5-2014

Corl8: A System for Analyzing Diagnostic Measures in Wireless Sensor Networks

Loren Klingman
Clemson University, loren@klingman.us

Corl8: A System for Analyzing Diagnostic Measures in Wireless Sensor Networks

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

by
Loren Klingman
May 2014

Abstract

Due to an increasing demand to monitor the physical world, researchers are deploying wireless sensor networks more than ever before. These networks comprise a large number of sensors integrated with small, low-power wireless transceivers used to transmit data to a central processing and storage location. These devices are often deployed in harsh, volatile locations, which increases their failure rate and decreases the rate at which packets can be successfully transmitted.

Existing sensor debugging tools, such as Sympathy and EmStar, rely on add-in network protocols to report status information, and to collectively diagnose network problems. Some protocols rely on a central node to initiate the diagnosis sequence. These methods can congest network channels and consume scarce resources, including battery power.

In this thesis, we present Corl8, a system for analyzing diagnostic traces in wireless sensor networks. Our method relies on diagnostic data that is periodically transmitted to a network sink as a part of the standard sensor payload to enable fault diagnosis. Corl8 does not require any specific data to be present in the system, making it flexible. Our system provides an interactive environment for exploring correlated changes across different diagnostic measures within an individual node. It also supports processing on a batch level to automatically flag interesting correlations. The system's flexibility makes it applicable for use in any wireless sensor network that transmits diagnostic measures. The analysis methods are user-configurable, but we suggest settings and analyze their performance. For our evaluation, we use data from five real-world deployments from the Intelligent River® project consisting of 36 sensor nodes.

Acknowledgments

I would first like to thank God through whom all things are possible (Matthew 19:26) and who gifted me with the intelligence to be able to complete this work. I owe gratitude and the continuing debt of love to so many people who helped to make this possible. My advisor, Dr. Jason Hallstrom, who helped me see the potential of this research project deserves many thanks. Without his advice and guidance, this thesis would have never been completed or would have been lacking in quality. I would like to thank my thesis committee, Dr. Brian Malloy, and Dr. Jacob Sorber, for their time and support to improve my work. This work is supported by the National Science Foundation through MRI award CNS-1126344. Finally, a special thanks to my mother, Dr. Nancy Phillips, who helped proof the manuscript and supported me throughout my studies.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Researchers use wireless sensor networks to monitor the physical world and to gather information that would otherwise be unavailable. These networks use a large number of sensors to gather data, often in harsh environments over a large area. Each node is equipped with a small, low-power wireless device used to transmit data to a central processing and storage location. These networks can be applied in a wide range of domains, such as habitat monitoring [35], finding nearby friends [31], microclimate monitoring [38], sow monitoring [5], pipeline monitoring [12], target detection [39], coal mine structural monitoring [14], early warnings for floods [4], and hospital patient monitoring [6]. Fault tolerance is critical in sensor networks because they are failure-prone in terms of data delivery resulting from link quality, packet corruption, and wireless congestion. These factors are often exacerbated due to multi-hop communication between nodes and the sink [18].

The nodes in a wireless sensor network have a unique set of requirements and hardware constraints. They must be able to endure the environment, require minimal maintenance, and function with little power in small spaces. Equally important, they must be able to react to and recover from failures, even though they function with much smaller storage and computational abilities than a desktop computer or even a smartphone. These limitations present challenges for debugging failures because there is insufficient space on a node to store a full debugging log, such as those commonly used on desktop computers. Further, energy limitations prohibit transmission of a complete log to a central node.

1.1 Motivation

Many solutions have emerged to address the challenge of debugging wireless sensor networks. Tools such as Sympathy [23] and EmStar [8] rely on add-in network protocols to report status information. EmView, a component of EmStar, also relies on a central node to initiate the diagnosis sequence. These methods can congest network channels and consume extra energy to transmit the data.

Our sensing system includes built-in reporting of a variety of diagnostic measures. These measures are typical analyzed manually when nodes are not reporting as expected. However, it is difficult to understand what these numbers mean without tools to interpret them. As preliminary work for this, we converted the raw diagnostic measures into graphs of rates per minute, per sensor reading attempt, and per transmission attempt. For example, errors having to do with wireless transmission were best represented based on transmission attempts, and errors associated with sensor readings were best based on reading attempts. Our programmers and deployment technicians were then able to use the associated graphs to quickly track deployment status.

