LiveRAC - Live Reorderable Accordion Drawing

by

Peter Jono McLachlan

B.A.H., Queen's University, 2001

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

September, 2006

Abstract

LiveRAC is a scalable focus+context approach for monitoring computer systems and networking data that provides user-directed data reordering and details-on-demand. LiveRAC maps alarm and metric data from network devices including servers, routers and switches into a visual metaphor called accordion drawing. In accordion drawing, users interact with the display as though it were a rubber sheet tacked down at the borders. Regions of the display can be stretched and compressed, but the fixed borders ensure the visibility of the entire information space. Compressed regions of the display aggregate the underlying data. We implement guaranteed visibility, a mechanism for ensuring the visibility of important features such as critical alarms. LiveRAC extends existing accordion drawing techniques in two ways: it can add, remove and reorder objects to the data set in logarithmic time, and provides an infrastructure for semantic zoom, a visualization technique where an optimal data representation is selected based on the screen space available to a data cell. Using a client-server approach we allow the user to query the underlying data in a context-rich, visually salient metaphor while maintaining interactive frame rates.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank everyone who contributed to the successful completion of this thesis. A thesis is not the product of a single mind, but a synergy of comrades who participate directly, or in spirit through friendship and support. Any list is bound to be incomplete, but thanking a few of my lab (in)mates is a good start: Andrew Adam, Meghan Allen, Dan Archambault, Aaron Barsky, Chris Batty, Derek Bradley, Tyson Brochu, Ciáran Llachlan Leavitt, Abhijeet Ghosh, Jennifer Gluck, Stephen Ingram, Vladislav Kraevoy, Heidi Lam, Barry Po, Tiberiu Popa, Ken Rose, James Slack, David Sprague, Matt Trentacoste, David White and Steve Yohanan. My supervisor Tamara Munzner deserves a special mention for putting up with my wandering ideas and helping to steer them into something tangible.

I'd like to single out my good friends from undergrad Christian Van Oullet and Barry Po, for persuading me to pursue graduate school. Without you I'd still be stuck in the 'grind'.

Sarah, you're a hard-working, good-natured, talented person. Stick to it, I know you'll make your dreams come true.

To my family: you have all been a source of inspiration to me. Jackie, you're an amazing person. Never stop being your fun, dynamic, I-might-do-anything-next self. You've accomplished an amazing amount, done and seen many things few others will, and I'm pretty sure you're just getting started. Mom, your attitude towards life is fantastic. You taught me a lot of what I know about taking responsibility for my own destiny. I know I didn't understand this when I was younger, but you've won a convert now. Dad, you have one of the gentlest, most tenacious spirits of anyone I know. Your commitment, honesty and work ethic is an example to others. You make us all proud.

Chapter 1

Introduction

Computer networks and network devices are pervasive in almost every aspect of modern society. Interactions with these devices may include single purpose terminals such as automated teller machines, or general purpose equipment like personal data assistants and desktop computers. Data for this infrastructure is generally centralized on network servers. Increasingly, servers are being centralized in data warehouses managed and monitored by a Network Operations Centre (NOC). NOC facilities may be responsible for many terabytes of data, with billions of bits transferred over the network each second. The scale and importance of managing these facilities presents interesting research challenges. The complexity of NOCs makes understanding the big picture difficult. Details are important for diagnosing failures but often require proprietary tools to obtain and may force the user to sacrifice higher level overviews. The high cost of system failures makes it critical to provide salient visibility of high severity system alarms and notifications.

This thesis presents LiveRAC, a scalable information visualization system that supports NOC management tasks. Information visualization is a field of computer science that develops visual tools to augment human cognition. Leveraging a growing body of knowledge about the human visual system and human attention, information visualization systems provide insights concerning data sets comprising millions of objects. Using information visualization, operators at NOCs can better understand the real-time state of their network, with interactive support for exploring individual system details.

We make three contributions to the field of information visualization: a visualization system for network and alarm data called LiveRAC, data structures and algorithms for adding and removing split lines dynamically in the PRISAD [34] accordion drawing infrastructure using a modified implementation of red-black trees, and a technique for integrating semantic zoom [51] into accordion drawing. These contributions are applied to create a visualization system that operates on a real-world data set to provide a unique focus+context [63] interface for large scale network data.

1.1 Motivation

Network devices extend beyond individual server or desktop systems, and include file storage units, fiber disk storage, uninterruptable power supplies, printers and many other pieces of equipment that are typical in a corporate or academic environment. Although these devices are network aware and can be queried on an individual basis, in a large organization the monitoring and management of these devices is an enormous task. For Simple Network Management Protocol (SNMP) aware devices there are numerous commercial tools available for monitoring, called Network Management Stations (NMS), such as HP OpenView [31] or Tivoli NetView [32]. In the free software world there is OpenNMS [50]. The primary objective of these tools is to: act as a central dispatch for alarms generated by devices, provide a browsable device hierarchy, obtain specific information regarding device status, analyse potential problems and provide a high-level overview of system state.

Existing software packages typically lack data context, or a high level dashboard that summarizes system state, when drilling for detail information. Although

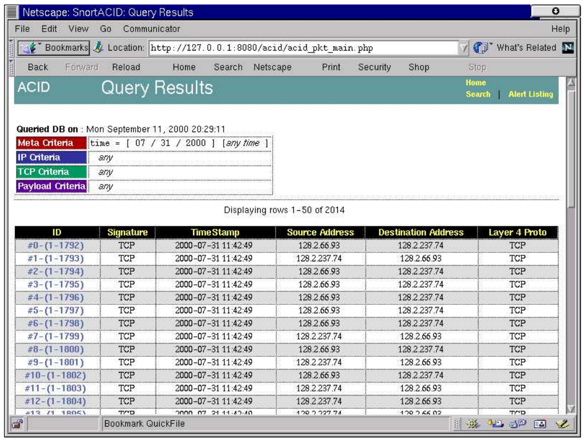
NMS systems provide some overview capabilities, the user interacts with these software packages by browsing, usually opening up new windows or changing the content of the main application window. Although this allows the user to see specific information about a system, during this navigation activity awareness of the overall system state is lost. The user can also lose spatial awareness of how one system relates to another while descending device hierarchies.

Another problem with current network management solutions is the inability to integrate and aggregate disparate data sources. Most modern computer networks are largely heterogeneous and consist of a mix of Windows and UNIX systems, and network appliances. Although there are some standard protocols for information gathering such as SNMP, there are also numerous proprietary data collection protocols which are platform specific, such as Windows performance monitor, and SGI Performance Co-Pilot. Although tools that target a single protocol or platform are useful, a visualization application which integrates data from multiple sources would be ideal. A multi-protocol and multi-platform tool better facilitates the side-by-side comparison of systems for root-cause analysis of problems and for planning activities.

Table 1.1: NOC operation staff tasks

Overview	**Detail**
Situational awareness	Incident investigation
Critical alarm monitoring	Capacity planning
Trend analysis	

Some high-level network management tasks include those listed in Table 1.1 which we have divided into two categories: overview tasks and detail tasks. This breakdown is imperfect as some elements of overview and detail are required for all of these tasks, however, it broadly categorizes the nature of the activity. Situational awareness means providing the user with a working mental model of the system state. Critical alarm monitoring requires the user to be aware of critical alarm states in the system, and changes to these states. Users perform trend analysis by examining detailed trend information, or getting an overview of system-wide trends. Incident investigation requires that a user obtain specific details from one or more systems concerning an incident such as a network intrusion or service failure. The activity of capacity planning is a blanket phrase for deciding when to deploy additional network or server capacity, and where to best deploy new services. Capacity planning involves examining detailed information from one or more devices, gathering data regarding current system-wide state, and integrating projections of future usage patterns.



http://127.0.0.1:8080/acid/acid_pkt_main.php

Queried DB on : Mon September 11, 2000 20:29:11

Meta Criteria	time = [07 / 31 / 2000] [*any time*]
IP Criteria	*any*
TCP Criteria	*any*
Payload Criteria	*any*

Displaying rows 1-50 of 2014

ID	Signature	TimeStamp	Source Address	Destination Address	Layer 4 Proto
#0-(1-1792)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#1-(1-1793)	TCP	2000-07-31 11:42:49	128.2.237.74	128.2.66.93	TCP
#2-(1-1794)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#3-(1-1795)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#4-(1-1796)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#5-(1-1797)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#6-(1-1798)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#7-(1-1799)	TCP	2000-07-31 11:42:49	128.2.237.74	128.2.66.93	TCP
#8-(1-1800)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#9-(1-1801)	TCP	2000-07-31 11:42:49	128.2.237.74	128.2.66.93	TCP
#10-(1-1802)	TCP	2000-07-31 11:42:49	128.2.66.93	128.2.237.74	TCP
#11-(1-1803)	TCP	2000-07-31 11:42:49	128.2.237.74	128.2.66.93	TCP
#12-(1-1804)	TCP	2000-07-31 11:42:49	128.2.237.74	128.2.66.93	TCP
#13-(1-1805)	TCP	2000-07-31 11:42:49	128.2.237.74	128.2.66.93	TCP

Figure 1.1: ACID [2] is a tool for querying and viewing intrusion alarm data.

In traditional applications these activities require manually navigating through large numbers of network devices, and synthesizing multiple data views. For example, alerts are typically aggregated using an alert manager, often in a table format, such as the view provided by ACID [2] shown in Figure 1.1, a web based application for querying and viewing intrusion alarm data. Getting information about any single device requires locating the device in the list views and loading details either in a new window or in the main content window of the network management application. This operation entails a loss of context. The system state may change while the operator is performing the navigation, or while examining the details of an individual system, and important information can be missed.

The objective of information visualization is to provide representations of large data sets that can harness our perceptual and cognitive abilities. This includes making use of visual characteristics, such as pre-attentive visual cues, and pattern recognition. As the volume of network data grows it becomes increasingly important to provide administrators with the tools to interact with the data in a fluid manner, maximizing the use of limited screen real-estate, and providing an overview of current system state during navigation operations.

LiveRAC is a visualization system we have developed to provide a scalable context-rich approach for system and network monitoring. LiveRAC explicitly targets the operations personnel of network operation centres and administrators of large networks, although LiveRAC may be of interest to small organizations as well. NOCs are monitoring centres for data hosting facilities where thousands or tens of thousands of devices are hosted in a physically secure data warehouse environment with highly redundant power and network connectivity. Some NOCs are operated by large companies for their own internal networks, but many provide hosting services to a variety of other companies, selling server space and connectivity. LiveRAC can be useful in either of these contexts.

The application domains for LiveRAC include analysis of high performance computer clusters, administration of large subnets, monitoring network application server farms, and large network capacity planning. LiveRAC provides operators with situational awareness through a high level overview where critical alarms are always visible regardless of the number of machines monitored. Incident investigation is supported by allowing operators to get increasing levels of detail by enlarging regions of interest. Semantic zoom adjusts the detail level of a representation as the area for the representation increases. LiveRAC simplifies capacity planning by making it possible to look at large numbers of devices simultaneously. Variances inside of devices are easily detected through small-multiples views. By viewing archived data administrators have access to a large volume of trend information, and can use this to baseline the current system status.

Our goal with LiveRAC was to design a scalable solution, and to do so our implementation needed to scale well across several dimensions. The first dimension is the number of devices that can be monitored. Large networks can have hundreds of thousands of workstations, with tens of thousands of supporting servers and network appliances. LiveRAC scales well both in the efficiency of its data structures, and the visual accordion dawing [47] encoding used for displaying data. Other dimensions include the number of metrics to be monitored and the quantity of archive data to be stored. The SWIFT [41] back-end used by LiveRAC integrates dozens of different metrics and alarms, and provides access to months worth of archived data. The scalability of SWIFT suggests the duration of archived data can potentially be expanded to years as monitoring continues.

1.2 Terminology

A **network device** is any device from which we can gather statistics or alarm data. This includes, but is not limited to, network routers, switches, servers and workstations. A **metric** is a descriptive value about a network device which can be sampled. The units of metrics may vary. Examples of metrics include the number of network packets transferred, or percentage of CPU load. A **bundle** is a grouping of metrics, or alarms, which will be placed in the same graphical representation. For example, CPU and memory might be grouped into a 'System' bundle.

An **alarm** in the context of network devices is an alert generated by a monitoring system, typically a service availability system, or an intrusion detection system. A **ticketed alarm** is an alarm which has been tagged for attention by a human operator. Alarms are either automatically tagged through a script, or manually by operators. Due to the volume of alarms, typically not all alarms are ticketed in enterprise environments.

Focus+Context is a family of visualization strategies that involve one or more regions of user focus, or attention, while providing a situational context for that focus. **Guaranteed visibility** is a technique for creating data landmarks which remain visible irrespective of navigation actions. **Rubber sheet navigation** [54] [58] is a focus+context technique for viewing and interacting with data where the data is projected on a plane which can be stretched or squished to bring different regions into focus. **Accordion drawing** [34] refers to rubber sheet navigation in conjunction with guaranteed visibility [47]. **Semantic zoom** is a visualization technique that displays different data representations at different levels of zoom. All of these techniques are described in more detail in Section 2.

1.3 Thesis Statement

Our objective is to create a scalable visualization system that supports the monitoring and planning tasks of NOC administrators for large-scale networked computer infrastructure. We create a unique visualization and supporting infrastructure which merges the techniques of accordion drawing and semantic zoom. The visualization allows users access to logged trend data, and uses streaming data to provide a live view of activity on monitored network devices.

1.4 Collaboration

This research has been conducted in collaboration with Eleftherios Koutsofios and Stephen North of AT&T Labs Inc. - Research who provided access to AT&T data, insights into network operations staff requirements, and assistance with integrating LiveRAC into the SWIFT data collection back-end.

1.5 Thesis Organization

This chapter has provided a motivation for this thesis, background on existing tools, our thesis statement, and outlines our research contributions. Chapter 2 discusses previous work related to this thesis. Chapter 3 provides an overview of the components and features of the LiveRAC application. Chapter 4 details our approach and algorithms for implementing LiveRAC. Chapter 5 discusses the performance, qualitative and quantitative results of LiveRAC. Chapter 6 suggests future directions and concludes this thesis.

Chapter 2

Related Work

LiveRAC draws on research in the fields of information visualization, perception, interaction design and computer systems. Our related work is presented as follows: Section 2.1 discusses visualization and interaction techniques. Section 2.2 describes computer network and systems analysis visualization tools. Section 2.3 discusses the SWIFT system which LiveRAC uses as a data source, and Section 2.4 discussed work relating to accordion drawing, the data display and interaction metaphor used by LiveRAC.

2.1 Information Visualization

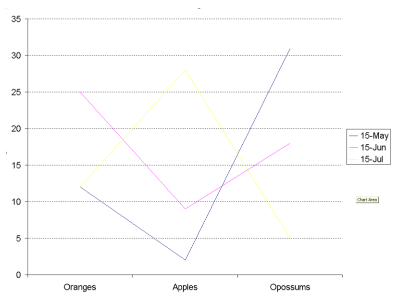
The objective of information visualization is to connect the data collection and aggregation power of computers with the human mind, a sophisticated pattern finding device. The human visual channel is the highest bandwidth perceptual system we possess for information acquisition [49]. Information visualization systems aim to sort and display information in a manner that allows a human operator to create mental models of the underlying data, and mine this data for correlations and outliers. Information visualization tools act as external memory aids for cognition, with more advanced systems providing capabilities for dynamic exploration and manipulation of the underlying structures.

Information visualization must provide spatial and representational encodings for data that is abstract in nature. For example, statistics from network devices such as number of bytes transferred, CPU load, or ICMP alarms received have no analogue in the physical world. Information visualization has many tools for displaying data, including the selection of a relevant spatial layout, colour, shape, patterns, and motion. Many of the guidelines for applying these techniques are available in Colin Ware's book *Information Visualization* [71].

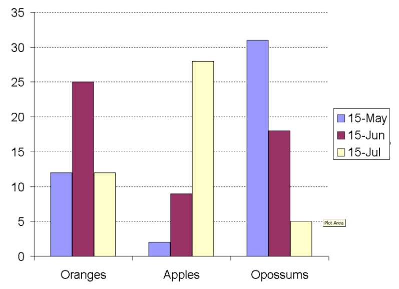
One high-level rule for designing visualizations has become immortalized in a mantra by Ben Shneiderman: 'Overview first, zoom and filter, then details-on-demand' [60]. Several disciplines of information visualization have evolved, taking different approaches to fulfilling the mantra's creed. Focus+context, discussed in more detail in Section 2.1.4, is a family of techniques for situating regions of focus, or user attention, in the context of the data space. Many of these techniques use distortion or aggregation to concurrently view the entire data field. Semantic zoom, a technique for creating area-aware representations, is implemented in systems such as Pad [51], discussed in more detail in Section 2.1.5. LiveRAC takes a focus+context approach, and integrates semantic zoom infrastructure.

Various toolkits have been released to support the development of information visualization applications. The InfoVis Toolkit by Jean-Daniel Fekete [22] provides scalable data structures and data representations, as well as supporting user interface widgets. Prefuse [27] by Jeffrey Heer *et al.* provides a wide range of visualization capabilities. These toolkits offer powerful visualization capabilities, but they do not provide infrastructure for all information visualization techniques. The visual metaphor for data layout in LiveRAC is accordion drawing which is not provided by either of these toolkits. However, LiveRAC does use a modified range slider widget from the InfoVis toolkit.

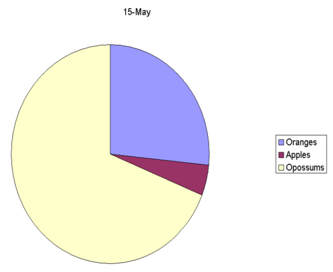
2.1.1 Statistical Graphics



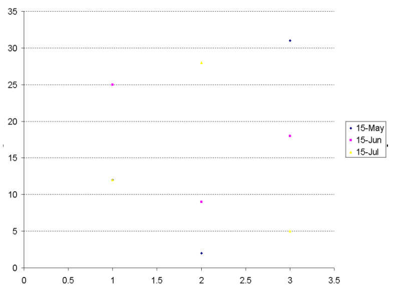
(a) Line chart



(b) Bar chart



(c) Pie chart



(d) Scatter chart

Figure 2.1: Statistical graphics are the projection of abstract shapes representing observed quantitative data onto a co-ordinate system. Here we present a number of examples of statistical graphics. (a) is a line chart, the slope of the line connecting plotted points visually embodies trend information. (b) is a bar chart, allowing users to do easy side by side comparisons. (c) is a pie chart, facilitating visual area comparisons. (d) is a scatter plot, an effective means of assimilating a large number of data points.

Statistical graphics, also called data graphics or quantitative graphics, is the projection of abstract shapes representing observed quantitative data onto a co-ordinate system. Traditional statistical graphics, which has been in use for centuries [8], uses dots and lines with labelled text to indicate values. Statistical representations are abstract from the data they represent; for example, a line chart can be used to indicate the relative numbers of apples, oranges and opossums at different sampling periods, but bears no resemblance to any of these objects. Many statistical graphics are commonly referred to as graphs, which are drawings representing one or more sets of data, often having two dimensions and using horizontal and vertical scalars for encoding. Graphs are relatively well explored and understood. Good resources on graphs include *The Elements of Graphing Data* [16] by William Cleveland, *The Semiology of Graphics* by J. Bertin [10] and numerous texts by Edward Tufte [67] [68] [69] [70]. Statistical graphics are in day-to-day use in academic and corporate environments. Popular and pervasive products such as Microsoft Excel provide computer assisted creation of statistical graphics, supporting charts such as column, bar, line, pie and scatter charts. LiveRAC implements a number of traditional statistical graphics, discussed in Section 3.4.

Information visualization is also concerned with finding novel data representations, and expanding the vocabulary of statistical graphics. Two recently developed statistical data representations that have been implemented by LiveRAC are two-tone pseudo colouring [57] described in Section 2.1.1.1 and sparklines, described in Section 2.1.1.2.

2.1.1.1 Two-Tone Pseudo Colouring

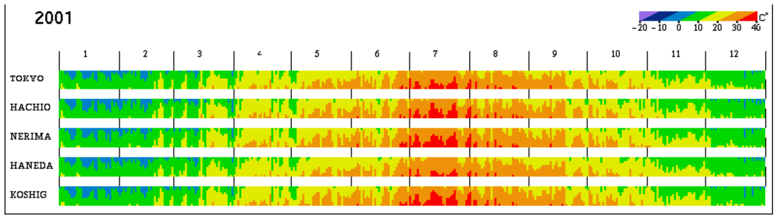


Figure 2.2: Two-tone pseudo coloring [57], is a visualization technique for large-scale univariate data. By using two colours for each scalar it is possible to more accurately read the value. Image © 2005 IEEE, reproduced with permission.

Two-tone pseudo colouring [57] is a statistical graphic for displaying large-scale univariate data. Pseudo colouring is a common approach for the visual representation of scalar data, where scalar data is mapped to a colour. Previous implementations of pseudo colouring have been either continuous, where colour value is directly linked to a continuous range of data values, or discrete where a data value range is mapped to a single solid colour.

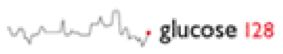
In two-tone pseudo colouring two colours are allocated to each value interval and are shaded to reflect the value. A scale of colours is created, with each colour building on the last, allowing a very small two-tone pseudo coloured graphic to represent a very large range.

Using two-tone pseudo colouring it is possible to precisely determine the value at each point. The rate of increase and decrease is also easily visible along the colour boundaries. One draw back of two-tone pseudo colouring is the requirement for a minimum height which ranges between 8 and 20 pixels for a typical display. The continuous and discrete pseudo colouring methods only minimally require one pixel, but in practical use typically need between 4 and 10.

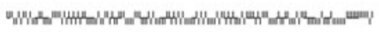
LiveRAC implements two-tone pseudo colouring as one possible representation for data.

2.1.1.2 Sparklines

Sparklines were introduced by Edward Tufte in Beautiful Evidence [69]. These small graphics represent trend information in a compact space, and can be compared to line charts with minimal axis and label information. The objective of sparklines is to display the current value and the trend leading up to the current value. This facilitates small multiple comparisons amongst dozens of graphics aligned side by side or vertically. At the smallest size a sparkline can appear inside



(a) Sparkline



(b) Binary sparkline

Figure 2.3: A sparkline [69] is a small graphic for representing trend information in a compact space. In (a) the label on the right represents the nature of the data being represented and a final value. A line plot indicates trend information. In (b) binary data is encoded in vertical lines called ‘whiskers’. This graphic represents a series of binary states.

of a single line of text.

