This greatly reduces our ability to perceive the behavior of the component.

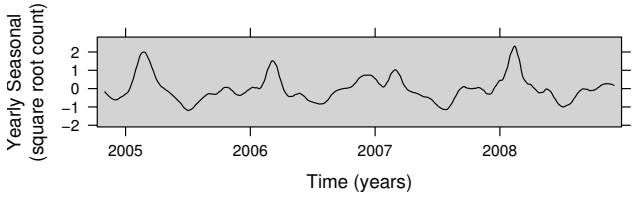


Figure 1.4.: The yearly seasonal component for one ED is displayed with the aspect ratio of 0.12 from the resultant-vector algorithm.

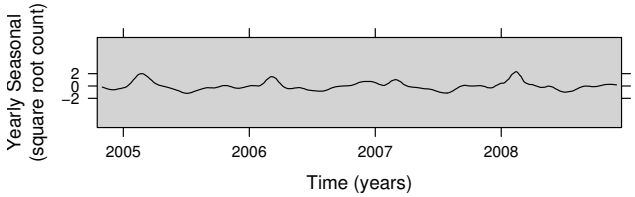


Figure 1.5.: The aspect ratio for the display in Figure 1.4 has been reduced to 0.04.

1.6 Design Methods for Gestalts

Our extraordinarily powerful and flexible human visual system can readily process very large displays by methods of viewing not utilized for the single-page display. One is rapid scanning. It is possible to click at a fast rate through the pages of a document

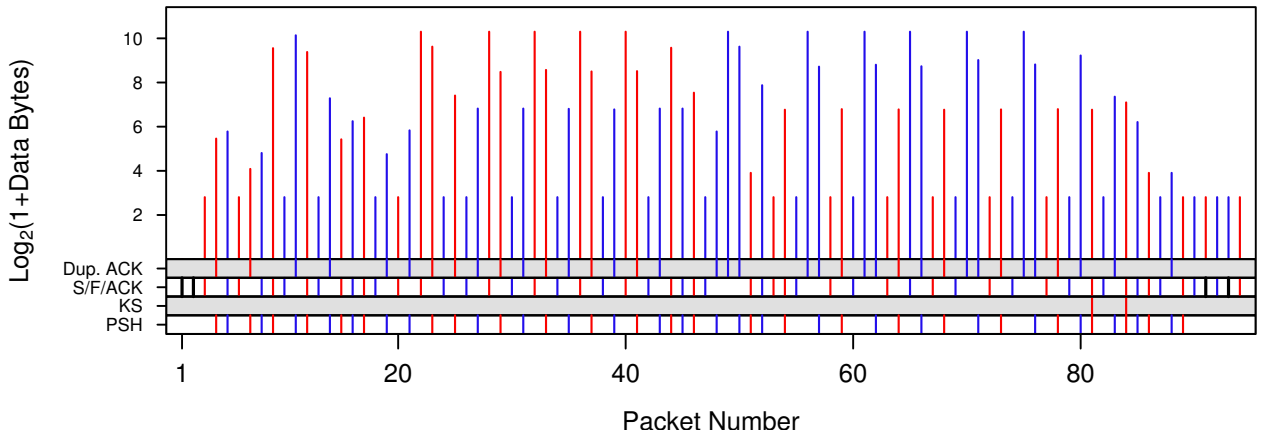


Figure 1.6.: First experiment in gestalt perception: color.

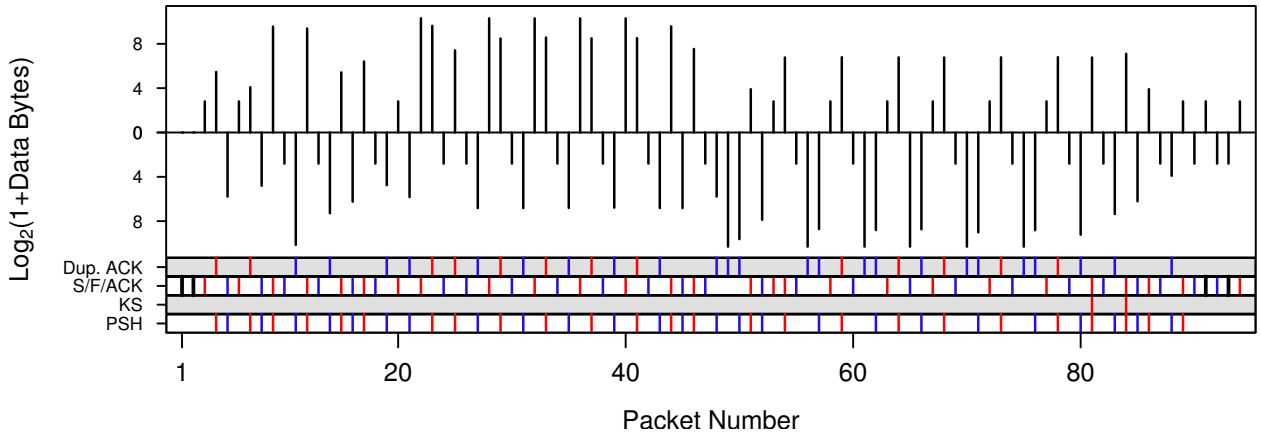


Figure 1.7.: Second experiment in gestalt perception: justaposition.

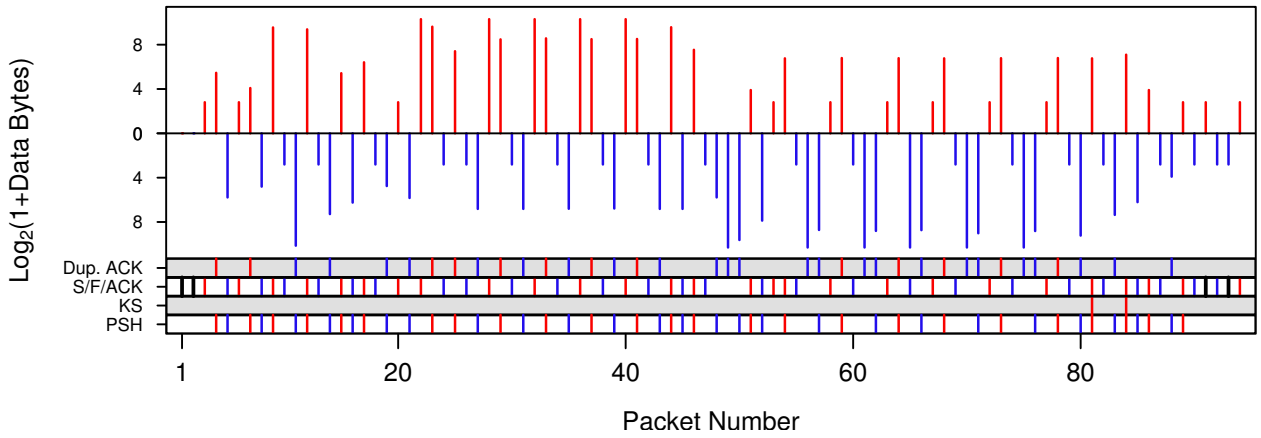


Figure 1.8.: Third experiment in gestalt perception: color and justaposition.

with the same display type replicated across the panels and pages, one subset per panel. But for this to succeed, it must involve the assessment of a gestalt: a pattern that forms effortlessly without attentive search of basic elements of the display. In other words, the pattern "hits you between the eyes."

When there are two or more types of gestalts to assess, it is best to scan through pages assessing one at a time. Attempting simultaneous assessment of two or more slows down the process remarkably because a cognitive shift of assessment is time consuming.

Gestalts can sometimes form as a matter of course for many common display methods; Figure 1.1 is one example. But in general, gestalt formation must get special attention in the design of a display. While principles of gestalt psychology can give some guidance (Koffka (1935)), some level of iterative experimentation for a particular type of display is typically needed.

The top panels of Figures 1.6 to 1.7 show an experiment with gestalt formation for a method of visualizing the detailed packet dynamics of sampled connections of the connection dataset. The connection in the displays is ssh. In the top panels, y = log(1 + payload size) for packets with payload bytes greater than 0 are plotted against the packet arrival order by the lengths of the vertical lines. Also encoded is the packet direction. In Figure 1.6 it is encoded by color; red is from the client and blue is from the server. We want to see each direction as a gestalt, mentally filtering out the packets of the other direction. In Figure 1.7 direction is encoded by juxtaposition instead of the superposition of the top panel. The server packets are displayed downward on the vertical scale and the client packets upward. We could juxtapose by an upward scale for the server with the lines emanating from the bottom of the panel, but this interferes with our ability to judge arrival order for client and server combined. In Figure 1.8, direction is encoded by by color and juxtaposition. Encoding direction just by color is the poorest method for gestalt formation, juxtaposition is better, and using both the best method, also allowing effective assessment of the sequence and size patterns in payload packets.

1.7 VDB Components

The software and hardware components of a VDB (in italics below) begin with a display *generator* in an interactive environment for data analysis that produces the VDB display documents in an image format such as PostScript or PDF. A *storage schema* might have the display documents as fundamental objects or the pages collectively of the documents. A display *viewer* renders the display documents on a display *device*; there can be different viewer designs to accommodate different document types and different amounts of screen space of display devices. *managers* organize the displays, and set out identifiers and descriptions of the VDB displays to enable easy selection for viewing. We use a plural, "managers", because there can be more than one. All provide direct access to display storage objects but vary in the documentation of the displays and project of which they are a part. The most basic provides simply object names. A next level might be notes for project participants: object names, descriptions of what is plotted, and conclusions from objects. At the most detailed, the manager provide a comprehensive narrative of the whole project of which the VDB is a part that can be understood by those not a part of the project.

Our projects using VDBs have included analysis of datasets ranging from moderate sizes to large and complex, and have included methodological research projects in statistics and machine learning that are developed and tested by their application to datasets. The display viewer and devices discussed in Section 1.4 resulted from considerations of VDBs. The other components are off-the-shelf, using what is readily available and already in use widely. We work in the R interactive environment and use lattice graphics in R as the display generator. R is the public domain version of the S language for data analysis whose commercial version is S-PLUS. Lattice is the public domain version of S-PLUS software for the trellis display framework for visualization discussed in Section 1.3. For large complex datasets, we employ distributed computing using RHIPE, the merger of R and Hadoop discussed in Section 1.2. In our storage schema, the displays are the database objects. Our managers are html pages with

the display files as links. There has typically been two types of managers: a basic manager just logging all display files, a project analysis manager containing short descriptions of displays, and notes that evolves from project to a narrative for others to read.

2. DETECTION OF SSH KEYSTROKE PACKETS IN TCP CONNECTIONS

2.1 Introduction

The input to the streaming algorithm presented here is packet arrival timestamps and information from TCP-IP packet headers in both directions of a TCP connection measured at a point along the path of the two hosts. In the algorithm, the host sending the SYN is the client and the other host is the server. The output is a classification of each client packet with data as an SSH keystroke or non-keystroke.

The variables used in the algorithm are the following: whether there are more than 22 packets or not, whether there are client packets after packet 22 or not, data size for client packets, data size for server packets, the number of packets between a client packet with data and the next acknowledging server packet with data, interarrival time of two successive client packets with data, interarrival time of two successive echoing server packets, the ratio of a client interarrival time and the interarrival time of the corresponding echoing server packets, and the number of packets between two successive client packets with data.

These variables constitute a description of the *packet dynamics* of a TCP connection. Packet dynamics have been been the object of vast and intensive study of TCP to understand performance and control, e.g. (Paxson (1997); Stevens (1994)). Some of the above algorithm variables are included in such study, but the goals and work here are very different. Some of the above variables are included in algorithms to detect interactive TCP logins. This past work, discussed in Section 2.7, involves looking for statistical properties that signal keystroke packets occurring, but does not attempt identifying keystroke packets individually.

Our algorithm succeeds because a keystroke creates an identifiable dynamical pattern. Such success has already been hypothesized by Donoho et al. (Donoho et al. (2002)):

> There are also other sources of information that we haven't discussed, the key one being the two-way nature of interactive sessions. There is far more information than just the keystrokes on the forward path through the stepping stones, there are also the echoes and command output on the reverse path, and it should be possible to use information about these to substantially improve detection.

There are many potential applications for identified keystrokes. One, mentioned above, is identification of SSH interactive sessions in any TCP connection, using port 22 or any other port. A session is interactive if there are any packets classified as keystrokes, and is non-interactive if there are none. This application is discussed extensively in this article. Our contribution is practical in that the algorithm is streaming and needs to retain only a limited amount of information at any given point, and so is suitable to be adapted as a component in realtime monitoring.

More generally, the article demonstrates the potential of detailed packet dynamics. This is important because algorithms that avoid deep packet inspection of payload mitigate privacy concerns and are largely insensitive to encryption. Much of the information in the TCP-IP headers are required to move and deliver packets and so is harder to counterfeit or subvert.

2.2 Details of the Algorithm

The algorithm inputs are derived from the timestamps and TCP/IP header fields in both directions of a single connection. The client is the host sending the SYN packet, and the server is the other host. The algorithm is streaming, processing the packets of the connection as they arrive; it is necessary to maintain the values of certain variables from a packet beyond the arrival of subsequent packets, but the

number of such values is small. The output of the algorithm is a classification of each client packet with data as a keystroke or a non-keystroke. For the application of the algorithm to detecting interactive connections, the output is a classification of a connection as interactive or non-interactive. An interactive connection contains one or more keystrokes; a non-interactive connection contains no keystrokes.

A keystroke is defined as a key depression that is part of typing to create text, up to and including the `Enter/Return` that ends the line. The definition of typing excludes key depressions that result in navigation through a document being viewed.

The primary variables for the algorithm are from TCP and IP packet headers: (1) Timestamp; (2) Source: *client* or *server*; (3) Size of the data in bytes, d_c for a client, d_s for a server packet; (4) Sequence number of a client packet with $d_c > 0$; (5) Acknowledgement number of a server packet with $d_s > 0$. The algorithm also uses secondary variables, described below, that are derived from the primary variables.

The algorithm consists of a sequence of 10 rules, applied in order. As soon as a packet violates any rule it is deemed not a keystroke. Rules I and II are the *SSH-Protocol Rules* and are applied first to exclude connections entirely. Rules 1 to 8 are the *Packet-Dynamics Rules* for keystroke detection. There are 8 statistical tuning factors that provide thresholds for the primary and secondary variables used by Rules 1 to 8. The values for the parameters are chosen based on the statistical performance of the algorithm.

SSH-Protocol Rules

Rule I. Candidate SSH Session. The algorithm counts packets in a connection. Packets 1 to 22 are the TCP handshake followed by 19 packets required for SSH session negotiation. if the packet number is less than 23, it is ignored.

Rule II. SSH packet length. Each packet starting with packet 23 where $d_c > 0$ or $d_s > 0$ is checked to see if its size is of the form $4k$ where k is a positive integer. This rule derives from the SSH Transport Protocol (Ylonen and Lonvick (2006b)) . If a packet size does not conform, the connection is abandoned.

Packet-Dynamics Rules

Rule 1. Minimum Client Data Size. $d_c \geq d_c^{(m)}$. Even though the current candidate data size d_c conforms to the SSH sizes of Rule II, there is a smaller range of highly likely values of d_c. $d_c^{(m)}$ is a statistical tuning factor that specifies the minimum value of the likely range for d_c.

Rule 2. Maximum Client Data Size. $d_c \leq d_c^{(M)}$. $d_c^{(M)}$ is a statistical tuning factor that specifies the maximum value of the likely range for d_c.

Rule 3. Maximum Server Echo Gap Size. $g_s \leq g_s^{(M)}$. The server *echo packet* of the current candidate packet, which is determined by matching the sequence number of the latter with the acknowledgement number of the former, plays a part in the rules. The number of packets between the current candidate packet and its server echo is the echo gap size, g_s. For a keystroke, the gap is not large; the maximum, $g_s^{(M)}$, is a statistical tuning factor for the maximum of the likely range.

Rule 4. Minimum Server Echo Data Size. $d_s > d_s^{(m)}$. As with d_c, the server echo data size d_s has a likely range of observed values. $d_s^{(m)}$ is a statistical tuning factor that specifies the minimum value of the likely range for d_s.

Rule 5. Maximum Server Echo Data Size. $d_s \leq d_s^{(M)}$. $d_s^{(M)}$ is a statistical tuning factor that specifies the maximum value of the likely range for d_s.

Rule 6. Minimum Client Candidate Interarrival. $i_c \geq i_c^{(m)}$. This rule uses the interarrival time, i_c, between the current candidate packet and the previous candidate packet. Client keystroke packets created by typing in an interactive SSH login have cadences that speed up and slow down. There are typing bursts, short cognitive pauses, and long cognitive pauses. Thus, i_c can vary substantially for true keystrokes. However, it has a lower bound determined by the limit of human typing speed. The statistical tuning factor $i_c^{(m)}$ is a lower bound of the likely interarrival time for human typing. If the current candidate packet is the first of the connection with $d_c > 0$, then it is not tested by this rule, and automatically conforms.

Rule 7. Maximum Absolute Log Interarrival Ratio. $\ell_{cs} \leq \ell_{cs}^{(m)}$. i_s is the interarrival time between the server echo of the previous candidate packet and the server echo of

the current candidate. This rule compares the ratio of i_c and i_s using the secondary variable $\ell_{cs} = |\log_{10}(i_c/i_s)|$. Under the assumption that jitter observed at the monitor is negligible, the ratio of i_c and i_s would be close to 1 whatever the absolute value of i_c. The statistical tuning factor $|\log_{10}(i_c/i_s)|$ places a limit on the deviation from 1. If the current candidate packet is the first of the connection with $d_c > 0$, then it is not tested by this rule, and automatically conforms.

Rule 8. Maximum Previous-Current Gap. $g_{pc} \leq g_{pc}^{(M)}$. The current candidate can pass this rule from the initial evaluation, making it a keystroke positive, or it can be deferred, making it a tentative candidate until the next candidate packet has its initial evaluation. In either case the current candidate will become the previous candidate when a new current candidate is encountered. g_{pc} is the previous-current gap, the number of packets between the previous and current candidates. The rule is based on behaviors during an interactive session. The typical pattern is an alternation between typing of text at the client and the sending of command output by the server. During a typing episode, a common transmission pattern seen by the monitor is a keystroke from the client, an echo from the server, an ACK from the client, and then the next keystroke. If this happens for the previous and current packets, $g_{pc} = 2$. $g_{pc}^{(M)}$ is a statistical tuning factor that is the maximum of the likely range of values of gaps between keystrokes during a typing episode. During a server output episode, the pattern tends to be packets from the server to the client carrying the output, and ACKs from the client. This tends to create a gap between the two keystrokes occurring before that is larger than $g_{pc}^{(M)}$. The principle of the rule is the following: if the two gaps before and after a candidate packet are larger than $g_{pc}^{(M)}$, then it unlikely that the candidate is a keystroke because it would imply output from typing one letter, whereas a normal line requires at least one character plus `Enter/Return`. The principle is applied by the following actions, which are based on the value of g_{pc} and the status of the previous packet:

1. $g_{pc} > g_{pc}^{(M)}$, previous = tentative.

   Action: previous $\leftarrow$ keystroke negative; current $\leftarrow$ tentative.

2. $g_{pc} > g_{pc}^{(M)}$, previous = keystroke positive.
   Action: current ← tentative.

3. $g_{pc} \leq g_{pc}^{(M)}$, previous = tentative.
   Action: previous ← keystroke positive; current ← keystroke positive.

4. $g_{pc} \leq g_{pc}^{(M)}$, previous = keystroke positive.
   Action: current ← keystroke positive.

2.3 Packet Trace Collection and Algorithm Development

2.3.1 Packet Trace Collection

Our research required packet traces — arrival timestamps and TCP/IP headers — on a gateway link carrying both directions of traffic between an inside network and the outside. The requirement corresponds to the targeted initial application of our keystroke algorithm, which is protection of an inside network by a monitor on the gateway link.

Furthermore, the research required trace collections for two types of connections: *commodity* and *scripted*. The former are the everyday traffic on a link carrying out the many applications running on the inside network. The latter are interactive connections initiated specifically for development and testing of the algorithm; they consist of prescribed commands and prescribed text input to programs that create or display documents. The commodity connections enable testing the classification of connections as interactive or non-interactive. The scripted connections enable testing of both the classification of connections, and the classification of client packets with data as keystroke or non-keystroke.

Traces of commodity connections were first obtained from a previous collection of 4 days 18 hours from the University of Leipzig Internet access link (Wand Network Research Group). The inside network in this case is the University of Leipzig campus and certain off-campus subnets.

The research also needed trace collection for an inside network for which we could access log files for SSH to allow us to have highly accurate determinations of whether a port 22 connection was interactive or non-interactive.

We set up trace collection on a subnet of the Purdue Statistics Department. A monitor collected traces with `tcpdump` running on a server connected to the span port of a switch that sees all traffic in and out of the subnet and between two virtual lans making up the subnet. Both commodity and scripted connections were collected. For connections with inside host port 22 we were able to access log files. But logging for the OpenSSH 5.3 server `sshd` was inadequate for verifying the correct classification, so simple modifications were made to the source code to emit additional messages for each connection. These messages captured the details of authentication and session characteristics, such as whether it was a login, single command, or subsystem request; had an allocated pseudo-tty; had requested port forwarding or X-window tunneling. Additional log messages recorded the intra-session forwarding and tunneling activity and, at session close, summary statistics on data transfer. This additional logging permits independent determination of SSH session attributes. In particular, it enabled highly accurate determinations of whether a connection with inside host port 22 was interactive or non-interactive. For scripted connections at inside host port 22, the `ssh` client program was modified to emit a tracer UDP packet every time it sent a keystroke packet. These tracer packets were collected by the subnet monitor along with packets from the SSH session, yielding a stream where the location of keystroke packets for scripted connections was precisely known.

2.3.2 Algorithm Development

Our development of the keystroke classification algorithm was done together with the development of its use in an application: classification of a connection as interactive or not.