For phase one of the work presented here, we apply standard statistical techniques to identify correlated error rates. We analyze error rates for multiple types of errors on the same node to attempt to identify failure patterns. For example, do both SDI-12 and 1-Wire have collection error rates that increase at the same time, suggesting a possible clock rate issue? In this phase of our project, we produced a system, Corl8, which is able to analyze multiple nodes in combination with any number of diagnostic measures.

In the second phase of the project, we created a web portal which allows researchers who are less comfortable in a command line setting to adjust settings and view specific graphs on an interactive basis. This assists researchers in diagnosing errors by allowing them to quickly view a specific error plot and to change the time span under consideration with only a few clicks. Researchers planning to use the command line tool can use the web portal to help determine the settings they would like to use for the parameters and to familiarize themselves with the functionality of each setting.

1.2 Problem Statement

Programmers must be able to localize and correct errors in wireless sensor networks. Debugging tools come with a price of resource utilization that must be minimized. Simply transmitting diagnostic measures gives the programmer some knowledge of where the problem may be but provides little information on how they are related. Understanding the relationships among diagnostic measures allows programmers to localize problems more quickly. For example, a large number of TCP disconnect errors could indicate any number of problems with the initiating node's hardware or software, the target system's hardware or software, or external environmental factors, but knowing that the TCP disconnect errors occur in correlation with low signal strength reduces the problem to environmental issues.

1.3 Solution Approach

To improve the comprehension of root cause errors, we begin by designing a system which graphs "count" measures from any set of diagnostic data. The counts can be viewed as raw data (an increasing count which is reset to zero each time the node is restarted), counts per minute, counts per sample attempt (e.g., for sampling errors), and counts per transmission attempt (e.g., for wireless errors). The system supports any number of error categories and names, as appropriate to specific deployments.

To provide automated diagnostic assistance, we extend the system to use R to automatically correlate error rates based on a number of user configurable settings. Using the command-line batch interface, Corl8 generates and saves a graph set for each pair-wise error pattern that is statistically correlated. Each such graph set contains a graph of all the relevant data and a graph for each requested sub-division of the data, with the associated best fit based on linear regression. The developer may want access to a specific graph to test a theory or to further investigate one of the graphs generated in batch mode, so Corl8 also provides access via an interactive web interface which allows developers to interactively view the various graphs.

In order to allow Corl8 to be integrated with any sensor network, it reads the diagnostic data out of a specifically formatted MySQL table. A simple conversion script can feed in data from the target sensor network without having to modify Corl8. In our approach, we run a preprocessing step to move the data from MongoDB into MySQL using a Python script. During this step, we compute the per minute, per sensor attempt, and per transmission attempt data and place each diagnostic measure into its own row. Computing the various measures during preprocessing saves resources at the node level and when Corl8 processes the data.

1.4 Contributions

The contributions of this thesis are as follows: We present an integrated system which supports the analysis of diagnostic data produced by wireless sensor nodes. This system generates user-friendly graphs of diagnostic data and supports the basic mining for error rate correlations, resulting in graphs of highly correlated data likely to be causally related. The analysis allows users to exclude repeated data points (e.g., no errors on either diagnostic) to prevent inflation of the correlation statistic. The user may also request that the data be analyzed in any number of equal width segments to support the discovery of error rates that may only be correlated during certain periods. We apply this system to five deployed wireless sensor networks that are part of the Intelligent River® project. Finally, we evaluate the system in terms of its ability to identify errors in the code of the system.

1.5 Thesis Organization

Chapter 2 surveys some of the most closely related work in debugging wireless sensor network applications. Chapter 3 provides a brief overview of R, which we use to implement our system. Chapter 4 presents the design and implementation of our system. Chapter 5 discusses various use-case scenarios for the system. Chapter 6 surveys results identified by the system and evaluates its effectiveness. Finally, Chapter 7 presents a summary of our work and conclusions.

Chapter 2

Related Work

Paradis and Han borrow from the distributed systems community [37], defining fault isolation in wireless sensor networks as correlating "different types of fault indications (alarms) received from the network, and propose various fault hypotheses" [18]. We consider prior work focused on fault hypotheses (Section 2.1), pattern detection (Section 2.2), and tracing and visualization (Section 2.3).