Binary sparklines are similar to standard sparklines but show only binary information in the plot. The blocks are referred to as whiskers. Additional information can be encoded with text on the left or right label, and certain whiskers of the binary sparkline can be marked in a different colour to denote unusual events during that interval.

LiveRAC provides data representations for both sparklines and binary sparklines.

2.1.2 Time Series Data

Time series data consists of any data elements which have a time dimension. A typical form of time series data is a collection of [time, value] pairs. One example of this might be monitoring data from a network interface which stores a time stamp and the number of bytes transferred: [16 AUG 2006, 25000]. Many of the traditional statistical graphics discussed in Section 2.1.1 can be used to represent time series data, such as line and scatter charts.

Because any regularly sampled data has a time dimension, time series data is common and it is an area of interest and active research in information visualization. Wijk *et al.* present a paper for cluster and calendar visualizations of univariate time series data [73]. They use a linked view between a calendar, and a data display which clusters similar daily patterns. Weber *et al.* use spirals [72] as an alternative to bar and line charts for visualizing time series data. They assert that spirals better support identifying periodic structures in the data. In TimeSearcher [29] by Hochheiser and Shneiderman, users retrieve time series through dynamic queries. The user selects time regions of interest called a 'time box' which provides details-on-demand in a linked view. The visual elements of TimeSearcher are placed in traditional scrollable GUI frames. Viz-Tree [44] discretizes time series data, and builds tree structures from the results. Similarities between regions of the time series plot can be detected by pattern matching the discretized data. Differences between two time series can be shown by highlighting components of the tree structure.

All data visualized by LiveRAC has a time dimension. We implement a number of the standard statistical graphics techniques and two-tone pseudo colouring for visualizing this data. However, where previous approaches to visualizing time-series will often use a clustering technique [73], LiveRAC provides a matrix view of time series data. LiveRAC also uses dynamic queries in a linked view for interacting with the time dimension, similar to TimeSearcher. LiveRAC differs from TimeSearcher by using the accordion drawing visual metaphor described in Section 2.4.

2.1.3 Interaction

Interaction is the activity of manipulating some aspect of a visualization. This may involve navigation actions in the visualization such as zooming, panning, and selecting regions of interest, or altering the underlying data. Interaction is one area where computer-supported information visualization provides capabilities entirely distinct from what is available with print graphics. An early paper introducing visualization interaction techniques is Selective Dynamic Manipulation of Visualizations (SDM) [15]. SDM proposed a new set of interactive techniques for 2D and 3D visualizations that supported user selection, dynamic system response, and the direct manipulation of visual objects. Other resources for interaction include several chapters in the book *Information Visualization* by Ware [71]. LiveRAC supports navigation of the data space and the selection of regions of interest.

Tightly coupled interaction has been shown to be an important component of visualization design. Work by Shneiderman on dynamic queries [59] suggests that allowing users to explore a data set by modifying parameters tightly coupled to the visualization can improve user satisfaction and lead to dramatic performance improvements. Later work by Ahlberg and Shneiderman [3] showed the importance of rapid response to input for this type of interaction. LiveRAC uses linked views to provide dynamic, progressive queries on data elements.

2.1.3.1 Matrix Visualization

Matrix layouts, also called grid layouts, are a common way to encode tabular data, and should be familiar to many users in the form of spreadsheets. Information visualization tools have also used a matrix approach to encoding data, which has been explored in various forms [4] [53] [40] [61] [62] [64].

2.1.3.2 Reorderable Matrix Visualization

With computer supported visualization, it is possible to interactively reorder a matrix view to help find correlations in the data, an idea introduced by Bertin [9] in his book *Graphics and Graphic Information Processing*. In Bertin's example, he replaces numeric cell values with correspondingly scaled ink shapes. A paper by

Siirtola [61] studied how users interact with reorderable matrix layouts using these ink shapes, and how they were used to find correlations. The results of this study indicate that user satisfaction was high and that users were able to find correlations easily using this approach. Since the study by Siirtola, a number of systems have taken a reorderable matrix approach including Online Partner Lens [26] and VistaClara [37]. Kincaid *et al.* explore the use of a reorderable matrix of line charts in a focus+context display with Line Graph Explorer [38]. Polaris [64] is an interactive table-based visualization system for rich hierarchical data structures that supports reordering.

LiveRAC uses a reorderable matrix layout for data. The key differences between LiveRAC and previous approaches are the use of the accordion drawing metaphor. A reorderable matrix layout has not previously been implemented in accordion drawing. Furthermore, other reorderable matrix visualization systems such as Polaris have used comparatively simple cells that do not support semantic zoom.

2.1.4 Focus+Context

LiveRAC uses focus+context [63], a family of visualization strategies that involves situating one or more regions of user attention in a surrounding context field. In many focus+context displays, all available data can be made visible regardless of navigation actions inside of the data space, although regions of the display may be expanded, compressed or distorted in these approaches.

Some approaches to focus+context displays use distortion, such as fisheye techniques [23] [13] and nonlinear magnification fields [14]. These techniques often yield better visual flow from context to focus at the expense of orthogonality. Rectilinear fisheye attempts to address this issue by preserving orthogonality in the distortion. An early application of rectilinear fisheye lenses is the Document Lens [54] by Robertson *et al.* The Document Lens provides a view of a complete document projected onto a two dimensional plane, zoomed to fit entirely in the display area. The user interacts with the document by using a rectilinear fisheye lens to view regions of interest inside of the document. Context is provided along the edges of the lens. Contextual information can also be displayed through layered lenses as in the work by Bier *et al.* [11].

Distortion-free approaches to focus+context displays also exist. One example of a distortion-free technique is aggregating context regions into glyphs [12] [52]. Table Lens [53] is a non-distortion based approach for viewing tabular data, and is described in more detail in Section 2.1.4.1 below.

2.1.4.1 Table Lens

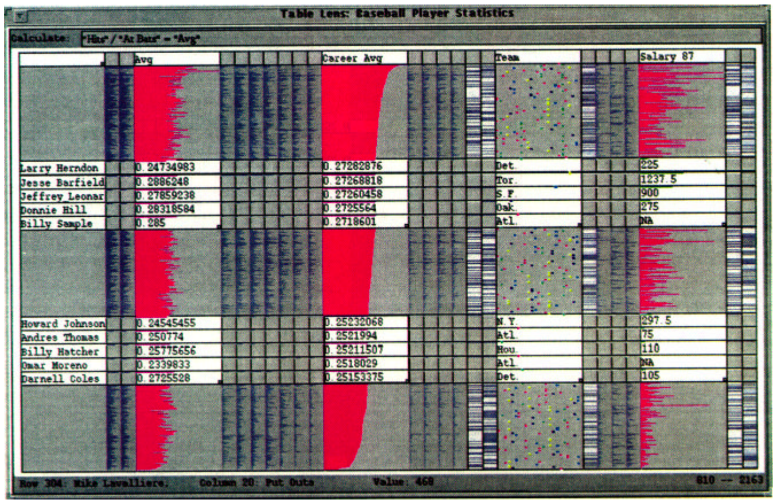


Table Lens: Baseball Player Statistics

Calculate: "Hits" / "At Bats" = "Avg"

	Avg	Career Avg	Team	Salary 87
Larry Herndon	0.24734983	0.27282876	Det.	225
Jesse Barfield	0.2886248	0.27268818	Tor.	1237.5
Jeffrey_Leonar	0.27859238	0.27260458	S.F.	900
Donnie Hill	0.28318584	0.2725564	Oak.	275
Billy Sample	0.285	0.2718601	Atl.	NA
Howard Johnson	0.24545455	0.25232068	N.Y.	297.5
Andres Thomas	0.250774	0.2521994	Atl.	75
Billy Hatcher	0.25775656	0.25211507	Hou.	110
Omar Moreno	0.2339833	0.2518029	Atl.	NA
Darnell Coles	0.2725528	0.25153375	Det.	105

Row 304: Mike Lavalliere. Column 20: Put Outs Value: 468 810 -- 2163

Figure 2.4: Table Lens [53], a focus+context visualization tool for large scale tabular data. Regions of the display can be expanded by user navigation.

Table Lens [53] is a focus+context visualization for large scale tabular data. It mutates table layout without distorting rows or columns, allowing cells to retain their rectangular shape during magnification. Users can manipulate the view by expanding and contracting regions of the display.

Table Lens offers a variety of graphical representations for data including text, colour, shading, length and position. Table Lens attempts to determine the best representation using a variety of information including the value, value type, region type, cell size, user preferences, and a user directed highlighting system called spotlighting.

LiveRAC shares the same rectilinear focus+context approach to visualizing tabular data as Table Lens. The key differences between LiveRAC and Table Lens are the use of the accordion drawing focus+context metaphor, an expanded set of statistical graphic options for cells, LiveRAC's time range selection capabilities, and the specific systems application domain. LiveRAC is designed for viewing live, streaming computer system and network data using a client-server architecture, whereas Table Lens views static abstract data loaded from files.

2.1.4.2 Rubber Sheet Navigation

Rubber sheet navigation is a focus+context interaction metaphor developed by Sarkar *et al.* [58], where the user manipulates the display as though it was a rubber sheet tacked down at the borders. Navigation operations can stretch and compress arbitrary regions of the display. Multiple focus areas are possible with rubber sheet navigation, as different regions can be stretched to a desired size.

There are two different approaches to rubber sheet navigation, one using orthogonal stretching and the other using polygonal convex hulls, representing non-distorting and distortion based approaches respectively. In orthogonal stretching, illustrated by Figure 2.5, the user is limited to rectilinear operations on the display.

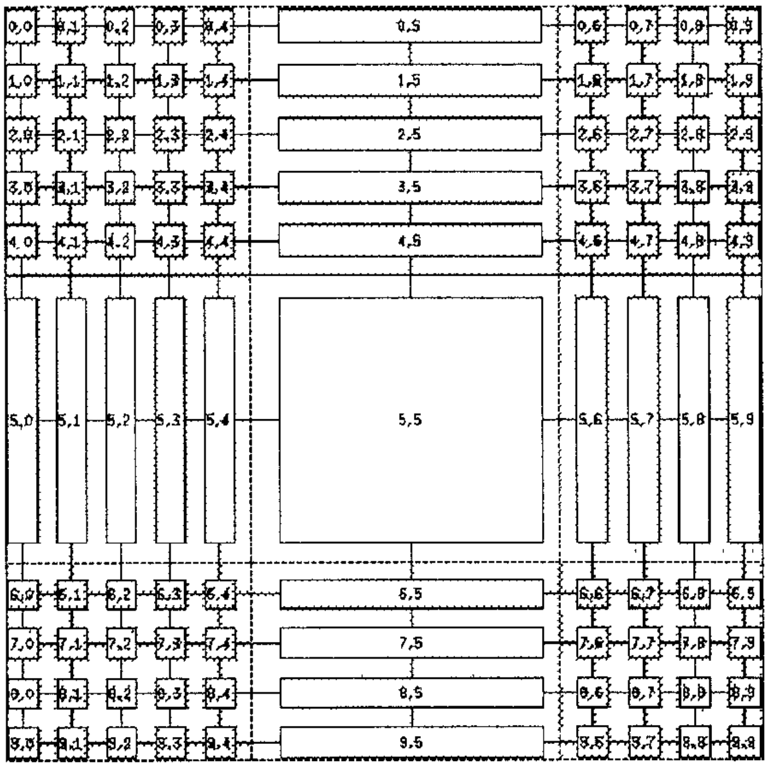


Figure 2.5: Rubber sheet navigation [58] with orthogonal stretching. Symmetry is preserved, but discontinuity is visible in the scaling between the focus and context regions.

Stretch and compress operations are locked to the cell boundaries. The advantage of this approach is the preservation of the orthogonality of the data set. However, it results in discontinuities in data density at the boundaries between the focus and context regions. Polygonal convex hull stretching, as shown in Figure 2.6, enables the selection of arbitrary polygonal interest areas within the visualization. The distortion between context and focus regions takes place in a smooth gradient; however, the symmetry of the grid boundaries is violated.

The data display and interaction mechanism in LiveRAC is accordion drawing, which implements and extends rubber sheet navigation. Accordion drawing is

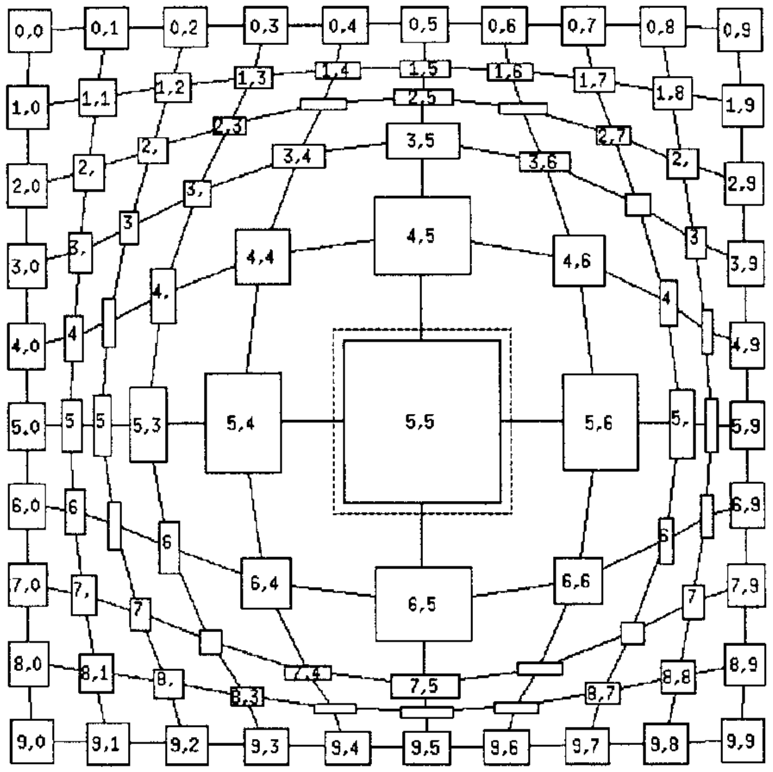
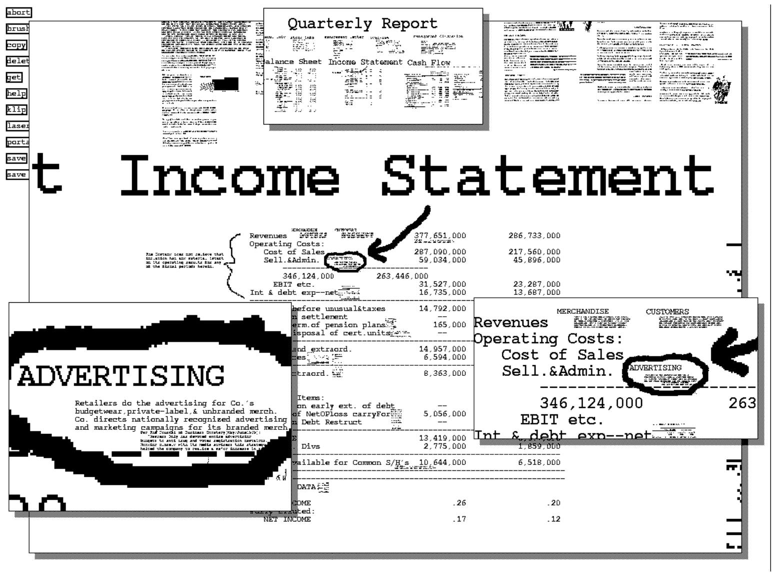


Figure 2.6: Rubber sheet navigation [58] with polygonal stretching. Scaling between focus and context regions is smooth but symmetry is lost.

discussed in Section 2.4.

2.1.5 Semantic Zoom

Semantic zoom is a visualization technique that represents graphic objects differently depending on the apparent size of the object to the user. This technique does not simply increase the polygon count or detail of an object that is occupying more display pixels, but actually changes the physical manifestation of the object to better suit its display size. For example, when occupying a small region of space a calendar object might display only a list of high priority events and the dates of



```
abort
brush
copy
delet
get
help
klip
laser
porta
save
save

Quarterly Report
Balance Sheet  Income Statement  Cash Flow

t Income Statement

Revenues                               377,651,000      286,733,000
Operating Costs:
   Cost of Sales                       287,090,000      217,560,000
   Sell.&Admin.                         59,034,000       45,896,000
    ----------------------------------------
      346,124,000           263,446,000
     EBIT etc.                          31,527,000       23,287,000
Int & debt exp--net                     16,735,000       13,687,000
before unusual&taxes                    14,792,000
n settlement                                --
erm.of pension plans                       165,000
isposal of cert.units                       --
nd extraord.                            14,957,000
es                                       6,594,000
traord.                                  8,363,000
Items:
on early ext. of debt                       --
f NetOPloss carryFor                     5,056,000
n Debt Restruct                             --
E                                       13,419,000
. Divs                                   2,775,000        1,859,000
ailable for Common S/H's                10,644,000        6,518,000
DATA
COME                                        .26              .20
ted:
NET INCOME                                  .17              .12

ADVERTISING
Retailers do the advertising for Co.'s
budgetwear,private-label,& unbranded merch.
Co. directs nationally recognized advertising
and marketing campaigns for its branded merch.

                MERCHANDISE      CUSTOMERS
Revenues
Operating Costs:
   Cost of Sales
   Sell.&Admin.   ADVERTISING
   ----------------------------------
      346,124,000                 263
     EBIT etc.
Int & debt exp--net
```

Figure 2.7: The Pad [51] user interface presents the visual metaphor of an infinite desktop. Objects and documents can be accessed by zooming and panning inside of the interface. Representations of objects can change depending on the perceived size of the object to the user, a technique referred to as semantic zoom.

those events in text. When the region is enlarged, the calendar can lay out the more familiar month view. The first visualization system to implement semantic zoom was Pad [51].

The Pad visualization system is a pan and zoom interface intended to leverage the human brain's natural spatial awareness. Pan and zoom interfaces provide a two-dimensional field of data over which the user navigates. The user's perspective can be thought of as a camera floating over a planar surface. The user looks at the entire field of data by zooming out to the maximum level, where the edges of the data field align with the view frustum of the virtual camera. The user obtains details by zooming on regions of the display, and then performing pan operations.

In Pad, information is presented on an infinite two dimensional plane. Applications can be located anywhere within this space. Typical navigation activity involves the use of a ‘portal’, a navigation action analogous to using a magnifying glass that allows the user to zoom into one aspect of the display. Figure 2.7 shows a screen shot from Pad.

Semantic zoom in Pad is provided by using portals and portal filters to provide non-literal views of the data. As an example, a portal could be created that displays tabular data as a bar chart, where another portal would reveal only the underlying text and numbers.

The primary limitation of pan and zoom interfaces is a loss of information context. The user cannot see beyond the edges of the view frustum, and it may be difficult for the user to arrange a zoom level which allows side-by-side comparisons of data at a suitable focus level if the regions of interest are geographically distant on the data plane. Overview windows have been investigated as a mechanism for providing an overview and improving navigation. However, quantitative evaluations using overview windows have been mixed [30] [48] [28].

Semantic zoom has also been implemented in focus+context visualizations such in work by Bartram *et al.* [5]. Bartram uses fisheye distortions to create user-directed regions of interest. Expanded regions can show multiple graphic representations, depending on the degree-of-interest.

LiveRAC implements a variant of semantic zoom. Unlike previous implementations, LiveRAC’s semantic zoom is in an accordion drawing visualization. Unlike Pad, LiveRAC is specifically targeted for providing a matrix layout of time series data, rather than a new desktop metaphor.

2.2 Network and Systems Visualization Tools

A large body of visualization tools have been developed for performing security and fault analysis on computers and computer networks. These range from basic tools such as packet analysis software like Ethereal [18] to simple web based viewers for intrusion detection alarm data such as Analysis Console for Intrusion Databases (ACID) [2].

Early work in computer network information visualization research used node-link as well as matrix representations of network structure, such as those done by Bertin [9]. Becker *et al.* developed an application called SeeNet [6] which focused on the data in the network rather than the network structure. SeeNet was used to visualize circuit switched long distance telephone data, rather than packet switched internet traffic. SeeNet introduced a mechanism for directly manipulating parameters of the visualization. SeeNet3D [19], introduced by Cox *et al.* and also using telephone data, provided a number of three-dimensional data views in an attempt to reduce edge crossings. Although metrics associated with the nodes and links was displayed, both systems were focused on using the underlying network topology for their representations.

In this section we survey some recent visualizations for computer cluster, log, network and intrusion alarm data.

2.2.1 Cluster Visualization Tools

A tightly integrated and centrally managed group of computer systems is often referred to as a cluster or a farm. Although the term cluster typically refers to scientific or research usage for executing a parallel program, and a farm typically refers to a group of systems running individual instances of applications amongst whom user load is automatically distributed, the two terms are often used interchangeably. Managing these distributed systems has led to a new group of monitoring tools, both from the open source and research community. Clumon [17] is a system for gathering cluster performance data and displaying it in simple web-based graphic and list based formats. Ganglia [24] is a similar tool to Clumon which uses multicasting to reduce network load, and also presents a mixed graphic and list based web interface, and uses RRDTool [56] to manage data storage and generate statistical graphics. RRDTool is a popular open source system for storing data in round robin databases, a mechanism for maintaining fixed size databases and aggregating archive data. NVisionCC [74] is a scalable visualization for computer clusters, though it is focused on system security rather than performance monitoring.

LiveRAC can be used to visualize cluster data, and is also well suited to more heterogeneous network environments, with access to many disparate data sources. It is also unique in using the accordion drawing approach over other cluster visualization tools.

2.2.2 Computer Log Visualizations

Most computer systems maintain log files of system activities. UNIX based systems use a log interface called syslog, and Microsoft Windows systems have an event logger. Many individual applications maintain separate log files and log formats. Several visualizations have used computer log files as a data source. Eick *et al.* applied the SeeSoft [21] tool for visualizing line oriented statistics to log files in *Graphical Analysis of Computer Log Files* [20]. Using this implementation of SeeSoft Eick *et al.* were able to present the user with a colour coded overview of the log file, and linked navigation for increasing detail to the level of viewing raw log text. MieLog [65] is a visual log browser which does statistical analysis and displays results in an interactive display with linked views. Users are presented with a log file overview, and multiple levels of detail up to the full text of the log contents.