The first phase of the work was an exploratory, unstructured study of traffic using both visualization methods and numeric methods of statistical analysis to gain a basic understanding of connection packet dynamics: arrival times, packet sizes, flags, sequence numbers, and acknowledgement numbers, as well as secondary variables derived from these primary variables. There was a comparison of the dynamics when keystrokes occurred and when they did not using the commodity traces from Leipzig, and commodity and scripted connections from Purdue. Conclusions from this empirical study, guided by descriptions of the SSH Architecture (Ylonen and Lonvick (2006a)) and of Transport (Ylonen and Lonvick (2006b)) Protocols, led to a succession of versions ending in the the streaming rules-based statistical algorithm described in Section 2.2. Details of this first phase work are not conveyed here.

The second phase of work was formal testing to verify previous conclusions, and to select certain tuning factors of the algorithm. This used commodity and scripted connections from Purdue, and is described in Sections 2.4-2.6.

2.4 Formal Test Regimens

The first data source for the formal tests consisted of 12 precisely defined interactive scripts, each resulting in a connection. They ranged from the simple command `ls` to more complex activities using the `emacs` editor or utilizing sequences of commands. Each script was executed 3 separate times yielding 36 connections. These scripted connections enabled the determination of false positives and false negatives for classification of client packets with data into keystroke and non-keystroke. In addition, the 36 connections enabled, for the application of of classification of connections into interactive and non-interactive, determination of the number of false negatives, `fn.22script`, among the 36. This is connection classification Test Regimen 1.

The second data source was five days of traffic collection on the Purdue subnet monitor, which resulted in 1,021,336 commodity connections.

Of the 7964 commodity connections which had one host using port 22, Rule I eliminated the 6672 with fewer than 23 packets. Of those, 207 connections with no client data packets were dropped. Rule II, which checks conformance of packet with SSH protocol eliminated another 38. Of these we were able to resolve all but one as actual packet errors leading to session termination. We were unable to deploy the revised `sshd` on all inside hosts and certainly not on any external servers, which reduced the 1047 commodity port 22 connections to 369 verifiable SSH sessions; 195 interactive logins and 174 non-interactive sessions (123 single commands and 51 sftp transfers). Performance of the classification of the 195 interactive connections as interactive or non-interactive is measured by the number of false negatives, `fn.22`. This is connection classification Test Regimen 2. Performance of the classification of the 174 non-interactive connections as interactive or non-interactive is measured by the number of false positives, `fp.22`.This is connection classification Test Regimen 3.

Of the 1,013,372 commodity connections which did not involve port 22, Rule I eliminated 703,353 with fewer than 23 packets leaving 310,019 connections which were all assumed to be non-interactive for purposes of the test. Of those, 150,228 connections with no client data packets after packet 22 were dropped. Application of Rule II dropped another 158,516 leaving 1275 candidate connections. Performance of the classification of these connections as interactive or non-interactive is measured by the number of false positives, `fp.not22`. This is connection classification Test Regimen 4.

2.5 Tuning Factors: A Multi-Factor Designed Experiment

The Packet-Dynamics Rules have 8 statistical tuning factors, as described in Section 2.2, that are thresholds for the variables used in the rules. The factors, their mathematical notation, and their units of measurement are shown in the first 3 columns of Table 2.1. Performance of the algorithm is measured by the 4 responses — `fp.not22`, `fp.22`, `fn.22`, and `fn.22script` — described in Section 2.4. We ran

Table 2.1: Statistical tuning factors and their values in the designed experiment.

			Increasing candidate packet pass-through: $\longrightarrow$		
Tuning Factor	**Notation**	**Units**	**1**	**2**	**3**
Minimum Client Data Size	$d_c^{(m)}$	bytes	48	32	24
Maximum Client Data Size	$d_c^{(M)}$	bytes	64	128	256
Server Echo Gap Size	$g_s^{(M)}$	number	2	3	4
Minimum Server Echo Data Size	$d_s^{(m)}$	bytes	44	32	24
Maximum Server Echo Data Size	$d_s^{(M)}$	bytes	64	128	256
Minimum Client Candidate Interarrival	$i_c^{(m)}$	$\log_{10}$ sec	-0.7	-1	-1.3
Maximum Absolute Log Interarrival Ratio	$\ell_{cs}^{(M)}$	$\lvert \log_{10} \text{ratio} \rvert$	0.05	0.075	0.1
Maximum Previous-Current Gap	$g_{pc}^{(M)}$	number	2	4	7

a multi-response designed experiment to determine the dependence of the responses on the levels of the tuning factors.

In the course of our pilot studies of performance we developed insight about ranges of the statistical factors for which the performance is quite good. Based on this we designed and ran a multi-factor fractional factorial designed experiment that varied all 8 factors in a systematic way in a region deemed to have reasonable performance, and studied how the above 4 responses changed with the values of the factors.

The experiment used the 1680 connections remaining after application of the SSH-Protocol Rules to the total 1,021,336 connections. For Test Regimens 1 to 4, remaining are 36, 195, 174, and 1275 connections, respectively. While only 0.16% of the connections remain, an absolute number of 1680 would be far too many for security analysts to review, making the Packet-Dynamics Rules critical. The Packet-Dynamics Rules were applied to the first 1500 packets of each of the 1680 connections.

The choice of tuning factors for statistical models and algorithms is often approached in the statistics and machine learning literature as just an optimization:

the best choice of the values of the tuning factors for the responses. Running a designed experiment goes well beyond this by providing valuable knowledge about the impact of the factors on the responses, the relationship of the responses as the factors change, the sensitivity of the algorithm to the changes in the values of the factors chosen for the experiment, and the trade-off between false positives and false negatives.

2.5.1 Experimental Design

The first step in the design was to select 3 values of each tuning parameter, which are shown in the last 3 columns of Table 2.1 so that candidate packet pass-through values increase from left to right. The factors that are minima, shown with superscript m, allow more candidate packets to pass-through as their values decrease; their values are ordered largest to smallest. The factors that are maxima, shown with superscript M, allow more candidate packets to pass-through as their values increase; their values are ordered smallest to largest.

Given 8 tuning parameters, there are $3^8 = 6561$ possible combinations requiring one experimental run. A run means applying the algorithm to each of the 1680 connections. We chose a fractional factorial design of with 243 runs from (Xu (2005)). It is a minimum-aberration resolution V design that spreads the points across the 8 dimensional space of the tuning factors to enable good characterization of the effects of the factors on the responses. The design has a certain balance of the values of the factors. Each of the 3 values of a factor occurs in 81 runs ($81 \times 3 = 243$). Each of the 9 combinations of values of 2 factors occur 27 times ($27 \times 9 = 243$).

2.5.2 Experimental Results: Dependencies Among the Responses

Each run of the experiment produces a 4-tuple of values of the four response variables. So the 243 runs produce 243 points in a 4-dimensional space. Figure 2.1 is a scatterplot matrix: all pairwise scatterplots of the four variables. The response

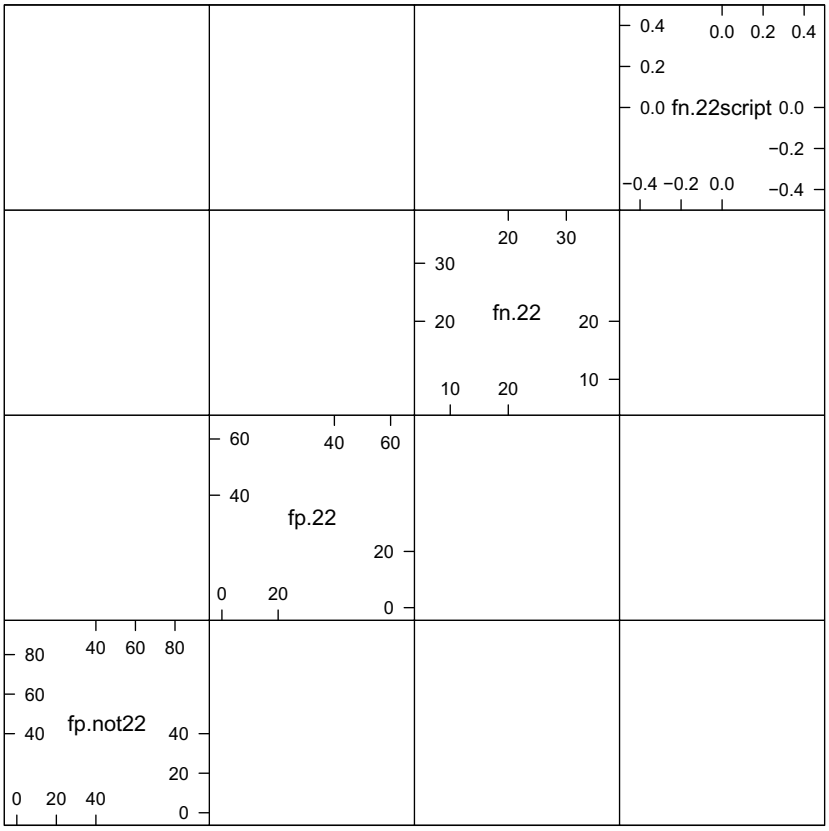


Figure 2.1.: Scatterplot matrix of the 4 responses — `fp.22`, `fn.22`, `fp.not22`, and `fn.22script` — for the 243 runs of the fractional factorial designed experiment.

names appear in a diagonal of the matrix. The vertical scales of the 4 scatterplots in each row of the matrix are the variable whose name appears in the row. The horizontal scales of the 4 scatterplots in a column of the matrix are the variable whose name appears in the column. Points have been jittered, a small amount of noise added, because a number of plotting locations have multiple points. Given a lower-left origin, the notation for the scatterplot in column i and row j is scatterplot ij. So the lower left scatterplot is 11. Note that scatterplot ij has the same variables as scatterplot ji, but with the scales reversed.

Figure 2.1 reveals important relationships among the responses. Scatterplot 32 shows a substantial trade-off between `fn.22` and `fp.22`, a strong negative dependence. The same negative dependence occurs for the three other pairs of one false positive and one false negative in scatterplot 31. Scatterplot 21 shows `fp.22` nd `fp.not22` have a positive dependence, although there are there are a number of quite low values of `fp.not22` for large values of `fp.22`. This likely occurs because non-interactive connections at port 22, since they are SSH, can behave differently from those at other ports.

What is surprising is the lack of correlation between `fn.22` and `fn.22script`. This might be occurring because the scripted typing, which requires having to follow prescribed commands under test conditions, is likely somewhat different from that for the every-day commodity connections. However, as we will see, there is a region of the tuning factors for which both these responses are small, which suggests a robustness in the algorithm to different typing conditions.

2.5.3 Pass-Through Analysis of False-Positives and False-Negatives

The trade-off between false positives and false negatives occurs through different levels of candidate packet pass-through. As discussed earlier, for each rule, as we go left to right in Table 2.1 through the levels of the tuning parameter, more candidate packets pass the rule. This, in principle, increases the number of false positives and decreases the number of false negatives. We can use this notion to determine the rules whose changing levels are most responsible for the trade-off.

We define a pass-through level variable for the values of the tuning factors: level 1 = the lowest level (column 4 in Table 2.1), level 2 = the middle level (column 5), and level 3 = the highest level (column 6). Next we select two sets of runs, the good-negative set, for which `fn.22` ≤ 10, and the good-positive set, for which `fp.22` ≤ 2. The cut-off values were chosen to achieve two goals. The first, which we can see achieved in Figure 2.1, is that different runs are in the two sets. The second is that

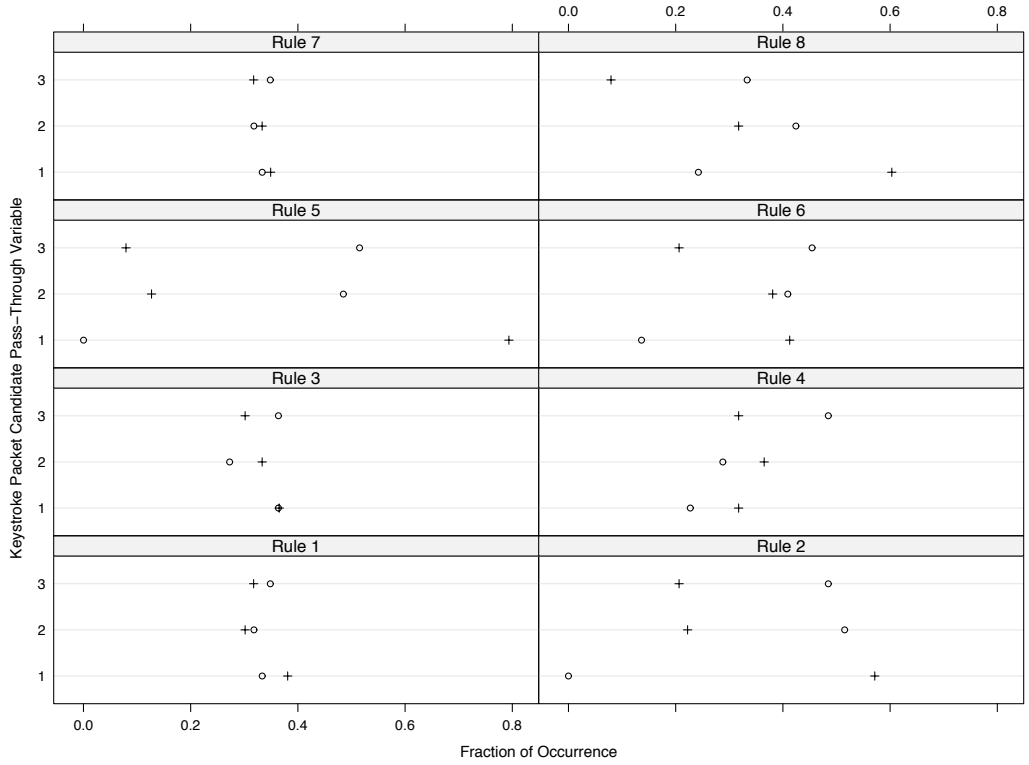


Figure 2.2.: For each rule, the fraction of occurrences of 3 levels of the pass-through variable for a set of runs with low `fn.22` (∘) and another set of runs with low `fp.22` (+).

the two sets have about the same number of runs; there are 66 in the good-negative and 63 in the good positive. For each rule, the fractions of occurrence of of 1, 2, and 3 are compared for the good negative and the good positive. The fractions are displayed in Figure 2.2.

Figure 2.2 shows little difference between the good-negative and good-positive sets for Rules 1, 3, 4, and 7; these rules play, overall, little role in the trade-off between false positives and false negatives. The other rules do substantially more. The fractions for the good-negative set tend to be higher for levels 2 and 3, and lower for level 1, than

the good-positive set, creating more pass-through. The most dramatic differences are Rules 2 and 5; pass-through level 1 is never taken by the good negative runs.

2.5.4 Tuning Parameter Values for Small Values of the Responses

While the trade-off of false positives and false negatives is informative, we want to select values of the tuning factors that make all errors as small as possible. However, considerations of false positives and false negatives are not necessarily same for many inside networks. For example, this is true for the Purdue inside subnet from which the data come. Other inside networks might have different considerations, so while the process of choice of tuning parameters is the same, specifics can be different. For the Purdue subnet, we seek false negatives as low as possible, and 0 is best; a missed intrusion can have major consequences. False positives can be more readily tolerated provided they do not become more than a minor burden for security analysts. With this in mind, we select runs with a partial compromise that gives somewhat higher priority to control of false negatives: `fn.22` ≤ 10, `fp.not22` ≤ 10, `fp.22` $<= 20$, and `fn.22script` $= 0$. This was achieved by 8 runs. Figure 2.3 displays the fraction of occurrences of the pass-through variable in the same manner as Figure 2.2. Comparison of the two figures shows that for the compromise runs in Figure 2.3, there is a greater balancing of pass-through than in Figure 2.2. One run that is a very reasonable choice from the 8 runs is shown in Table 2.2 which has 2 values of the pass-through variable at level 2, and 2 each at levels 1 and 3, exactly balancing the pass-through.

2.5.5 Rule Impact

The analysis of the 4 responses has given insight about the tuning factors for Rules 1 to 8, showing how much they change the responses over the ranges chosen in the experiment, but this does not speak directly to the impact that each rule has on the

Table 2.2: Values of pass-through variables to minimize false positives and false negatives, but with a preference for greater control of false negatives.

Rule	1	2	3	4	5	6	7	8
Tuning Factor	$d_c^{(m)}$	$d_c^{(M)}$	$g_s^{(M)}$	$d_s^{(m)}$	$d_s^{(M)}$	$i_c^{(m)}$	$\ell_{cs}^{(M)}$	$g_{pc}^{(M)}$
Pass-Through Variable	1	2	2	3	3	3	1	1

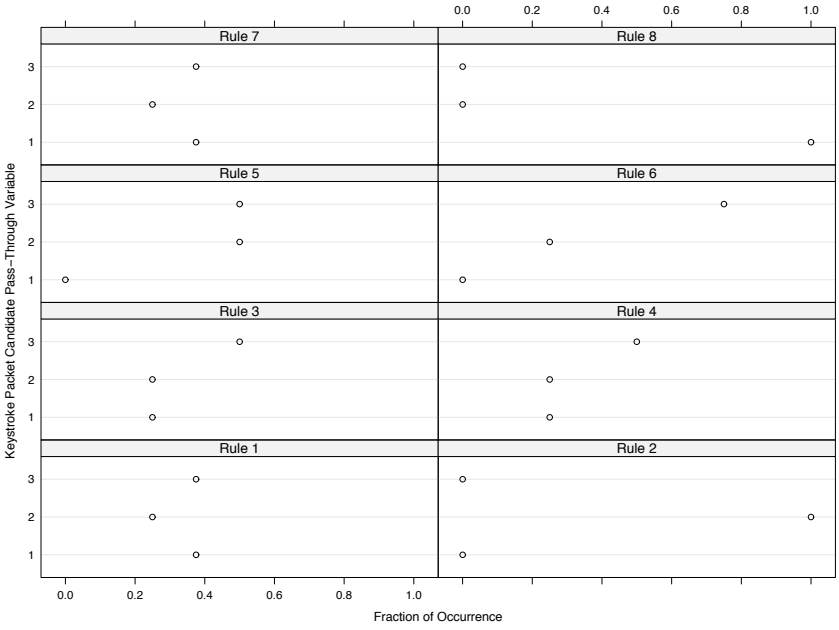


Figure 2.3.: For each rule, the fraction of occurrences of 3 levels of the pass-through variable for runs that are a partial compromise of false positves and false negatives, but but with somewhat higher priority to control of false negatives.

classification We studied the impact by analyzing the numbers of dropped packets by the rules.

For each of Rules 1 to 8, the impact metric for a rule based on one connection is the count of client candidate packets dropped by the rule. If after application of a rule, for example Rule 4, no client packets with data remain in the connection, then the values of the metric for succeeding rules, Rules 5 to 8 in our example, are 0. Below we will study the impact metric for a collection of m connections; in this case each rule has m values of the impact metric, and we will study the 8 distributions of metric values, each with m values, to determine the impact. Here, we describe results of the analysis of the impact metric for the connection collections of Test Regimen 3 and 4.

The value of m for Regimen 4 is 1275 (the number of connections). Thus for each of Rules 1 to 8, there are 1275 by 8 counts of packets dropped, one count per connection. If the rule did not drop packets for a connection then the count for the rule is 0. For Rules 1 to 8, the number of connections with nonzero packets dropped are the following:

$$(1)\ 182,\ (2)\ 1135,\ (3)\ 31,\ (4)\ 75,\ (5)\ 33,\ (6)\ 42,\ (7)\ 14,\ (8)\ 92.$$

Let the above number for Rule k be v_k The panel for Rule k of Figure 2.4 graphs the v_k nonzero numbers for each the rule. Let n_i be the i-th largest value of these numbers. Then $\log_2(n_i)$, where $\log_2$ is log base 2, is plotted against i/v_k. So a fraction i/v_i of the values on the vertical scale are less than or equal to $\log_2(n_i)$.

The values of v_k and Figure 2.4 show that Rule 2 has the largest impact by far. It dropped packets in 1135 of the 1275 connections, Rule 1 has the next largest impact, and then Rules 4 and 8. Rule 7 appears to have the least impact, with only 4 connections affected.

The value of m for Regimen 3 is 174. For Rules 1 to 8, the number of connections with nonzero packets dropped are the following:

$$(1)\ 150,\ (2)\ 173,\ (3)\ 49,\ (4)\ 55,\ (5)\ 70,\ (6)\ 164,\ (7)\ 7,\ (8)\ 173.$$

Figure 2.5 displays the nonzero counts of packets dropped using the same display method as Figure 2.4. In the case, the impact of Rules 1, 2, 6, and 8 are quite substantial. Rule 7 remains as the one with substantially less impact than the others.

A rule with low impact, such as Rule 7, might be a candidate for removal, but more investigation is needed. If it removes non-keystroke packets that other rules miss, and has a very low removal rate of keystrokes, then it can still be useful. This topic is discussed further in the future work discussion of Section 2.8.