2.1 Fault Hypotheses

2.1.1 Sympathy

Sympathy [23, 24] is designed to be deployed in tree-based, multi-hop topologies. It assigns each failure a localized source (root cause). This source indicates where the failure most likely originated and is where the developer should begin when debugging the problem. These sources can be *self*, *path*, or *sink*. Self denotes that the failure likely originated from the node itself (e.g., crash, reboot, local bug, or connectivity issue). Path denotes that the failure is likely a routing issue between the node and the sink; Sympathy will identify the node on the path that is likely causing the issue. Finally, sink indicates that the whole network appears to be failing.

Sympathy works by monitoring network traffic and extracting flow and node metrics from messages received at the sink. Flow metrics for transmitted and received packets are collected at each node, along with the node (or sink) the packets were transmitted to/from. Node metrics include uptime, and good/bad packet counts. Sympathy uses this information to compute its own diagnostic support data, such as the routing table and neighbor list. The transmit period of these metrics can be adjusted to control network traffic.

Sympathy introduces overhead less than or equal to 31% of data traffic. Ideally, we would like the overhead to be as small as possible, but we would also like to avoid modifying existing deployments. Our system's flexibility allows it to analyze any diagnostic measures that developers may already be periodically transmitting in their diagnostic process. Sympathy's purpose is to identify the node causing packet or data loss in the network. Our system, however, focuses on isolating failures to specific components of a sensor node by identifying the interactions of those components through correlation of diagnostic measures.

2.1.2 Fault Management in Event-Driven Wireless Sensor Networks

Ruiz et al. [28] describe a system for using MANNA [27] to detect faults in event-driven wireless sensor networks. MANNA is a management architecture that enables information collection from sensor nodes and provides a number of associated services, including coverage area maintenance and failure detection. Ruiz et al. use a simulated environment to decide whether individual nodes have failed. The nodes in the simulation are event-driven, meaning they only report when an event is detected. This means that the monitoring system must introduce communication overhead to detect nodes that are still alive but have not had any events to report. The diagnostic information is used to develop a visualization of node failures. Similar to our work, Ruiz et al. collect information across sensor nodes to assist in finding faults, but their system indicates only whether individual nodes have stopped responding. Our data includes information on a variety of sensor, network, and system diagnostic measures. MANNA itself is also limited, as it requires that users implement business rules over the collected data. Our system relies on the collected data to uncover interactions that may have been unexpected in order to help developers quickly debug and tune their systems.

2.1.3 Passive Diagnosis for Wireless Sensor Networks

Liu et al. [15] report a network diagnosis system, PAD, which uses a packet marking scheme to lower the network overhead associated with discovering the network topology. The packet marking scheme marks normal communication packets to capture how they traveled through the network, which reveals dependencies. Using a hierarchical inference model, PAD's inference engine enables reasoning about the root cause of faults, even with incomplete or suspicious inputs. In field tests, PAD consumes less than 10% of total network traffic and achieves a 90% failure detection rate, with approximately 80% accuracy in terms of correctly identifying the cause of failure.

While PAD may be useful in determining whether an individual node caused a failure, whether a group of nodes has gone down, or the system as a whole is suffering from excessive traffic reroutes, it is not well-suited for identifying application bugs or performance anomalies since it only collects data on network dependencies. Our system collects data across many different devices and software categories to allow for more extensive error analysis.

2.2 Pattern Detection

2.2.1 Lessons from a Sensor Network Expedition

Szewczyk et al. [36] evaluate a four-month-long deployment of a habitat monitoring system off the coast of Maine. After the deployment, they used the collected statistics to analyze packet loss, node level crashes, and power management. They were further able to use sensor data and mote clock skew (compared to the base station), combined with node failure information to analyze states that frequently led to mote crashes. In their tests, this detection method had very low false positive rates.

Their results shed light on how data can be used to predict failures, but since only one deployment was used, it is unknown how well their rules generalize. Our work seeks to expand upon this, such that data analysis can be applied in a simple fashion to any wireless sensor network to expose correlated error rates.