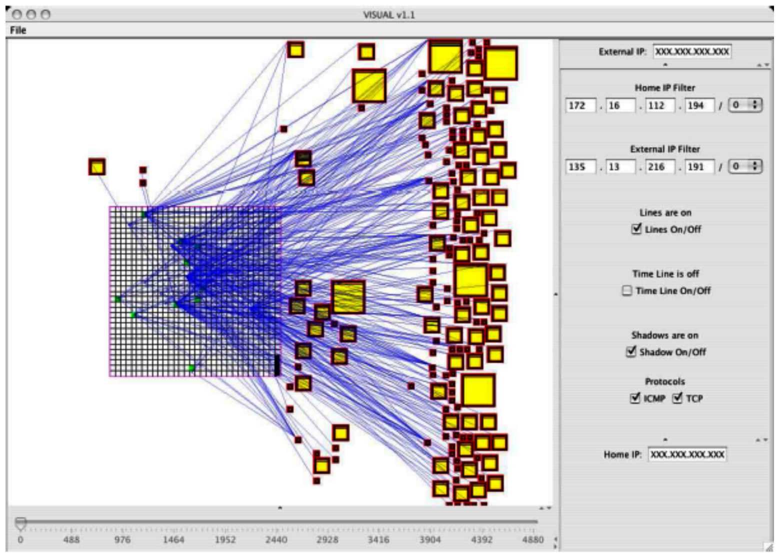
The alarms and metrics monitored by LiveRAC have many similar properties to system log files as they are essentially time sorted lists of text. LiveRAC differs from the above tools by integrating the visualization of this data with other computer metrics. Furthermore, neither MieLog nor SeeSoft operate on streaming data, and therefore can only visualize snapshots of the current system state. LiveRAC uses SWIFT [41] as a source for streaming systems and network data. This provides a springboard for creating an interactive visualization that can monitor systems in real time.

2.2.3 Network Traffic Visualization

In this section we discuss visualization tools for looking at network traffic patterns. Although LiveRAC allows administrators to view network statistics like many of the tools described below, it also encodes data from internal system metrics such as CPU and memory, and provides an interface to the alarm and ticketing system.

NVisionIP[43] is a cluster security visualization tool developed at NCSA. It uses NetFlow data to display a visualization of filter-selected traffic on an entire class B network which can be based on the number of bytes transmitted or the number of flows. NetFlow is a Cisco proprietary IOS software feature and protocol for exporting network traffic flow records. The global view of NVisionIP presents a host-by-subnet grid based co-ordinate system of the monitored IP space. The global view supports interaction by allowing users to investigate more detailed information about an individual host. A small-multiple view allows an administrator to view an entire region of NetFlows in more detail, including two bar charts;

one displays flow data for a common subset of ports, and the other shows flow data for all other ports. The objective of NVisionIP is to provide network administrators with a single interface for viewing the state of the network. Although NVisionIP is intended primarily as a security tool, it can also serve as a useful tool for gaining an overview of network status and utilization. NVisionIP was later enhanced [42] to allow administrators to build rule sets based on observed visual patterns in the data. LiveRAC differs from NVisionIP by taking a focus+context approach to viewing network and systems data. Furthermore, it arranges the data in a reorderable grid as opposed to a fixed IP address view. LiveRAC focuses on the state of the monitored system and does not track or display per-connection data.



External IP: XXX.XXX.XXX.XXX

Home IP Filter
172 . 16 . 112 . 194 / 0

External IP Filter
135 . 13 . 216 . 191 / 0

Lines are on
[✓] Lines On/Off

Time Line is off
[ ] Time Line On/Off

Shadows are on
[✓] Shadow On/Off

Protocols
[✓] ICMP [✓] TCP

Home IP: XXX.XXX.XXX.XXX

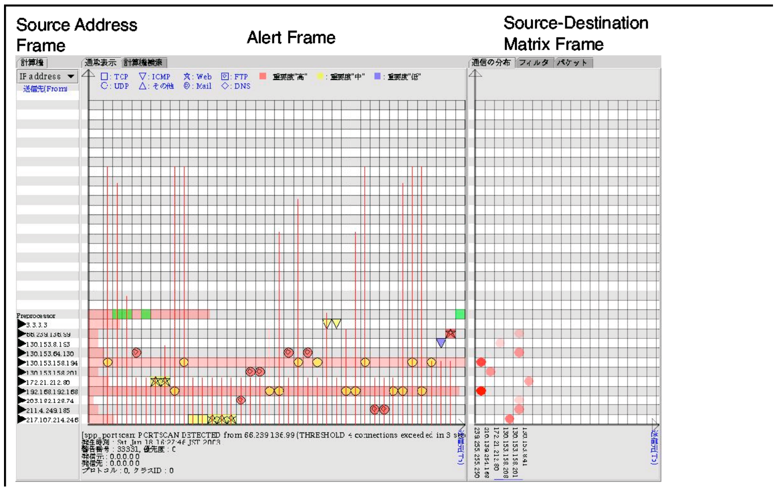
Figure 2.8: VISUAL [4] is a tool to help administrators of small network computer systems. A home IP filter allows users to specify local IP ranges.

Visual Information Security Utility for Administration Live (VISUAL) [4], shown in Figure 2.8, is a tool for assisting system administrators of small computer networks to quickly understand the security state of their network. VISUAL divides network space into local network address space and remote network address space (the rest of the Internet). It uses data from the log files of Ethereal [18] or tcpdump [33] to produce its visualizations. VISUAL provides a quick overview of the current and recent communication patterns in the monitored network. A grid view represents a matrix of local hosts; lines are drawn to indicate connections to remote hosts drawn outside of the matrix. Using the visualization it is trivial to see whether a single external host communicated with a large number of internal hosts, which may be relevant for detecting network scanning activity. The volume of network traffic is represented by the size of the external host box. The system can scale to a size of approximately 2500 home hosts and 10,000 external hosts. Unlike VISUAL, LiveRAC does not take a connection-oriented approach. Although LiveRAC provides visualizations for network interface statistics, it does not visualize per-connection data. Furthermore, LiveRAC visualizes system and alarm metrics which are not handled by VISUAL.

2.2.4 Intrusion Detection Visualization

In this section we discuss recent visualizations for Network-based Intrusion Detection Systems (NIDS). Although LiveRAC handles intrusion alarms, it also visualizes network statistics and many types of ticketed alarms which are not limited to intrusion events. By drawing on more data sources, LiveRAC provides administrators with additional context for alarm events.

SnortView [39] is a visualization for NIDS, and uses the logs of a popular open source intrusion detection package called Snort [55]. In a typical enterprise NIDS environment, administrators must cope with a large number of false positive and false negative results. Typically a NIDS implementation must be tuned, pruning signature entries that lead to large numbers of false positives, at the expense of increasing the number of false negatives. SnortView allows the visual analysis of



```
[spp_portscan: PORTSCAN DETECTED from 66.239.136.99 (THRESHOLD 4 connections exceeded in 3 s
発生時刻 : Sat Jan 18 16:27:46 JST 2003
警告番号 : 33331, 優先度 : 0
発信元 : 0.0.0.0 0
発信先 : 0.0.0.0 0
プロトコル : 0, クラスID : 0
```

Figure 2.9: SnortView [39] is a visualization for Network-based Intrusion Detection Systems (NIDS). IP addresses are arranged on the Y axis and time on the X axis. Glyphs represent the different types of alarms.

NIDS log files allowing administrators to visually explore the alarm data and find patterns, reducing the need to over cull the signature database. SnortView uses a matrix based approach as seen in Figure 2.9. IP addresses are arranged on the Y axis and time on the X axis. Glyphs represent the different types of alarms. One problem with SnortView is the scalability of a fixed matrix of machines and alarms. On large networks users may be forced to scroll to see details, and could easily overlook alarms arriving in a region not currently visible on the display. SnortView is also limited to a four hour time monitoring window.

IDS RainStorm [1] is another approach to visualizing network intrusion detection using the Snort [55] intrusion detection system. IDS RainStorm divides display space into columns that represent 2.5 class B address spaces for a total of 163,830 hosts. Coloured dots represent hosts where alarms have been received. The user is able to zoom on regions of the display in a pan and zoom metaphor, and can select individual alarms for more details. One significant problem with

IDS RainStorm is that it renders an entire address space, rather than only the monitored hosts within that space. In a typical network, the number of nodes can be relatively sparse in comparison with the size of addressable space. LiveRAC uses screen space efficiently by rendering only monitored nodes.

2.3 SWIFT

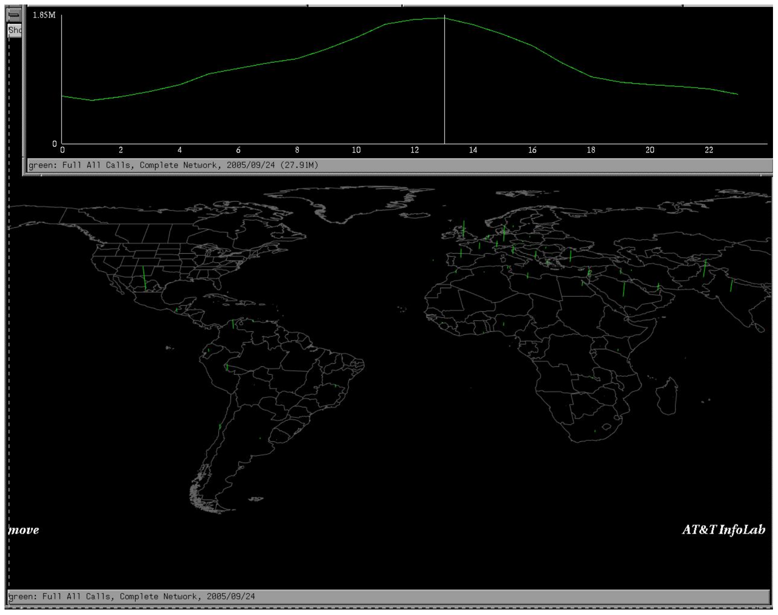

1.85M
0
0
2
4
6
8
10
12
14
16
18
20
22
green: Full All Calls, Complete Network, 2005/09/24 (27.91M)
move
AT&T InfoLab
green: Full All Calls, Complete Network, 2005/09/24

Figure 2.10: SWIFT [41] is a streaming data processing and query back-end combined with two and three dimensional front-end visualization techniques. SWIFT gathers data from multiple unique data sources and merges them into well-defined, self-describing records which can be streamed or queried by visualization applications.

SWIFT [41] is a set of data storage, aggregation and visualization tools that allows data from many distinct sources to be integrated into a single self-describing data format. These data sources could include SNMP, intrusion detection systems, or Microsoft Windows system monitors. The underlying system is extensible so

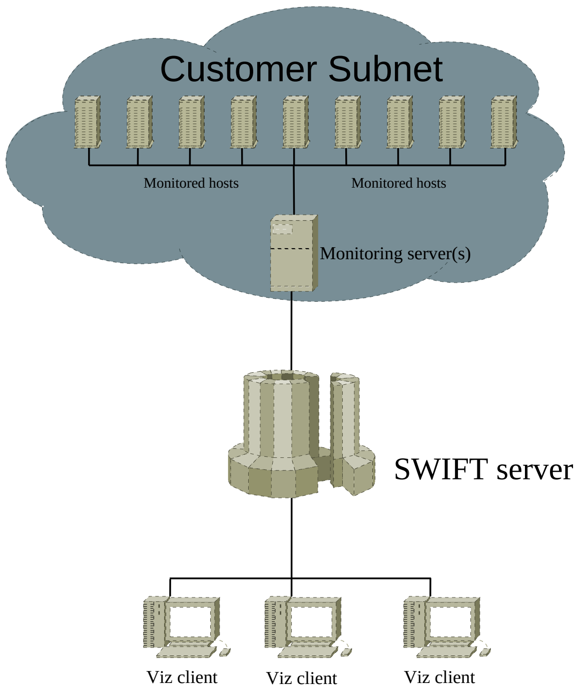


Figure 2.11: Multiple monitoring hosts sitting on customer subnets gather statistics and update the SWIFT database server. The visualization clients query the SWIFT server for metric, alarm and aggregate data. LiveRAC provides a new visualization client for SWIFT that takes an information-dense matrix approach as opposed to previous visualizations which focused on node-link, geographic and list views.

virtually any data source can be mapped into the schema. Although SWIFT has many features in common with other database systems, SWIFT has been optimized for streaming data.

SWIFT queries and filters can be compiled into shared libraries that execute rapidly over a set of streamed records. SWIFT also provides a number of visualizations such as the geographic view of telephone data shown in Figure 2.10. Other visualizations provided by SWIFT include text, node-link, and other geographic views.

The SWIFT system uses a client-server architecture illustrated in Figure 2.11. LiveRAC harnesses the data collection and processing capabilities of SWIFT to provide a new front-end visualization for the SWIFT data. Where previous SWIFT visualizations have sought to show the physical relationships between network devices, LiveRAC's goal is to provide a scalable, information-dense matrix encoding for all of the data associated with these devices. LiveRAC uses a threaded architecture to load data from the SWIFT system, and provides its own unique visualizations based on the SWIFT data. We use a modified Java interface from SWIFT to query the server.

2.4 Accordion Drawing

LiveRAC uses an orthogonal descendant of rubber sheet navigation called accordion drawing, first introduced in TreeJuxtaposer [47] (TJ1) by Munzner *et al.* Accordion drawing is a combination of rubber sheet navigation and Guaranteed Visibility (GV). During navigation landmarks or regions of interest may become compressed such that they are not visible if another region of the display is dramatically expanded, or when viewing a very large data set. In many application domains it can be desirable to have regions of the display which are always visible irrespective of navigation activities. These regions, called critical zones, are formally identified in a paper by Jul *et al.* [36] and addressed by GV. Visualization systems implementing GV ensure that marked regions will remain visible regardless of the information density.

In accordion drawing, screen space is divided into a grid of cells by horizontal and vertical lines called split lines which form the cell boundaries. These split lines can be manipulated interactively, changing the relative sizes of the underlying cells. TJ1 was designed to provide a highly scalable mechanism for the side-by-side comparison of phylogenetic trees. TJ1 uses a quadtree data structure for split lines, with prioritized rendering for marked regions. Context nodes are filled in on a best-effort basis and no limit is placed on the amount of compression that can be applied to them. TJC [7] is a C-based implementation of TreeJuxtaposer that removed the quadtree data structure for layout in favour of two separate data structures representing each axis of the split line grid separately. The resulting system was able to render fifteen million nodes in under one second and used significantly less memory than TJ1.

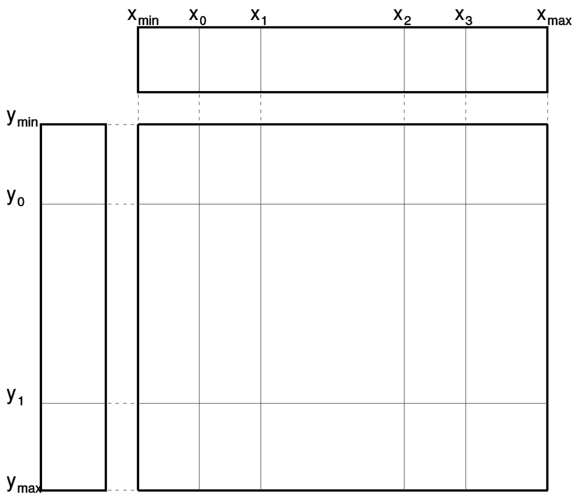


Figure 2.12: This grid shows horizontal and vertical split lines. The X_{min}, x_{max}, Y_{min}, and Y_{max} have immovable absolute co-ordinates; these are called the 'stuck lines'. There are four movable lines on the horizontal axis, and two movable split lines on the vertical axis forming a grid with fifteen cells.

Other accordion drawing applications have included SequenceJuxtaposer [62] in Figure 2.13 for viewing genetic sequences and PowerSetViewer [40] shown in Figure 2.14 which displays power sets. Both of these applications render rectangular regions of colour to represent the underlying data which is mapped into a grid. PowerSetViewer has the ability to add and delete data in the split line grid, and uses

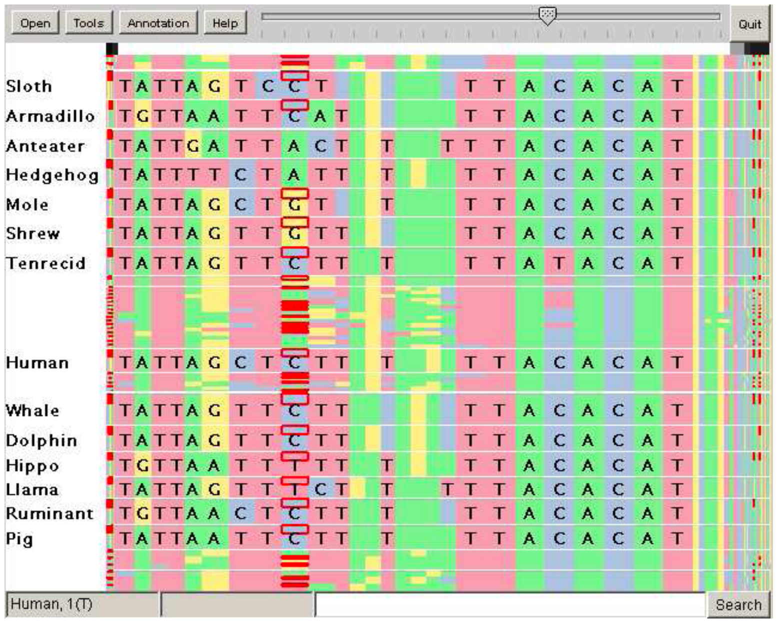
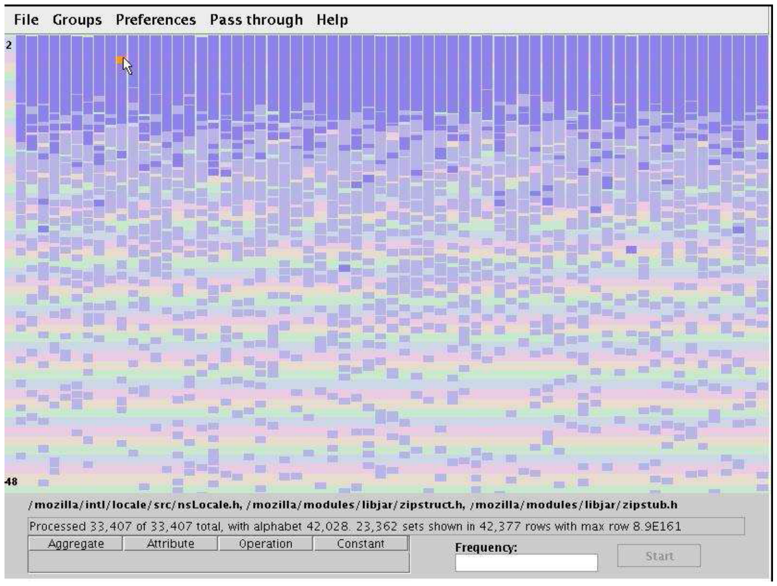


Figure 2.13: Sequence Juxtaposer [62] is a visualization system for genetic sequences using the accordion drawing infrastructure.

a client-server architecture to retrieve data from a server. Power sets are enumerated as a single one dimensional list, placed in a rectangular grid of fixed width. The result is a grid with a small number of columns and very large number of rows. PowerSetViewer does not allocate structures for the entire addressable space of the power set, but instead creates a grid in which it can draw sparsely distributed subsets within the power set. PowerSetViewer uses an algorithm for enumerating the power sets governed by Formula 2.1. The resulting enumeration is used as a key for inserting new grid lines into the red-black tree [66] based data structure. One limitation of the implementation of dynamic split lines in PowerSetViewer is that the enumeration approach is restricted to use by power sets, and each power set has a fixed location. If a power set exists, it has a well defined location corresponding to the index computed by the enumeration algorithm. Power sets cannot therefore be arbitrarily located elsewhere in the visualization through user interaction.

$$\sum_{i=1}^{m-1} \binom{A}{i} + \sum_{i=1}^{m} \left[\binom{A - p_{i-1}}{m-i+1} - \binom{A - p_i + 1}{m-i+1} \right] \tag{2.1}$$



/mozilla/intl/locale/src/nsLocale.h, /mozilla/modules/libjar/zipstruct.h, /mozilla/modules/libjar/zipstub.h

Processed 33,407 of 33,407 total, with alphabet 42,028. 23,362 sets shown in 42,377 rows with max row 8.9E161

Figure 2.14: PowerSetViewer [40] is a visualization system for viewing power sets. Power sets are represented by a single enumerated sequence of nodes which are line wrapped repeatedly creating a grid with a small number of columns and very large number of rows. Set cardinalities are separated using alternating background colours.

Partitioned Rendering Infrastructure for Scalable Accordion Drawing (PRISAD) [34] by Slack *et al.* is a Java-based framework for building applications requiring rubber sheet navigation and guaranteed visibility. Several applications have been developed for the PRISAD infrastructure, including ported implementations of TreeJuxtaposer and SequenceJuxtaposer. PRISAD introduces rendering algorithms that are based on the partitioning of the data structures into screen space, which significantly improves the rendering performance of large data sets, and helps prevent the over-and-under culling of nodes. Because the implementation is pixel based, rendering time is bounded by a function of the number of pixels being displayed rather than the size of the entire data set. LiveRAC uses and extends the PRISAD infrastructure described in detail in Section 4 by introducing fully dynamic data structures that allow split lines to be added and removed at run time, and providing a framework for semantic zoom in accordion drawing.

Figure 2.12 shows a split line grid for PRISAD. In PRISAD the drawing area is divided into cells using horizontal and vertical split lines. The minimum and maximum lines on each axis have fixed absolute co-ordinates, a condition which is referred to as being ‘stuck’. The position of the other lines can be manipulated by user navigation actions.

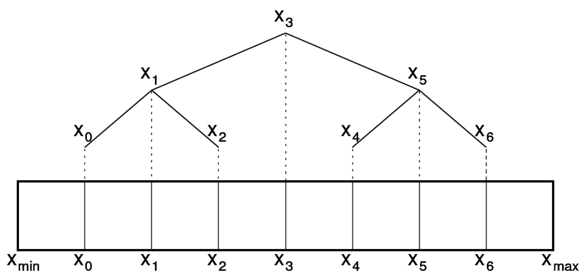


Figure 2.15: Split lines are stored in a balanced binary tree implemented over a one dimensional array. The min and max positions are not stored in the tree. Each split line has a relative value that determines its position between its min and max bound.