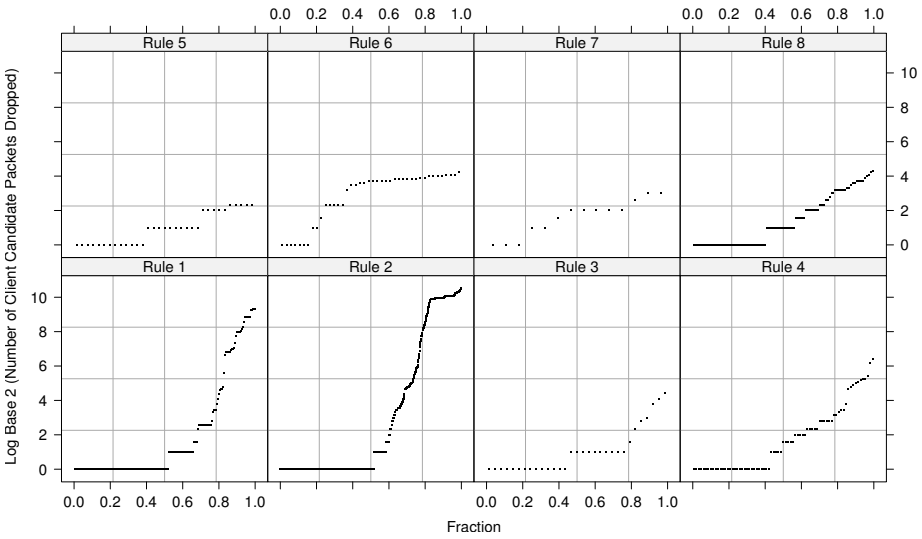


Figure 2.4.: Each panel plots the log base two of the nonzero counts of client keystrokes dropped by one rule for the 1275 connections from Test Regimen 4.

2.6 Testing the Keystroke Packet Classification

The methodology for Test Regimen 1 allows us to test the accuracy of the classification of client packets with data as keystrokes or non-keystrokes. The SSH connections of the regimen consist of 12 sessions of scripted activities. All the activities were performed by one individual from a client outside the Purdue subnet to a server within it. Each session was repeated 3 times for a total of 36 connections. For each connection, the SSH packets and UDP key-press tracer packets were collected which

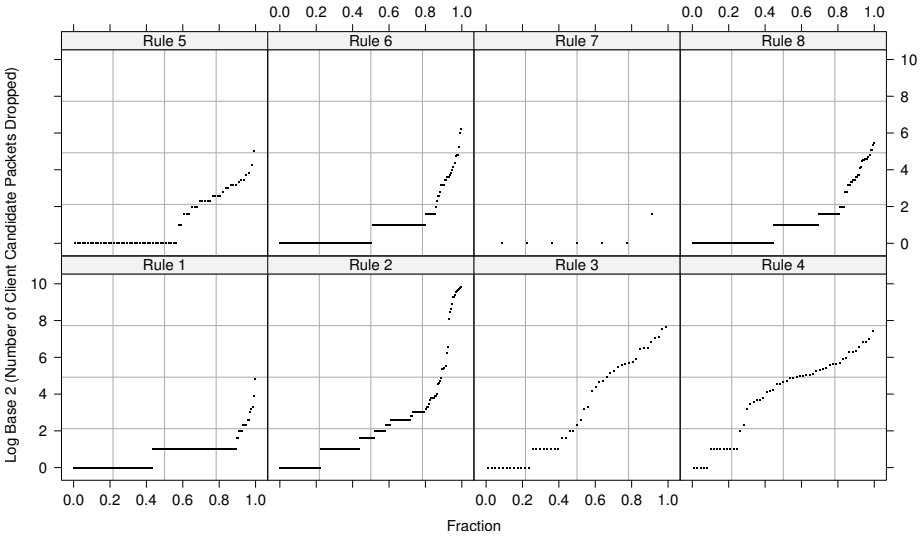


Figure 2.5.: Each panel plots the log base 2 of the nonzero counts of client keystrokes dropped by one rule for 176 connections from Test Regimen 3.

allowed determination of the client keystroke packets. The algorithm was then applied to the SSH packets using parameter values from Table 2.2.

Client packets containing data are candidates to be keystrokes. In the sessions, candidates are either (1) keystrokes, (2) packets in an SSH negotiation if it extends past packet 22, or (3) program document navigation packets. Classification errors in category (1) are false negatives, and false positives in category (2).

Each of Sessions 1-3 consisted of a single command resulting in a small number of packets. For example, one session is

```
uname -a <return>
```

Table 2.3 shows results for Sessions 1-3. There were no false positives or false negatives.

Table 2.3: Test Regimen: Results for Sessions 1 to 3

Session	Keystrokes	False Neg.	False Pos.	Client Data Pkts.	Client Pkts.	Server Data Pkts.	Server Pkts.
1	3	0	0	9	25	15	17
	3	0	0	9	26	16	18
	3	0	0	9	26	16	18
2	13	0	0	19	46	25	26
	13	0	0	19	44	25	26
	13	0	0	19	44	23	24
3	9	0	0	15	38	22	24
	9	0	0	15	37	21	23
	9	0	0	15	37	21	23

Each of Sessions 4-8 consisted of multiple commands over several lines. Compared with Sessions 1-3, there were more keystrokes and sessions were longer. One session is

```
cd tmp <return>
cd git-1.6.3.1 <return>
./configure <return>
```

Table 2.4 shows results for Sessions 4-8. There were no false positives or false negatives.

In Sessions 9-12, programs which required text typing during execution, e.g. the `vi` editor, were used. Editors also have key presses that are not keystrokes by our definition, but act as navigation or editing aides, e.g. pressing the `j` key when `vi` is not in text-entry mode moves the cursor down one line. Table 2.4 shows results

Table 2.4: Test Regimen: Results for Sessions 4 to 8

Session	Keystrokes	False Neg.	False Pos.	Client Data Pkts.	Client Pkts.	Server Data Pkts.	Server Pkts.
4	32	0	0	38	84	45	47
	32	0	0	38	85	46	48
	32	0	0	38	84	45	47
5	53	0	0	59	262	202	204
	53	0	0	59	262	202	204
	53	0	0	59	264	204	206
6	31	0	0	37	452	414	416
	31	0	0	37	517	479	481
	31	0	0	37	518	480	482
7	203	0	0	209	462	252	254
	203	0	0	209	460	250	252
	203	0	0	209	462	252	254
8	173	0	0	180	1771	1591	1594
	173	0	0	180	1788	1608	1610
	173	0	0	180	1791	1611	1613

for Sessions 9-12. The causes of false negatives and positives for the session were the same for the 3 connections.

Of the 3 false negatives in Session 9 (creating a document in console based `emacs`), one was a text creation character that also cleared the screen. The data size of the server echo for this was larger than the maximum for Rule 5. The remaining two, the keystroke sequence (`[ctrl-x] [ctrl-s]`) to save a file, caused zero byte server acknowledgments and were discarded by Rule 2.

The false negative in Session 10, was caused by typing `q` to quit a `man` page browser. This keystroke triggered more than 5 packets to be sent between the client and server. This results in a value of g_{pc} larger than the Rule 8 value of $g_{pc}^{(M)} = 2$. In addition, the previous key press was also for navigation and resulted again $g_{pc} > g_{pc}^{(M)} = 2$. This led to Rule 8 rejecting the `q`.

Session 11 consisted of launching `vi` and transcribing a paragraph. The first time `i` was pressed, `vi` switched to text-entry mode, cleared the screen, and changed the status line. The server responded with 1448 bytes and the keystroke was rejected by Rule 5. The first keystroke of the text-mode typing entry was a false negative because its first server acknowledgment contained no data which caused rejection by Rule 4.

In each connection of Session 12, browsing `man` pages, there were 6 false negatives due to the typing of `q` as described above for Session 10. One of the activities in this session was to search for a word in a `man` page. The search was initiated by pressing `/`, typing the search word and pressing `Enter`. The `man` program then found and highlighted the searched word. The packet corresponding to the `Enter` was rejected by Rule 5 because the server response was 1448 bytes. The 2 false positives were the two consecutive presses of the space bar to navigate to the following page.

2.7 Past Work

There is a large literature on classifying network traffic by attributes of its connections or packets rather than payload Examples of this research from Montigny-Leboeuf (De Montigny-Leboeuf (2005)) , Dunigan (Dunnigan and Ostrouchov (2000)), Hernandez (Hernandez-Campos et al. (2005)), Karagiannis (Karagiannis et al. (2005)), Wright (Wright et al. (2006)), Horton (Horton and Safavi-Naini (2006)), and Alshammari (Alshammari et al. (2009)) employ statistical and machine-learning methods to cluster and classify traffic by application protocol and across dimensions such as interactive vs. bulk transfer.

Table 2.5: Test Regimen 1: Results for Sessions 9 to 12

Session	Keystrokes	False Neg.	False Pos.	Client Data Pkts.	Client Pkts.	Server Data Pkts.	Server Pkts.
9	54	3	0	60	163	102	106
	54	3	0	60	165	104	108
	54	3	0	60	164	103	107
10	9	1	0	19	57	37	39
	9	1	0	19	57	37	39
	9	1	0	19	57	37	39
11	338	2	0	344	697	353	357
	338	2	0	344	697	353	357
	338	2	0	344	696	352	356
12	177	7	2	198	490	303	307
	177	7	2	198	489	301	305
	177	7	2	198	495	306	310

Our work is most similar to research in detecting backdoors and stepping stone traffic. Zhang and Paxson (Zhang and Paxson (2000a)) label connections with a high proportion of small packets with inter-arrival times in the range of 10 ms to 2 sec as interactive. It also checks packet lengths for conformance to the SSH protocol: if 75% of the packets meet the specification, the connection is declared SSH. In contrast, our work performs differently by identifing individual keystroke packets.

Timing analysis along with packet size has been used in the detection of stepping stones by Zhang (Zhang and Paxson (2000b)) and detection of interactive terminal sessions over stepping stones by Yung (Yung (2002)). Ding (Ding et al. (2009)) estimated the full RTT for long chains in a stepping stone attack by identifying the interval from the end of command output from the server to the next client input

using TCP sequence and acknowledgement numbers. This is similar to our approach although our algorithm analyzes intervals in the reverse (client to server) direction.

Donoho (Donoho et al. (2002)) was able to demonstrate that even if traffic is jittered for purposes of evasion, there are theoretical limits on the ability of attackers to disguise traffic and showed a wavelet-based approach for multi-scale detection. The stepping stone detection Donoho was engaged in sought to correlate the inbound and outbound connections of an intermediate stepping stone machine. Our interest is in detecting the traffic at the ultimate target of the attack and being the endpoint of the chain, our algorithm must find patterns within an single connection.

2.8 Discussion

2.8.1 Results

The algorithm succeeds because the keystroke and echo packets create an identifiable dynamical pattern. The size and timing relationships of these packets is clearly different from those which are seen for machine-generated traffic.

The SSH protocol plays an important role in the algorithm by dramatically reducing the number of connections under consideration. Here, encryption is our friend. In the first instance, setting up the encrypted session requires almost two dozen packets. This eliminates the majority of connections from consideration. In the second instance, the sizes mandated by the smearing required for encryption create a stark length signature which drops half of the remaining connections. In what remains, only an handful of client packets will qualify as candidates.

And yet, because a human will eventually need to resolve connections classified as interactive, eliminating 99.84% of the traffic still leaves far too much. This is where the Packet-Dynamics Rules become crucial. The formal testing and the experiment to explore the effect of the tuning factors reveal how thresholds can be set to achieve different goals.

One might object that the combination of SSH protocol attributes with features unique to human typing is an isolated opportunity and that studying network traffic dynamics will prove a dead end. We argue that this will not be the case. Machine-to-machine communication appears to be full of patterns induced by applications, libraries, and other software layers providing a rich surface for detecting both expected an unexpected behavior.

2.8.2 Limitations

The Packet-Dynamics Rules focus on the timing characteristics between client keystroke and server echo and its performance could suffer in conditions of heavy or widely fluctuating load. All of our experience to date has been on networks where (a) there is no substantial jitter for the packets during an individual connection; (b) servers are not so burdened as to have to defer generating echoes; and (c) the monitor is close enough, in terms of network distance, that there are no congestion delays between it and the inside servers. We have tried informally to stress the algorithm by using stepping stones on remote networks, but formal testing is warranted.

The SSH protocol permits the multiplexing of data streams within a single active connection through a variety of tunneling mechanisms. This could result in keystroke traffic being shrouded by other data. In simple instances of X11 tunnels, the algorithm still makes correct decisions, but we did not test all of the myriad possible tunnel and port forwarding configurations.

Of course attackers would try to evade detection by the algorithm. In giving sample evasion scenarios, we divide the rules into roughly three groups: *SSH Protocol:* Rules I and II which limit connections to SSH traffic; *Size:* Rules 1,2,4,5 identify candidate keystrokes by packet size ranges; *Timing:* Rules 3,6,7,8 identify timing characteristics of interactive traffic.

In *Timing*, Rule 6, which depends on the cadences derived from the human factors of typing, can be directly attacked using an unmodified SSH client if the session is

launched with the "-T" flag. This prevents the server from allocating a pseudo-tty and forces the client program to assemble keystrokes into entire lines of input before sending them to the server. However, this has significant negative consequences for the attacker because the session no longer appears interactive to many commands useful for *ad hoc* exploration.

To evade *Size* constraints an attacker able to modify the SSH client can convince the algorithm that the session was non-interactive by padding all keystroke packets to have lengths larger than the maximum client packet size in Rule 2.

The *SSH Protocol* (Ylonen and Lonvick (2006b)) Rule II requires that **every** keystroke packet have a length which is a multiple of 4. Attacking the algorithm here involves convincing it that Rule II has been violated even though the victim server believes it hasn't. This could be done if the monitor is several network hops from the victim A skilled attacker could inject a chaff packet whose length was not a multiple of 4 but which had a reduced TTL value such that it would be seen at the monitor but expire before reaching the victim.

2.8.3 Future Work

The algorithm has been developed against a set of collected packet traces from several sources. As a streaming algorithm, we hope to implement it as part of live traffic monitor either by building our own or augmenting an existing open source sensor. Implementation should be straightforward. There is a need to optimize both storage and execution time and there are practical considerations about such things as how to timeout connections which need to be addressed.

Although keystroke detection appears robust enough for the simple connection classification task at hand and even though we are greatly encouraged by its accuracy during the scripted regimen, we will need to do much more comprehensive testing. This is important because one can readily envision any number of useful things to study given strings of encrypted keystrokes and their timing information, e.g. set of

strings accumulated into lines whose lengths alone serve as a forensic clue. Indeed, we are hoping to collaborate with Maxion at CMU whose research suggests one could use signatures derived from typing rhythms (Kilourhy and Maxion (2009)) to potentially correlate actors across disparate intrusions.

We will continue basic research in the fundamentals of network traffic analysis where knowledge gained from packet dynamics can also inform the design of new connection summaries targeted at security.

Acknowledgements

This work was supported in part by the Army Research Office MURI Program under award W911NF-08-1-0238, the National Science Foundation under award CCF-0937123, and the U.S. Department of Homeland Security under a Center of Excellence award.

3. R AND HADOOP INTEGRATED PROCESSING ENVIRONMENT

3.1 Introduction

In the previous chapter upwards of hundred gigabytes of data for the keystroke analysis was collected and analyzed. The analysis of massive data has been (and in many cases still is) cumbersome and even infeasible for some types of analyses (e.g. time series analysis of long range dependent data). The defining characteristic of such data sets is that there are too many observations to be placed in main memory and computed with all at once. This is not a new problem: the size of data has always been greater than available fast access memory (RAM). *External memory* algorithms are a class of algorithms that compute with large data sets without out having to load everything into memory. Examples of external memory algorithms include matrix multiplication (Sibeyn (2004)), singular value decomposition (Toledo (1999)) and graph theoretic algorithms (Chiang et al. (1995)). SAS, the statistical analysis software system sold by SAS Incorporated, is a well known example of a statistical tool that uses external memory algorithms to compute with gigabytes of data. Programming libraries e.g. STLXX (Beckmann et al. (2009)) provides support at the data structure level for large data sets (it does not provide linear algebra routines for large data). Further reading on the topic of external memory algorithms can be found in (Vitter (2008)).

In contrast to the above approach of computing with all the data, sampling the data is a popular practice among analysts. This involves taking a uniform random sample of size n (less than N, the total size of the data) or a fraction p of the data. This is not feasible for data which is not in the row-column format. For example, taking a random sample of telephone customers to model their social connections (e.g.

who calls whom) would lead to an incomplete and possibly misleading analysis. Row column data sets of N rows and K columns, can be geometrically represented by N vectors in R^K. Simple random samples do not provide guarantees that the n sampled vectors cover the range of the N vectors. The approach discussed in (Dumouchel et al. (1999)) construct a smaller data set by partitioning the data space into smaller regions called spheres and pyramids. Within each region, m new data points and associated weights are constructed. The restriction on the new points is that their weighted likelihood is equal to the likelihood of the original data in the region. Using p'th order Taylor approximations to the likelihood reduces this constraint to moment matching of the new and original data. Because of some restrictions imposed by the data e.g the range of the new points should be within the range of the original data (though the restriction is only on marginal ranges), the solutions might not exist. The authors use a least square approach to estimate the new data points that minimizes the squared error loss between the new and original moments.

Rather than analyzing a massive data set as a single entity we can compute across subsets of the data. The subsets can run into millions. There can be different ways to subset the data. Each partition of the data is stored as a separate data set. This is still in the realm of high performance computing and not enough tools exist to analyze this efficiently. For example, in a packet collection of network data, it is possible to quickly collect hundreds of gigabytes. A connection can be defined as the flow of packets between two computers in a single communication. An analysis might involve computing across millions of connections, each small in themselves. We can subset by connection, or by source IP address. The compute-across-subsets approach is a possible approach to regression: partition the data into subsets. The subsets are *near replicates*: the joint distributions of the independent variables are similar across subsets and similar to that of the population. An open research problem is: what is the loss in recombining the coefficients from the regressions across subsets as compared to a regression on the whole data?

In the following sections, I will discuss MapReduce, a programming model for massive data. I introduce the Hadoop project for distributed computing and its relationship with the MapReduce programming model. I talk about R, a programming language for statistics and its support for high performance computing and computing with massive data. I then introduce the R and Hadoop Integrated Programming Environment (RHIPE), followed by several examples of programming with RHIPE to compute with large data sets across a cluster of computers. The RHIPE section can be treated as a manual, though it is instructive to read Sections 3.2 and 3.3 to gain an understanding of the MapReduce programming model.

3.2 What is MapReduce

Concurrent programming is known to be difficult to get right i.e. a program when evaluated in a concurrent environment returns correct results (Lee (2006)), to quote (Sutter and Larus (2005))

> ... humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code.

Given an input I, a program must always return an output $O(I)$ whether it is evaluated sequentially or concurrently. A statistician, not familiar with concurrent programming techniques needs to familiarize herself with concepts to ensure correct evaluation e.g. locks, monitors, semaphores and mutually exclusive regions. Concurrent programming in a distributed environment introduces new challenges such as inter-machine communication, machine failure and network bandwidth. The MapReduce programming model is a simplified concurrency programming framework. Using this framework, a programmer can write code to do more than embarrassingly parallel but less than fully distributed computations. Given this restriction, a program written in the MapReduce programming model can be automatically and correctly distributed across a cluster of computers. Key to the successful implementation of

MapReduce is that operations are side effect free i.e. the programmer is not allowed to perform operations on shared data structures.

The terms Map and Reduce have existed in the computer science literature for a long time. A map is a higher order function that applies a given function to a set of objects. Several languages refer to this by different names, e.g `map` in Python, Perl and Ruby, `Map` in R and `mapcar` in Lisp. For example, let `isprime` be a function that returns a boolean value according as a given integer is prime or not. We can use `Map` in R to apply this to a sequence of numbers,

```
primes <- Map(1:1000, isprime)
```

Reduce (also known as fold) applies a function to a series of elements aggregating the results of the function. For example in R,

```
Reduce(c(1,2,3),sum)
```

returns 6. For an ordered stream of elements e_i, a *left reduce*, computes $j_i = f(j_{i-1}, e_i)$ (where $j_0 = e_0$), and returns $f(j_{n-1}, e_n)$.

In the context of MapReduce as implemented in Hadoop, a map is a user provided function that takes a two elements K and V and returns two intermediate values K_I, V_I. The first element is called a *key* and the second is called the *value*. The Hadoop system then collects all intermediate values V_I that are associated with the same key K_I. These are streamed to the user provided reducer function which computes with the values (and key) to return a final key and value.

The key can be interpreted as a categorical variable that partitions the data. The value are the objects associated with the same level of the categorical variable. Categorical variables can be arbitrarily defined e.g. a discretization of a continuous variable and the cartesian product of several nominal variables. Technically, given the discrete nature of hardware level representations of numbers, everything is discrete e.g. a double precision number can take at most 2^{64} values, it is advisable, for memory and efficiency reasons to bin the values of such a variable if it is to be used as a key.

3.2.1 Examples

Sum of two columns of numbers The map function will be called for every pair of numbers as the values (V). The function returns (also called *emitting*) two intermediate key,value pairs. The first intermediate key is equal to 1 and the value is the first number in the pair. For the second number, the function emits, an intermediate key equal to 2 and the second number as the value. The reducer, receives a stream of intermediate values corresponding to the intermediate key 1. The reduce function returns the sum of these numbers. Similarly, it will receive a stream of intermediate values associated with the key 2. In Hadoop MapReduce, the reduce functions for the two intermediate keys will be evaluated in parallel.

Word Frequencies Often used for text mining analysis, the map and reduce functions will return a the frequency count of unique words. The map receives a line of text as the value which it tokenizes into distinct words. Each word is an intermediate key and the corresponding value is 1. The reducer will add the intermediate values for a given intermediate key to return the word and its frequency.

Quantiles For discrete data, by obtaining a frequency count of the unique values in the data set, the quantiles of the data can be exactly computed. The same approach works for continuous data though care must be taken to round the number to N significant places. Without rounding there can be as many unique values as there are rows in the data set. It may be possible to assume a straight line fit for the rounded data after the N significant figures and use interpolation to obtain approximate qauntiles.

3.3 The Hadoop Project

Hadoop is an open source implementation of the Google Filesystem and Google MapReduce. The project began with Doug Cutting but is now under the auspices of the Apache Software Foundation. The website for the project is (Hadoop). The Hadoop DFS and MapReduce is used by several major companies including The New

York Times, Cloudera, Amazon, DoCoMo and Bank of America. For a complete list of companies that use Hadoop, see (Hadoop Companies).