2.2.2 Dustminer

Khan et al. [11] detail Dustminer, a tool for identifying bugs associated with complex component interactions in networked sensing applications. These bugs are not localized to one faulty component, but occur due to unexpected interactions which may not be repeatable, making it difficult to reconstruct the anomalous situations and localize the bugs. Dustminer examines sequences of events across nodes to identify the root cause. Since events at devices that have not yet communicated are independent and could have happened in any order, the system uses non-determinism to help build sufficiently diverse behavior examples to train the system to recognize relevant correlations for “bad” system behavior. A front-end framework collects event logs; the back-end processes these logs using Apriori [1] to mine for frequent discriminative patterns using *minSup*[1] greater than 1. A second phase splits the logs into fixed-width segments. The number of discriminative patterns in a segment becomes the rank of that segment and consecutive segments containing discriminative patterns are combined into one larger segment. After this analysis, the K-highest ranking segments are analyzed for frequent patterns using *minSup*[1] of 1. This scheme helps to reduce the search space and makes it feasible to mine for longer, frequent patterns.

Dustminer can help identify event patterns leading to error states. The success of this method of course depends on logging the appropriate events. In their tests, the logs took 100-400 Kb of flash memory space and were not transmitted directly to the server. Unfortunately, they do not specify how long the tests took to run, which makes it difficult to determine the overall efficiency impact. In our study, we are able to reduce the amount of data transmitted by only transmitting a count of events, rather than sequence information. This improves efficiency, but comes at the cost of loss of causality; i.e., it is unknown which event came first. Our approach reduces the resources needed to transmit the diagnostic data and provides an integrated system for viewing and analyzing the results.

2.3 Tracing and Visualization

2.3.1 EmStar

EmStar [8] is a software environment for developing and deploying wireless sensor networks, composed of libraries that implement message-passing IPC primitives; tools for simulation, emulation, and visualization of real and simulated systems; and services that support networking, sensing, and time synchronization. EmStar favors ease of use and modularity over efficiency. Though the authors note that efficiency has not been a barrier in their current projects, it could be a problem in cases where efficiency is a top priority. The EmView system for network visualization must explicitly request updates from individual nodes, which could be a disadvantage in providing regular updates since it would require many data requests. While the approach is more efficient when the system is

[1]The minimum number of times the pattern must appear in the database to be considered frequent.

not being monitored, if the system crashes in such a state, there would be no data to help diagnose the cause of the failure.

The robustness of EmStar requires that it be run on a multi-process "microserver" style node [9]. It is not intended to be used on resource-constrained devices. This allows different processes to be isolated, and to tolerate process faults while continuing to operate (perhaps in a degraded state). EmStar also defaults to memory-based logs persisted to flash in the event of a critical failure or user request. This again assumes that the device has a relatively large amount of memory available for use.

The EmStar platform, while robust and modular, is not applicable to systems with limited on-board memory or processor resources. It relies on stored logs and concurrently running processes to be able to report failures after they have occurred. Our system focuses on providing enough information prior to failure or system degradation that the cause of failure can be inferred by the programmer.

2.3.2 Clairvoyant

Yang et al. [41] present Clairvoyant, a source-level debugger for wireless embedded networks. It allows developers to execute debugging commands, such as `break`, `step`, `watch`, and others specific to wireless sensor networks within a live environment. Other commands include `backtrace` to list all frames on the stack, `frame [n]` to select a frame, and `info frame` to detail a particular frame. The user selects a specific target and detaches when debugging is complete to allow the device to resume normal execution. The user can use network-level `stop`, `continue`, `detach`, and `reset` commands to issue the same command to all nodes.

While useful, basic Clairvoyant requires 875 bytes of data memory, and 31,360 bytes of program memory, which is nearly a quarter of the memory available on a Mica2 mote. Calling `break`, `watch`, `cond`, and `log` commands each add more memory requirements. The flash-based context switching required by `watch` and `step` could also quickly exhaust the number of flash memory writes possible on a mote. Our system requires no context switching or mote code changes, assuming the target system is already transmitting some amount of diagnostic data, which allows us to introduce only minimal overhead.

2.3.3 TOSSIM

TOSSIM is a system for accurate and scalable simulation of TinyOS-based wireless sensor networks [13]. TOSSIM achieves simplicity and efficiency by using a probabilistic bit error model, which enables detailed and scalable sensor network simulation. Included as part of TOSSIM is TinyViz, a graphical user interface for visualization, control, and analysis of TOSSIM simulations. TinyViz uses plugins to add functionality, including plugins for network traffic, mote state control, debugging, and power monitoring [29].

TOSSIM provides several other simulation control plugins, but these have been omitted in this discussion since they are not pertinent to our research. As a debugging platform, TOSSIM provides a wide range of tools in its graphical user interface (TinyViz). These tools allow visualization of data, much like the preliminary work for our project, but they are only applicable for simulated systems. TinyViz is not built to allow use in a live system. It also does not provide any details on relationships between errors, which would help programmers to more quickly localize the root cause of a fault.