In PRISAD split lines are stored in a balanced binary tree data structure implemented using a one-dimensional array shown in Figure 2.15. Each axis has its own data structure which is handled separately. Each split line has a relative value that determines its position within its boundaries. The root absolute position can be found by the equation $\frac{X_{min}}{X_{max}}Rel_{root}$ where X_{min} is the absolute position of the minimum stuck line, X_{max} is the position of the maximum stuck line and Rel_{root}

is the relative value of the root split line. The absolute position of any split line can be determined by ascending the tree from the target split line, and adjusting its position based on the position of its ancestor.

Chapter 3

LiveRAC Overview

LiveRAC is a visualization system for monitoring computer performance statistics and alarm data. Our objective with LiveRAC is to provide a scalable, cross platform tool which integrates multiple data channels and supports both real-time and historical visualizations.

LiveRAC is a Java-based application that addresses our cross-platform requirement, although it requires native bindings for OpenGL using the Java OpenGL (JOGL) library [35]. Currently bindings for JOGL are available for Windows, Linux, Mac OS X and Solaris, with source code available for building on other platforms. The plethora of networking hardware and vendors means that tools must work well in a heterogeneous environment if they are going to support the needs of network administrators. LiveRAC is designed to integrate different types of hardware platforms including network appliances, servers and workstations. Most platforms have multiple data sources for system monitoring data, and across platforms there are dozens of different protocols, proprietary and standardized, for gathering this data. LiveRAC leverages the back-end data services of SWIFT [41] to provide integration for monitoring many of these data sources.

LiveRAC takes a matrix visualization approach, presenting data in a grid of cells similar to a spreadsheet. Unlike a spreadsheet, LiveRAC uses a focus+context technique called accordion drawing that allows the user to navigate system data without losing an overview of system state. LiveRAC allows users to view data in real time, or examine historical data for trend information using dynamic queries.

In this chapter we provide a detailed overview of the LiveRAC application, shown in Figure 3.1. Section 3.1 describes the visual components of our system. Section 3.2 describes the semantic zoom infrastructure that allows the grouping of related alarms and metrics, and selecting representations for data. Section 3.3 describes how LiveRAC interaction supports user tasks. In Section 3.4 we discuss the features of jGLChartUtil, an OpenGL charting library developed for LiveRAC.

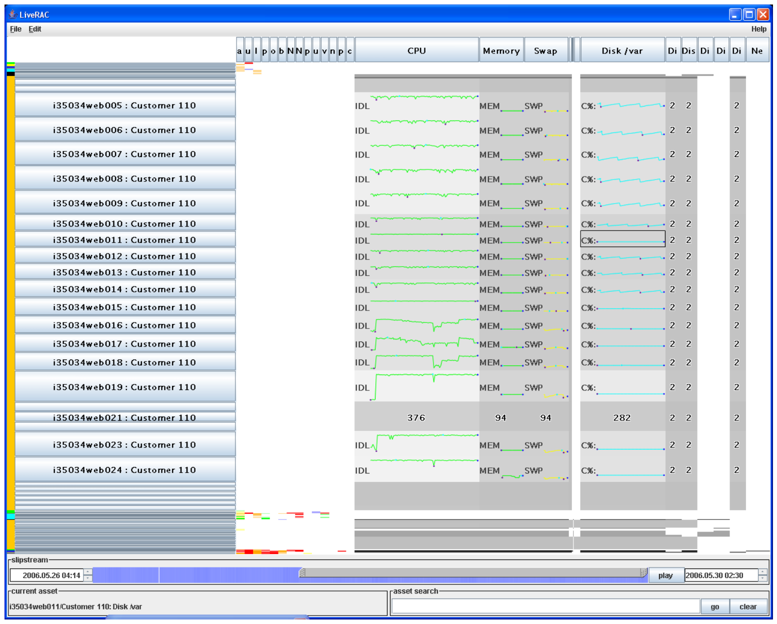

LiveRAC
File
Edit
Help
CPU
Memory
Swap
Disk /var
i35034web005 : Customer 110
i35034web006 : Customer 110
i35034web007 : Customer 110
i35034web008 : Customer 110
i35034web009 : Customer 110
i35034web010 : Customer 110
i35034web011 : Customer 110
i35034web012 : Customer 110
i35034web013 : Customer 110
i35034web014 : Customer 110
i35034web015 : Customer 110
i35034web016 : Customer 110
i35034web017 : Customer 110
i35034web018 : Customer 110
i35034web019 : Customer 110
i35034web021 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
IDL
MEM
SWP
C%:
376
94
94
282
slipstream
2006.05.26 04:14
play
2006.05.30 02:30
current asset
i35034web011/Customer 110: Disk /var
asset search
go
clear

Figure 3.1: LiveRAC is a visualization tool for performance statistics and alarm data. This screen capture shows five days of data. One region of the display has been expanded to show information about a cluster of web servers belonging to a single customer.

3.1 Interface Components and Visual Encoding

This section introduces each of the visual components of the LiveRAC application, divided into three distinct regions shown in Figure 3.2. These are the data region, the time range panel, and the query panel. External dialogs, described in Section 3.1.4, support reordering actions.

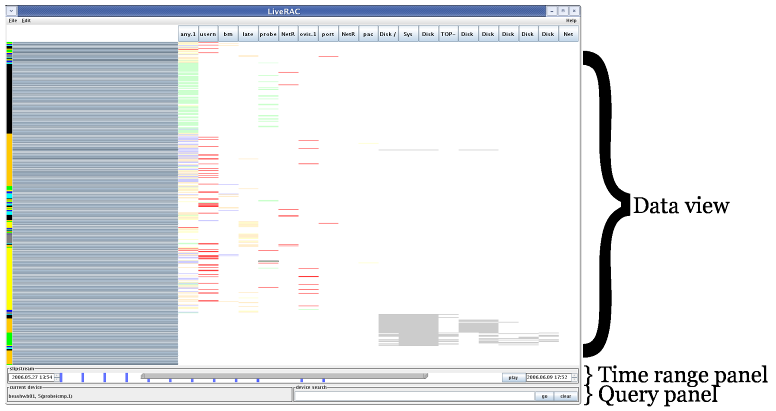


Figure 3.2: LiveRAC is divided into three display regions, a data view and two tool bars. The topmost region is the data view which is the only focus+context component of the display. Below the data view is the time range tool panel which provides visual and text components for choosing the time range of the data view. The query tool panel at the bottom provides information about the cell under the mouse pointer and a search dialog.

3.1.1 Data View

3.1.1.1 Layout

LiveRAC presents a matrix view of the data in an accordion drawing infrastructure. Rows represent network devices, and columns can represent metrics, alarms, or groups of metrics and alarms called bundles. A matrix cell represents the intercept between a row and a column containing an area-aware rendering of the underlying data. The number of nodes in the data set can be larger than the number of display pixels available. Regions with data density that exceed the number of pixels in the display are aggregated. The user specifies a focus by enlarging and compressing display regions through navigation actions on the data set. Expanded regions reveal more detailed information.

The key challenge in selecting a layout strategy for the visualization is to provide groupings that will locate relevant results adjacent to one another for comparison. Providing small-multiples comparisons, introduced by Bertin [9] and popularized by Tufte [67], has become a core principle in information visualization. Small multiples help humans locate differences and patterns by allowing them to saccade rapidly between multiple representations.

With the alarm and metric data that LiveRAC visualizes, there is no single correct way to lay out information. For example, it is unclear as to whether to group devices by IP address, by geography, or by functional grouping such as web servers, file servers, etc. Although it may be intuitive to segregate alarms and statistics, it is possible to construct scenarios where it is advantageous to look at alarm histograms adjacent to a particular metric which may be relevant to the alarm. Our approach with LiveRAC is to provide a logical initial ordering, then allow the user to manipulate the ordering to their preferences. This allows users flexibility to take an exploratory approach to interacting with the data. We initially order devices by customer. Because the underlying data is from a network operations centre, all of the devices are physically located in a small number of buildings and so geography is not relevant in this particular case. We then sub-key on device name. Because the device names have been selected to match the functional grouping of the device, this has the added benefit of grouping computers which perform similar activities.

The optimal initial layout of columns is less clear. Instead of imposing a layout, we allow columns to be added as relevant alarms or metrics are read from the

SWIFT database. The user is able to group related metrics or alarms into bundles as described in Section 3.2. Object ordering in the data view is preserved unless the user performs an explicit reorder operation described in detail in Section 3.1.4.

We selected critical alarms and search results as candidates for Guaranteed Visibility (GV). A critical alarm is a ticketed alarm that must be serviced by a technician to meet quality of service requirements for customers, and therefore we select it as our highest priority guaranteed visible marking. Red was chosen as the colour for the critical alarm marking because it was used previously by other applications at AT&T, and network operators are familiar with this mapping. The minimum size for GV regions was initially selected to be three pixels. This size can be tuned depending on the resolution of the displays being used for the visualization. Search results must be visible, even in highly compressed regions of the display, and therefore we use GV to support this activity, marking the results in highly saturated blue.

3.1.1.2 Aggregation

Data aggregation is used when there are more devices in a region of the display than pixels. Aggregation optimizes rendering by disposing of the redundant overdrawing of certain pixels of the display, and allows us to present a meaningful value to represent all of the occluded data. In LiveRAC we aggregate alarms by showing the highest priority alarm that has been occluded, and use saturation as a way of indicating how many alarms have been aggregated. Metrics are only assigned a single colour, but we can still use aggregation to give users an overview of the information density. Our aggregation algorithm is discussed in Section 4.3 of the Algorithms chapter. We describe the colour scheme and our saturation equation for LiveRAC below.

3.1.1.3 Colour

The default representation for a cell which spans only a few pixels on the X or Y axis would typically be a coloured box. LiveRAC inherits the colour scheme used by an AT&T internal visualization application called SWIFT (described in Section 2.3). This colour scheme encodes alarm severity data in the order defined in Table 3.1. Metrics are always coloured grey.

Table 3.1: Cell colour mapping

Severity	**Colour**
Critical	Red
Major	Orange
Minor	Yellow
Warning	Blue
Normal	Green
Non-ticketed or Metric	Gray

Saturation is used to encode information density. A non-aggregated has a base saturation of 25%. As a cell is expanded, the saturation level decreases proportionally with area to a minimum of .05%. We follow the colour use guidelines of Ware [71] recommending that large areas should use desaturated colours. Moreover the decreasing saturation level helps make the semantic representation contained within the cell more readable by providing sufficient contrast. The enforced minimum ensures the severity level of the cell will always be visible regardless of how large it is expanded. Our formula for computing cell saturation is provided in Equation 3.1, where we define B as being the base saturation of a non-aggregated cell, M as being the minimum saturation for a cell, K as the size at which minimum saturation is achieved and S as the current size of the cell.

$$\frac{B-M}{K}S \tag{3.1}$$

When regions of the display are compressed it is necessary to aggregate data. The hue and value of the representation for this aggregated region is that of the worst alarm that has been seen. The saturation starts at a minimum of 25% and increases linearly with the number of alarms to a maximum of 100% saturation when ten alarms have been aggregated.

3.1.1.4 Adding Devices and Metrics

When the user manipulates the time range of the application, described in Section 3.1.2, data regarding previously unseen devices may be retrieved. Devices are added dynamically to the split line structure whenever a record with a new device is retrieved. Devices are added based on the sort order of the device axis.

Columns are added in a similar manner to rows if new types of alarms or metrics are detected in the incoming data from the server as a result of user interaction. However, because columns are sorted arbitrarily through user interaction, new columns are added to the far right of the display. The user is then free to reorder columns however they choose.

3.1.1.5 Deleting Devices and Metrics

To maintain the stability of objects within the layout of the LiveRAC application, devices and metrics are never removed from the display without an explicit action from the user. Two shortcuts are provided for ‘cleaning’ the display if the user is looking at a very small time range and wishes to focus on data relevant to that particular moment. The first allows the user to drop any metric or alarm columns for which there is no data in the current time range. The second performs a similar operation for devices, clearing empty rows.

3.1.2 Time Range Selection Panel

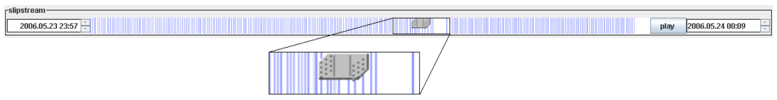


Figure 3.3: The time range selection panel is located underneath the data canvas of the LiveRAC application. The text field on the left displays the start date of the selected range. The widget in the center is the time range slider. The time range slider allows the user to define the time range by dragging either edge of the slider widget to a new location, or by clicking between the edges and dragging to move the entire range. The play button advances the time range at regular intervals. The text field on the right displays the end date of the selected range.

The time range selection panel, shown in Figure 3.3, is located immediately beneath the main focus+context data canvas. This panel allows the user to select a time range for viewing data. The user can select the time range using one of two mechanisms: a double edged slider, or a pair of date spinners.

LiveRAC uses a range slider, visible at the centre of Figure 3.3, for selecting the time range in the data view. A range slider allows the user to define the time range by dragging either edge of the slider widget to a new location, or by clicking between the edges and dragging to move the entire range. LiveRAC uses a modified version of the range slider from the InfoVis Toolkit by Fekete [22]. The range slider in LiveRAC has been enhanced to display the density of alarms in the underlying data. Each pixel of width of the slider bar is assigned an equal time range and alarms are then ‘binned’ to a representative pixel. Displaying the alarm density makes it easier for the user to detect periods of unusual alarm activity. A logarithmic scale is used such that pixels initially grow to a moderate saturation quickly, then increase saturation more slowly as the number of alarms increases. The function used is $\log I/\log T$, where I is the number of alarms seen in that interval and T is the total number of alarms in the time range of the slider.

Using the text fields to the left and right the user can directly manipulate the date range. The text fields disallow invalid actions such as setting the start date after the end date. The text fields and the slider are linked and update one another to reflect any changes from user interaction.

LiveRAC supports VCR-like interaction using a play button visible in Figure 3.3. After selecting a time range, the user can choose an update interval and a period of seconds for the update. When playback is initiated, the user can watch historical data, or live data as it is added. A typical activity might be to view historical data at an accelerated rate, by selecting an interval of five seconds for updates, and a period of one hour. This selection will advance the data view one hour for every five wall-clock seconds. When viewing live data in real time the update interval and period must match.

3.1.3 Current device and Search Panel

The current device and search panel in Figure 3.4 is located beneath the time range panel.

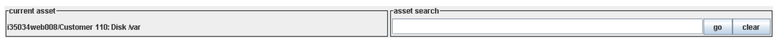


Figure 3.4: The device tool panel shows information about the device under the mouse on the left, and provides regular expression based search tools using the input box on the right.

The text area on the left side of the current device and search panel provides information about the device, customer and metric currently located beneath the mouse cursor. If the mouse cursor is outside of the accordion drawing canvas it indicates the information about the last cell that was moused over by the user.

On the right side of the panel an input dialog allows the user to type in the name of a device or a group name for devices. This search can be done progressively. The results of a search operation are marked with guaranteed visible thin blue horizontal lines, shown in Figure 3.5.

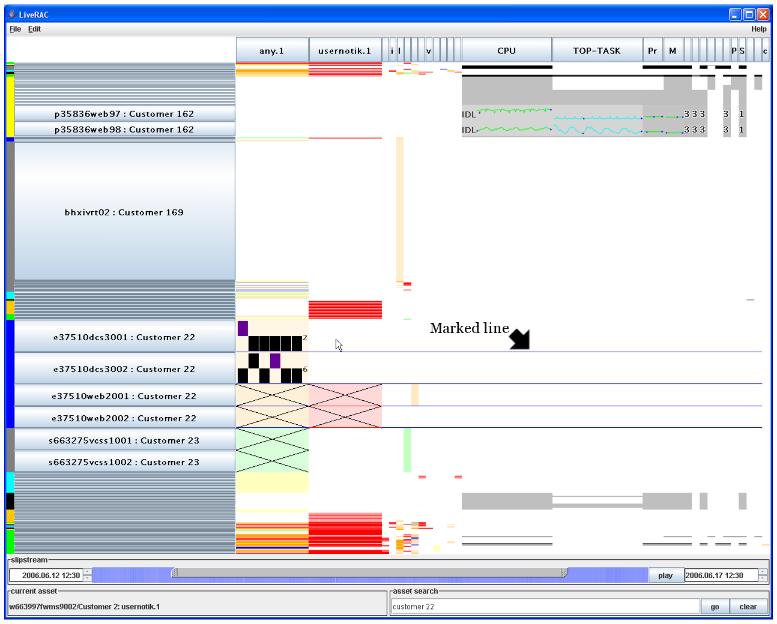
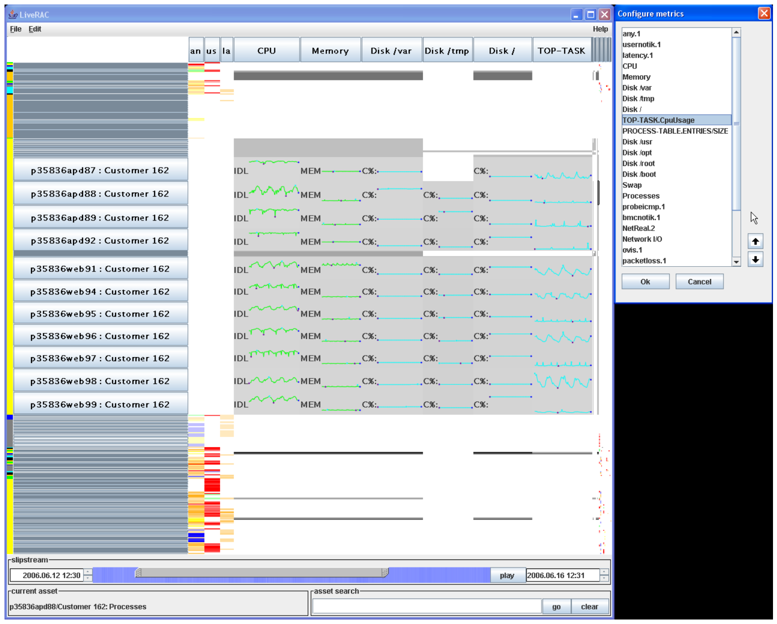

LiveRAC
File
Edit
Help
any.1
usernotik.1
CPU
TOP-TASK
Pr
M
p35836web97 : Customer 162
p35836web98 : Customer 162
bhxivrt02 : Customer 169
e37510dcs3001 : Customer 22
e37510dcs3002 : Customer 22
e37510web2001 : Customer 22
e37510web2002 : Customer 22
s663275vcss1001 : Customer 23
s663275vcss1002 : Customer 23
Marked line
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
w663997fwms9002:Customer 2: usernotik.1
asset search
customer 22
go
clear

Figure 3.5: The guaranteed visible thin blue horizontal lines indicate search results. Users can progressively search on an asset name or a customer name.

3.1.4 Reordering

LiveRAC allows rows and columns to be reordered to facilitate side-by-side comparisons and the grouping of similar metrics. This capability can be very important when directly comparing devices, grouping related devices, or exploring correlations between alarms and metrics. The ability to move data within the visualization empowers the user to explore patterns.


LiveRAC
File Edit Help
an us la CPU Memory Disk /var Disk /tmp Disk / TOP-TASK
p35836apd87 : Customer 162
p35836apd88 : Customer 162
p35836apd89 : Customer 162
p35836apd92 : Customer 162
p35836web91 : Customer 162
p35836web94 : Customer 162
p35836web95 : Customer 162
p35836web96 : Customer 162
p35836web97 : Customer 162
p35836web98 : Customer 162
p35836web99 : Customer 162
IDL MEM C%:
slipstream
2006.06.12 12:30
play
2006.06.16 12:31
current asset
p35836apd88/Customer 162: Processes
asset search
go
clear
Configure metrics
Ok
Cancel

any.1
usernotik.1
latency.1
CPU
Memory
Disk /var
Disk /tmp
Disk /
TOP-TASK.CpuUsage
PROCESS-TABLE.ENTRIES/SIZE
Disk /usr
Disk /opt
Disk /root
Disk /boot
Swap
Processes
probeicmp.1
bmcnotik.1
NetReal.2
Network I/O
ovis.1
packetloss.1

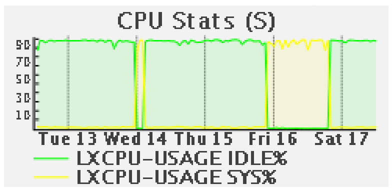
Figure 3.6: The control panel on the right allows users to interactively reorder metrics in the visualization. The visualization is updated with every operation.

Rows can be sorted using two different criteria: 1) customer name, device name or 2) device name. Individual devices can also be reordered arbitrarily, although this mode will result in new devices being added to the bottom of the display rather than inserted using the sorted order.

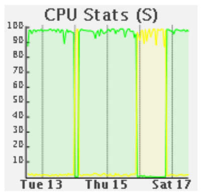
Columns can be ordered arbitrarily by the user. New columns are added on the right most side of the display. Generally, it is advantageous to group similar alarms and metrics. Metrics which are best placed in the same chart can be added to the same group, called a bundle in LiveRAC, described in Section 3.2. This capability ensures closely related data can be viewed in the same representation.

3.2 Semantic Zoom and Bundles

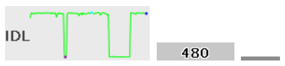
Semantic zoom [51] is a visualization technique where different representations of data are presented at different levels of zoom. Semantic zoom tries to provide the most meaningful possible representation at any given zoom level. This a significantly more sophisticated approach than simple scaling. For example, an object with text elements has very little value when scaled down such that the text can no longer be read. Features that were relevant at the larger zoom level cause clutter and illegibility when the image is scaled down. However, if the text elements are removed and a simpler graphic is used, it is still possible to extract meaningful information from the representation.



(a) Full size line chart



(b) Reduced line chart



(c) Sparkline (d) (e)

Figure 3.7: An example of semantic zoom on a line chart. (a) A full sized line chart is displayed. (b) The line chart has been reduced in size, the legend has been removed and labels are more sparse. (c) The chart has been changed to a sparkline graphic. Only one label is shown on the sparkline, indicating the subject of the sparkline. (d) Only a colour swatch and a numeric value indicating the number of records in this time range are shown. (e) Only a colour swatch is displayed.

LiveRAC provides a grouping system called bundles for collecting related alarms and metrics, and specifying their representation levels. Because alarms represent ordinal data and metrics represent ratio data, it is not possible to group alarms and metrics into the same bundle. Bundles are defined in an XML configuration file. In the future, it should be possible to construct the bundles visually using a dialog-driven wizard or similar GUI mechanism. A partial XML bundle configuration is shown in Figure 3.8. This bundle was used to generate the graphic images in Figure 3.7.