3.3.1 Hadoop DFS

A distributed file system (DFS) pools the storage of computers in a cluster into one unified file system. The user transparently interacts with the filesystem without knowledge of the distribution of files across the cluster. Typically, a DFS strips and replicates files. That is, a file when stored on the DFS, is partitioned into blocks and each block is replicated across the cluster. Replication entails storing copies of blocks on different cluster computers. The replication provides a measure of fault-tolerance. In case of machine failure, the block can be accessed from a another computer. In this section we shall discuss the Hadoop DFS.

The Hadoop DFS is modeled after the Google File System. It follows most of POSIX standards though sacrifices some e.g. appends for efficiency reasons. It is a *write once-read many* filesystem providing bulk throughput at the cost of latency and designed to run on commodity hardware.

The HDFS implementation language is Java and can be installed on top of the filesystem present on the cluster computers. The HDFS is implemented using a master-slave model. One computer plays the role of the *Namenode*. The Namenode is the gateway to the HDFS. This node manages the filesystem including the location and replication of files stored on the HDFS. Clients interact with the HDFS through the Namenode. Files that are stored on the HDFS are partitioned into blocks. The Namenode then replicates these blocks and requests the *Datanodes* to write the blocks. Datanodes are cluster computers that manage the data local to that machine. The Datanodes serve read/write requests from clients (via the Namenode) - actual access to the data occurs between the client and the Datanodes bypassing the Namenode. The Namenode stores the file to block mapping including file attributes e.g. file owner and file size. Typically, one node in the cluster is the Namenode and the rest

are Datanodes. Because there is only one Namenode, this becomes the single point of failure of the cluster.

When a client writes a file to the HDFS, the file is partitioned into blocks which are stored on the cluster computers. These blocks can be changed on a per file basis. The default value is 64MB, though for very large files, a block size of 128MB can improve performance (a large block size decreases the number of the entries in the HDFS allocation table). Initial writes occur locally on the clients filesystem and not the HDFS. Once the size of the local buffer has reached the block size, the client contacts the Namenode which creates an entry in the HDFS for the file's first block in the HDFS file allocation table. It also tells the client which Datanodes to write it to (as many as the replication number). The client then flushes the buffer to the first Datanode. This Datanode receives data in small 4KB chunks, which it writes to disk and streams to the next Datanode in the list which does the same for the next Datanode in the list up until the last Datanode (thus the client has only written to one Datanode, avoiding flooding the network with data). Once the entire file has been written, the client informs the Namenode which commits the file meta data e.g file creation time, file owner, file size to its HDFS meta table. Fig 3.1 is a schematic of the layout of HDFS.

3.3.2 Hadoop MapReduce

The Hadoop MapReduce Framework is a software environment for implementing MapReduce programs on data sets stored on the HDFS. Jobs are submitted to the *Jobtracker*. The Jobtracker maintains the queue of pending jobs, informs the worker nodes, called Tasktrackers, of new work and collects the results. Given the input data, the user supplies an *Inputformat*. The task of the Inputformat is to partition the data into splits. Typically, the splits correspond to the blocks of the file. However, the splits can also be filenames or blocks of rows in databases. Splits are assigned to the different Tasktrackers to compute with. The Tasktrackers launch a number

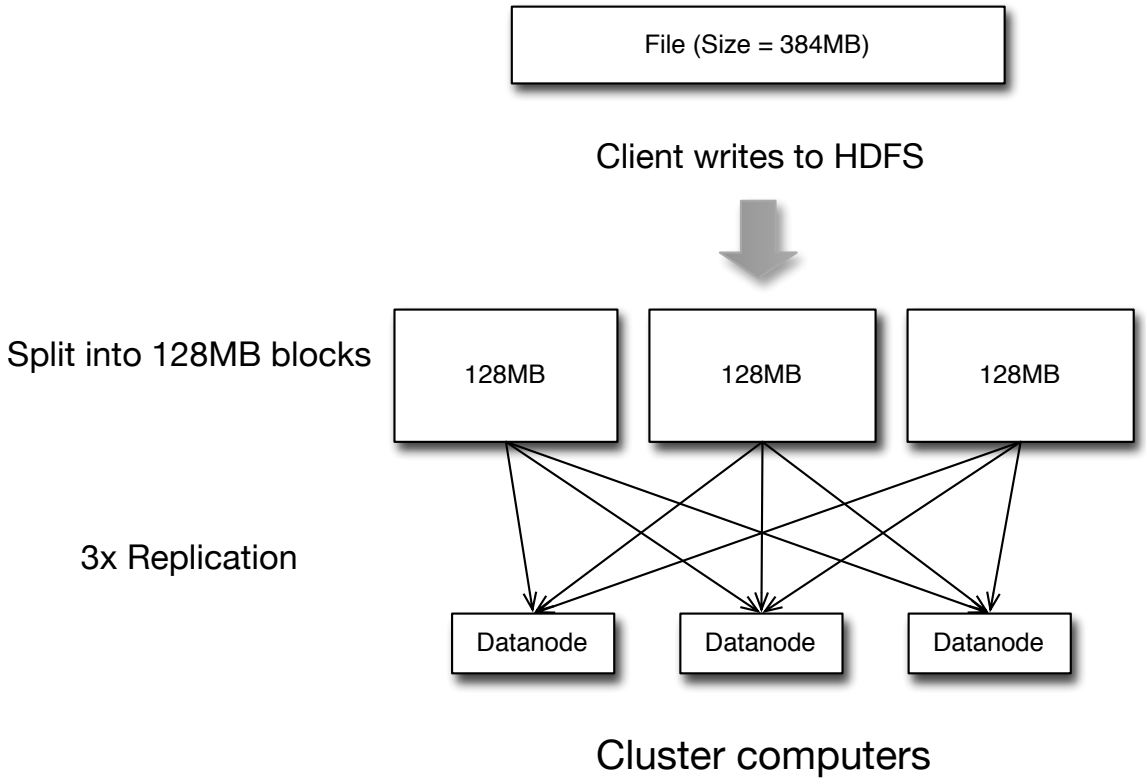


Figure 3.1.: A file of 384MB is written to an HDFS with 3 datanodes and blocksize of 128MB. The file is replicated 3 ways across the cluster datanodes

of Java Virtual Machines (JVM) each of which process the assigned splits in parallel taking advantage of the cores in a computer. The Hadoop MapReduce framework then calls the Inputformat's *Recordreader* to parse the data in the splits into records of key-value pairs which are processed by the `run` method in the user defined Map class (which is run by the child JVMs) which returns an inter-mediate key-value pair.

The Tasktrackers locally collect the intermediate key-value pairs and sort them by the intermediate key. Therefore the intermediate key must have an ordering imposed on them. The user can provide a custom ordering. The sort phase is run while the Map class is running. The space of intermediate keys are partitioned and each partition is assigned to one Reducer. It is possible for the user to change the partitioning scheme. Once the map phase is over each Tasktracker will retrieve (across the network) all values associated with the intermediate keys in its assigned partition. These are then merged together. This is called the *sort and shuffle* phase. The user supplied Reducer class is now executed. It iterates over intermediate keys and for each key, it is provided an iterator to loop over the associated values. The intermediate keys are sorted but the associated values are not. Usually, the Reducer summarizes the values but this is not required. The final key and value pairs are written to the output destination using the specified *Outputformat* and *Recordreader*, both of which can be provided by the user (see Fig 3.2 for a schematic).

Combining: The process of combining is an optimization to accelerate the MapReduce job. For a given cluster computer, the intermediate key,value output pairs of the maps are grouped by distinct keys. The user supplied reducer will then be called on that computer itself once for each distinct key. The output of the reduction, instead of the map output, is then sent to the sort and shuffle phase. For combining to work, the reduction operation should be associative and commutative. Typically, it decreases the size of the data to be sorted and shuffled which positively impacts the time to complete a MapReduce job.

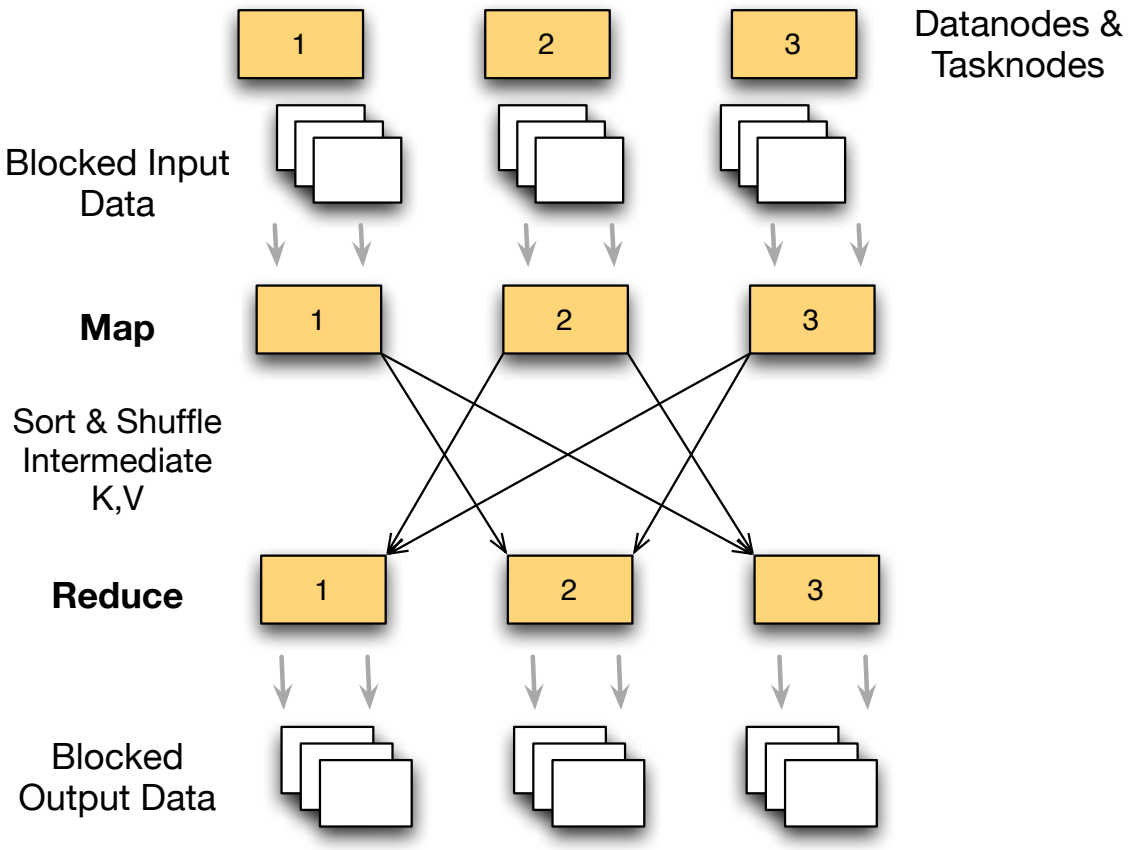


Figure 3.2.: The input is divided into blocks, mapped, sorted and shuffled and finally reduced

3.4 RHIPE

Hadoop provides several advantages to the R user. It provides (a) a scalable, fault tolerant distributed file system for storing many large files (b) a framework for the analysis of large and complex data across a cluster. There is a need for an analysis framework for massive data within the R environment. Though several packages for R exist for the parallelization of computation across cores and clusters none provide support for the above two features. Moreover, many packages that tackle the problem of applying numerical statistical routines are specific to small number of algorithms. Instead we discuss how the Hadoop Project offers a technology leap in computing with large data using the principle of Divide and Recombine. We shall first introduce R, describe the support for high performance computing and then introduce the R and Hadoop Integrated Processing Environment for the analysis of large data using the data set introduced in Chapter 1.

3.4.1 The R Project for Statistical Computing

R (R Development Core Team (2005)) is more than a language but also a software environment for statistical computing and graphics. It is the open source implementation of the S language developed by John Chambers at Bell Labs. The current version of R is 2.11 and runs on several operating systems and a variety of hardware platforms. The R language treats functions as first class objects (called closures), is lexically scoped and has lazy evaluation. It is dynamically typed and it is possible to evaluate strings as expressions. It supports object orientation via generic functions and its error handling mechanism is similar to that found in Lisp. The default version of R comes with support for linear and general linear regression, non linear regression, generalized regression, survival analysis and the functionality to generate publication quality graphs. It is the lingua franca for statisticians and academicians. Support for other features can be provided through packages which are written in the R language itself. For extra performance and interfacing to libraries in other languages, R

provides an API for interacting with the R language from C. There are about 2000 packages spanning Bayesian analysis, Clinical trials, Econometrics, Ecology, Finance, Spatial Statistics, Time Series Analysis and the Social Sciences to name a few topics.

3.4.2 R and HPC

Internally, R has not supported the parallelization of procedures across multiple cores. For example, the `sum` function to add the element of a vector, can be implemented to take advantage of multiple processor cores (e.g prefix scan). The `apply` family of functions which apply a given function to the rows or columns of a matrix (`apply`) or elements of a list (`lapply`) or a vector (`sapply`), in most cases can be implemented in parallel. Moreover, its parameter passing mechanism is *call by value*. This implies that objects get copied increasing the resident memory utilization. In effect, this reduces the size of data sets the R user can manipulate. For example, the default `lm` procedure for fitting linear models loads the entire data set into memory. Other software applications such as SAS use external memory algorithms which analyze the data without loading all of it in memory.

As mentioned before user contributed packages remedy many of these drawbacks. The package `rmpi` (Yu) supports the Message Passing Interface API (MPI) (Gabriel et al. (2004)) for multi-core and distributed computing. The `rmpi` packages expects a working MPI installation (e.g using OpenMPI or MPICH2). The user bootstraps worker nodes using `mpi.spawn.slaves`. Commands are executed remotely using `mpi.remote.exec`. `rmpi` provides a low level interface to MPI and can be difficult for the novice programmer to use. Moreover, it does not provide any support for computing across large data sets. The package `snow` (Tierney et al.), is a set of commands that depend on `rmpi`. This provides a much smaller set but cover a typical range of user commands e.g. `papply` which is distributed application of a function across elements of a list. Packages such `snowfall` (Knaus) provide support for starting the node resources, extended error checks and the ability to work in sequential mode

if no cluster is present. Other models of distributed computation such as Network Spaces (Gelernter (1985)) are supported by `nws` (REvolution Analytics). For a full list of R High Performance Packages (HPC) see (hpc).

Computing with large amounts of complex data is another significant problem. With the data sets rapidly scaling, the analyst is faced with the problems of (a) storing (b) querying (c) computing and (d) visualizing the data. Approaches to (a) have ranged from storing the data in many flat files taking into operating system limits on the number of files in directory. With large data sets it is essential for transparent access to the offline storage of networked computers. This has been tackled using networked file systems such as NFS. One drawback to this approach is that the computation is not aware of the locality of data, as a result the computation will be hampered by slow network transfers (Kielmann et al. (2001),Wattenhofer and Widmayer (1997)). Issues discussed in the aforementioned references are not considered by external memory regression and data management routines for R, contained in `biglm` (Lumley (2009)) and `bigmemory` (Michael, K. and Emerson, J. (2010)).

The default data file format for saving R data is not designed for moderately large data sets. All objects in memory are sequentially serialized to disk in a binary format. Meta information regarding the data set is not written to the file nor offset markers to indicate the location of the objects. Thus to read one object from an R data file, the user needs to read the entire data. Packages that support efficient and scalable data formats e.g. HDF5 ((HDF Group)) exist but have not replaced the default R data file format.

3.4.3 Overview of RHIPE

R and Hadoop Integrated Processing Environment (RHIPE) is an R package that provides a framework:

- To interact with the Hadoop Distributed File System(HDFS) e.g. read and write files created using RHIPE, to list, delete, copy and move files on the HDFS.
- To run R programs and compute with large data using a cluster of computers without having to manage the various vital housekeeping tasks involved in distributed computing.
- To submit, monitor and stop MapReduce jobs submitted from the R console.

All of this can be achieved using a flexible yet simple R commands and programming idioms. The package builds and installs on the UNIX family of operating systems (including RedHat Enterprise Linux and Fedora) and requires Hadoop installation of version higher than 0.21.

Installation

RHIPE is an R package, that can be downloaded at (RHIPE (b)). To install the user needs to

- Set an environment variable `$HADOOP` that points to the Hadoop installation directory. It is expected that `bin` folder in `$HADOOP` contains the Hadoop shell executable `hadoop`.
- A version of Google's `protobuf` (Google) greater than 2.3.0

Once the package has been downloaded the user can install it via

```
R CMD INSTALL Rhipe_version.tar.gz
```

where `version` is the latest version of RHIPE. The source is under version control at GitHub (RHIPE (a)).

3.4.4 Design and Features

RHIPE has three components: the R interface, the Java to R bridge and the engine written in C. The R interface has R functions (discussed in detail in Section 3.8) to interact with the HDFS. The R interface also has functions to submit computations across the cluster and monitor the status of the computations.

The R interface communicates with Hadoop via Java programs. When a function which interacts with Hadoop is called, a Java program is launched with a file containing the name of the function and its arguments (serialized to binary data with Protocol Buffers). The Java program, depending on the function, calls HDFS specific functions or launches jobs across the cluster. There is an alternative interface that communicates with Hadoop over Unix pipes. When the RHIIPE library is loaded via a call to `library`, RHIPE creates three pipes: for data from R to Java, from Java to R and one for errors. This approach bypasses the need for repeatedly starting a Java program and in some cases achieves a 50% speed up. However this latter approach is still in development and is yet to be documented.

Brief Outline of MapReduce Before we discuss the design of the Hadoop-R bridge, it is instructive to outline the flow of a MapReduce job.

In short (Dean and Ghemawat (2008)), a MapReduce program takes a set of *input* key-value pairs and returns a set of *output* key-value pairs. The value is the actual data to be stored whereas the key can be treated as meta data. However this definition is not rigid, the key and value can take on different roles depending on the problem. A statistician might view the key as the levels of a categorial variable or a combination of categorical variables (e.g. the cartesian product of 3 categorical variables A, B, C) that partition(s) the data into subsets. The value can be any data that will belong to the set defined by the levels of the categorical variable.

To generate the *output* key-value pairs, the RHIPE user needs to code the following:

Map This is an R expression (code fragment) that takes the *input* key-value pairs as input and outputs *intermediate* key-value pairs. Hadoop MapReduce will group all intermediate values for the same intermediate key I and send them to the Reduce expression - this stage includes bulk network transfer among the compute nodes.

Reduce This R expression takes as input an intermediate key I and receives a stream of intermediate values corresponding to this key. The role of the reducer is usually to summarize these values and return an *output* key-value pair (though it is possible for a reducer to return many key-value pairs). Each core across the cluster is assigned a number of intermediate keys to process. The Reduce expression is called for every distinct intermediate key and evaluated in parallel across the cores in the cluster.

The output key-value pairs are finally written to the HDFS in the user defined output destination.

The bridge component is used for the execution of MapReduce jobs. The code is based of Hadoop Streaming (Apache Foundation). It

- Creates the configuration file for jobs e.g. the location of distributed cache files.
- Defines the Java classes that will read in the data, to be used for the Map and Reduce phase, and save the data to the HDFS.
- Launch the MapReduce job across the cluster.

Both the Map and Reduce classes both have a helper member field. As mentioned previously, Hadoop willl decompose the work into splits (which is composed of key,value pairs) and call the Map class for every split. Before the Map class iterates over the key, value pairs, its `configure` method is called. In this method, a Java thread is launched. This thread launches the C engine (which evaluates the user's R code), sending data from Hadoop to R over the engine's standard input, reading data from the engine over standard output and monitoring for errors over its standard error. All

data is written in serialized binary form using Protocol Buffers. If the thread receives an R error over the standard error stream it cancels the running job.

Once the engine has started, RHIPE switches to different states depending on the stage of the computation. In the `configure` method, RHIPE sends a byte to the engine requesting it to run the user's `setup` expression (if given). Then RHIPE sends a byte to the engine informing it to start receiving key,value pairs belonging to the input split. Each element of the pair is written with a header (a variable length encoded integer) indicating the length of the serialized key and value. Thereafter, RHIPE iterates over the key,value pairs belonging to the split forwarding the data to the engine. As mentioned before, the helper thread monitors the engines standard output for the intermediate key,value pairs written by the user's R code. These are passed on to the MapReduce framework. When all the key,value pairs have been written, the bridge sends a byte to the engine requesting it to run (if provided) the `close` expression. Finally, the engine terminates. A failing in the design of RHIPE is that once the Map class executes its close method, any errors sent from the engine, though caught will not kill the task. The task is considered successfully completed if errors are not detected during the configure, the writing of key,value pairs and the close methods. A result of this is that for short Map phases, the job might appear to have succeeded whereas it has actually failed. RHIPE works around this by setting a flag describing the error. When the user is returned the result of the MapReduce job in the R console, the presence of the flag will indicate an error.

Similarly, during the Reduce phase, RHIPE starts in the `configure` state, launches the C engine and requests it to evaluate the `setup` expression. The bridge then sends the intermediate key followed by a stream of values. Once the bridge has iterated through all the keys and values, it sends the command byte to the engine to run the `close` expression after which the engine terminates. The RHIPE Reduce bridge has two different approaches to collating the intermediate key,value pairs.

- If the user does not provide an reduce expression, but would like the output of the map to be sorted by keys, or to merely collate the map output in to

smaller set of output files, the RHIPE Reduce bridge writes all the intermediate key,value pairs to the MapReduce framework and not to the engine. This is called *plain* or *simple reducing*.

- If the user does provide the reduce expression, the data is written to the engines standard input.

The engine is written in C and embeds the R interpreter. It reads data from the bridge and sends output on its standard output and error conditions on its standard error. Status updates and counter values (discussed below) are sent on standard error. The bridge gives the engine the user's Map expression (an R code fragment), the reduce expression and any setup/close expressions.