2.3.4 Lightweight Tracing

Sundaram et al. [32, 33, 34] present a lightweight code tracing algorithm to assist in wireless sensor network debugging. Their system uses an algorithm by Ball and Larus [3] which represents the particular path taken through a procedure with an integer between zero and the total number of possible paths less one. Using this algorithm, each path taken within a procedure can be identified. In order to reduce the number of possible paths, loops are treated as separate procedures. For example, a program with two if-statements followed by a loop would be treated as two procedures, one for the if-statements and one for the loop. Representing the flow for an entire program is done using numbers that represent each procedure, and the numbers representing the paths taken through the procedures. In representing a system, procedures may be interrupted by the system, which means a number of procedures may start before any of them complete. As a result, the system records procedure start and end markers. In the case of multiple threads, each procedure is also labeled with a session identifier.

Sundaram et al. compress the trace records based on procedures that ran uninterrupted, repeated sequences of procedures, and repeated paths within a procedure (e.g., paths taken within loops). Using their trace system increases the code size by a variable amount, depending on the number of procedures present. In their examples, the overhead ranges between 5KB to 15KB, which could be a several hundred percent increase, or as low as a 1 or 2 percent increase, depending on the original program size. The compressed logs require 344b to 93200b for a 30-minute run. The size could be reduced by tracing only key functions.

The trace results must be viewed by a programmer to infer the root causes of faults. If further information is needed, the programmer may need to use a heavier-weight tracing tool that captures more data, such as the exact function inputs. The system presented by Sundaram et al. provides an efficient way to log the program actions of individual nodes but requires time-consuming evaluation. The goal of our approach is to reduce the diagnostic data to only what is relevant, and to present the correlated diagnostic results to the programmer in a way that is easy to interpret.

2.3.5 Minerva

Minerva [30] is a testbed architecture for distributed debugging of wireless sensor networks. Since it is a testbed architecture, it uses direct connections (serial or otherwise) between the motes and the servers, rather than wireless links. It also requires that a diagnostic board be installed at each node to provide access to the debug port of the sensor node's processor. The debug board is connected to a server that is independent of the server used to collect data from the nodes. The debugging server controls the motes' processors. This enables tracking of the internal state of individual nodes, whole-network suspension, and distributed assertion checking.

Since Minerva assumes a direct serial (or other) testbed connection from the debug board to the server, it is able to perform functions that cannot be performed in a live environment, such as transmitting more data with higher throughput, speed, and reliability. Our approach is meant to help in both test and live environments to provide information on relationships between diagnostic metrics at a scale appropriate to resource-constrained devices.

Chapter 3

Overview of R

Our diagnostic system is implemented using R, a powerful statistical programming language, and the relational database structure of MySQL. It is assumed that the reader is already familiar with SQL-style databases. We focus on providing an overview of the R programming language. In Section 3.1, we provide an introduction to R. Section 3.2 provides an overview of how the R console can be used to evaluate data quickly without needing to first create and then execute a file. Section 3.3 details wrting functions so users can reuse code. Finally, Section 3.4 briefly discusses using packages in R, including the packages used in our system.

3.1 Introduction and History

R is a language and programming environment for statistical computation and data representation through plots, graphs, and figures. It is similar to the S language originally developed at Bell Laboratories by Chambers et al. [19]. Most code written for S can be run unmodified in R, though R contains several important differences.

R provides an open source route for people to access statistical methods, and it is highly extensible. R is available as free software under the GNU General Public License. It compiles on UNIX, Windows, and MacOS [19].

The language of R is very similar to that of Python. R is dynamically typed, and code can be compartmentalized using functions and packages. Packages may contain namespaces, but functions within a package intended for end-users are exported, meaning they can be called without

```
> ## Load the data set
> data(mtcars)
> ## Get the first few rows
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
> ## Get details on the mpg data
> summary(mtcars$mpg)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10.40   15.42   19.20   20.09   22.80   33.90
> ## Load the graphics library
> require(graphics)
> ## Plot all the points
> pairs(mtcars, main = "MTCars Data")
> ## Plot MPG vs Displacement using number of cylinders as a factor
> coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
+        panel = panel.smooth, rows = 1)
```

Listing 3.1: R Console Programming

using the namespace.