Once the metrics or alarms for a bundle have been specified, it is possible to configure every aspect of the representation levels for the bundle. The minimum width and height for each representation level is specified, followed by chart type, what data series to use at that representation level, colours and other chart details discussed in section 3.4.

```
<bundle>
  <bundlename>CPU</bundlename>
  <generic>0</generic>
  <type>1</type>
  <metric>LXCPU-USAGE.IDLE%</metric>
  <metric>LXCPU-USAGE.SYS%</metric>
  <representation>
    <chartlabel>CPU Stats (S)</chartlabel>
    <width>120</width>
    <height>80</height>
    <charttype>3</charttype>
    <drawgrid>1</drawgrid>
    <globalshadeunder>1</globalshadeunder>
    <drawlegend>1</drawlegend>
    <drawlabels>1</drawlabels>
    <drawticks>1</drawticks>
    <gridstipple>dotted</gridstipple>
    <series>
      <name>LXCPU-USAGE.IDLE%</mname>
      <shadeunder>1</shadeunder>
      <seriescolor>
        <color>
          <H>.3334</H> <S>1</S> <V>1</V>
        </color>
      </seriescolor>
    </series>
    <series>
      <mname>LXCPU-USAGE.SYS%</mname>
      <shadeunder>1</shadeunder>
      <seriescolor>
        <color>
          <H>.16667</H> <S>1</S> <V>1</V>
        </color>
      </seriescolor>
    </series>
  </representation>
  <representation>
    <chartlabel>Sparkline</chartlabel>
    <width>30</width>
    <height>20</height>
    <charttype>6</charttype>
    <globalshadeunder>0</globalshadeunder>
    <drawband>0</drawband>
    <markhigh>1</markhigh>
    <markhighcolor>
      <color>
        <H>.5</H> <S>1</S> <V>1</V>
      </color>
    </markhighcolor>
    <marklast>1</marklast>
    <marklastcolor>
      <color>
        <H>.6667</H> <S>1</S> <V>1</V>
      </color>
    </marklastcolor>
    <marklow>1</marklow>
    <marklowcolor>
      <color>
        <H>.7778</H> <S>1</S> <V>.6</V>
      </color>
    </marklowcolor>
    <series>
      <mname>LXCPU-USAGE.IDLE%</mname>
      <shadeunder>1</shadeunder>
      <seriescolor>
        <color>
          <H>.3334</H> <S>1</S> <V>1</V>
        </color>
      </seriescolor>
    </series>
  </representation>
</bundle>
```

Figure 3.8: This is an example of the partial XML definition for the CPU metric bundle, that is, a bundle handler for CPU metrics. For brevity we show only two representation levels from the bundle. The first representation is displayed graphically in Figure 3.7(b), and the second in Figure 3.7(c).

3.3 Interaction & Tasks

The primary interaction mechanism in LiveRAC is accordion drawing described in Section 2.4 of related work. Users can select a region of the display using a 'rubber band', a region of the data marked by the mouse and snapped to split line grid boundaries. This region can then be expanded or compressed in the X and Y directions through a mouse drag operation. This technique facilitates fluid interaction with the application, and helps the user maintain spatial awareness regarding their current focus within the context of the larger data set. This interaction technique, in conjunction with the time range and query panels described in Section 3.1 provide a rich set of tools for exploring data.

When no focus regions are selected, all cells in the display are of roughly equal size. Guaranteed visibility ensures that even if thousands or millions of cells exist in the visualization, critical events will be marked. This level of zoom gives an overview of system state, indicating the distribution of devices with critical alarms. LiveRAC marks all critical alarms at any level of zoom. The size of the critical marker can be adjusted to optimize visibility for large and small displays.

When investigating alarms, users of LiveRAC can explore rich historical data regarding a device. By enlarging the time window, users can easily see repeating patterns of alarms, such as alarms that recur at regular intervals, as well as correlation between metrics and alarms.

LiveRAC simplifies capacity planning by providing side-by-side comparisons of server load using multiple focus regions on a single display. This capability removes the requirement for switching between windows, and allows rapid navigation to expand new regions of interest. Adjacent and distant regions can be compared with nearly equivalent ease by compressing the regions between devices or columns.

By supporting a large time window, it is possible to view daily, weekly and monthly trends in LiveRAC.

3.4 jGLChartUtil

Providing multiple semantic zoom levels in LiveRAC required a charting library with support for a variety of graphic options at different levels of data density. There are two excellent charting packages for Java which fall under free software licenses: jFreeChart [25] and jRobin [45]. Although both of these solutions offered robust charting features, they use the 2D drawing library provided by Java2D. The resulting images then needed to be texture mapped in OpenGL, which is inefficient compared to drawing the polygons in OpenGL. Benchmarks using these two charts revealed that even a small chart of a few dozen data points could take over 100ms to render and texture map. During a navigation operation it is desirable to maintain a minimum frame rate of 10fps, or 100ms. Clearly, if there were two or more of these charts on the display at any given time, achieving this frame rate would become impossible.

One possible work-around for the performance problems of jFreeChart and jRobin would be to generate mip-maps of the textures and use them during scale operations, to limit the number of times a chart needs to be regenerated. One problem with this approach is that it would quickly consume a large amount of texture memory. Furthermore, the aspect ratio of the charts can change, creating distorted and potentially unreadable results when the charts are directly scaled as images, rather than redrawn with raster fonts. A chart which redraws with every resize also has the advantage of being able to re-evaluate how screen real-estate is used. For example, it can choose to decrease font size, or reduce the number of labels and grid lines to improve legibility on a smaller chart.

Because none of the off-the-shelf solutions provided the capabilities we desired, we elected to write a small, high performance charting library using OpenGL. The resulting chart library, jGLChartUtil 3.4, is completely abstracted from the LiveRAC application. It is a general purpose charting library with a well defined Java API [46]. Keeping the charting library separate from the application gives us the ability to replace the charting feature if a more capable library becomes available in the future, and also allows us to provide this charting library as a separate work product under an open source license for use in other projects.

3.4.1 Features

In this section we summarize some of the high level capabilities of jGLChartUtil. For brevity will not provide an exhaustive list of the jGLChartUtil features, for detailed information refer to the jGLChartUtil javadoc API reference [46].

jGLChartUtil supports a number of standard features and enhanced capabilities. jGLChartUtil allows charts to have multiple data series, which can be distinguished by a user assigned colour or different line stipples. Legends are provided for each data series on the chart, and labels can be assigned by the user. The legend is drawn on a best-effort basis, and may be dropped if the chart is too small to render it. The chart can be assigned a master label, with a user selected alignment. Axis labels are optional, and are provided by the charting library on a best effort basis. The charting library will try and provide as many labels as possible given the size of the labels and space constraints of the chart. It will attempt to fit the labels by reducing label size down to a user specified minimum, and reducing significant digits of a decimal value.

Many charts offer the option of shading regions with an alpha value. Region shading can be enabled and disabled globally, and for each individual data series.

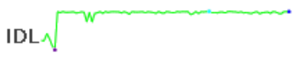
The use of alpha transparency and blending makes intersections between data series clearly visible. The amount of transparency is configurable by the user.

jGLChartUtil supports numerous types of charts detailed in the following sections. Table 3.2 provides a breakdown of the supported categorical and ratio charts.

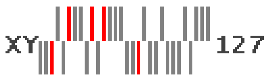
Table 3.2: Chart types

Ratio charts	Categorical charts
Sparkline	Binary Sparkline
Two-tone pseudo colouring	Histogram
Line chart	Bar chart
Scatter chart	

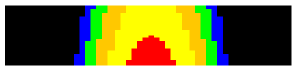
3.4.1.1 Sparklines



(a) Sparkline



(b) Binary sparkline



(c) Two-tone pseudo colour map

Figure 3.9: (a) Sparklines [69] are small graphics that compactly represent trend information. (b) Binary sparklines represent binary information in a compact space. (c) Two-tone pseudo colour maps [57] are a representation technique for displaying large scale one dimensional data sets at very high density.

Sparklines are small statistical graphics introduced by Edward Tufte in Beautiful Evidence [69] for representing trend information in a compact space. An example of a sparkline drawn by jGLChartUtil is shown in Figure 3.9(b).

Binary sparklines are similar to standard sparklines but show only binary information in the plot. An example of a binary sparkline drawn by jGLChartUtil is shown in Figure 3.9(b).

3.4.1.2 Two-tone Pseudo-Colour

Two-tone pseudo colour maps [57] are a representation technique for displaying large scale one dimensional data sets at very high density. We show an example of a two-tone pseudo colour map drawn by jGLChartUtil in Figure 3.9(c).

3.4.1.3 Line Charts

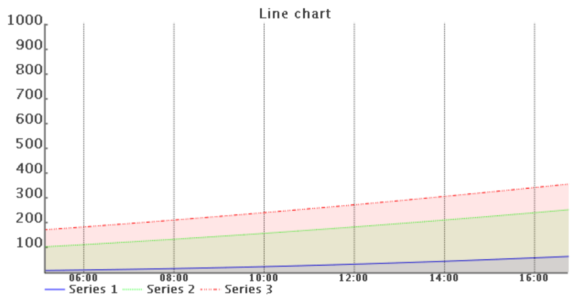


Figure 3.10: Line charts are used to compare data points in two dimensional data.

Line charts are used to compare data points in two dimensional data. The slope of the line simplifies trend finding. jGLChartUtil provides standard line charts with the added capability to shade under the lines. The resulting region-shaded graphic can make it easier to compare relative volumes visually. When multiple data series are being charted, the shaded region can be blended between the data series. Shading can be turned on or off on a per-data series basis, or globally. An example of a line chart can be seen in Figure 3.10.

3.4.1.4 Scatter Charts

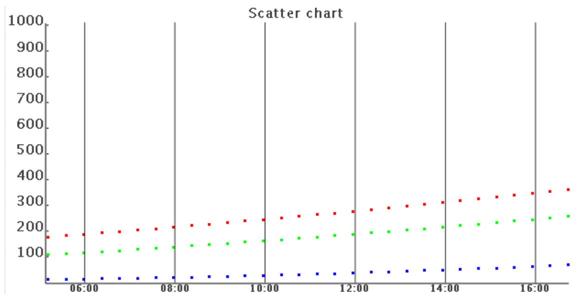


Figure 3.11: Scatter charts are typically used for mapping two dimensional data sets with large numbers of data points.

jGLChartUtil provides scatter charts, a plot typically used for mapping two dimensional data sets with large numbers of data points. Multiple data series can be shown in a single chart. The charting library does not currently support drawing a trend line through the scatter plot, but this feature may be added to future versions. We show a scatter chart drawn in jGLChartUtil in Figure 3.11.

3.4.1.5 Histograms

A histogram is a statistical graphic for showing frequency distributions. jGLChartUtil histograms support multiple data series, and alpha-blended region shading of the data points. We show a histogram drawn in jGLChartUtil in Figure 3.12.

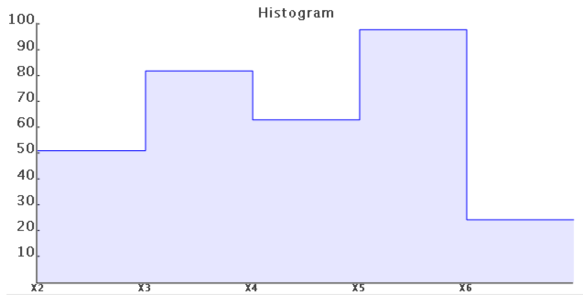


Figure 3.12: Histograms show frequency distributions

3.4.1.6 Bar Charts

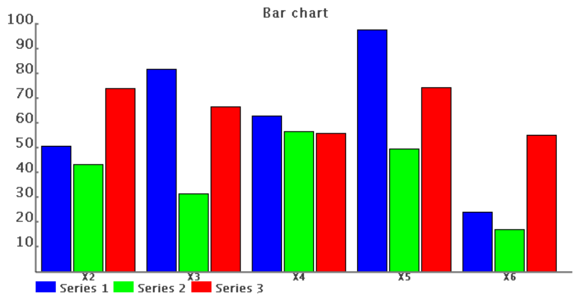


Figure 3.13: Bar charts show relationships in categorical data.

Bar charts are a common statistical graphic representation. jGLChartUtil supports two types of bar charts. The first is a standard coloured bar chart, with optional outlining to provide higher contrast. The second type of bar chart uses a

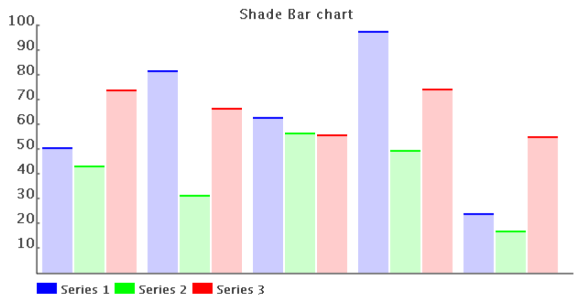


Figure 3.14: An example of the shade bar chart. Bar charts show relationships in categorical data.

less saturated shaded region underneath a solid marker. The reduced saturation in this type of chart helps reduce saturation overload on busy displays. We show a standard bar chart drawn in jGLChartUtil in Figure 3.13 and a shade bar chart in Figure 3.14.

Chapter 4

Approach and Algorithms

In this chapter we provide the approach and algorithms for overcoming the technical challenges of developing the LiveRAC visualization. Section 4.1 discusses the architecture of LiveRAC and interaction with the SWIFT data collection and processing engine. Section 4.2 discusses the design of dynamic structures for handling add and delete operations on rows and columns at run time. Section 4.3 describes our algorithm for node aggregation and Section 4.4 discusses our semantic zoom strategy. Section 4.5 describes the rendering process, and Section 4.6 discusses picking objects from the visualization. Section 4.7 discusses user-directed reordering of data within LiveRAC.

4.1 Architecture

LiveRAC implements a threaded, modular architecture that utilizes the PRISAD [34] accordion drawing API. The application layer manages user interaction, enqueues data requests, and renders the data view. The gridding and data management layer maintains the data structures necessary for addressing each cell in the matrix layout, stores data retrieved from the server and maintains information regarding cell representations. PRISAD provides an API for accordion drawing. Extensive modifications were made to the PRISAD split line infrastructure, described in Section 4.2 to provide dynamic functionality required by LiveRAC. Minor changes

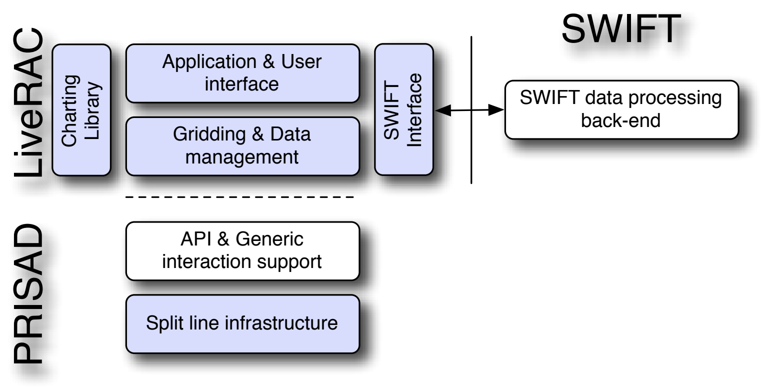

LiveRAC
Charting Library
Application & User interface
Gridding & Data management
SWIFT Interface
SWIFT
SWIFT data processing back-end
PRISAD
API & Generic interaction support
Split line infrastructure

Figure 4.1: LiveRAC uses a modular architecture. The PRISAD accordion drawing infrastructure provides API's on which the LiveRAC application is built. The SWIFT interface layer provides a network client-server interface to the SWIFT data collection and processing services. Shaded boxes indicate modules where we have made significant contributions.

were made to the API to incorporate these modifications.

The interface to the SWIFT [41] system runs as a separate thread that processes data requests and returns the data to the application and gridding layers. Separating data requests from rendering ensures that navigation and user interaction can continue while data is loading from the server. User interactions queue data requests on the data service thread, which aggregates relevant requests and culls unneeded requests, and interacts with the SWIFT server. The SWIFT back-end aggregates multiple data sources into a unified record structure. LiveRAC provides a dense, unique, matrix-layout visualization of this data that incorporates overview and detail.

Although large volumes of data must be parsed whenever the time window is changed, LiveRAC minimizes the amount of data that must be transferred to the client as much as possible. For most nodes in the display, the representation is only a colour index value which indicates the highest severity event during the time range, and an integer which indicates the number of events which occurred in that interval. At the overview level, metric cells indicate only a colour value and the number of metric records received during the sampling period specified by the user in the time range slider. For these highly compressed cells the server returns only these two integers, an order of magnitude less data than the full alarm or metric record contains. For cells occupying more screen real estate, detail requests are enqueued to the data service thread.

The data service thread makes data requests to the SWIFT [41] server described in section 2.3. SWIFT queries resemble web CGI requests. On the SWIFT server a number of custom written filters for LiveRAC have been pre-compiled and are applied to the selected range of data. These queries are written in a C-like language for a SWIFT parser, which generates additional code blocks, and then uses a standard C compiler such as GCC to create a native shared library for the server platform. SWIFT processes and searches the underlying data for LiveRAC. SWIFT is a multi-platform tool that is deployed both on SGI IRIX based systems, as well as Linux on PC hardware. Because most SWIFT tools work as small steps in a pipeline, SWIFT can leverage large scale multi-processor systems effectively.

The user can make navigation actions between the execution cycle of the data service thread, providing an opportunity for the data service thread to aggregate queries where possible, and discard outdated queries. For example, it is possible the user might have made two rapid changes to the time slider between data service thread execution. In this case, the first request can be discarded, saving server load and bandwidth. A user navigation could also reveal dozens of charts simultaneously. Each of these chart queries can be grouped into a single large query, which dramatically decreases load on the server and reduces query overhead because only a single pass needs to be made over the data rather than a full pass for each chart query. The data service thread sleeps for thirty milliseconds between execution cycles.

4.2 Split Line Hierarchy

As discussed in Section 2.1.4.2 of related work, PRISAD was designed to allocate a static 2-D grid of cells. Each of these cells is bounded by four movable lines, which are called split lines. Navigation in the accordion drawing metaphor is accomplished through the manipulation of these split lines. Placing cell data objects within this framework is called gridding. In previous PRISAD-based applications the number of cells was well defined when the data set was loaded, and PRISAD allocated two static one dimensional arrays for the split line grid. Elements could not be arbitrarily inserted into these static structures without copying all data to a new, resized, array structure incurring an *O(n)* penalty.

LiveRAC requires that devices (rows) and user defined alarm bundles, metrics, or alarms (columns) be interactively added, removed or relocated to any point within the grid. The static structures of PRISAD were not suitable for these requirements. In this section we present algorithms and dynamic structures for extending PRISAD to allow logarithmically bounded split line insert and remove operations at run time.

4.2.1 Challenges

We define the following design criteria for a dynamic structure to replace the static arrays in PRISAD:

- Worst case logarithmic insert and remove operation
- *O(log n)* worst case path to any node from the root
- Linear scalability in memory usage

- Support for arbitrary ordering with enumeration

The requirement for preserving arbitrary node ordering eliminates hash tables. Furthermore, because split lines subdivide the display area in a tree hierarchy, it was intuitive to use a tree-like data structure to store the split lines. A balanced binary tree structure such as red-black trees [66] seemed to be the optimal solution, as it resolves all of our first three requirements.

In LiveRAC we needed to support the arbitrary ordering of split lines within the hierarchy to support add, remove and reorder operations. Any split line can be placed at the beginning, end or between any two other split lines in the hierarchy. In this schema the data structure itself stores the user-specified ordering, but we need a mechanism for enumerating tree nodes for picking, range selection and split line order comparison operations. During picking, we may want to select the *n*-th node in the ordering. Furthermore, the most efficient way to store split line ranges is as a pair of integer indices, for example [10, 73], indicating a range between 10 and 73. This could be duplicated without needing indices by creating a pair of pointers, one to split line 10 and another to split line 73. However, the index encodes additional information not available to the pointer. For example, we need to store extra information to indicate which pointer is the start of the range, and which pointer is the end. We lose other important information about the range as well, for example we can no longer assert that the range spans 64 nodes without performing a traversal, again with a worst case of $O(n)$ if the range is the span of the entire tree. Without enumeration, it is impossible to tell whether a given node s lies previous or successive to a given node t without a complete traversal of the tree.

As discussed in Section 2.1.4.2, PowerSetViewer [40] addresses this enumeration issue by utilizing properties specific to power sets. With *a priori* knowledge of the power set alphabet it is possible to allocate a sufficiently large enumeration space to ensure each power set can have a unique index in the ordering. In LiveRAC we must enumerate objects that lack the properties particular to power sets.

Performing a simple linear enumeration of the split line objects in the tree carries an *O(n)* penalty for insert and remove operations, since a re-enumeration of all subsequent nodes would be required for either operation, which violates our first design requirement. One alternative is to evenly distribute the split line enumeration over a 64 bit space, and then distribute new split lines as evenly as possible within this range, performing a *O(n)* key redistribution pass when we run out of resolution. This solution might be acceptable if we can assert that split lines will generally be added and removed evenly across the layout. However, no such assertion can be made and we have every reason to expect the possibility of hundreds or thousands of adjacent split lines being added concurrently, making re-enumeration a common operation. The requirement for frequent re-enumeration makes this solution unacceptable.

Implementing a data structure which allows arbitrary ordering but preserves a mechanism for enumerating nodes in a balanced tree was the primary challenge in implementing a dynamic grid structure.

4.2.2 Our Approach

We determined that red-black trees [66] were a suitable candidate data structure for each split line axis. Red-black trees satisfy most of our requirements: insert and remove operations are *O(log n)*, objects can be ordered arbitrarily if we do not require the ability to binary search the tree, and the tree structure scales linearly in memory usage. The only outstanding challenge is the issue of enumerating the

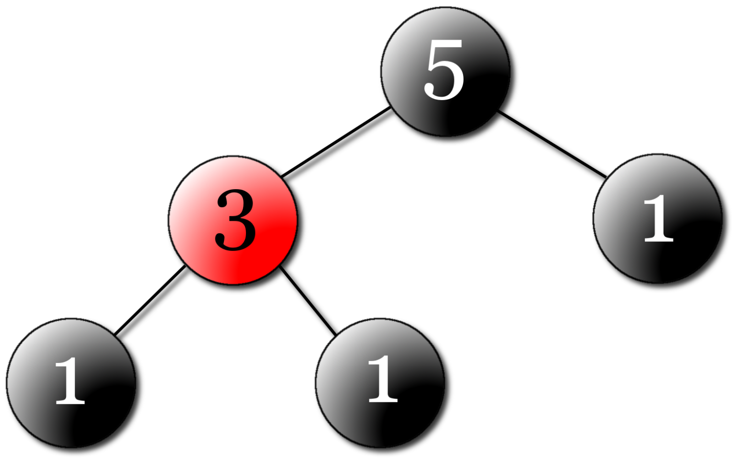


Figure 4.2: Each node of the red-black tree stores an integer value of its subtree size. Leaf nodes have a subtree size of 1: themselves. The subtree size of the root is the size of the tree.

tree nodes described in Section 4.2.1. We provide a solution based on a balanced red-black tree that allows for an unlimited number of split line add or remove operations in logarithmic time on either axis, and allows retrieval of tree nodes based on an integer index.