How RHIPE Evaluate the User's Code When the bridge instructs the engine to run in the setup state, the engine evaluates the map specific setup expression. The user defined R map expression is given two R lists: `map.values` and `map.keys` which contain the input values and keys respectively. Providing the user with lists allows for the possibility of vector operations. If a split contains several thousands of key,value pairs the user defined map expression will be called several times. For each call, the vectors `map.keys` and `map.values` will contain *rhipe_map_buffsize* number of key and values respectively. If a split has N pairs, the map expression will be called $\lceil N/rhipe\_map\_buffsize \rceil$ times.

The reduce is a named R expression consisting of three elements:

pre This is called when RHIPE receives a new intermediate key, I from Hadoop. The expression is provided with the variable `reduce.key` which contain the intermediate key. This is also available to the subsequent expressions. This expression is called before any intermediate values have been received. Use this expression to create any initialization code needed by the subsequent expressions.

reduce The `reduce` is called repeatedly when new intermediate values have been received. The intermediate values are sent in groups of size specified by the user. The values are not sorted and can appear in arbitrary order. The expression

is provided with `reduce.values`, an R list containing the intermediate values. For example, if intermediate key I, has $30,000$ values associated with it (emitted by the map expression), this expression is called sequentially 3 times within the same R process. With each call, the `reduce.values` is populated with $10,000$ values.

post An R expression called once all intermediate values for I have been received. This can be used to emit the output key/value pair.

For a given intermediate key, the reduce expression is executed in a single R session. Thus variables created in `pre` will be present in `reduce` and `post`. The function to return key-value pairs in RHIPE is `rhcollect` which takes a key and a value and can be called multiple times. Note, RHIPE enforces a limit on the size of the object that can be read. Objects of any size can be serialized, but if the serialized size is larger than 256MB, it cannot be read. This is a limit imposed by Protocol Buffers. Support for distributed *counters*, e.g a count of the number of occurrences of some event is provided via `rhcounter` and status updates through `rhstatus`.

rhcounter This is related to the Hadoop Counter API. Counters are distributed sums. The user can create groups and subgroups of counters. To increment a counter F that belongs to group G, call `rhcounter(G,F,n)` where G and F are strings and n is an integer.

rhstatus This function plays the dual role of providing information messages to the user (which can be viewed from the Hadoop JobTracker website) and at the same time informs the Hadoop Tasktracker that the R code is still running. If not informed, Hadoop will assume the code has an error in it (e.g. an infinite loop) and will kill the task, reassigning it to another Tasktracker.

Getting Data from RHIPE The `rhcollect` function at the end of the reduce expression writes data to the output folder on the HDFS. The function `rhread` takes an output folder and returns a list of key-value pairs. The raw serialized bytes are

first read from the HDFS and then unserialized in R. To improve the speed of the latter operation (which can take time), RHIPE can take advantage of the `multicore` package to unserialize in parallel.

RHIPE as a large key-value database The unit of data in Hadoop MapReduce key and its associated value. For unique output keys, the output can be treated as an out-of-core *hash table*. For example, we have stored 1.3MM data frames associated with unique identifiers across 140GB. The function to retrieve the values for a key is `rhgetkey` and the time to retrieve a few hundred keys in this example is a few seconds. When the number of keys are in the billions, one may consider the use of dedicated distributed key-value stores e.g HBase and MongoDB.

Data Types RHIPE supports many R types, including scalar vectors (complex, real, integer, strings and booleans), `NA` of all types, lists and attributes. These include factors, data frames, matrices, class attributes and expressions but excludes functions, environments, pairlists etc. For these object types, the R `serialize` function can be used to save data.

Side effect files: Not all results can be saved as output values (there is a 256MB limit). Results can also be written to files at the end of the job. These files (e.g. PDFs and R data sets) can be automatically copied to the output folder on the HDFS at the end of the job.

Simulations: For simulations, the user can implement a distributed version of `lapply` within RHIPE (see 3.7). For very short lived computational units, this is not as fast as MPI based alternatives. For a large number of trials, RHIPE does make for a viable alternative. Also with the Hadoop back end, RHIPE offers fault tolerance and load balancing automatically.

Language Interoperability: No one language or tool is perfect for massive data. To get best results the data sets created by such tools should be language agnostic. Data written using RHIPE is serialized using Google's Protocol Buffers (Google), which has interfaces for Java, C, Python and other languages. The upshot of this

that input data sets can be created in other languages and data created with RHIPE can be read in other languages.

3.5 Analyzing Large Data Sets Using RHIPE

I will cover the use of RHIPE by studying a large data set. Through this section we shall cover functions of RHIPE and their various uses. We will also see how to debug error conditions using RHIPE.

3.5.1 Divide and Recombine

Our goal is to analyze massive data sets with the same level of detail as in the case of smaller data sets. There are several thousands of statistical algorithms, not all of which have external memory solutions. Our approach is to partition the data into subsets. This has been discussed in Chapter 1. We can apply statistical routines across subsets, or visualize a subset. Often the results will need to be aggregated: displays will need to be tiled and results of computations need to be displayed. This is the recombine. Computing subsets and recombining has been used in the R community. For example, the `xyplot` command in the `lattice` package can divide the data frame into subsets by the values of a categorical variable. A display is generated for each subset and aggregated into a final trellis visualization. With the advent of computing frameworks such as Hadoop DFS and MapReduce, the Divide and Recombine approach can be generalized to handle massive data sets across a cluster across arbitrary subsets.

3.5.2 Network Data Set

We set up a trace collection on a subnet of the Purdue Statistics Department. A monitor collected traces with `tcpdump` running on a server connected to the span port of a switch that sees all traffic in and out of the subnet and between two virtual

lans making up the subnet. This data set was used in Chapter 2. It is approximately 150GB of text data. Each row of text has 13 columns:

hash Connection identifier (MD5 hash of the sorted tuple of source IP and port, destination IP and port), all packets in a connection have the same hash.

isrequester This takes the value 1 if the packet is from the machine that initiated the connection, 0 otherwise. The machine that sent the first SYN packet is the initiator, thereafter any packet with the same hash and source IP that is equal to the initiator machine has `isrequester` equal to 1.

sip The source IP address.

dip The destination IP address.

srcport The source port number.

dstport The destination port.

srcporthr A human readable version of source port. This is the entry from `/etc/services` which maps port numbers and protocols to application names e.g. 22 is mapped to SSH.

destporthr The human readable version of destination port.

datasize The number of bytes in the payload of the packet: this ignores the header content.

hdrsize The size of the header in bytes.

timeOfPacket The time of arrival of the packet.

flagsnum A binary OR operation over the TCP flags, computed as:

$$I_{FIN} + I_{SYN} * 2^1 + I_{RST} * 2^2 + I_{PSH} * 2^3 + I_{ACK} * 2^4$$

where I_X is equal to 1 or 0 if the corresponding flag X is on or not in the packet.

ACKN The acknowledgement number.

SEQN The sequence number.

Before we can begin to compute with it, we need to store the data on the HDFS and then store them as R data objects. There are several ways the data set can be stored. Storing the data is equivalent to creating different partitions of the data.

1. As blocks of N rows and 13 columns. This hides any structure present in the data set. The 1.2 billion rows will be stored as tables of N rows.

2. De-multiplex the data set, so that packets belonging to the same connection are stored as one R object. Note, we might have very long connections, which can occupy millions of packets. If stored as one R object, it would require many hundreds of megabytes. Instead we save the connection as multiple R sub objects indexed by the $hash$ and a number ranging from 1 to $round(N/M) + 1$ where N is the number of packets in the connection and M is the number of rows of the sub object (we intend to save it as data frames hence M is the number of rows of the sub data frame). The packets sub data frame 1 will, in all likelihood, not be time ordered before those of sub data frame 1. In the reducer, though the keys are in order, the values are not. We cannot expect to accumulate the millions of packets corresponding the connection, sort them by time and then save to disk.

3. De-multiplex the data set, but store the sub data frames in time order. Useful if we need to analyze by the first T seconds of every connection.

4. Save the first (by time) F packets of every connection.

3.5.3 Converting text files to R objects

Network data has a natural partition: the connection. Option (1) does not take advantage of this; since we wish to analyze by connection, this is not an attractive

format. Option (2) stores the connections as blocks of data frames each with up to e.g. 10,000 rows. Each data frame is indexed by the *hash* and an integer from 1 to the number of blocks required to save the connection. If we wish to compute across the entire connection, this approach serves us well. We reach a hurdle if we desire to compute on the first F packets, e.g. in Chapter 2 we applied the keystroke algorithm to the first 1500 packets. Option (2) does not readily provide us a way to get this. This because the each intermediate value is a packet and the packets for a connection hash do not arrive at the reducer ordered by time. Thus each sub data frame contains 10,000 packets but not ordered by time. Sorting this table will not help since the sub tables corresponding to a connection need not be sequential in time. Option (3) stores the data in blocks indexed by hash and integer, where packets in data frame i have time stamps less than every packet in data frame $j > i$. Option (4) is similar to Option (3) though we only store the first F packets. We will outline the code for Options (1),(2) and (4). It should be pointed out that our guideline is the following: *d*isk is cheap. So, ideally, one should store the data in different ways without worrying about running out of disk space.

Option (1) Option (1) is useful when the data has no obvious partition. For example the airline data (air) in which each line of data is a record of a passenger flight, offers many different partitions but no single partition is better than the other for repeated analyses. The 150 GB of text files for the network data is stored on the HDFS in blocks of 128MB. Each block is replicated across 3 different computers. When the job to convert text to R objects is started, the map expression (called mapper) will be run on all the cores of the cluster. Each mapper will process one split (in this case one split is equivalent to one block of 128MB of text data). For textual input data, the value is a line of text (including the newline) and the key is the byte offset from beginning of the file (which will not be of use to us).

The code in (3.3) is called across cores on all cluster computers. Each core is given a series of splits to process. Each split consists of key,value pairs. RHIPE places these key,value pairs in the `map.keys` and `map.values` (two R lists). The lengths of the

```
map <- expression({
  textline <- strsplit(unlist(map.values),"[[:space:]]")
  textline <- do.call("rbind",textline)
  colnames(textline)<- c("hash","isrequester","srcip","dstip"
                         ,"srcport","dstport","srcporthr"
                         ,"dstporthr", "datasize","timeOfPacket"
                         ,"flagsnum","ackn","seqn")
  textline <- textline.frame(hash=textline[,'hash']
                             ,isrequester=as.integer(
                                              textline[,'isrequester'])
                             ,srcip = textline[,'srcip']
                             ,dstip = textline[,'dstip']
                             ,srcport = as.integer(textline[,'srcport'])
                             ,dstport = as.integer(textline[,'dstport'])
                             ,srcporthr = textline[,'srcporthr']
                             ,dstporthr = textline[,'dstporthr']
                             ,datasize = as.integer(
                                            textline[,'textlinesize'])
                             ,timeOfPacket = as.numeric(
                                           textline[,'timeOfPacket'])
                             ,flagsnum = as.integer(textline[,'flagsnum'])
                             ,ackn = as.numeric(textline[,'ackn'])
                             ,seqn = as.numeric(textline[,'seqn'])
                             ,stringsAsFactors=FALSE)
  rhcollect(map.keys[[1]],textline) })
```

Figure 3.3.: Option 1: R code to convert text data to R objects

lists can be configured but the default length is $10,000$. The values in the `map.values` correspond to contiguous lines of text in the input file. Once the number of key,values have reached the buffer size, the map expression is called. Once the expression has finished evaluating, the lists are filled with the next $10,000$ key,value pairs and the map expression is evaluated once again in the same R session. This is repeated until there are no more key,value pairs for that split. In this example, `map.values` has 10,000 lines of text. For efficiency we unlist the vector and call `strsplit` to tokenize the lines into character vectors of 13 elements. This list of tokenized lines is aggregated into a character matrix. We define the column names and convert it into a data frame. Note the this will fail if all the lines do not have the same number of columns. It is essential we set `stringsAsFactors` to $FALSE$: R stores factors as integers with character labels. This is useful for analyzing smaller data sets but is inefficient for larger ones. Finally, the function `rhcollect`, a RHIPE function, sends the final key,value pair to the MapReduce framework to either save as output or pass onto the reducer. The function `rhcollect` can be called any number of times. The type of data that can be sent include

- Atomic vectors: character, integer, real, complex, raw,logical including NA values.
- Lists of the above and lists.
- Objects with attributes, thus this includes: matrices, data frames, factors and class attributes.

Note, closures and environments will not be successfully serialized. The objects will be replaced with NULL. To transmit such data, serialize using R's `serialize` function. In the above, we are blocking the data into arrays of $10,000$ rows: there is nothing unique about a block.

We will not need a user defined reduce. Each split is processed by the map and written to disk. With 150GB, with each split equivalent to 128MB, there are 1200 splits and therefore 1200 output files. We can request a *plain reduce* to aggregate the

```
z<-rhmr(map=map,
     ifolder="/net/d/textinput",
     ofolder="/net/o/blocks",
     inout=c("text","sequence"),
     mapred=list(mapred.reduce.tasks=0))
rhex(z)
```

Figure 3.4.: R code to start a MapReduce job

files into a few group of files, but this takes additional time. To call this expression we can run the code in (3.4).

The call for `rhmr` specified the map expression (always required), the reduce expression (plain reducing if none given), the output and input folder (the input folder is not required, see 3.7) and input and output formats. The actual command is executed via a call to `rhex`. The source to the job is text and the output is binary data (which is indicated by "sequence"). The advantage to binary data is that a variety of R objects, including their attributes, can be saved. Finally Hadoop specific information is sent via the `mapred` argument which is a named list. The variable `mapred.reduce.tasks` instructs Hadoop how many reduce tasks to run - when this is zero, the output of map is the written to the output folder, there is no sort and shuffle phase, and the job completes. For a full list of Hadoop specific variables see (had). All values passed via `mapred`, will be converted to characters. In any user provided R expression, their values can be read with the `Sys.getenv`. For example, should the user want to pass the value of some variable (there are several ways to do this) whose object size is less than 32MB, one approach is outlined in (3.5)

In any expression (e.g. the map expression), the value can be obtained as in (3.6) RHIPE adds all variables in the `mapred` argument to the job configuration. This file is loaded in memory for every split task. Large job configurations increase the time to

```
rhmr(..., mapred=list(usefulVar =
        rawToChar(serialize(usefulVar,ascii=TRUE,NULL))
```

Figure 3.5.: R code to send parameters to MapReduce jobs

```
usefulVar  <- unserialize(charToRaw(Sys.getenv('usefulVar')))
```

Figure 3.6.: R code to retrieve parameters to MapReduce jobs

start the job. Hence, this is not a good approach for anything more than a megabyte. For such objects, consider the use of the shared files (see 3.8.2).

Option (2) With analysis of the network data mostly by connection, storing the data by connection makes for efficient querying. In this approach, we will de-multiplex the data so that packets belonging to the same connection are stored in at least one data frame. For connections that are millions of packets long and too big to fit in RHIPES key,value size limit (265MB), we store the packets for one connection across multiple data frames, each data frame containing at most 10,000 packets. The data frames are indexed by connection hash and an integer running from 1 to $\lceil N/10000 \rceil$. The data is partitioned by *hash*, see (3.7).

```
map <- expression({
  textline <- strsplit(unlist(map.values),"[[:space:]]")
  textline <- do.call("rbind",textline)
  apply(textline,1, function(value)
      rhcollect(value[1],value))
      })
```

Figure 3.7.: R code to split lines of text into columns

Packets with the same *hash* are aggregated in the reduce expression and every 10,000 packets are written to disk indexed by $(hash, integerId)$ pairs (see (3.8)). Before we begin aggregating a connection, the `reduce$pre` expression defines a function `doempty` and initializes the per connection integer counter `integerid` to 1 (this is called once for every new connection). The function `doempty` creates a data frame from the text entries in `map.values`. The expression `reduce$reduce` is called for every $10,000$ packets (though this can be set to any number by setting `rhipe_reduce_buff_size` in the `mapred` argument of `rhmr`).

We can run this code across the network data using code outlined in (3.9).

How do we access the connections? We will need a scheme to access the data by connection. The first approach is to store the data as a *map* file. The map file format is a directory containing two files. One with the actual data and another with a sample of the key,value pairs. Each reducer creates its own map file for output. The intermediate keys received by the reducer are sorted. Assuming the user passes `reduce.key` unmodified to `rhcollect`, the keys in the output file will also be sorted. This sample acts as a quick index into the data file for querying. Rather than iterating through the data file for the requested key, the reader loads the index file, finds the closest index entry less than or equal to the key and from there iterates through the data file. It is possible to change the number of entries inserted into the map file index. The map file is not intended as a query system for many billions of key,value pairs and for such cases requires, a dedicated distributed key,value databases such as HBase and Hypertable (see (hba; hyp)) is recommended. As described above, a condition for the map output format to work is that the output keys (i.e the call to `rhcollect`) in the `reduce` must be in the same order as the intermediate keys. Hadoop will provide the reduce expression with keys in sorted order, changing the key in the reduce expression will invariably change the order that they arrived in. A map file is only effective as a fast key retrieval system only when the keys are guaranteed to be in sorted order - a guarantee that cannot be met if the key written to disk with `rhcollect` in the reduce expressions is different from `reduce.key` . Hence, we cannot use *map* as the output

format in the above call to `rhmr`. We need to run one more pass over the above data set to save it as a map file, see (3.10) The above code, is an *identity* map. Because a reduce is present (since `mapred.reduce.tasks` is not zero), the output keys from the map will be sorted, shuffled and written to disk. We can now access connections: for example to access a connections with *hash* in {`0000131163cd7e167b72ee987311ab224b54f`, `00002e3a6481673fc1845b86a8f750f0fded2`}, we call the function `rhgetkey`. The function `rhgetkey` takes a list of keys and a path to the location of the map files. It returns a list of lists, each of which consists of a key,value pair. Recall, each key is the *hash* and a number. For the complete data for a connection we would have to request all the pairs $c(hash, n)$ where n is from 1 to $\lceil N/10000 \rceil$. Another approach is to subset the connection from the database (see (3.12)) We introduce a few new concepts here.

`rhsave` The list of keys are saved in the variable `mykeys` and saved in an R data set via the call to `rhsave` which works exactly as the R command `save` but its destination is the HDFS

`shared` The expression `map.setup` is a code fragment that is evaluated once for every split. We use this to load the data set containing the connection hashes. The data set is distributed to the local storage of the cluster computers *once* at the beginning of the computation. The location of the destination on the HDFS is specified in the call to `rmr` via the argument `shared.files`. Hadoop will copy this file to the current directory of the R engine, hence the call to load in `map.setup` specifies the current directory.

We advocate the use of environments (see line 1) since these are implemented using hash tables. For millions of comparisons, they will be faster than the `%in%` operator. We also use vector operations: we subset `map.keys` having checked which keys in `map.keys` match the ones in `mykeys` (see lines 11-14). These are then written to disk. Though slower than the previous method (i.e using map file formats), this will scale across any amount of data.

Option (3) Some algorithms and displays only need the first F packets of the connection. For example, the keystroke algorithm was applied to the first 2000 packets: if keystrokes is not detected in the first 2000 packets it is quite unlikely there is a human present. Visual displays were vital to the development of the algorithm. Patterns in the inter-host communication dynamics are revealed in graphical displays that are hidden in numerical summaries. The following code demonstrates how we save the first 1500 packets of a connection from the R data frames (stored in the previous example). The code is in (3.13).

The map expression associates each connection sub data frame with the same connection *hash*. Thus all the data frames the make up a connection are aggregated as a whole. In the reducer, before the start of a new connection, the object `first1500` is initialized. In the second part of `reduce`, the data frames are combined together and sorted by time and the first 1500 (or the entire data frame if less than 1500 rows) are selected. This is then saved using `rhcollect`.

We have used a combiner: instead of sending all the data frames corresponding to a connection (possibly thousands), the reducer is evaluated on the computer running the map expression. That is, the reducer runs on a subset of data frames associated with the connection hash. When all the data has been partitioned by the map and local reductions have completed aggregating the map output, the output of the local reduction is sorted,shuffled and sent to the Hadoop reduce phase for a final reduction and saving to disk. This saves on excessive network transfer and can greatly accelerate the time to complete the data creation.

The output from Option (2) serves as the input to Option (3). Each value written in Option (1) is data frame of $10,000$ packets. If the value of `rhipe_*_buff_size` is left unchanged, then $10,000$ data frames each of possibly 10,000 rows will be loaded in memory. Decreasing these numbers defends against memory overflows. However, the map and reduce expressions will be called many more times than before. R function calls can be costly and more calls decreases possibilities of vectorizing operations. Increasing the length of `rhipe_*_buff_size` increases the scope of vector operations

at cost of large resident memory usage and time to create the vector. In our network data a vast majority of the connections have much fewer than 10,000 packets. We have been very conservative in our numbers.

Since we have not changed the key in the reduction, we can use a map output format. The connections can now be queried by *hash* e.g.

```
rhgetkey(list('158d4bca1ac237a2b0f81420592fa704aee5f'),'/net/o/first1500')
```

To summarize, we have converted the text files to R objects in such way that we can compute at the connection level. We can compute summaries across all packets for a connection (using the data stored via Option (2)) or analyze the first 1500 packets of connection (by Option (3)).