3.2 Console Programming

Using the R console, short code fragments can be run to test ideas or perform one-off data analyses. In Listing 3.1, we show simple commands run on the Motor Trend Car Road Tests (mtcars) data set, which comes preloaded in the data sets package for R. The mtcars data set was extracted from the 1974 Motor Trend US magazine and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models) [22]. The graphs generated would normally display in a separate viewer on screen, but for the sake of presentation, we have included them as Figures 3.1 and 3.2. Figure 3.1 shows scatter plots of all 11 aspects of the cars plotted against each other. The aspects are listed along the diagonal. Following an aspect along its row shows the aspect on the y-axis plotted against each other aspect on the x-axis. Similarly, following an aspect along its column shows a scatterplot where that aspect is plotted on the x-axis and each other aspect is plotted on the y-axis. Figure 3.2 uses the number of cylinders in the car to separate the data and plots the displacement versus the miles per gallon for cars with four, six, and eight cylinders.

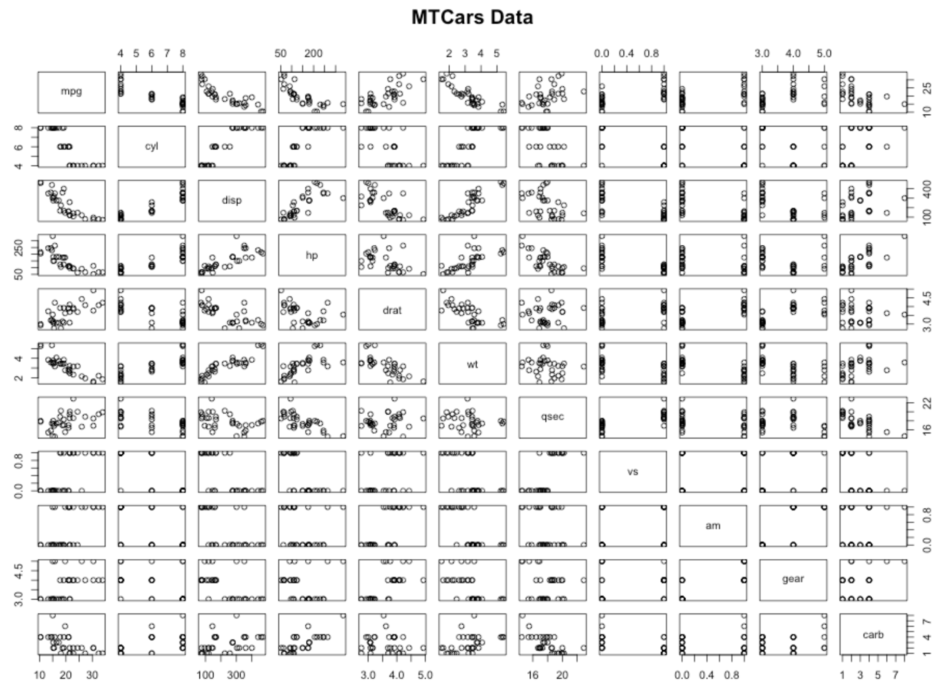


Figure 3.1: MTCars Data Plotted with All Pairs

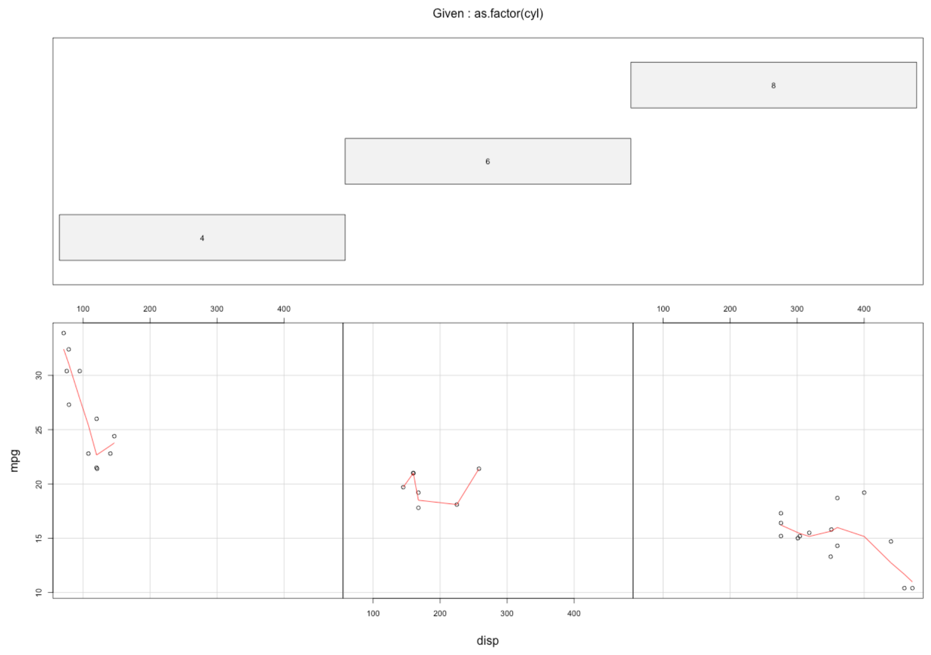


Figure 3.2: MPG vs Displacement using Number of Cylinders as a Factor

```r
power <- function(number, raiseToPower=2) {
  temp <- number
  while (raiseToPower > 1) {
    raiseToPower <- raiseToPower-1
    temp <- temp * number
  }
  temp ## This is the return value
}
```

Listing 3.2: R User-Defined Function, `power.R`

```
> ## Import the function file
> source("power.R")
> ## Run with the default raiseToPower of 2
> power(3)
[1] 9
> ## Run with specified raiseToPower
> power(3,3)
[1] 27
> ## Run with out-of-order named parameters
> power(raiseToPower=4, number=3)
[1] 81
> ##Run with out-of-order partially named parameters
> power(r=4, 3)
[1] 81
> ## Run without the number, error
> power(raiseToPower=4)
Error in power(raiseToPower = 4) : argument "number" is missing, with no default
```

Listing 3.3: Using the Power Function in the Console

3.3 Writing Functions

Operations that are often run in series are easily converted to user-defined functions. In R, functions are assigned to a variable, which becomes the function name. Functions can be called using in-order parameter passing, out-of-order parameter passing (with parameter names), and default parameter values. The return value of a function is simply the result of the last line of code executed in the function. In Listing 3.2, we show a simple R function for raising a number to a specified power.[1] We have saved this function in a file called `power.R`. In Listing 3.3, we demonstrate using the power function with various parameter passing methods.