As described in Section 4.2.1 any static enumeration assignment will result in a requirement to re-enumerate during an insert or remove operation on the tree. The solution to the problem requires the use of relative value assignments. In our split line tree, we store an added integer value for each split line called ‘subtree size’ shown in Figure 4.2. This value indicates the number of descendants of a given node, inclusive of the node itself. The root of the tree therefore has a subtree size that is identical to the size of the tree. Using this additional information it is possible to find a tree node by index number or determine an index number given a node in *O(log n)* time.

```
getSplitFromIndex Function
input:   target index $I$
         running count from leftmost node in the tree, $L$
         current node $N$
output:  node corresponding to input index, $N$

  $myIndex \leftarrow (N.left \neq null)\ ?\ myIndex + N.left.subtreeSize : L$
  if $myIndex$ = I
      return $node$
  else if $I < myIndex$
      return $getSplitFromIndexRecursive(I, L, N.left)$
  else
      return $getSplitFromIndexRecursive(I, myIndex+1, N.right)$
  end if
```

Figure 4.3: *getSplitByIndex O(log n)* recursive function locates a node of the red-black tree based on its index number within the range of the tree from left to right. It is first called with the root of the tree as the current node, *N*.

To find a node given an index number, we use the recursive function getSplit-ByIndex shown in Figure 4.3. We descend the tree from the root, adding the left subtree index on every right descent. Because this descent can be no further than the depth of the tree, which can be no deeper than *O(log n)* in a red-black tree [66], and requires only *O(1)* add or compare operations at each step, we can assert that this is an *O(log n)* algorithm.

Computing a node's index number is the inverse of finding the node given an index number. We begin from the specified node, and ascend towards the root using the algorithm in Figure 4.4. A counter is incremented by the left subtree size each time the ascent is made from a right child node. Because this ascent can be no longer than the depth of the red-black tree, which is bounded to *log n*, as described

getIndexFromSplit Function
input: target node, *N*
output: index of target node, I

```
$lineIndex \leftarrow$ (N.left $\neq null$) ? N.left.subtreeSize : 0
$p \leftarrow$ N.parent
$current \leftarrow$ N
while $p \neq null$
    if $current = p$.right
        $lineIndex \leftarrow$($p$.left $\neq null$) ? $lineIndex$ + $p$.left.subtreeSize + 1 : $lineIndex$ + 1
    end if
    $current \leftarrow p$
    $p \leftarrow p$.parent
end while
return $lineIndex$
```

Figure 4.4: *getIndexFromSplit O(log n)* recursive function returns the index number of a target node. The algorithm ascends from the target node, incrementing a counter by left subtree size each time the ascent is made from a right child node.

in CLR [66], and requires only *O(1)* add or compare operations at each step, we can assert that this is an *O(log n)* algorithm.

Using these algorithms we are able to theoretically enumerate a tree of split lines of size 2^{31}, based on the positive number space available to a Java signed integer.

New devices and columns are added to the split line tree structure as leaf nodes. Because the new devices are added as leaf nodes, they effectively split the distance between their parent split line and the successive split line in the hierarchy in two. Because the user generally only has a relatively small number of focus regions of the screen (a few dozen split cells or less) expanded, relative to the size of the hierarchy, the effective result is that dense regions of the display typically grow denser with very little effect on the stability of the stretched regions. If a new device is added between devices which have been stretched, the result would cut the parent cell's space in half, while preserving the overall geometry of the stretched region. This approach provides the minimum possible impact of dynamic add operations on the user.

4.3 Aggregation

The PRISAD [34] accordion drawing infrastructure supports aggregation in dense regions of the split line hierarchy. A tuneable parameter specifies in world space co-ordinates the threshold in pixels beneath which aggregation will take place. We aggregate devices, represented by rows in LiveRAC, as this represents our largest data dimension in most circumstances. We perform aggregation when multiple split lines subtend a single display pixel.

4.3.1 Challenges

The aggregation algorithm must present the most relevant information regarding the culled nodes. This algorithm must execute quickly, executing in at worst *O(k)* time where *k* is the number of nodes being aggregated, as a value must be returned for every pixel during each rendering pass.

4.3.2 Our Approach

Our algorithm traverses the subtree of aggregated nodes as shown in Figure 4.5. It maintains a record of the highest priority cell value and a count of the number of cells with this value from all culled nodes in a RowColorAggregator object. Since this algorithm is a traversal of the subtree, and performs only *O(1)* compare and addition operations, it has *O(k)* complexity, where *k* is the number of nodes aggregated. The highest severity value determines the hue of the colour used to represent the aggregated cells, and the saturation is selected by the number of alarms received.

To improve performance we cache the RowColorAggregator object by associating it with the root split line of the aggregated subtree. The cache is dirtied if the split line hierarchy is modified. To prevent these cache objects from occupying too much memory, we delete any cached RowColorAggregator objects associated with split lines that are descendants of the aggregated subtree.

4.4 Semantic Zoom

As discussed in Section 3.2, LiveRAC can have multiple data representations per cell depending on the cell size, a visualization technique called semantic zoom [51].

4.4.1 Challenges

LiveRAC required an infrastructure for semantic zoom that was fast enough to maintain interactive frame rates during navigation operations, and that was highly configurable by users of the system. The solution needed to support grouping multiple metrics together into a single column, but preserving the ability for different representation levels to show subsets of these groupings.

4.4.2 Our Approach

We designed a highly configurable semantic zoom infrastructure that allows users to create customized groups of alarms and metrics called bundles. Bundles are configured using an XML formatted configuration file which permits the user to create

recurseCulledRowRange Function

input: root split line of the subtree to be aggregated, *s*
column index number, *i*
RowColorAggregator object *r*

output: modified RowColorAggregator object

```
$dev \leftarrow$ s.$getRowObject()$
$cell \leftarrow dev.getMetric($i$)$
if $cell \neq null$
    $severity \leftarrow cell.getHighestSeverity()$
    if $severity > agg.worstAlarm$
        $r.worstAlarm \leftarrow severity$
        $r.numFound \leftarrow 1$
    else if $severity = r.worstAlarm$
        $r.numFound{+}{+}$
    end if
end if
if $s.getLeft() \neq null$
    $recurseCulledRowRange(s.getLeft(), metricID, agg)$
    $s.getLeft().cullingObject \leftarrow null$
end if
if $(s.getRight() \neq null)$
    $recurseCulledRowRange(s.getRight(), metricID, agg)$
    $s.getRight().cullingObject \leftarrow null$
end if
return
```

Figure 4.5: The *recurseCulledRowRange* recursive function returns a modified RowColorAggregator object which contains a record of the highest priority alarm from the aggregated range as well as a count of the number of alarms received from the culled rows.

explicit groups of alarms and metrics. The XML format provides a syntax for specifying the features available in jGLChartUtil to represent the selected alarms and metrics. Users define a bundle, specify the metrics to be included, define global charting options, and then create a series of representations. Each representation has a minimum X and a minimum Y size.

LiveRAC uses a greedy algorithm to select the largest representation it can find for a given bundle. Note that the available screen size and the representation do not need to match. If the available size is larger than the minimum required for the representation, the representation is expanded to use the available space. This scaling is handled by the supporting chart library, which takes as a parameter the number of pixels of width and height available. It uses this information to determine the draw area of chart, and the optimal font size to avoid any font overlap.

Although they are defined in the same file and configured in an identical manner, metric bundles and alarm bundles are separate entities distinguished through a type flag. This is due to the incompatibility of the underlying data types: alarm representations are of a categorical data type, and system metrics are ratio data, which cannot be meaningfully mapped together with traditional representations. It may be possible to create representations that synthesize these data types in useful ways, and this will be explored in future work.

Three bundles have special status in LiveRAC. The first is called the garbage bundle. In this bundle the user places the names of metrics or alarms they do not wish to appear in the layout. The other two special bundles are a generic metric bundle, and a generic alarm bundle. These are the bundles that describe how any metric or alarm that has not been explicitly handled by other bundles should be represented, and will typically be the most common column type being displayed.

4.5 Rendering

Cells are rendered by locating the left, right, top and bottom split line bounds of the cell. The absolute values of these boundaries within the world space are then computed. Once we have these four world space co-ordinates, any desired OpenGL rendering operation can take place within this display region. When finding the boundaries of a cell in PRISAD, we first locate the split line that bisects the cell. This may be a 'leaf' split line, that is, it has no children in the split line tree and therefore its relative position has no value for layout, or it may be an internal node in which case the relative position of the split line between its boundaries is relevant for finding the cell boundaries of its descendants.

4.5.1 Challenges

The boundaries of a given cell S are always formed on one side by its parent. If S is a left child, the parent forms its right boundary. If S is a right child, the parent forms its left boundary. An example of this can be seen in Figure 4.6. Finding the parent bound is a trivial *O(1)* operation, requiring only an ascent to the parent of S to determine whether S is a left or right child. We call the non-parent bound the opposite bound. Finding the opposite bound quickly is a key challenge for rendering. Figures 4.6 and 4.7 provide graphical examples of opposite bounds for nodes.

4.5.2 Our Approach

To locate the opposite bound of S we ascend through each of the ancestors of S. After each ascent, we check whether the path to S lies to the left or right of the current ancestor A. If the path to A lies in the opposite direction of the path from S to its parent, then A is the opposite bound of S. Figure 4.8 provides pseudocode

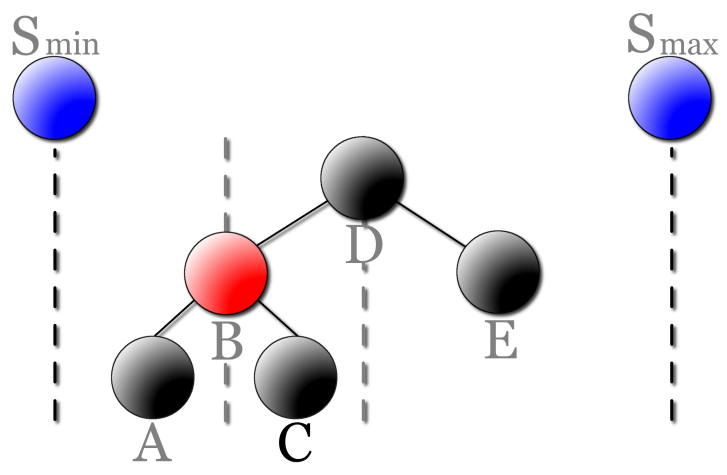


Figure 4.6: The left and right boundaries for node C are B and D respectively where B is also the parent bound and D is the opposite bound. S_{min} and S_{max} represent the stuck lines which are not stored in the data structure but can exist as boundaries for exterior nodes of the tree and the root. For example, S_{min} is the left bound for A, B and D. S_{max} is the right bound for D and E.

for computing the left bound of *S*. Finding the opposite bound is *O(log n)* because in the worst case it is an ascent from a leaf node to the root in the red-black tree structure, which has a maximum depth of *log n* [66]. The right bound can be found using a functionally identical algorithm appropriately substituting left for right.

The root of the tree is a special case with two virtual bounds called the minimum and maximum stuck lines. These lines have fixed absolute co-ordinates and form the basis for all absolute positioning computations performed in the tree as presented in PRISAD [34].

Once an opposite bound is computed for a split line we cache the result. During an insert operation it is necessary to fix the bounds of all rotated nodes. We do this for each rotation. In Figure 4.9 we show our algorithm for a left rotate operation, a modified form of the red-black tree left rotate algorithm from *Introduction to*

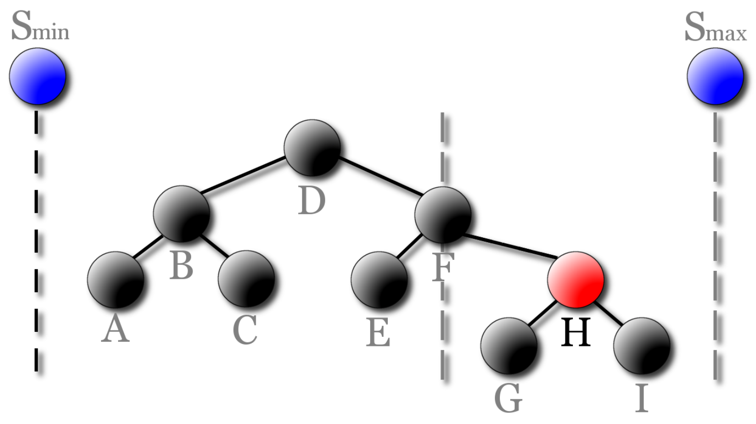


Figure 4.7: The left and right boundaries for node H are F and S_{max} respectively, where S_{max} represents the opposite bound and F represents the parent bound. S_{min} and S_{max} are stuck lines which are not stored in the data structure but can exist as boundaries for exterior nodes of the tree, and the root. As an additional example, S_{min} is the left bound for A, B and D.

Algorithms by Cormen, Leiserson, Rivest, and Stein [66].

When a split line is deleted from the tree it is necessary to update the bounds of all the rightmost descendants of the node's left child, and all the leftmost descendants of the node's right child. We call this an echelon bound update. Our algorithm for updating a deleted node's left child bounds is provided in Figure 4.10. Updating both sets of bounds can take a worst case of *O(2 log n)*.

computeLeftBound Function
input: split line for which the left bound is needed, *S*
output: left bound of *S*, *R*

```
  if $S = S.parent.left$
      while true
        $parent \leftarrow S.parent$
          if $parent = root$
              return $minStuckLine$
          end if
        $gp \leftarrow parent.parent$
          if $parent = gp.right$
              return $gp$
        else
            $S \leftarrow parent$
        end if
      end while
else
      return $S.parent$
end if
```

Figure 4.8: The *computeLeftBound* function finds the split line that forms the left boundary of the input split line. The right bound can be found using an identical algorithm appropriately substituting left for right.

rotateLeft Function
input: split line to rotate, *P*
output: a rebalanced tree structure

```
$R \leftarrow P.right$
$P.right \leftarrow R.left$
$P.subtreeSize \leftarrow P.subtreeSize - R.subtreeSize$
if $P.right \neq$ null
    $P.subtreeSize \leftarrow P.subtreeSize + P.right.subtreeSize$
    $P.right.parent \leftarrow P$
end if
$R.parent \leftarrow P.parent$
if $P.parent =$ null
    $root \leftarrow R$
else if $P.parent.left = P$
    $P.parent.left \leftarrow R$
else
    $P.parent.right \leftarrow R$
end if
$R.left \leftarrow P$
$P.parent \leftarrow R$
$R.subtreeSize \leftarrow (R.right = null)\ ?\ 1 + R.left.subtreeSize : 1 + R.right.subtreeSize + R.left.subtreeSize$
setBounds(R)
setBounds(R.left)
if $P.right \neq$ null
    setBounds(P.right)
end if
```

Figure 4.9: The *rotateLeft* function performs a left rotation on the tree. The algorithm is based on that provided from *Introduction to Algorithms* [66]. During the rotation we perform subtree size accounting and re-compute the bounds for the affected nodes.

updateBoundLeftEchelon Function

input: split line which has replaced the deleted split line in the tree hierarchy, *S*

output: updated opposite ancestors

```
if $S.left$ = null
    return
end if
if $S.left.right$ = null
    return
end if
$current \leftarrow S.left.right$
while $current \neq null$
    $current.opBound \leftarrow S$
    $current \leftarrow current.right$
end while
```

Figure 4.10: The *updateBoundLeftEchelon* updates the rightmost descendants of the left child of the replacement node in the tree after a delete operation.

LiveRAC streamlines some of the rendering procedures from the PRISAD implementation of SequenceJuxtaposer. For both LiveRAC and SequenceJuxtaposer rendering requires finding all four split line bounds of a given cell. In SequenceJuxtaposer, the rendering queue consists of a list of split line ranges. A range is a set of index numbers into the split line hierarchy. During rendering in SequenceJuxtaposer, it is necessary to compute which split lines have boundaries that best match a given range, because the full range, or regions of a range, may have been aggregated into a single display block. If this is the case it is desirable to stretch and align the range span to the display block. The look up to discover whether a range subtends a display block is a binary search, an *O(log n)* operation. LiveRAC adds the split line root of a range directly to the queue, thus during rendering we need only do a bounds lookup on the split line. Since split line bounds are cached and maintained in the dynamic infrastructure each time the tree is modified, this is an *O(1)* operation during rendering.

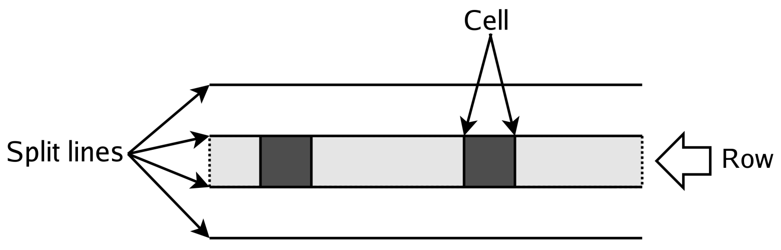
4.6 Picking

A picking operation is the process of locating a cell in the accordion drawing infrastructure given X and Y absolute co-ordinates in screen space. We first translate these into our rendering canvas world space co-ordinate system. To locate the appropriate device we descend the device axis split line tree in a binary search operation using the computed absolute values of the split lines. When we find the split line with the largest value that is smaller than our screen position, this split line is returned. The split line contains a pointer to the correct device which represents a row in our application. To find the appropriate column we perform a similar binary search on the column split line axis, and return the column index number. The index number permits an *O(1)* look up of the column name in the column ordering array, which we then use to find the correct cell in the device object's hash table.

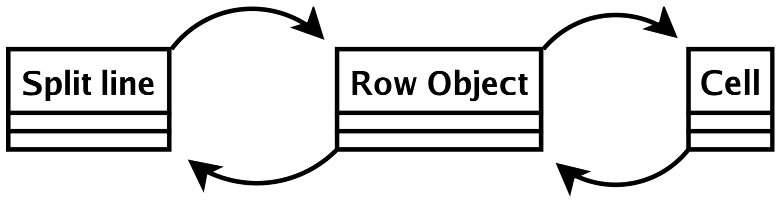
4.6.1 Challenges

The primary rendering and picking challenge was to look up an device or column name in *O(1)* time with any one of the following: a pointer to the corresponding split line, the index number of the corresponding split line, or the name of the device or bundle.

4.6.2 Our Approach



(a) Relationships of grid objects in split line hierarchy



(b) Pointer relationships between objects

Figure 4.11: (a) displays the relationships of grid objects within the split line grid hierarchy. A row is physically located between adjacent split lines, and can contain multiple cells. The column split lines are not shown in this figure for improved visual clarity. (b) demonstrates the pointer relationships between the objects. A split line has a pointer to the row located beneath it in the layout, and the row has a back pointer to its top split line. Rows maintain pointers to each of their cells, which contain back pointers to the row.

Our approach begins by distinguishing between rows which represent network devices, and columns which represent alarms and metrics. Because we know *a priori* that our data consists of many thousands of rows but only dozens or hundreds of columns we can exploit these characteristics and handle the two axes differently. If there are *N* rows and *M* columns, a complete matrix data structure would consist of a two dimensional array of *N* x *M* cells. However, not every cell in LiveRAC is populated with data. We implement row objects, discussed in more detail below, which store information about each row, including the list of cells containing representations within that row and the column index numbers where the cells are located. There is then no requirement for a corresponding column object. Because each row has pointers only to cells which are actually populated we save a considerable amount of heap memory, especially in the case of sparsely populated rows. A diagram illustrating grid object relationships is provided in Figure 4.11(a).

4.6.2.1 Row objects

We introduce the concept of a row object, a program data structure that represents a network device. The ordering of row objects is determined by the horizontal split lines, each split line object contains a pointer to a row object located immediately beneath itself and spanning the distance to the next visible split line. Thus, the row object *r* would span the distance between split line *n* and *n+1*. The decision to make this pointer point to the row beneath the split line rather than the row above was arbitrary and has no impact on the application as long as it is applied consistently throughout the data structure. Each row object also has a pointer back to its corresponding split line to make lookups possible in either direction as shown in Figure 4.11(b).

Each row object implements a comparator that defines its ordering relative to other row objects. For LiveRAC two comparators were implemented. The first comparator sorts row objects by their customer, and then sub-sorts by device name. The second comparator applies only to device names. When a split line and the associated row object is added to the data structure the split line tree is descended starting from the root, doing a binary search comparison at each step. Because the split line tree is derived from a red-black tree this is an *O(log n)* operation to find the correct place to insert. We also want to support the case where the ordering of the row objects could be a completely arbitrary arrangement of the user. In this case new devices are added as children of the last element in the data structure, which is rebalanced with each add operation.

When data is received by the server it is necessary to find a device by its string identifier. Because it is possible that the split line tree is in arbitrary order, thus denying us the ability to binary search, we need to provide a fast mechanism for obtaining the pointer to the correct row object. A hash table is maintained using the device's string identifier as the key. This allows for *O(1)* look up of row objects by name.

Each row object contains a table of corresponding entries for populated cells. This is further described in section 4.6.2.2.

4.6.2.2 Cells

LiveRAC must support three types of column operations for rendering and data storage: determining the column of a cell using a global ordering, getting the cell's column split line index numbers based on its associated bundle name and locating a bundle name based on an index number.

We address the first issue using a static global array associated with the row object which contains the complete list of columns in the accordion drawing framework, and maintains global ordering information. There is only a single instance of this list regardless of the number of devices and columns, and because the number of columns is small in comparison to the number of devices, *O(n)* operations for insert and delete from this table are acceptable.

During a rendering operation a non-culled row must look up the column split lines for each cell in its table of populated cells. This lookup needs to take place in *O(1)* time to ensure fast rendering. To provide this capability we use a hash table built from the global ordering array which is keyed using the string name of the cell's bundle. The result of the hash table look up is the integer index number of the cell's left split line.