3.5.4 Computing Summary Information

The connections are stored as component data frames of $10,000$ packets each. Without a means to find the total number of packets for a connection, we will not be able to access all the component data frames that make up the connection. Besides, we would like to describe our connections using the standard five tuple summary: source port and IP address, destination port and IP address, duration of connection. We define the client of a connection as the IP address that sent the first SYN packet without an ACK. In cases where the first SYN packet cannot be found the connections origin is not known. We can calculate bytes and packets sent in both directions but will not know which IP address initiated the connection. Using heuristics, it is possible to figure out the client even when the required SYN cannot be found e.g., for SSH connections, the IP address associated with port 22 is the server , the IP address associated with the higher port is the client. We do not use these heuristics. To compute the five tuple summary, the connection table is partitioned by the source IP and ports. For each subset (at most two) compute the bytes and packets transferred, time when the connection started and duration of connection indexed by connection

hash. All operations are commutative and associative. For example, the *min* operator to find the minimum of a set of elements $x_1, \ldots, x_n$

$$min(x_1, x_2, x_3, \ldots, x_n) = min(min(min(x_1, x_2), min(x_3, x_4)), x_5, x_6, \ldots, x_n)$$

The values computed for the component data frames can be locally combined using a combiner. The range operator is not associative but the minimum and maximum are. The variable `isrq` is a flag that indicates whether the direction initiated the connection. If it is 1 in the output, then the associated direction i.e source IP, port initiated the connection. It is possible for it to be zero for both directions when it cannot be determined which machine initiated the connection.

In the reducer (3.15), we combine the sums, minimums and maximums into a matrix. It the reduce expression is evaluated locally as a combiner, on partial data emitted in the map, the variable `rhipe_combiner` is 1, hence we return the minimum of the minimums. Otherwise, the reduce expression has ben invoked after the sort and shuffle phase as part of the Reduce phase. The range (giving us the duration) and the start time of the connection is written to the output destination. Notice, in the negative branch of the `if` condition names are attached to the object and not in the positive branch. The output of the negative branch is written to disk once, whereas the output of the first branch is exchanged between computers. Not including names reduces the amount of data exchanged during the computation.

The code in (3.14 and 3.15) compute the summary statistics for each direction of a connection indexed by hash, source IP and port. Some connections have packets in one direction, e.g. probes where the client attempts to establish a connection with the host but is denied. The final data set will have at most two rows for every connection hash. To aggregate these into one object per connection hash, e.g. a row in a table indexed by connection hashes, map the summary objects indexed by connection hash, source port and IP to objects indexed by the hash. The number of distinct intermediate keys is equal to the number of unique connection hashes. Each reduce expression will receive up to two summary objects for each connection hash: one for each direction of the connection. These are aggregated as a single object.

3.5.5 Visualizing Subsets of the Data

Summaries do not substitute a detailed analsys of the data. Visualization tools are the first step towards a comprehensive analysis of the network connection data. We create static displays that work at two levels as discussed in Chapter 1: for detailed inspection and gestalt formation. The two types of displays are: time series displays of the packet bytes against the time of packet and against the order of the packet. (Cleveland and Sun (2000); Cao et al. (2002)) discuss such displays, however we add elements to enhance pattern detection. In a network connection over TCP/IP, one machine is the client and the machine it connects to is the server. The client initiates the connection by sending a SYN packet: a packet without data but the TCP SYN flag is sent. The server responds with a ACK and the client responds with an ACK to the SYN-ACK. This is called the three-way handshake (Comer (2006)). The connection ends when a packet has the RST flag or a FIN flag set. The displays plot the log (base 2) of $1 + datasize$, with the client packets in red, directed vertically downwards and the server packets in blue directed vertically upwards. The color and direction difference help delineate pattersn in the data (see MacDonald (1999)). The display includes tick marks, corresponding to ACKS,Duplicate ACKS, SYN, FIN and RST flags, in regions below the packet size bars. These are placed on a grey colored background to distinguish the region from the main data region. In the first display, the packet sizes are displayed against their sequential order. In the time order display, the connection packets are graphed against transformed time, i.e. let t_i be the time of arrival of the i'th packet. Let ϕ be a function from R^+ to R^+, where R^+ is the positive real line. Modified time τ_i is defined as

$$\tau_0 = t_0 \tag{3.1}$$

$$\tau_i = \tau_{i-1} + \phi(t_i - t_{i-1}) \tag{3.2}$$

Transforming time stretches the distance between consecutive packets. Transformations must be carefully chosen so that the difference in inter-arrival times do not

visually disappear. Finally, we display approximately 200 packets per panel and split a connection across several panels.

Packet order displays capture the communication dynamics of the two endpoints in a connection. It quickly captures the relative data transferred between the two endpoints and detailed inspection also reveals transmission failures (by the presence of duplicate ACKs). Time order displays capture the differences inter-arrival times of the packets. These reveal differences in short and long duration connections and periods of no communication between the connection end points and assist in distinguishing automated traffic from human initiated traffic. Figure 3.16 is packet order display for two SSH connections. The displays in Figure 3.19 are demonstrate the effect of ϕ on time.

3.5.6 Running the Designed Experiment for the Keystroke Analysis

The Packet-Dynamics Rules have 8 statistical tuning factors, as described in Chapter 2, that are thresholds for the variables used in the rules. The factors, their mathematical notation, and their units of measurement are shown in the first 3 columns of Table X of Chapter 2. Performance of the algorithm is measured by the 4 responses — `fp.not22`, `fp.22`, `fn.22`, and `fn.22script`. We ran a multi-response 3^{8-3} minimum aberration fractional factorial design of experiments (Xu (2005)) to determine the dependence of the responses on the levels of the tuning factors.

In the course of our pilot studies of performance we developed insight about ranges of the statistical factors for which the performance is quite good. Based on this we designed and ran a multi-factor fractional factorial designed experiment that varied all 8 factors in a systematic way in a region deemed to have reasonable performance, and studied how the above 4 responses changed with the values of the factors.

The experiment used the 1680 connections remaining after application of the SSH-Protocol Rules to the total 1,021,336 connections. For Test Regimens 1 to 4, remaining are 36, 195, 174, and 1275 connections, respectively. While only 0.16% of the

connections remain, an absolute number of 1680 would be far too many for security analysts to review, making the Packet-Dynamics Rules critical. The Packet-Dynamics Rules were applied to the first 1500 packets of each of the 1680 connections.

The first 1500 packets of the 1680 connections are stored as individual objects on the HDFS. The design has $243 = 3^{8-3}$ trials, each trial is a separate map job which assigns the computation of the keystroke algorithm of the 1680 connections to the computers in the cluster. A data frame of connections and number of detected keystroke packets are written to the HDFS.

There are two approaches for parallelizing the $408,240 = 243 * 1680$ trials

1. Divide the 408,240 trials into N chunks of size $\lceil 408,240/N \rceil$ each. In the map, each connection is replicated 243 times (once for each list of algorithm parameters) with keys from a simple random without replacement from $\{1, 2, \ldots, N\}$. The reducer receives all connections with the same number k and applies the algorithm sequentially to all. We chose $N = 50,000$. However the drawback to this approach is the amount of data sent from the map to the reduce. The 1680 connections take up 94MB, but because each connection is replicated 243 times, approximately, 3.4GB of data is sent on the network. Clearly this will not scale with the number of connections.

2. The second approach makes use of RHIPE's asynchronous job launch feature. We submit 243 jobs corresponding to a list of algorithm parameters. Each job is launched asynchronously, i.e the job is launched on Hadoop and the R shell returns to the user. Hadoop will queue the jobs to run one after the other, and share resources among the jobs. After submitting all the jobs, the R user can wait for all pending trials to complete and then read the results, as demonstrated in (3.20).

3.6 Streaming Computations in a Distributed Environment

Streaming computations are inherently sequential. The update of a function at time T depends on time points $t < T$. However RHIPE can be used for parallel sequential computations.

Some algorithms are left associative in their operands $t_1, t_2, \ldots, t_n$algorithm that computes the inter-arrival times of time series data for different levels of a categorical variable. That is, the triangular series $t_{k,1}, t_{k,2}, \ldots, t_{k,n_k}$ where k takes the levels of a categorical variable C (which takes the values $1, 2, 3, \ldots, m$). The input are pairs $(i, j), i \in \{1, 2, \ldots, m\}, j \in \{t_{ik}\}$. In the following code (see 3.21), the data structure F is updated with the *datastructure* contained in the values (see the map). The set of *datastructure* are indexed in time by the *timepoint* - they need to be sent to the reducer (for a given level of the categorical variable *catlevel*) in order of time. Thus the map sends the pair $(catlevel, timepoint)$ as the key. By using the *part* parameter (see line 39) **all** the data structures associated with the *catlevel* are sent to the same R reduce process. This is vital since all the component R expressions in the reduce are run in the same process and namespace. To preserve numeric ordering we insist on the special map output key class (see line 38). With this special key class, we cannot have a map output format. In the reduce, the setup expression *redsetup* is run upon R startup (the process assigned to several keys and their associated values). Then for each new intermediate key $(catlevel, timepoint)$, it runs the *pre*, *reduce* and *post*. The lack of a *post* is because we have exactly one intermediate value for a given key (assuming the time points for a category are unique). The *edclose* expression is run when all keys and values have been processed by the reducer and R is about to quit.

3.7 Task Parallel Jobs using RHIPE

Simulations are an example of task parallel routines in which a function is called repeatedly with varying parameters. These computations are processor intensive and

consume/produce little data. The evaluation of these tasks are independent in that there is no communication between them. With N tasks and P processors, if $P = N$ we could run all N in parallel and collect the results. However, often $P << N$ and thus we must either

1. Create a queue of tasks and assign the top most task on the queue to the next free processor. This works very well in an heterogeneous environment e.g. with varying processor capacities or varying task characteristics - free resources will be automatically assigned pending tasks. The cost in creating a new task can be much greater than the cost of evaluating the task.

2. Partition the N tasks into n splits (see 3.3.2) each containing $\lceil N/n \rceil$ tasks (with the last split containing the remainder). These splits are placed in a queue, each processor is assigned a splits and the tasks in a split are evaluated sequentially.

The second approach simplifies to the first when $n = N$. Creating one split per task is inefficient since the time to create,assign launch the task contained in a split might be much greater than the evaluation of the task. Moreover with N in the millions, this will cause the Jobtracker to run out of memory. It is recommended to divide the N tasks into fewer splits of sequential tasks. Because of non uniform running times among tasks, processors can spend time in the sequential execution of tasks in a split σ with other processors idle. Hadoop will schedule the split σ to another processor (however it will not divide the split into smaller splits), and the output of whichever completes first will be used.

RHIPE provides two approaches to this sort of computation. To apply the function F to the set $\{1, 2, \ldots, M\}$, the pseudo code would follow as (here we assume F returns a data frame) (3.22).

Here F is applied to the numbers $1, 2, \ldots, M$. The job is decomposed into 1000 splits (specified by `mapred.map.tasks`) each containing approximately $\lceil M/1000 \rceil$ tasks. The expression, FC sequentially applies F to the elements of `map.values` (which will contain a subset of $1, 2, \ldots, M$) and aggregate the returned data frames

with a call to `rbind`. In the last line, the results of the 1000 tasks (which is a list of data frames) are read from the HDFS, the data frame are extracted from the list and combined using a call to `rbind`. Much of this is boiler plate RHIPE code and the only varying portions are: the function F, the number of iterations M, the number of groups (e.g. `mapred.map.tasks`) and the aggregation scheme (e.g. I used the call to `rbind`). R lists can be written to a file on the HDFS(with `rhwrite`), which can be used as i input to a MapReduce job . All of this could then be wrapped in a single function:

```
rhipe.lapply(function, input, groups=number.of.cores, aggregate)
```

where `function` is F, `input` could be a list or maximum trials (e.g. M). The parameter `groups` is the number of groups to divide the job into and by default is the number of cluster cores and `aggregate` is a function to aggregate the intermediate results. With this function, the user can distribute the `lapply` command and rely on Hadoop to handle fault-tolerancy and the scheduling of processors in an optimal fashion.

3.7.1 A Note on Random Number Generators

RHIPE does not include parallel random generator e.g. Scalable Parallel Random Number Generators Library and the `rstreams` package for R (L'Ecuyer and Leydold; Mascagni and Srinivasan (2000)). Parallel RNGs can create streams of random numbers that are not correlated across cluster computers (i.e enforce 'statistical independence') and ensure reproducibility of streams for research. RHIPE can guarantee independent streams since each task has a unique identifier obtained from the environment variable *mapred.task.id*. Since the identifier is unique for every task it can be used to seed random number generators. This cannot be used for reproducible results. There is ongoing work to integrate parallel random generator packages for R with RHIPE.

3.8 Functions in RHIPE

Functions in RHIPE fall under filesystem related or MapReduce programming related. We discuss each function in its own section. The syntax for the call is declared first followed by the description of the function.

3.8.1 Filesystem Commands

File Deletion

```
rhdel(folders)
```

This function deletes the folders contained in the string vector `folders`, which are located on the HDFS. The deletion is recursive, so all subfolders will be deleted too.

Listing Files

```
rhls(path, recurse=FALSE)
```

Returns a filesystem information for the files located at `path`. If `recurse` is TRUE, `rhls` will recursively travel the directory tree rooted at `path`. The returned object is a data frame consisting of the columns: permission, owner, group, size (which is numeric), modification time, and the file name. The `path` may optionally end in '*' which is the wildcard and will match any character.

Copying from the HDFS

```
rhget(src,dest)
```

Copies the files (or folder) at `src`, located on the HDFS to the destination `dest` located on the local filesystem. If a file or folder of the same name as `dest` exists on the local filesystem, it will be deleted.

Copying to the HDFS

`rhput(src,dest)`

Copies the local file called **src** (not a folder) to the destination `dest` on the HDFS.

Copying on the HDFS

`rhcp(src,dest)`

Copies the file (or folder) `src` on the HDFS to the destination `dest` also on the HDFS.

Writing R data to the HDFS

`rhwrite(list,dest,N=NULL)`

Takes a list of objects, found in `list` and writes them to the folder pointed to by `dest` which will be located on the HDFS. The file `dest` will be in a format interpretable by RHIPE, i.e it can be used as input to a MapReduce job. The values the list of are written as key-value pairs in a SequenceFileFormat format. Since the list only contains values, the keys are the indices of the value in the list, stored as strings. Thus when used as a source for a MapReduce job, the variable `map.keys` will contain numbers in the range $[1, length(list)]$. The variable `map.values` will contain elements of `list`. Providing a value N instructs RHIPE to write the list values across N files. Thus if N is 10 and `list` contains $1,000,000$ values, each of the 10 files (located in the directory `dest`) will contain $100,000$ values.

Reading data from HDFS into R

`rhread(files,type="sequence",max=-1,mc=FALSE)`

Reads the key,value pairs from the files pointed to by `files`. The source `files` can end in a wildcard (*) e.g. $/path/input/p*$ will read all the key,value pairs contained

in files starting with p in the folder */path/input*. The parameter `type` specifies the format of `files`. This can be one of `map` or `sequence` which imply a MapFileFormat or a SequenceFileFormat respectively. Thus data written by `rhwrite` can be read using `rhread`. The parameter `max` specifies the maximum number of entries to read, by default all the key,value pairs will be read. Setting `mc` to TRUE will read the key,value pairs in parallel using the `multicore` package (Urbanek). The user must have first loaded `multicore` via call to library. This often does accelerate the process of reading data into R.

Reading Values from Map Files

```
rhgetkey(keys, path)
```

Returns the values from the map files contained in `path` corresponding to the keys in `keys`

3.8.2 MapReduce Related

Creating a MapReduce R Object

```
rhmr(map,reduce=NULL, combiner=FALSE,
     setup=NULL,cleanup=NULL,
     ofolder='',ifolder='',
     inout=c("text","text"),mapred=NULL,
     shared=c(),jarfiles=c(),
     partitioner=NULL,copyFiles=F,
     N=NA,opts=rhoptions(),jobname="")
```

map The map is an R expression (created using `expression`) that is evaluated by RHIPE during the map stage. For each task, RHIPE will call this expression multiple times. If a task consists of W key,value pairs, the expression `map` will be

called $\lceil \frac{W}{\text{rhipe\_map\_buffsize}} \rceil$ times. The default value of rhipe_map_buffsize is $10,000$ and is user configurable. Each time `map` is called, the vectors `map.keys` and `map.values` contain rhipe_map_buffsize keys and values respectively.

reduce The general form the Reduce phase is best explained with this pseudo code

```
1:while more_keys_present?
2:   let reduce_key := get_new_key()
3:   ...
4:   while more_values_for_key_present?
5:     let value := get_new_value_for_key
     ...
6:   end while
7:   ...
8:end while
```

Each Reduce task is a partition of the intermediate keys produced as the output of the Map phase. The above code is run for every Reduce task. RHIPE implements the above algorithm by calling the R expression `reduce$pre` at line 3. In this expression, the user will have the new key present in `reduce.key`. After which RHIPE will call `reduce$reduce` several times until the condition in line 3 is false. Each time `reduce$reduce` is called, the vector `reduce.values` will contain a subset of the intermediate map values associated with `reduce.key`. The length of this vector is a default $10,000$ but can be changed via the `rhipe_reduce_bufsize` option. FInally when all values have been processed, RHIPE calls `reduce$post` at line 7. At this stage, all intermediate values have been sent and the user is expected to write out the final results. Variables created in `reduce$pre` will be visible in the subsequent expressions. Thus to compute the sum of all the intermediate values, see (3.23)

`reduce` is actually optional, and if not specified the map output keys will be sorted and shuffled and saved to disk. Thus it is possible to set `inout[2]` to

map. To turn of sorting and shuffling and instead write the map output to disk directly, set `mapred.reduce.tasks` to zero in `mapred`.

combiner If set to TRUE, RHIPE will run the `reduce` expression on the output of the `map` expression locally i.e. on the same computer that is running the associated map. For every *io.sort.mb* megabytes of key,value output from the map, the keys are sorted, and the expression `reduce` will be called for all keys and their associated values. The calling sequence of the elements of `reduce` is the same as above. The only difference is that the expression will not be sent *all* the values associated with the key.

If `combiner` is TRUE, **it** will be called.

The outputs from the reduce are sorted and shuffled and sent to the Hadoop MapReduce reduce phase. Since the output from `map` is sent to `reduce` and the output from `reduce` is also sent to the `reduce` (during the final reduce phase of Hadoop MapReduce), the `reduce` expression must be able to handle input from the `map` and from `reduce`.

If `combiner` is TRUE, the `reduce` expression can be invoked during the local combine, in which case the output is intermediate and not saved as final output. It will also be invoked during the final reduce phase, in which case the will receive all the values associated with the key (note, these are values outputted when `reduce` is invoked as a combiner) and the output will be committed to the destination folder. To determine in which state `reduce` is running read the environment variable `rhipe_iscombining` which is '1' or '0' for the former and latter states respectively.

setup and cleanup In RHIPE, each tasks is a sequence of many thousands of key, value pairs. Before running the `map` and `reduce` expression (and before any key, value pairs have been read), RHIPE will evaluate expressions in `setup` and `cleanup`. Each of these may contain the names `map` and `reduce` e.g `setup=list(map=,reduce=)` specific to the `map` and `reduce` expressions. If

just an expressions is provided, it will be evaluated before both the Map phase and Reduce phase. The same is true for `cleanup`. Variables created, packages loaded in the `setup` expression will be visible in the `map` expression.

ifolder This is a path to a folder on the HDFS containing the input data. This folder may contain sub folders in which case RHIPE use the all the files in the subfolders as input. This argument is optional: if not provided, the user must provide a value for `N` and set the first value of `inout` to *lapply*.

ofolder The destination of the output. If the destination already exists, it will be overwritten.

inout A character vector of two components which specify the formats of the input and output destinations. Values are

sequence The keys and values can be arbitrary R objects. All the information of the object will be preserved. To extract a single key,value pair from a sequence file, either the user has to read the entire file or compose a MapReduce job to subset that key,value pair.

text The keys, and values are stored as lines of text. If the input is of text format, the keys will be byte offsets from beginning of the file and the value is a line of text without the trailing newline. R objects written to a text output format are written as one line. Characters are quoted and vectors are separated by `mapred.field.separator` (default is space). The character used to separate the key from the value is specified in the `mapred` argument by setting `mapred.textoutputformat.separator` (default is tab). To not output the key, set `mapred.textoutputformat.usekey` to FALSE.

map A map file is actually a folder consisting of sequence file and an index file. A small percentage of the keys in the sequence file are stored in the index file. Using the index file, Hadoop can very quickly return a value corresponding to a key (using `rhgetkey`). To create such an output format,

use `map`. Note, the keys have to be saved in sorted order. The keys are sent to the `reduce` expression in sorted order, hence if the user does not modify `reduce.key` a query-able map file will be created. If `reduce.key` is modified, the sorted guarantee does not hold and RHIPE will either throw an error or querying the output for a key might return with empty results

shared This is a character vector of files located on the HDFS. At the beginning of the MapReduce job, these files will be copied to the local hard disks of the Tasktrackers. User provided R code can read theses files from the current directory (which is located on the local hard disk). For example, if `/path/to/file.Rdata` is located on the HDFS and shared, it is possible to read it in the R expressions as `load('file.Rdata')`. Note, there is no need for the full path, the file is copied to the current directory of the R process.

copyFiles Will the files created in the R code e.g. PDF output, be copied to the destination folder, `ofolder`?

jobname The name of the job, which is visible on the Jobtracker website. If not provided, Hadoop MapReduce uses the default name i.e. *job_date_time_number*

jarfiles Optional JARs that need to be used during Hadoop MapReduce. This is used in the case when a user provides a custom InputFormat. Specify the JAR file to handle this InputFormat using this argument and specify the name of the InputFormat in the `mapred` argument.

opts RHIPE launches the C engine on the remote computers using the value found in `rhoptions()$opts$runner`. This is created from the local R installation which is possibly different from the Tasktrackers. If this is the case, specify the command that launches the R session via this parameter.

N To apply a computation to the numbers $1, 2, \ldots, N$ set `inout[1]` to *lapply* and specify the value of N in this parameter. Set the number of map tasks in

`mapred.map.tasks` (hence each task will run approximately $\lfloor \frac{N}{mapred.map.tasks} \rfloor$ computations sequentially).

partitioner A list of two names elements: *lims* and *type*. A partitioner forces all keys sharing the same property to be processed by one reducer. Thus, for these keys, the output of the reduce phase will be saved in one file. For example, if the keys were IP addresses e.g. `c(A,B,C,D)` where the components are integers, with the default partitioner, the space of keys will be uniformly distributed across the number of reduce tasks. If it is desired to store all IP addresses with the same first three ordinates, use a partitioner as `list(lims=c(1:3), type='integer')`. RHIPE implements partitioners when the key is an atomic vector of the following type: integer, string, and real. The value of *lims* specifies the ordinates of the key to partition on. The numbers must be positive.

mapred Specify Hadoop and RHIPE options in this parameter (a list). For a full list of RHIPE options see (3.1) and for Hadoop options (had).