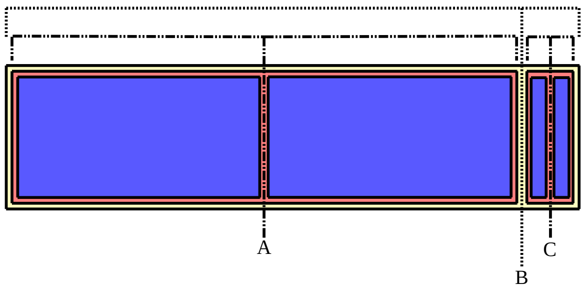
When we are drawing column labels we need to look up bundle names based on the index number of the column split line being drawn. This can be accomplished by using the index number of the column as the index number of the global ordering array.

4.7 Reordering Data

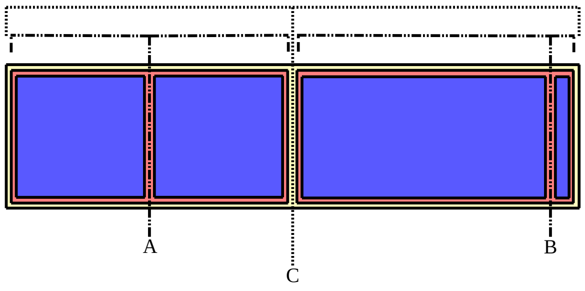
Reordering data is the process of swapping rows or columns of data inside of the accordion framework to facilitate side by side comparisons.

4.7.1 Challenges

There are two possible approaches for reordering rows and columns. The naive approach is to move split lines within the split line hierarchy, this operation is analogous to removing a split line and adding it at a different location in the hierarchy. The most significant drawback to this approach is that such a move operation causes problems maintaining the visual stability of the underlying tree. Let us say the line being moved is the root node, and it is positioned at 0.9. Both of its chil-



(a) User navigation has moved the root split line B to 0.9, enlarging the regions to the left of it.



(b) Swapping split lines B with C results in a dramatic shift in layout.

Figure 4.12: Naive reordering of a split line hierarchy on one axis. Three split lines are visible in both (a) and (b). We remove one split line and insert it in another location of the hierarchy. This is a poor approach to reordering because it can dramatically change the location of objects on the screen if relative values are not fixed appropriately. In Figure (a), the root split line B is set to a relative position of 0.9. Both children, A and C of the root have a relative position of 0.5. In Figure (b), we show the hierarchy after transposing the right child C with the root B from (a). There is a dramatic shift in cell layout as a result of this transposition that may cause confusion for the user.

dren are positioned at the default of 0.5. Because of the hierarchical nature of the accordion layout, the entire subtree on both sides of the new root will experience a significant shift in its on-screen location if we swap the root with the right child node, as demonstrated in Figure 4.12.

The problem of this dramatic shift could be fixed with additional bookkeeping operations in the split line hierarchy. However, there is a simpler alternative. Instead of moving the split lines, references can be swapped in the gridding data structures. This is the approach we have pursued in our implementation of data reordering.

4.7.2 Our Approach

As described in Section 4.2 we handle rows and columns differently. The row split lines maintain a pointer to a row object, and the row object has a back pointer to its corresponding split line. We can trivially relocate rows by swapping these two pointers with the corresponding pointers on another device. This makes the actual swap an *O(1)* operation on the data structures. To display the results a redraw must be performed.

We avoid having to modify each row object during a column reorder by storing the ordering of columns as a global structure. During rendering, a row object can lookup the position of a given cell by keying the name of that cell against this global hash table which returns an integer value representing the cell's column. We also maintain a list structure of column names for drawing the column labels. When a column is reordered, it is necessary to rebuild the list at *O(n)* cost, where *n* is the number of columns. This operation must be coupled with a fix to the hash table structure that allows the *O(1)* lookup of column index numbers, another *O(n)* operation. Although the column reorder operation has a worst case run time of *O(2n)*, where *n* represents the number of columns, we know that *n* is bounded to less than one thousand and typically is in the dozens, making this a trivial operation despite the high complexity cost. The performance benefits of O(1) lookups during rendering offset the up front expense.

Chapter 5

Results

In this chapter we present the qualitative and quantitative results from the LiveRAC visualization system. In Section 5.1 we examine the performance of operations on the split line tree. Section 5.2 presents and discusses qualitative results using the LiveRAC application with real network operation centre data.

5.1 Performance

Maintaining interactive performance in LiveRAC depends on the underlying split line infrastructure, rendering operations, and statistical graphic drawing time.

5.1.1 Split Line Data Structure

The split line data structure contains the largest number of data elements in LiveRAC. Typical operations on this data structure include: a shallow partial traversal while building the partitioned tree for rendering as discussed in Section 2.4, finding nodes by index, finding index numbers for nodes and finding node bounds. Insert and delete operations are less common, and therefore we accept a higher run time cost to pre-compute and cache values that are required during rendering to improve interactive performance. The performance of common split line tree operations is summarized in Table 5.1 and will be discussed in more detail below.

Because node bounds are cached during insert and delete operations, lookups

Table 5.1: Complexity for split line tree operations

Operation	Significant Operation Cost
Find node bounds	*1*
Find node by index	*log n*
Find index of node	*log n*
Insert node	*9 log n*
Delete node	*12 log n*

on node bounds require only one significant operation during actual rendering operations. Without the cache, finding bounds would require *log n* significant operations for each bound. The cache is maintained during insert and delete operations on the tree structure, with only affected nodes being updated.

All *log n* operations require only a descent from the root of the tree to a leaf node. All of the accounting to support node indexes is performed during node insert and delete operations.

The most expensive component of the insert and delete operations is updating node bounds during tree rotations. Red-black trees require a maximum of 2 rotations during insert and three rotations during delete operations. Each of these rotations can affect a maximum of three nodes, requiring that the bounds of these nodes be re-computed at *log n* cost for each. Node inserts have a base cost of *3 log n*: locating the node position to insert at (*log n*), updating the subtree count (*log n*), and locating the bounds of the inserted node (*log n*).Therefore, maintaining an accurate cache of node bounds can require in the worst case *9 log n* for inserts and *12 log n* for deletes, when we add in the base cost of a red-black insert and index accounting. We suggest ways of reducing this cost in Section 6.1.

5.1.2 Rendering

We optimize rendering performance in LiveRAC by caching the most critical operations and re-computing only when changes are made to the underlying split line data structure. These caches include the bounds cache from the split line infrastructure, a column index cache, and a node aggregation cache. The design goal of the PRISAD infrastructure is to partition the split line hierarchy into pixel space, bounding rendering performance to *O(p)* where *p* is the number of pixels on a given split line axis. LiveRAC adheres to this design goal.

5.1.3 Charts

Because most cells in LiveRAC will typically be too small to show representations more sophisticated than simple colours, charts are not as critical to the performance of the application as the split line structure. Even on a large display it is difficult to have more than a few dozen charts visible at any given time. However, because charts can be complex visual objects, care was taken to optimize the performance of the charting library.

To achieve interactive rendering speeds, jGLChartUtil uses the simplest data structures possible. Data is stored in arrays using the primitive type double, rather than in more flexible but slower object wrappers and dynamic data structures. The charting utility was written to do the minimum amount of layout possible. All representation awareness is maintained outside of the charting library. The only area-aware components of jGLChartUtil are the scalars for drawing the chart, auto font sizing and label collision detection. The charting library detects changes to these underlying data or charting options and rebuilds the chart on demand.

5.1.3.1 Chart Meta-Data Caching

All meta-data required by the chart is computed once and cached. This includes information with respect to the scale of the axis, labels, legends and positions within the chart co-ordinate system. The cache is marked as dirty when updates are made to the individual data series. Dirty markers are maintained for each data series separately, so a modification to one data series will require only that data series to be re-evaluated. Resize operations on the chart do not require the rebuilding of any data series meta-data. All data series meta-data computations are *O(d)* worst case, where *d* is the number of data points in the chart.

5.1.3.2 Drawing

Chart drawing operations without labels are *O(d)* where *d* is the number of data points in the chart, plus a fixed cost for setting up the rendering.

Charts are drawn using OpenGL and are stored as display lists. Display lists cache OpenGL instructions on the graphics card, which can result in significant performance improvements on newer graphics hardware. Using a 3GHz Pentium IV system, a chart with three data series of 100 data points takes under 50ms for the first draw, and under 5ms for subsequent redraws. Typical time for rendering a cached draw list is under 1 ms.

If the chart is resized, it may be necessary to re-compute chart meta-data, and recreate the display list. Although a complete redraw of the chart is fast, we can optimized further by scaling the chart to fit the new dimensions instead of redrawing if the change is only a few pixels, making the distortion almost invisible. Because navigation activities on one side of the display may have only minor effects on the size of objects on the other side of the display, small changes to charts can be quite common.

5.2 Discussion

Although more work is needed for LiveRAC to become a software package that can be deployed to end users, we are able to make a number of observations based on the current state of the application. Performance is quite good. Navigation actions are smooth, and the only visible delay in chart loading is retrieving data from the SWIFT server. When loading aggregate alarm and statistic data in queries with a very large time range, such as a week or more of data, the query can take over a second. Although this could stand improvement, the amount of data that must be parsed on the server side dictates that some perceivable delay will always exist on large queries.

Positive feedback from users is encouraging, as is enthusiasm for seeing this research develop into a tool which can be deployed in the field. That administrators feel that context has value for situational awareness and navigation is an important validation of our focus+context approach to addressing this visualization challenge.

5.2.1 Qualitative Results

LiveRAC has been developed using an iterative process which has consistently involved personnel from a Network Operations Centre at AT&T and feedback from researchers familiar with creating visualizations for this domain. During the early stages a first generation mock-up prototype showing the capabilities of semantic zoom in accordion drawing was presented, using randomly generated data. The positive feedback gleaned from this first generation system, and the dialog that was generated, have guided the evolution of the LiveRAC prototype.

The most salient results for LiveRAC are the discovery of real world conditions using the tool. Using LiveRAC we have been able to discover the following:

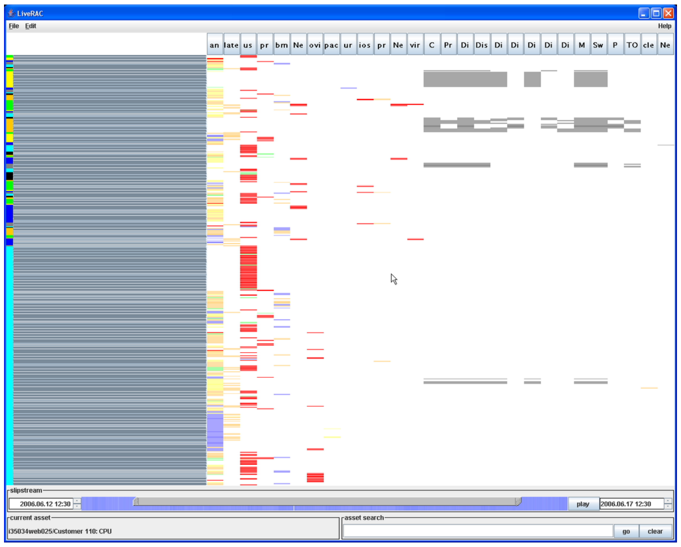
- Variance in /var debris accumulation and scheduled cleanup patterns

- Evidence of idle processes on some web servers
- Cyclical load patterns
- Uneven distribution of load across clusters
- Impact of web server load on filers

Feedback regarding the prototype has been positive. A senior manager at AT&T has said the tool has promise as a next generation platform for their executive dashboard of data centre operations. Funding for the project is ongoing and the prototype will continue to evolve.

5.2.2 Visual Case Study

In this section we document how the LiveRAC application can be used to examine and diagnose problems on a large web farm. The data we are examining is real, but names have been obfuscated for reasons of confidentiality. During the monitoring period, the web farm we are examining was under heavy web load due to a large sporting event.

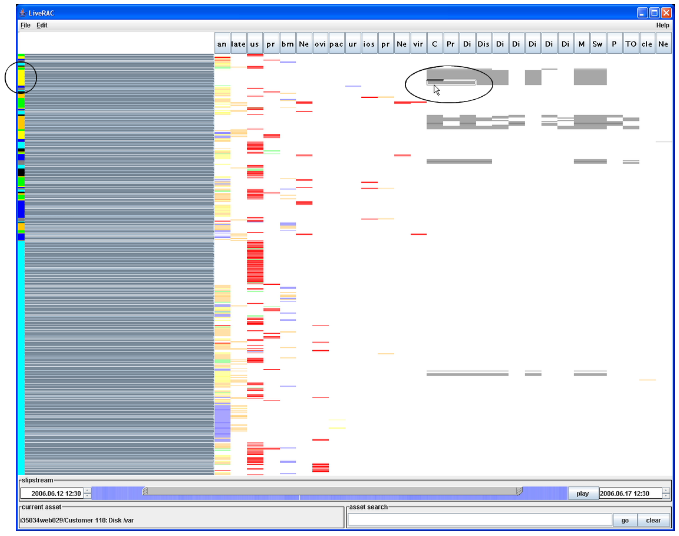

LiveRAC
File Edit
Help
an late us pr bm Ne ovi pac ur ios pr Ne vir C Pr Di Dis Di Di Di Di Di Di M Sw P TO cle Ne
slipstream
play
current asset
asset search
go
clear

2006.06.12 12:30

2006.06.17 12:30

i35034web025/Customer 110: CPU

Figure 5.1: This screen capture displays 5 days worth of alarm and metric data. No focus regions have been selected.

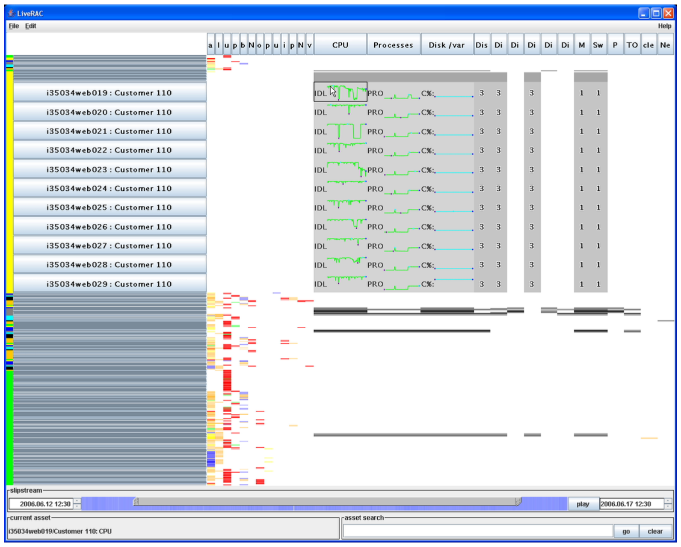

LiveRAC
File Edit
Help
an late us pr bm Ne ovi pac ur ios pr Ne vir C Pr Di Dis Di Di Di Di Di Di M Sw P TO cle Ne
slipstream
play
current asset
asset search
go
clear

2006.06.12 12:30

2006.06.17 12:30

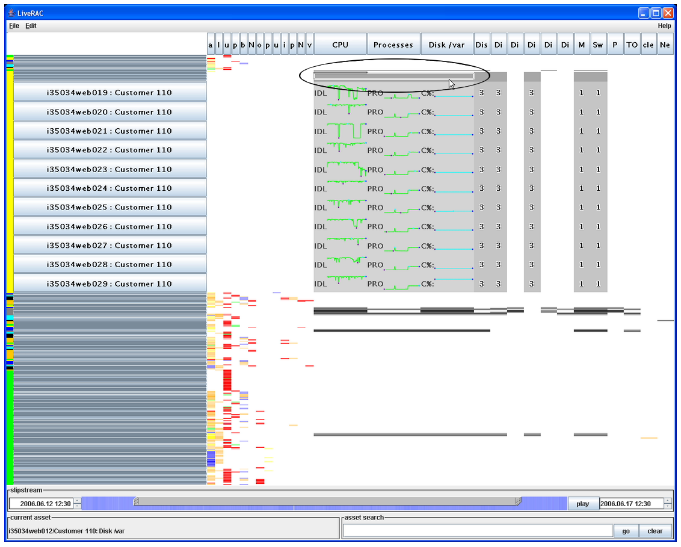
i35034web029/Customer 110: Disk /var

Figure 5.2: The user selects a region to stretch. Using the group bar located on the far left of the display, it is possible to tell that this entire region represents a single customer grouping.


LiveRAC
File Edit
Help
CPU
Processes
Disk /var
i35034web019 : Customer 110
i35034web020 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
i35034web027 : Customer 110
i35034web028 : Customer 110
i35034web029 : Customer 110
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
asset search
go
clear

i35034web019/Customer 110: CPU

Figure 5.3: The user expands the region sufficiently to reveal a series of spark lines. We can see some interesting trend information in the CPU and Processes columns.


LiveRAC
File Edit
Help
CPU
Processes
Disk /var
i35034web019 : Customer 110
i35034web020 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
i35034web027 : Customer 110
i35034web028 : Customer 110
i35034web029 : Customer 110
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
asset search
go
clear

i35034web012/Customer 110: Disk /var

Figure 5.4: The user selects another region from the same customer grouping.

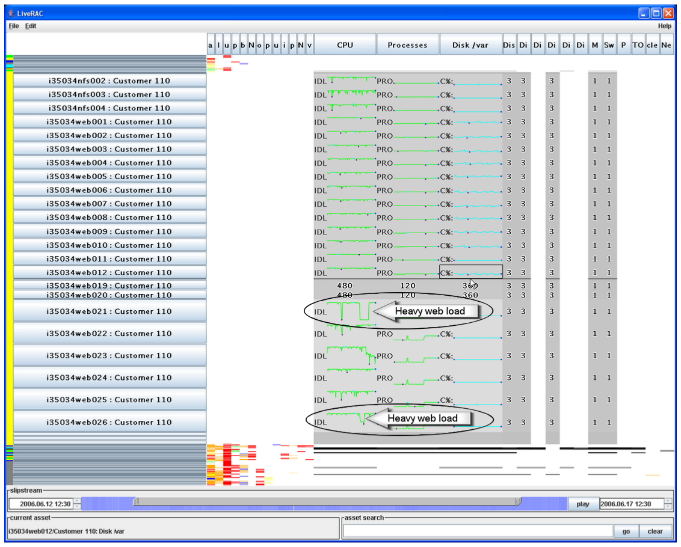

LiveRAC
File Edit
Help
a l u p b N o p u i p N v
CPU
Processes
Disk /var
Dis Di Di Di Di Di M Sw P TO cle Ne
i35034nfs002 : Customer 110
i35034nfs003 : Customer 110
i35034nfs004 : Customer 110
i35034web001 : Customer 110
i35034web002 : Customer 110
i35034web003 : Customer 110
i35034web004 : Customer 110
i35034web005 : Customer 110
i35034web006 : Customer 110
i35034web007 : Customer 110
i35034web008 : Customer 110
i35034web009 : Customer 110
i35034web010 : Customer 110
i35034web011 : Customer 110
i35034web012 : Customer 110
i35034web019 : Customer 110
i35034web020 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
IDL
PRO
C%:
480
120
360
Heavy web load
Heavy web load
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
i35034web012/Customer 110: Disk /var
asset search
go
clear

Figure 5.5: Periods of heavy CPU utilization are visible on several of the servers in the web farm as downward pointing spikes on the idle CPU chart. Note the corresponding jump in number of processes on these machines. The servers closer to the top of the screen capture did not experience similar load spikes in CPU, indicating an imbalance in load distribution. Also of interest is the column indicating the use of /var. The systems towards the top of the screen capture show periodic growth and cleanup in the use of this file system, whereas the systems below do not. This could indicate configuration differences between the systems.

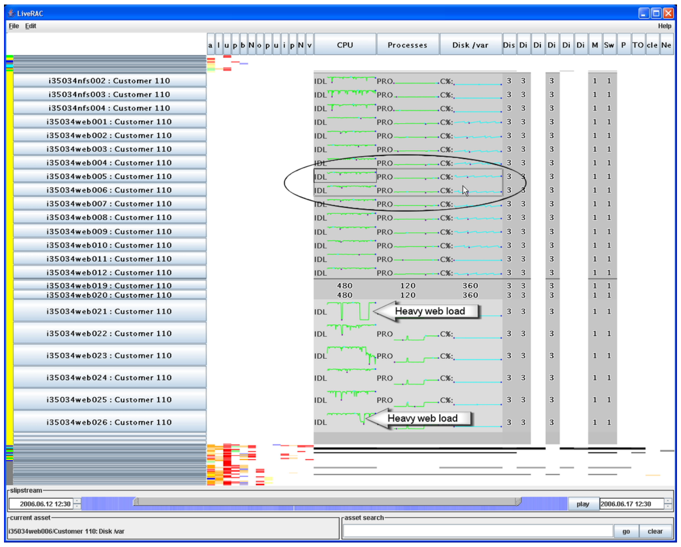

LiveRAC
File Edit
Help
CPU
Processes
Disk /var
i35034nfs002 : Customer 110
i35034nfs003 : Customer 110
i35034nfs004 : Customer 110
i35034web001 : Customer 110
i35034web002 : Customer 110
i35034web003 : Customer 110
i35034web004 : Customer 110
i35034web005 : Customer 110
i35034web006 : Customer 110
i35034web007 : Customer 110
i35034web008 : Customer 110
i35034web009 : Customer 110
i35034web010 : Customer 110
i35034web011 : Customer 110
i35034web012 : Customer 110
i35034web019 : Customer 110
i35034web020 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
Heavy web load
Heavy web load
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
i35034web006/Customer 110: Disk /var
asset search
go
clear

Figure 5.6: A region of web servers is selected to expand for more detailed information. The machines selected have been chosen to use as baseline systems to compare against the more unusual behaviours from the systems located below.