Submitting a MapReduce R Object to Hadoop

```
rhex(mrobj, async=FALSE,mapred)
```

Submits a MapReduce job (created using `rhmr`) to the Hadoop MapReduce framework. The argument `mapred` serves the same purpose as the `mapred` argument to `rhmr`. This will override the settings in the object returned from `rhmr`. The function returns when the job ends (success/failure or because the user terminated (see `rhkill`)). When `asynch` is TRUE, the function returns immediately, the job running in the background; the function also returns an object of class *jobtoken*. The generic function `print.jobtoken`, displays the start time, duration (in seconds) and percent progress.

Table 3.1: Configurable options for RHIPE and MapReduce

Name	Description	Default
rhipe_map_buffsize	The length of map.keys and map.values	10,000
rhipe_reduce_buffsize	The length of reduce.values	10,000
io.sort.mb	The number of megabytes of intermediate output before it is locally reduced when combiner is true	Hadoop site file
mapred.textoutputformat.separator	Character that separates the key and value in text output	[tab]
mapred.field.separator	Character that separates the elements of vectors in text output	[space]
mapred.textoutputformat.usekey	For text output, if false, the key is not written	TRUE
mapred.map.tasks	The number of map tasks to divide the job into	For sequence and text file input, it is derived from the number of blocks. For lapply input, it is derived from *N* in the call to `rhmr`
mapred.reduce.tasks	The number of reduce tasks to launch to aggregate the intermediate keys	Hadoop site file
mapred.tasktimeout	The number of minutes after which the Tasktracker will terminate the R process. Set to zero, if the job consists of unpredictable long running tasks	10 min
rhipe_stream_buffer	The size of the buffer in KB for transferring data between Java and R	1MB
mapred.input.filename	the name of current input file for the given task, read it using `Sys.getenv`	

Monitoring a MapReduce Job

`rhstatus(jobid)`

This returns the status of an running MapReduce job. The parameter `jobid` can either be string with the format *job_datetime_id* or the value returned from `rhex` with the `async` option set to TRUE. A list of 4 elements: the state of the job (one of $START, RUNNING, FAIL, COMPLETE$), the duration in seconds, a data frame with columns for the Map and Reduce phase. This data frame summarizes the number of tasks, the percent complete, and the number of tasks that are pending, running, complete or have failed. In addition the list has an element that consists of both user defined and Hadoop MapReduce built in counters.

Waiting on Completion of a MapReduce Job

`rhjoin(jobid)`

Calling this functions pauses the R console till the MapReduce job indicated by `jobid` is over (successfully or not). The parameter `jobid` can either be string with the format *job_datetime_id* or the value returned from `rhex` with the `async` option set to TRUE. This function returns the same object as `rhex` i.e a list of the results of the job (TRUE or FALSE indicating success or failure) and a counters returned by the job.

Stopping a MapReduce Job

`rhkill(jobid)`

This kills the MapReduce job with job identifier given by `jobid`. The parameter `jobid` can either be string with the format *job_datetime_id* or the value returned from `rhex` with the `async` option set to TRUE.

3.8.3 Functions Available during the MapReduce

In the `map`,`reduce`, `setup` and `cleanup` expressions, RHIPE provides three functions to communicate the Hadoop MapReduce framework.

rhcollect Called with two R objects. Sends a key,value pair to the Hadoop system. In the Map phase, it will pass it on for reduction if `mapred.reduce.tasks` is not zero (by default it is non zero) or it will be written to disk if `mapred.reduce.tasks` is zero. In the Reduce phase, it will be sent for further reduction if `reduce` is being run as a combiner or it will be written to the final destination if it is being run as the reducer.

rhstatus Takes a character vector. A textual message that is visible on the Jobtracker website. This also informs Hadoop that the task is still running and it is not to be killed. In the absence of `rhstatus` and if `mapred.task.timeout` is non zero (by default it is 10 minutes) Hadoop will kill the R process.

rhcounter Called as `rhcounter(A,B,N)`. This increments the distributed counter B belonging to the family A by the integer N.

3.8.4 Miscellaneous Functions

The functions `rhuz` and `rhsz` serialize an object using the same serialization used by the RHIPE Java Bridge and C engine. As mentioned before, RHIPE enforces a 256MB read limitation on the size of any key or value. The `rhsz` function is useful for the programmer who wishes to know how many bytes the serialized object will take. The `rhsz` function is also useful for sharing data structures among various languages e.g. Python, Java, C++ since bindings exist in these languages to read RHIPE objects.

3.9 Conclusion

Though MapReduce has gained much popularity in the recent years, the principles of defining and computing across subsets has been prevalent in the statistical community for a long time. Hadoop MapReduce and the Hadoop Distributed Filesystem brings the approach of analysis to massive distributed data across a cluster. RHIPE integrates the R programming environment with the Hadoop distributed computing framework. Using RHIPE, the data analyst can stay within R and yet compute across massive amounts data in parallel across a cluster. RHIPE is designed to be used with the divide and recombine approach to data analysis i.e create different subsets, compute across all or some, visualize a sample and recombine the results. We have not implemented statistical algorithms such as regression routines into RHIPE, though there is active research into ways of framing the regression problem in the language of divide and recombine. RHIPE can be used as a query-able data base to store millions of objects across gigabytes of storage and retrieve them with relatively low latency. Future work will include integration of RHIPE with HBase to query, in a scalable fashion, arbitrarily large data sets.

```
reduce <- expression(pre = {
      doempty <- function(d,part0){
        data <- do.call("rbind",d)
        colnames(data)<- c("hash","isrequester","srcip","dstip"
                           ,"srcport","dstport","srcporthr"
                           ,"dstporthr", "datasize","timeOfPacket"
                           ,"flagsnum","ackn","seqn")
        data <- data.frame(hash=data[,'hash']
                           ,isrequester=as.integer(
                                    data[,'isrequester'])
                           ,srcip = data[,'srcip']
                           % ,dstip = data[,'dstip']
                           % ,srcport = as.integer(data[,'srcport'])
                           % ,dstport = as.integer(data[,'dstport'])
                           % ,srcporthr = data[,'srcporthr']
                           % ,dstporthr = data[,'dstporthr']
                           ....
                           ,datasize = as.integer(data[,'datasize'])
                           ,timeOfPacket = as.numeric(
                                     data[,'timeOfPacket'])
                           ,flagsnum = as.integer
                                     (data[,'flagsnum'])
                           ,ackn = as.numeric(data[,'ackn'])
                           ,seqn = as.numeric(data[,'seqn'])
                           ,stringsAsFactors=FALSE)
        key <- c( as.vector(data[1,c('hash')]), integerid)
        rhcollect(key,data)
      }, integerid <- 1
    },
    reduce = {
      doempty(reduce.values,apart);
      integerid <- integerid + 1})
```

Figure 3.8.: Option (2): R code to convert text to R data frames

```
rhmr(map=map,reduce=reduce,
     ifolder="/net/d/textinput",
     ofolder="/net/o/blocks",
     inout=c("text","sequence"))
```

Figure 3.9.: R code to run MapReduce job

```
map <- expression({
  lapply(seq_along(map.keys),function(r)
         rhcollect(map.keys[[r]],map.values[[r]]))
})
rhmr(map=m
     ifolder="/net/o/blocks",
     ofolder="/net/o/mapblocks",
     inout=c("sequence","map"))
```

Figure 3.10.: R code to convert sequence files to indexable map files

```
a <- rhgetkey(list(
  c("0000131163cd7e167b72ee987311ab224b54f",1),
  c("00002e3a6481673fc1845b86a8f750f0fded2",1)
  ),"/net/d/dump.12.1.14.09.map")
4.43 kb read,unserializing, please wait
> length(a)
2
> a[[1]][[2]][1:2,c('hash', 'srcport','timeOfPacket')]
hash                                          srcport     timeOfPacket
00002e3a6481673fc1845b86a8f750f0fded2            80          1259765941
00002e3a6481673fc1845b86a8f750f0fded2          3282          1259765941
```

Figure 3.11.: R code to retrieve keys from the database

```
mykeys <- new.env()
mykeys$'0000131163cd7e167b72ee987311ab224b54f' <- 1
mykeys$'00002e3a6481673fc1845b86a8f750f0fded2' <- 1
rhsave(mykeys,file='/tmp/mykeys.Rdata')
map.setup <- expression({
  load("mykeys.Rdata")
})
map <- expression({
  hashes <- lapply(map.keys,function(r) r[1])
  a <- lapply(hashes, function(r)
         if(!is.null(mget(r,mykeys,ifnotfound=list(NULL))))
         TRUE else FALSE
         )
  map.keys <- map.keys[a]
  map.values <- map.values[a]
  lapply(seq_along(map.keys),function(r)
    rhcollect(map.keys[[r]],map.values[[r]]))
})
z <- rhmr(map=map,setup=list(map=map.setup),
     ifolder="/net/o/blocks",
     ofolder="/tmp/subset",
     inout=c("sequence",'sequence'),
     shared='/tmp/mykeys.Rdata'
     mapred=list(mapred.reduce.tasks=0))
rhex(z)
```

Figure 3.12.: R code to use MapReduce to retrieve a key

```
map <- expression({
 lapply(seq_along(map.keys), function(r)
        rhcollect(map.keys[[r]][[1]],map.values[[r]]))
})
reduce <- expression(
    pre = {
      first1500 <- NULL
    },
    reduce = {
      first1500 <- rbind(first1500,do.call("rbind",reduce.values))
      first1500 <- first1500[order(first1500\$timeOfPacket),]
                      [1:min(1500,nrow(first1500)),]
   },
    post = {
      rhcollect(reduce.key, first1500)
    })
z <- rhmr(map=map,reduce=reduce,
          ifolder="/net/o/blocks",
          ofolder="/net/o/first1500",
          inout=c("sequence",'map'),
          combiner=TRUE,
          mapred=list(rhipe_buff_size=10,rhipe_map_buff_size=20))
rhex(z)
```

Figure 3.13.: R Code to store first 1500 packets

```
map <- expression({
  lapply(map.values,function(r){
    lapply(split(r,c(r$srcip,r$srcport)),function(d){
    num.packets <- nrow(d)
    databytes <- sum(d$datasize)
    min.time <- min(d$timeOfPacket)
    max.time <- max(d$timeOfPacket)
    isrq <- max(d$isrequester)
    rhcollect(d[1,c('hash','srcip','srcport')],c(packets,databytes,
                        min.time,max.time,isrq))
    }
  })
})
```

Figure 3.14.: R code for map portion of the summary computation

```
reduce <- expression(
    pre = {
      a <- NULL
    },
    reduce = {
      a <- rbind(a,do.call('rbind', reduce.values))
    },
    post = {
      if(as.integer(Sys.getenv("rhipe_combiner"))==1L)
        rhcollect(reduce.key,c(sum(a[,1]),sum(a[,2]),min(a[,3]),
                                                max(a[,4]),max[a,5]))
      else
        rhcollect(reduce.key,c(packets=sum(a[,1]),
                               bytes=sum(a[,2]),
                               start=min(a[,3]),
                               duration=max(a[,4])-min(a[,3]),
                               isrq=max(a[,5])))
    })
```

Figure 3.15.: R code for the reduce portion of the summary computation

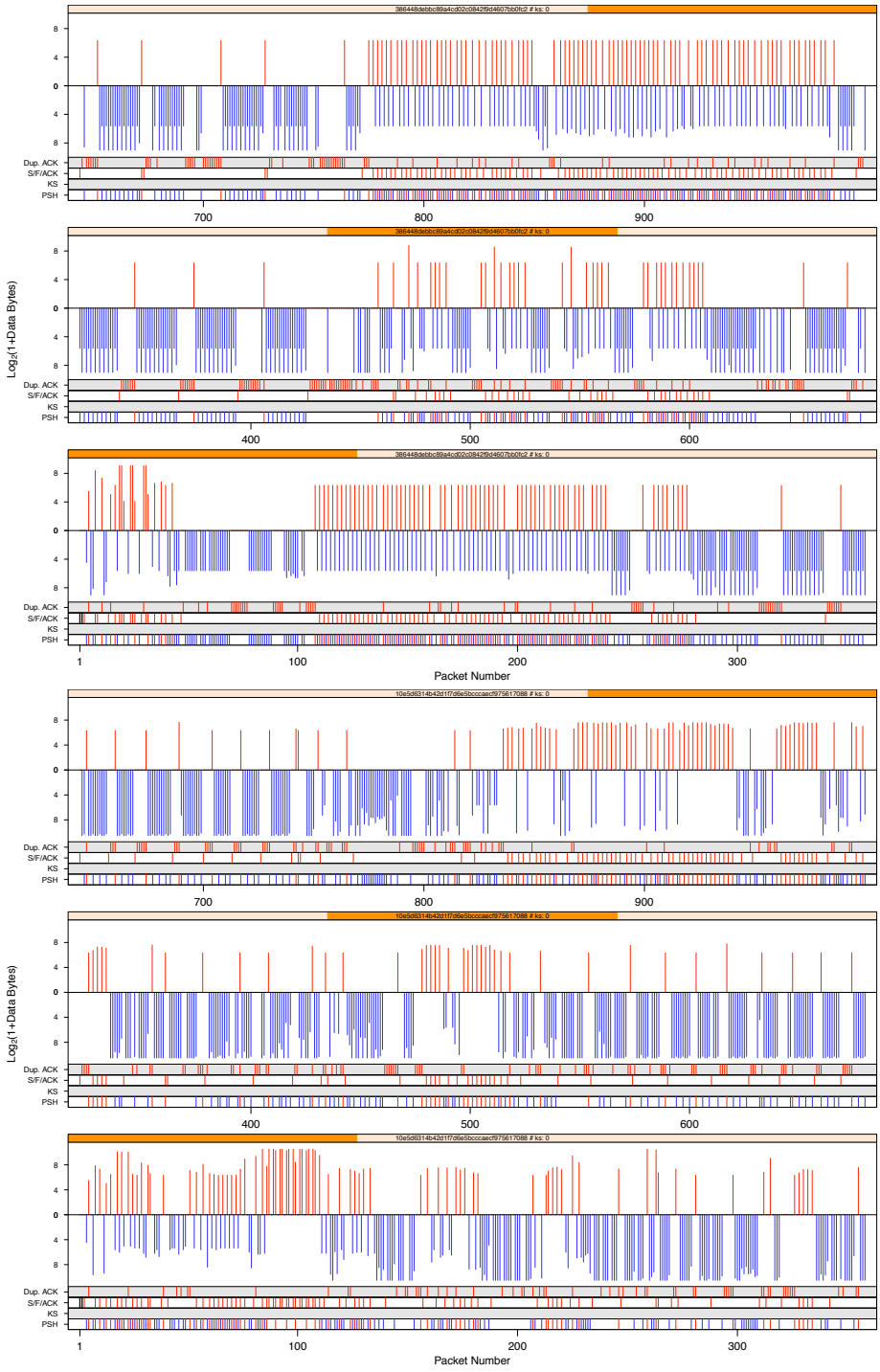
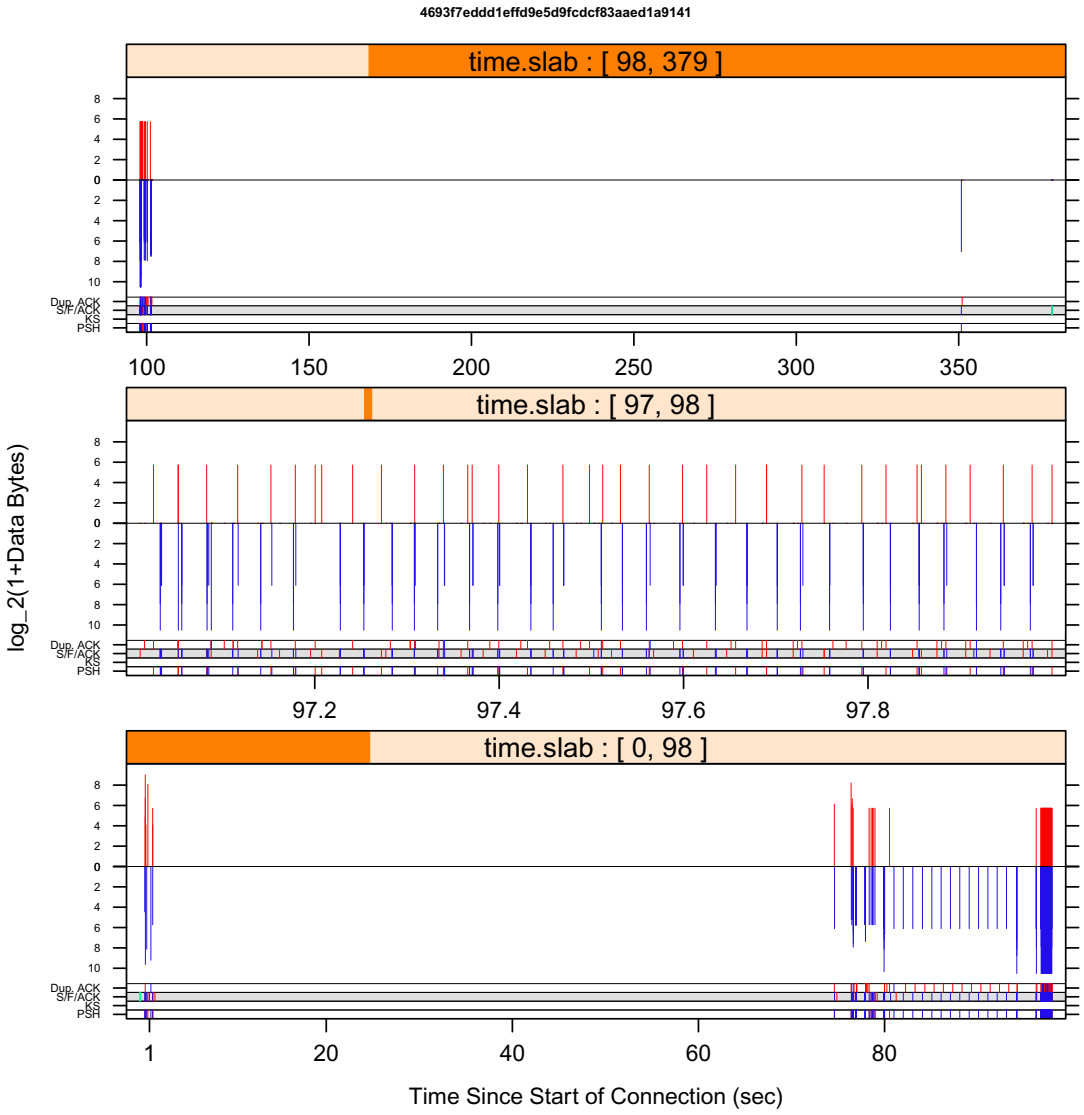
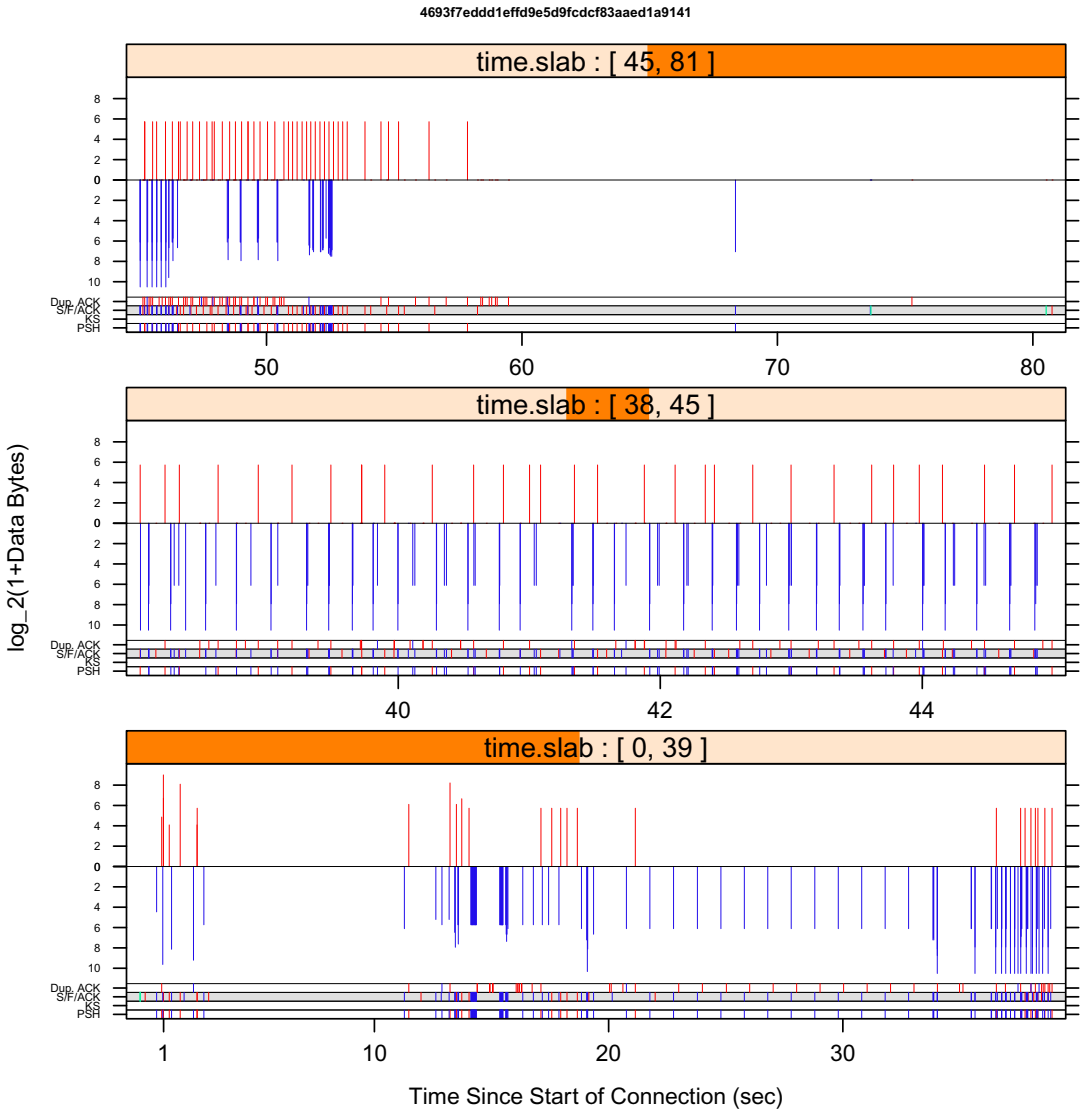


Figure 3.16.: Packet order displays by connection for two connections. The bottom 3 are for the first connection and the top 3 are for the second connection. Each panel displays 300 packets.The connections start from the first and third panel respectively.