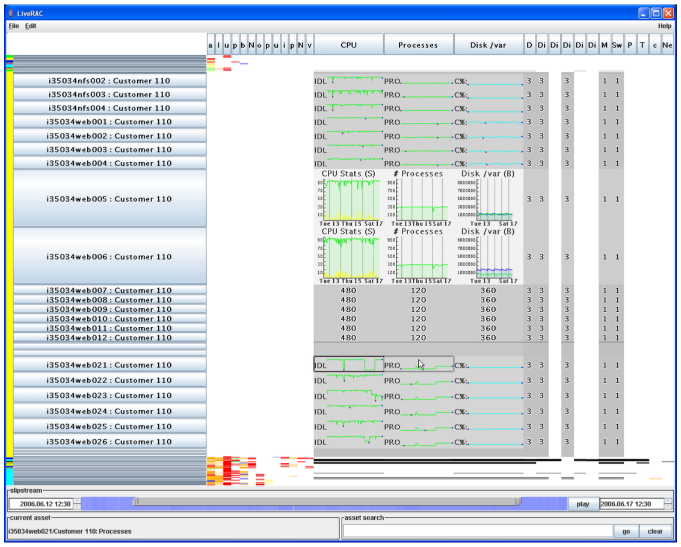

LiveRAC
File Edit
Help
CPU
Processes
Disk /var
i35034nfs002 : Customer 110
i35034nfs003 : Customer 110
i35034nfs004 : Customer 110
i35034web001 : Customer 110
i35034web002 : Customer 110
i35034web003 : Customer 110
i35034web004 : Customer 110
i35034web005 : Customer 110
CPU Stats (S)
# Processes
Disk /var (B)
i35034web006 : Customer 110
i35034web007 : Customer 110
i35034web008 : Customer 110
i35034web009 : Customer 110
i35034web010 : Customer 110
i35034web011 : Customer 110
i35034web012 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
i35034web021/Customer 110: Processes
asset search
go
clear

Figure 5.7: The user has expanded i35034web005 and i35034web006. We see more details of server behaviour during the selected time range. The user has now selected one of the web servers that exhibited unusual behaviour, and prepares to expand it. The legend is not yet visible.

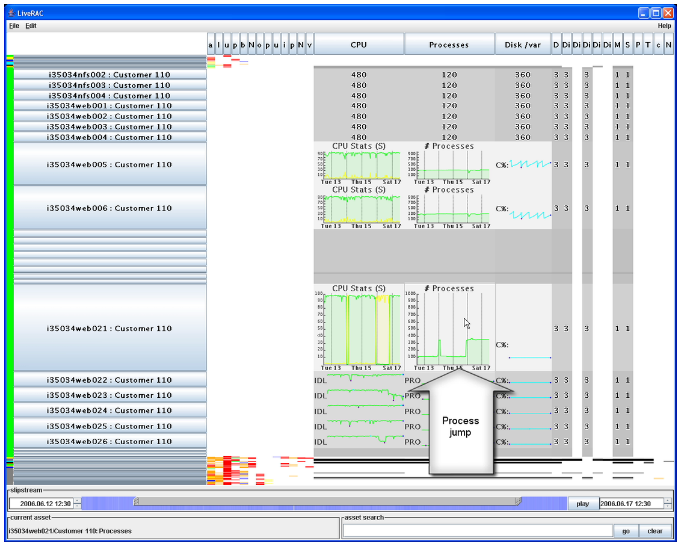

LiveRAC
File
Edit
Help
CPU
Processes
Disk /var
i35034nfs002 : Customer 110
i35034nfs003 : Customer 110
i35034nfs004 : Customer 110
i35034web001 : Customer 110
i35034web002 : Customer 110
i35034web003 : Customer 110
i35034web004 : Customer 110
i35034web005 : Customer 110
i35034web006 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
CPU Stats (S)
# Processes
Tue 13
Thu 15
Sat 17
Process jump
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
i35034web021/Customer 110: Processes
asset search
go
clear

Figure 5.8: The user has expanded i35034web021. We now have more details on the extent of the process spike, and can see the system was at 100% utilization from Friday 16 until Sat 17 on the chart. There were spikes at corresponding periods on the baselines system but not to nearly the same extent. Also, notice that even though the load reduced on Sat 17 for i35034web021, the number of processes has stayed high. This may indicate apache is not cleaning up idle server threads, a potential performance problem.

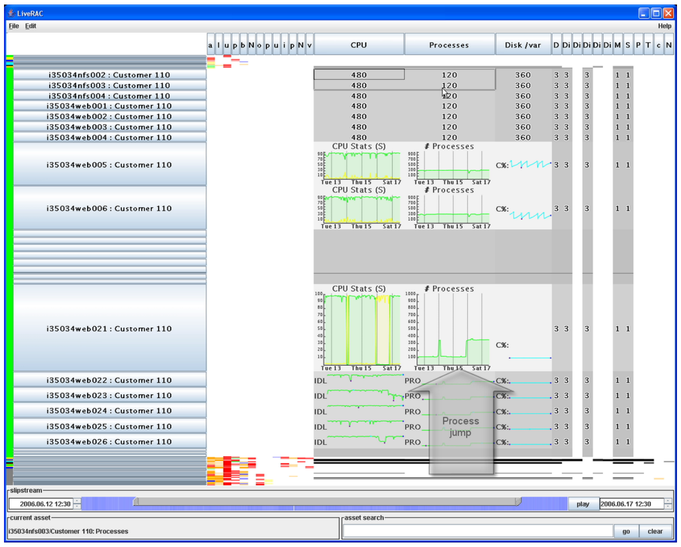

LiveRAC
File Edit
Help
CPU
Processes
Disk /var
i35034nfs002 : Customer 110
i35034nfs003 : Customer 110
i35034nfs004 : Customer 110
i35034web001 : Customer 110
i35034web002 : Customer 110
i35034web003 : Customer 110
i35034web004 : Customer 110
i35034web005 : Customer 110
i35034web006 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
CPU Stats (S)
# Processes
Process jump
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
i35034nfs003/Customer 110: Processes
asset search
go
clear

Figure 5.9: Near the top of the screen capture the user selects two of the NFS filers associated with the web farm. These filers provide the disk storage for the web farm.

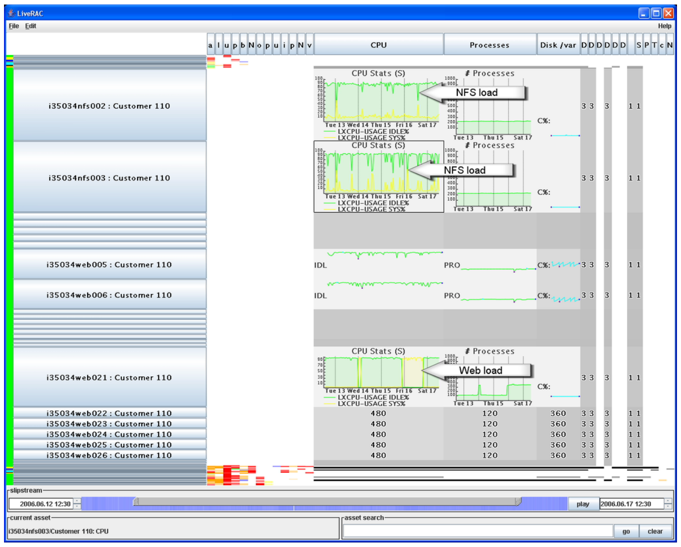

LiveRAC
File Edit
Help
CPU
Processes
Disk /var
i35034nfs002 : Customer 110
i35034nfs003 : Customer 110
i35034web005 : Customer 110
i35034web006 : Customer 110
i35034web021 : Customer 110
i35034web022 : Customer 110
i35034web023 : Customer 110
i35034web024 : Customer 110
i35034web025 : Customer 110
i35034web026 : Customer 110
CPU Stats (S)
# Processes
NFS load
NFS load
Web load
LXCPU-USAGE IDLE%
LXCPU-USAGE SYS%
slipstream
2006.06.12 12:30
play
2006.06.17 12:30
current asset
i35034nfs003/Customer 110: CPU
asset search
go
clear

Figure 5.10: The NFS filers experienced an increased frequency of load spikes starting around the same period as i35034web021 experienced heavy CPU utilization. However, neither NFS filer approached capacity, indicating there were sufficient resources to cope.

Chapter 6

Conclusions and Future Work

6.1 Future Work

Interaction with the LiveRAC application suggests a number of directions which could improve the value of the tool to network administrators and management users. We have divided these directions into three categories: interaction, data processing and performance.

6.1.1 Interaction

Although the accordion drawing interaction technique offers the advantages of context and a smooth navigation metaphor, the interface does have a significant learning curve. Adding capabilities to automatically expand regions of the display using keyboard shortcuts, for example by hotkeying certain groups of devices, would make it easier for administrators to quickly look at a group of devices.

Administrators may want to lock the application to view a fixed set of devices and alarm bundles. For example, an administrator may be responsible for only a single customer. An option to specify explicitly which machines, alarms and metrics are to be displayed would help filter data for users who have specific areas of responsibility or who are only interested in a narrow subset of the system.

Currently row and column reordering takes place using external dialogs. This offers the advantage of being easier to use because the user can perform the reordering irrespective of compressed regions. However, the user should be permitted to interact directly with rows and columns in the accordion drawing view. Using a lensing technique that spreads the region directly beneath the cursor when the user is dragging a column or row to a new location it should be possible to mitigate some of the difficulties posed by a direct manipulation technique.

LiveRAC currently permits users to build columns which represent aggregates of multiple metrics or alarms, called bundles. This functionality could extend to allowing the user to add groups of devices together, and to average the values from this group into aggregate statistics for each bundle. This provides a good view of the state of closely related devices, such as a web farm.

LiveRAC does not currently allow the user to view actual alarm text, or values of the statistic data, even though this information is stored in memory. Adding a separate panel or window for viewing raw data elements in a table would provide access to this low-level information which may be valuable during deep analysis.

6.1.2 Data Processing

Currently the SWIFT system creates a running average of all gathered statistics. This trend information is stored for different time scales, such as days, weeks and months. Using this data provides both an excellent baseline for making comparisons against recorded data, and a method for conducting predictive analytics by projecting the trends into the future. This functionality could be further extended to alarm data, perhaps leading to a system that de-prioritizes alarms that are expected on a daily basis, reducing operator workload.

Currently there is no correlation computation performed between alarm events and metrics. An obvious next step is to add causal analysis, to determine if there is a significant relationship between when alarm events are generated and the underlying state of the monitored system. The result may be 'hybrid' cells which map both alarm and supporting statistical information.

6.1.3 Performance

As described in Section 5.1.1, split line bounds are re-computed with every add and remove operation to the split line axis. This allows us to improve rendering performance at the expense of an up-front charge when the split line tree is modified. An alternative solution would be to store a modification count on the split line axis, and lazily compute bounds as needed, storing the index of when each bound was computed in the split line object, and caching the bounds. Because many split lines are never actually rendered if they fall beneath the partitioning range, this might offer a significant performance improvement in very large data sets at the expense of increased memory usage and a slow initial rendering after data structure modification. The cost of significant insert and delete operations would drop to *3 log n*.

6.2 Conclusion

We have presented LiveRAC, a focus+context visualization system for monitoring large collections of networked devices. The system scales to thousands of network devices, dozens of monitored metrics and alarm types and can display data ranging over a period of months. LiveRAC supports NOC operations staff by providing situational awareness in parallel to other activities that require detail views such as incident investigation and capacity planning.

We have modified the split line data structure for logarithmic time insert and delete operations, allowing data elements to be added and removed from the visualization at run time. Data reordering significantly improves the ability of users to perform side-by-side comparisons for regions of interest [61]. The dynamic split line code is fully integrated into PRISAD [34] and can be used for future accordion drawing applications that require this capability.

We have created a library and infrastructure for semantic zoom which allows multiple data representations to be mapped into each cell created by the split line grid. These representations can be specified hierarchically to allow the optimal representation to be displayed for different focus levels. The library is generic and can be used for extensions to the LiveRAC visualization system and for future projects.

Quantitative measurements and qualitative feedback indicate that LiveRAC is a scalable, effective visualization system for monitoring system metric and alarm data in complex NOC environments.

Bibliography

[1] Kulsoom Abdullah, Chris Lee, Gregory Conti, John A. Copeland, and John Stasko. IDS RainStorm: Visualizing ids alarms. In *VIZSEC '05: Proceedings of the IEEE Workshops on Visualization for Computer Security*, page 1, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Analysis Console for Intrusion Databases (ACID). downloadable at: http://acidlab.sourceforge.net/, cited July 26, 2006.

[3] Christopher Ahlberg and Ben Shneiderman. Visual information seeking: tight coupling of dynamic query filters with starfield displays. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 313–317, New York, NY, USA, 1994. ACM Press.

[4] Robert Ball, Glenn A. Fink, and Chris North. Home-centric visualization of network traffic for security administration. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 55–64. ACM Press, 2004.

[5] Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. The continuous zoom: a constrained fisheye technique for viewing and navigating large information spaces. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 207–215, New York, NY, USA, 1995. ACM Press.

[6] Richard A. Becker, Stephen G. Eick, and Allan R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995.

[7] Dale Beermann, Tamara Munzner, and Greg Humphreys. Scalable, robust visualization of very large trees. In *EuroVis 2005*, 2005.

[8] James R. Beniger and Dorothy L. Robyn. Quantitative graphics in statistics: A brief history. *American Statistician*, 32(1):1–11, 1978.

[9] J. Bertin. *Graphics and graphic information processing*. Walter de Gruyter, Berlin, Germany, 1981.

[10] J. Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[11] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. Toolglass and magic lenses: the see-through interface. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 73–80, New York, NY, USA, 1993. ACM Press.

[12] Stuart Card and David Nation. Degree-of-interest trees: A component of an attention-reactive user interface. In *Proceedings of Advanced Visual Interface*, pages 231–246, 2002.

[13] M. Sheelagh T. Carpendale, David J. Cowperthwaite, and F. David Fracchia. 3-dimensional pliable surfaces: for the effective presentation of visual information. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 217–226, New York, NY, USA, 1995. ACM Press.

[14] Sheelagh Carpendale, John Ligh, and Eric Pattison. Achieving higher magnification in context. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 71–80, New York, NY, USA, 2004. ACM Press.

[15] Mei C. Chuah, Steven F. Roth, Joe Mattis, and John Kolojejchick. SDM: Selective dynamic manipulation of visualizations. In *ACM Symposium on User Interface Software and Technology*, pages 61–70, 1995.

[16] William S. Cleveland. *The elements of graphing data*. Wadsworth Publ. Co., Belmont, CA, USA, 1985.

[17] Clumon - The Cluster Monitoring System. http://clumon.ncsa.uiuc.edu/, cited Aug 10, 2006.

[18] Gerald Combs. Ethereal. downloadable at: http://www.ethereal.com/, cited July 5, 2006.

[19] Kenneth C. Cox, Stephen G. Eick, and Taosong He. 3D geographic network displays. *SIGMOD Rec.*, 25(4):50–54, 1996.

[20] Stephen G. Eick, Michael C. Nelson, and Jerry D. Schmidt. Graphical analysis of computer log files. *Commun. ACM*, 37(12):50–56, 1994.

[21] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[22] Jean-Daniel Fekete. The InfoVis Toolkit. In *Proc. of InfoVis '04*, pages 167–174, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[23] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23, New York, NY, USA, 1986. ACM Press.

[24] Ganglia. http://ganglia.sourceforge.net/, cited Aug 10, 2006.

[25] David Gilbert. jFreeChart. downloadable at: http://www.jfree.org/jfreechart/, cited July 4, 2006.

[26] Martin Graham and Jessie Kennedy. Exploring and examining assessment data via a matrix visualisation. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 158–162, New York, NY, USA, 2004. ACM Press.

[27] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430, New York, NY, USA, 2005. ACM Press.

[28] I. Herman, G. Melancon, and M.S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[29] Harry Hochheiser and Ben Shneiderman. Interactive exploration of time series data. In *DS '01: Proceedings of the 4th International Conference on Discovery Science*, pages 441–446, London, UK, 2001. Springer-Verlag.

[30] Kasper Hornbæk, Benjamin B. Bederson, and Catherine Plaisant. Navigation patterns and usability of zoomable user interfaces with and without an overview. *ACM Trans. Comput.-Hum. Interact.*, 9(4):362–389, 2002.

[31] HP Inc. HP OpenView. http://www.managementsoftware.hp.com/, cited July 5, 2006.

[32] IBM Ltd. IBM Tivoli NetView. http://www.ibm.com/software/tivoli/products/netview/, cited July 5, 2006.

[33] Van Jacobson, Craig Leres, and Steven McCanne. TCPdump public repository. http://www.tcpdump.org/, cited December 13, 2004.

[34] James Slack and Kristian Hildebrand and Tamara Munzner. PRISAD: A partitioned rendering infrastructure for scalable accordion drawing. In *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 6, Washington, DC, USA, 2005. IEEE Computer Society.

[35] The JOGL API Project. https://jogl.dev.java.net/, cited August 6, 2006.

[36] Susanne Jul and George W. Furnas. Critical zones in desert fog: aids to multiscale navigation. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 97–106, New York, NY, USA, 1998. ACM Press.

[37] Robert Kincaid. VistaClara: an interactive visualization for exploratory analysis of DNA microarrays. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 167–174, New York, NY, USA, 2004. ACM Press.

[38] Robert Kincaid and Heidi Lam. Line graph explorer: scalable display of line graphs using focus+context. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, pages 404–411, New York, NY, USA, 2006. ACM Press.

[39] Hideki Koike and Kazuhiro Ohno. Snortview: visualization system of snort logs. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 143–147, New York, NY, USA, 2004. ACM Press.

[40] Qiang Kong. Visual mining of powersets with large alphabets. Master's thesis, University of British Columbia, 201-2366 Main Mall, Vancouver, B.C., V6T 1Z4, May 2006.

[41] Eleftherios E. Koutsofios, Stephen C. North, Russell Truscott, and Daniel A. Keim. Visualizing large-scale telecommunication networks and services (case study). In *VIS '99: Proceedings of the conference on Visualization '99*, pages 457–461, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[42] Kiran Lakkaraju, Ratna Bearavolu, Adam Slagell, William Yurcik, and Stephen North. Closing-the-loop in NVisionIP: Integrating discovery and search in security visualizations. In *VIZSEC '05: Proceedings of the IEEE Workshops on Visualization for Computer Security (VizSec'05)*, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[43] Kiran Lakkaraju, William Yurcik, and Adam J. Lee. NVisionIP: netflow visualizations of system state for security situational awareness. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 65–72. ACM Press, 2004.

[44] Jessica Lin, Eamonn Keogh, Stefano Lonardi, Jeffrey P. Lankford, and Donna M. Nystrom. Visually mining and monitoring massive time series. In *KDD '04: Proceedings of the tenth ACM SIGKDD international confer-*

ence on Knowledge discovery and data mining, pages 460–469, New York, NY, USA, 2004. ACM Press.

[45] Sasa Markovic and Arne Vandamme. jRobin. downloadable at: http://www.jrobin.org/, cited July 4, 2006.

[46] Peter McLachlan. jGLChartUtil. downloadable at: http://www.sourceforge.net/projects/jglchartutil, cited July 4, 2006.

[47] Tamara Munzner, François Guimbretiére, Serdar Tasiran, Li Zhang, and Yunhong Zhou. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Trans. Graph.*, 22(3):453–462, 2003.

[48] Dmitry Nekrasovski, Adam Bodnar, Joanna McGrenere, François Guimbretiére, and Tamara Munzner. An evaluation of pan & zoom and rubber sheet navigation with and without an overview. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 11–20, New York, NY, USA, 2006. ACM Press.

[49] Tor Norretranders. *The User Illusion*. Viking, 1999.

[50] OpenNMS. http://www.opennms.org/, cited July 5, 2006.

[51] Ken Perlin and David Fox. Pad: an alternative approach to the computer interface. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 57–64, New York, NY, USA, 1993. ACM Press.

[52] Catherine Plaisant, Jesse Grosjean, and Benjamin B. Bederson. SpaceTree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *INFOVIS '02: Proceedings of the IEEE Symposium on In-*

formation Visualization (InfoVis'02), page 57, Washington, DC, USA, 2002. IEEE Computer Society.

[53] Ramana Rao and Stuart K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 318–322, New York, NY, USA, 1994. ACM Press.

[54] George G. Robertson and Jock D. Mackinlay. The document lens. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 101–108, New York, NY, USA, 1993. ACM Press.

[55] Marty Roesch. SNORT. downloadable at: http://www.snort.org/, cited July 26, 2006.

[56] RRDTool. http://oss.oetiker.ch/rrdtool/, cited Aug 10, 2006.

[57] Takafumi Saito, Hiroko Nakamura Miyamura, Mitsuyoshi Yamamoto, Hiroki Saito, Yuka Hoshiya, and Takumi Kaseda. Two-tone pseudo coloring: Compact visualization for one-dimensional data. In *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 23, Washington, DC, USA, 2005. IEEE Computer Society.

[58] Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss. Stretching the rubber sheet: a metaphor for viewing large layouts on small screens. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 81–91, New York, NY, USA, 1993. ACM Press.

[59] Ben Schneiderman. Dynamic queries for visual information seeking. *IEEE Softw.*, 11(6):70–77, 1994.

[60] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336. IEEE Computer Society, 1996.

[61] Harri Siirtola. Interaction with the reorderable matrix. In *IV '99: Proceedings of the 1999 International Conference on Information Visualisation*, page 272, Washington, DC, USA, 1999. IEEE Computer Society.

[62] James Slack, Kristian Hildebrand, Tamara Munzner, and Katherine St. John. Sequencejuxtaposer: Fluid navigation for large-scale sequence comparison in context. In *German Conference in Bioinfromatics*, 2004.

[63] Robert Spence and Mark Apperley. Data base navigation: an office environment for the professional. *Readings in information visualization: using vision to think*, pages 333–340, 1999.

[64] Chris Stolte, Diane Tang, and Pat Hanrahan. Query, analysis, and visualization of hierarchically structured data using polaris. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 112–122, New York, NY, USA, 2002. ACM Press.

[65] Tetsuji Takada and Hideki Koike. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 133–144, Berkeley, CA, USA, 2002. USENIX Association.

[66] Thomas H. Cormen and Charles E. Lieserson and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[67] Edward Tufte. *The Visual Display of Quantitative Information*. Graphics Press, first edition, 1981.

[68] Edward Tufte. *Envisioning Information*. Graphics Press, 1990.

[69] Edward Tufte. *Beautiful Evidence*. Graphics Press, first edition, 2006.

[70] Edward R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, February 1997.

[71] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, second edition, 2004.

[72] Marc Weber, Marc Alexa, and Wolfgang Műller. Visualizing time-series on spirals. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, page 7, Washington, DC, USA, 2001. IEEE Computer Society.

[73] Jarke J. Van Wijk and Edward R. Van Selow. Cluster and calendar based visualization of time series data. In *INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization*, page 4, Washington, DC, USA, 1999. IEEE Computer Society.

[74] William Yurcik, Xin Meng, and Nadir Kiyanclar. NVisionCC: a visualization framework for high performance cluster security. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 133–137, New York, NY, USA, 2004. ACM Press.