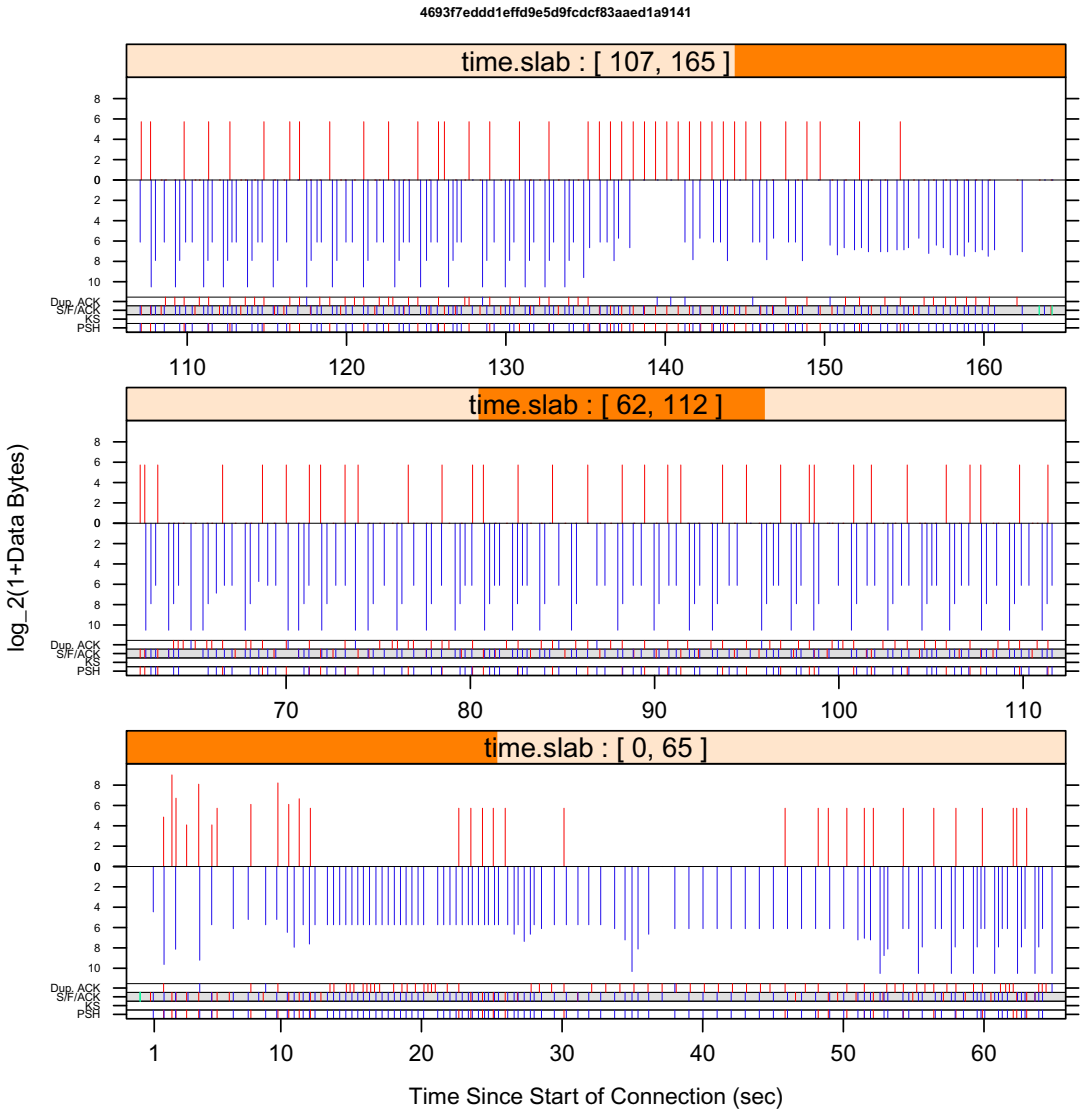
(a) $\phi(t) = t$

Figure 3.17.: Time order displays for different ϕ



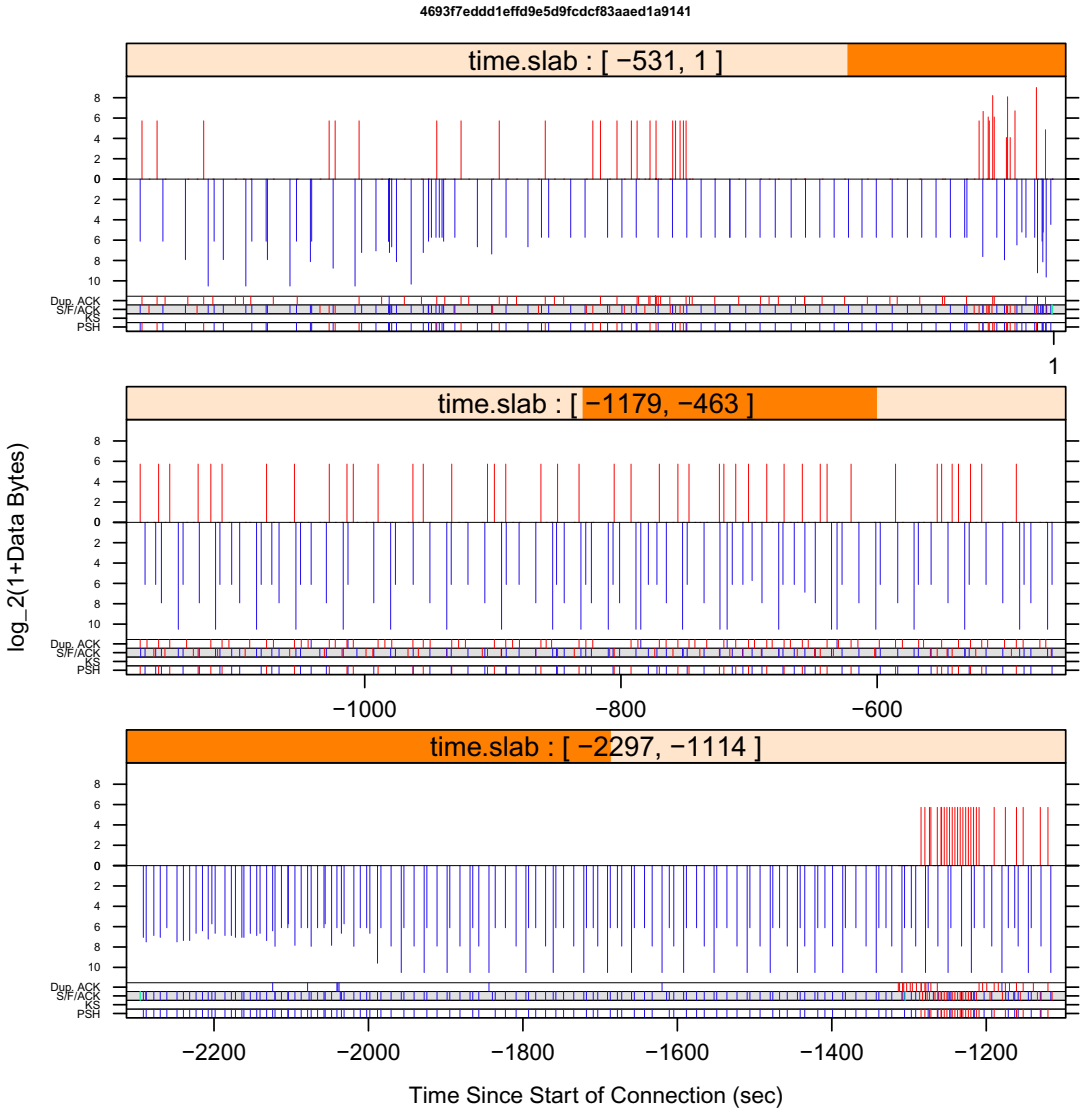
(b) $\phi(t) = t^{0.5}$

Figure 3.17.: Time order displays for different ϕ



(a) $\phi(t) = t^{0.1}$

Figure 3.18.: Time order displays for different ϕ



(a) $\phi(t) = log_2(t)$

Figure 3.19.: Time order displays for different ϕ. The skewed nature of the inter-arrival times disappear as the transform goes to the logarithm

```
map <- expression({
  param <- unserialize(charToRaw(Sys.getenv("params")))
  lapply(seq_along(map.keys),function(i){
    rhcollect(map.keys[[i]], run.ks.algo(map.values[[i]],param))
  })
})
rhids <- list()
for (id  in seq_along(list.of.params)){
  z <- rhmr(map=algo,ifolder="/path/to/data",
            ofolder=sprintf("/path/to/out/result.%s",id),
            inout=c("sequence","sequence"),
            mapred=list(param=
            rawToChar(serialize(params[[id]],NULL,ascii=TRUE))))
  rhids <- c(rhids,rhex(z,async=TRUE))
}
for( id in rhids){
  rhjoin(id)
}
results <- rhread("/path/to/out/")
```

Figure 3.20.: R code to submit asynchronous jobs and wait for results

```
map <- expression({
  lapply(seq_along(map.values),function(r){
    catlevel <- map.keys[[r]] #numeric
    timepoint <- map.values[[r]]$timepoint #numeric
    datastructure <- map.values[[r]]$data
    key <- c(catlevel,timepoint)
    rhcollect(key,datastructure)
  })
})
redsetup <- expression({
  currentkey <- NULL
})
reduce <- expression(
    pre={
      catlevel <- reduce.key[1]
      time <- reduce.key[2]
      if(!identical(catlevel,currentkey)){
       if(!identical(currentkey,NULL))
          FINALIZE(F)
       currentkey <- catlevel
        ## initialize computation for new level
        INITIALIZE(F)
      }
    },
    reduce={
      F <- UPDATE(F, reduce.values[[1]])
    })
redclose <- expression({
FINALIZE(F)
})
```

Figure 3.21.: Pseudo RHIPE code for streaming computations

```
FC <- expression({
  results <- do.call("rbind",lapply(map.values,F))
  rhcollect(1,results)
})

rhmr(map=FC,ofolder='tempfolder',inout=c('lapply','sequence'),N=M
     ,mapred=list(mapred.map.tasks=1000))

do.call('rbind',lapply(rhread('/tempfolder', mc=TRUE),'[[',2))
```

Figure 3.22.: R code to distribute `do.call('rbind',sapply(1:M,F))`

```
reduce <- expression(
  pre        = { s <- 0 },
  reduce = { s <- sum(s,unlist(reduce.values)) },
  post      = { rhcollect(reduce.key,s) }
)
```

Figure 3.23.: Demonstration of the `reduce` expression to compute a sum

LIST OF REFERENCES

LIST OF REFERENCES

Jsm 2009 DataExpo, http://stat-computing.org/dataexpo/2009/. (page 70).

http://hadoop.apache.org/common/docs/current/mapred-default.html, Configurable Options for Hadoop DFS and Hadoop MapReduce. (pages 73, 93).

HBase, http://hbase.apache.org/. (page 75).

CRAN HPC task view, http://cran.r-project.org/web/views/HighPerformanceComputing.html. (page 59).

Hypertable, http://hypertable.org. (page 75).

Jester Joke Ratings ,www.ieor.berkeley.edu/~goldberg/jester-data. (page 3).

Purdue Visualization Databases,http://ml.stat.purdue.edu/vdb. (page 2).

Riyad Alshammari, Peter I. Lichodzijewski, Malcolm Heywood, and A. Nur Zincir-Heywood. Classifying SSH Encrypted Traffic with Minimum Packet Header Features using Genetic Programming. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2539–2546, New York, NY, USA, 2009. ACM Press. ISBN 978-1-60558-505-5. (page 40).

Apache Foundation. Hadoop Streaming, http://hadoop.apache.org/common/docs/current/streaming.html. (page 62).

R. A. Becker, W. S. Cleveland, and M. J. Shyu. The Visual Design and Control of Trellis Display. *Journal of Computational and Graphical Statistics*, 5:123–155, 1996. (page 6).

A. Beckmann, R. Dementiev, and J. Singler. Building a parallel pipelined external memory algorithm library. In *International Parallel and Distributed Processing Symposium*, 2009. (page 47).

Jin Cao, William S. Cleveland, and Don X. Sun. S-net: A software system for analyzing packet header databases,. In *Proceedings of Passive and Active Measurement*, pages 34–44, 2002. (page 80).

Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149. Society for Industrial and Applied Mathematics, 1995. (page 47).

W. S. Cleveland. *Visualizing Data*. Hobart Press, Chicago, 1993. (page 1).

William S. Cleveland and Don X. Sun. Internet traffic data. *Journal of the American Statistical Association*, 95(979-9785), 2000. (page 80).

Douglas Comer. *Internetworking With TCP/IP Volume 1: Principles Protocols, and Architecture,*. Prentice Hall, 2006. (page 80).

R. D. Cook and S. Weisberg. *Applied Regression Including Computing and Graphics*. Wiley, New York, 1999. (page 1).

Annie De Montigny-Leboeuf. Flow Attributes for use in Traffic Characterization. Technical Report CRC-TN-2005-003, CRC, December 2005. (page 40).

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008. (page 61).

Wei Ding, Matthew J. Hausknecht, Shou-Hsuan, Stephen Huang, and Zach Riggle. Detecting Stepping-Stone Intruders with Long Connection Chains. *International Symposium on Information Assurance and Security*, 2:665–669, 2009. (page 41).

David L. Donoho, Ana Georgina Flesia, Umesh Shankar, Vern Paxson, Jason Coit, and Stuart Staniford. Multiscale Stepping-Stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay. In *Proceedings of The 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 17–35. Springer, 2002. (pages 20, 42).

William Dumouchel, Chris Volinsky, Theodore Johnson, Corinna Cortes, and Daryl Pregibon. Squashing flat files flatter. pages 6–15. ACM Press, 1999. (page 48).

T. Dunnigan and G. Ostrouchov. Flow Characterization for Intrusion Detection. Technical report, Oak Ridge National Laboratory, 2000. (page 40).

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004. (page 58).

David Gelernter. Generative communication in linda. In *ACM Transactions on Programming Languages and Systems*, volume 7, 1985. (page 59).

K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A Constant Time Collaborative Filtering Algorithm. *Information Retrieval*, 4:133–151, 2001. (page 3).

Google. Protocol Buffers, `http://code.google.com/p/protobuf/`. (pages 60, 66).

S. J. Grannis, M. Wade, J. Gibson, and J. M. Overhage. The indiana public health emergency surveillance system: Ongoing progress, early findings, and future directions. In *Proceedings of the Annual Symposium*, pages 304–308. American Medical Informatics Association, 2006. (page 3).

Hadoop. Hadoop: Open source implementation of mapreduce, `http://hadoop.apache.org/`. (pages 5, 51).

Hadoop Companies. List of companies using Hadoop, `http://wiki.apache.org/hadoop/PoweredBy`. (page 52).

R. P. Hafen, D. E. Anderson, W. S. Cleveland, R. Maciejewski, D. S. Ebert, A. Abusalah, M. Yakout, M. Ouzzani, and S. Grannis. Syndromic Surveillance: STL for Modeling, Visualizing, and Monitoring Disease Counts. *BMC Medical Informatics and Decision Making*, 2009, to appear. (page 3).

HDF Group. *HDF5: Hierarchical Data Format*. URL http://www.hdfgroup.org/HDF5/. (page 59).

F. Hernandez-Campos, F. Donelson-Smith, K. Jeffay, and A.B. Nobel. Understanding Patterns of TCP Connection Usage With Statistical Clustering. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 35–44, 2005. (page 40).

J Horton and R Safavi-Naini. Detecting Policy Violations through Traffic Analysis. In *22nd Annual Computer Security Applications Conference*, pages 109–120, December 2006. (page 40).

Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Sigcomm '05: Proceedings of the 2005 Conference On Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 229–240, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-009-4. (page 40).

Thilo Kielmann, Henri E. Bal, Sergei Gorlatch, Kees Verstoep, and Rutger F. H. Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431–1456, 2001. (page 59).

Kevin Kilourhy and Roy Maxion. Comparing Anomaly-Detection Algorithms for Keystroke Dynamics. In *International Conference on Dependable Systems and Networks (DSN-09)*, pages 125–134, Los Alamitos, CA, USA, 2009. IEEE Computer Society. (page 45).

Jochen Knaus. *snowfall: Easier cluster computing (based on snow)*. URL http://cran.r-project.org/web/packages/snowfall/index.html. (page 58).

K. Koffka. *Principles of Gestalt Psychology*. Harcourt, New York, 1935. (page 15).

Pierre L'Ecuyer and Josef Leydold. rstream: Streams of random numbers for stochastic simulation. (page 85).

E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. (page 49).

Thomas Lumley. *biglm: bounded memory linear and generalized linear models*, 2009. URL http://cran.r-project.org/web/packages/biglm/index.html. (page 59).

Lindsay W. MacDonald. Tutorial: Using color effectively in computer graphics. *IEEE Computer Graphics and Applications*, 19, 1999. (page 80).

M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000. (page 85).

Michael, K. and Emerson, J. *bigmemory: Manage massive matrices with shared memory and memory-mapped files*, 2010. URL http://cran.r-project.org/web/packages/bigmemory/index.html. (page 59).

V. Paxson. End-to-End Internet Packet Dynamics. In *Proceedings of ACM SIGCOMM*, pages 139–152, 1997. (page 19).

R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. URL http://www.R-project.org. ISBN 3-900051-07-0. (pages 5, 57).

REvolution Analytics. *nws: R functions for NetWorkSpaces and Sleigh*. URL http://cran.r-project.org/web/packages/nws/index.html. (page 59).

RHIPE. RHIPE on GitHub,http://github.com/saptarshiguha/RHIPE/, a. (page 60).

RHIPE. RHIPE, http://www.stat.purdue.edu/~sguha/rhipe, b. (pages 5, 60).

Jop F. Sibeyn. External matrix multiplication and all-pairs shortest path. *Information Processing Letters*, 91, 2004. (page 47).

W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994. (page 19).

H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):62, 2005. (page 49).

Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*. URL http://cran.r-project.org/web/packages/snow/index.html. (page 58).

Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 161–179. American Mathematical Society, 1999. (page 47).

Simon Urbanek. *multicore: Using Cores of Computer*. URL http://cran.r-project.org/web/packages/multicore/index.html. (page 88).

Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers Inc, 2008. (page 47).

Wand Network Research Group. Wand Network Research Group. http://www.wand.net.nz/wits/leipzig/1/. (page 24).

Roger Wattenhofer and Peter Widmayer. An inherent bottleneck in distributed counting. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computin*. ACM Press, 1997. (page 59).

Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On Inferring Application Protocol Behaviors in Encrypted Network Traffic. *J. Mach. Learn. Res.*, 7: 2745–2769, 2006. ISSN 1532-4435. (page 40).

Hongquan Xu. A Catalogue of Three-Level Regular Fractional Factorial Designs. *Metrika*, 62:259–281, 2005. (pages 29, 81).

T. Ylonen and C. Lonvick. RFC 4251: The Secure Shell (SSH) Protocol Architecture. http://www.ietf.org/rfc/rfc4251.txt, 2006a. URL http://www.ietf.org/rfc/rfc4251.txt. This is an electronic document. Date of publication: January, 1996. Date retrieved: April 12, 2010. (page 26).

T. Ylonen and C. Lonvick. RFC 4253: The Secure Shell (SSH) Transport Layer Protocol. http://www.ietf.org/rfc/rfc4253.txt, 2006b. URL http://www.ietf.org/rfc/rfc4253.txt. This is an electronic document. Date of publication: January, 1996. Date retrieved: April 12, 2010. (pages 21, 26, 44).

Hao Yu. *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*. URL http://cran.r-project.org/web/packages/rmpi/index.html. (page 58).

Kwong H. Yung. Detecting Long Connecting Chains of Interactive Terminal Sessions. In *Proceedings of The 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–16, 2002. (page 41).

Yin Zhang and Vern Paxson. Detecting Backdoors. In *Proceedings of the 9th USENIX Security Symposium*, pages 157–170, 2000a. (page 41).

Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *Proceedings of the 9th USENIX Security Symposium*, pages 171–184, 2000b. (page 41).

VITA

VITA

Saptarshi Guha was born in the remote town of Larne, N. Ireland in 1978. After 9 years in England, he moved to Kolkata, India. Saptarshi graduated from St. James' School and obtained his B.Sc in Statistics from Presidency College, Kolkata. After his M.Stat from the Indian Statistical Institute, New Delhi, he worked for two years as a Business Analyst in GE Capital, India Services. His academic interests include statistical model building, exploratory data analysis, visualization, massive data, computational statistics, network security and programming languages for statistics.