3.4 Packages

At the time of writing, R had 5,322 packages available in the CRAN repository [7]. Most downloads of R include a number of the basic packages. Additional packages can be downloaded

[1]R provides the ^operator for raising numbers to powers. This function is for demonstration purposes only.

and installed at any time. RStudio [25], used to develop our system, comes with a GUI for installing and loading packages. The `install.packages` command can also be used at the console to add packages. To load a package, the user employs either the `require` or `library` command. The `require` command will return false on failure and throw a warning, whereas `library` will throw an error if it fails.

We make use of ten of the available packages in our project. RMySQL [10] and DBI [17] facilitate interfacing with our MySQL database. ggplot2 [40], graphics [20], grDevices [20], grid [20], and gridExtra [2] are used for graphing. Finally, methods, stats, and utils, all of which are part of the R-core [20], provide the rest of the statistical and support functions.

Chapter 4

Framework Design

In this chapter, we introduce the design and implementation of the Corl8 analysis system. The system relies on data from a MySQL table and allows users to choose settings that affect how the system analyzes and classifies the data. The "interesting" data will automatically be output in the form of an image showing a set of scatterplots that depict how the two diagnostic measures in question vary. The system can be used to analyze a single node or all the nodes in a wireless sensor network. First, we present an overview of the system in Section 4.1, then describe each of the individual components.

4.1 Overview

An overview of the flow of diagnostic data through the system is show in Figure 4.1. Data originates at the *Sensor Nodes* and flows into the *Sensor Diagnostic Log Database (MongoDB)*. Next, a *Python Preprocessing Script* parses the data into per minute, per sensor attempt, and per transmission attempt metrics. As the data is processed, the Python script inserts it into the *MySQL Database.* The data may also be preprocessed to reduce the number of points or to round the measures, allowing the system to flag more points as duplicates.

The central component of our framework is the *R Analyzer Function*, which reads information from the *MySQL Database* and accepts *User-specified Parameters.* The function parameters tailor various aspects of the data analysis and how the resulting graph sets should be output. The output for each unique combination of (i) node, (ii) independent diagnostic measure, and (iii) de-

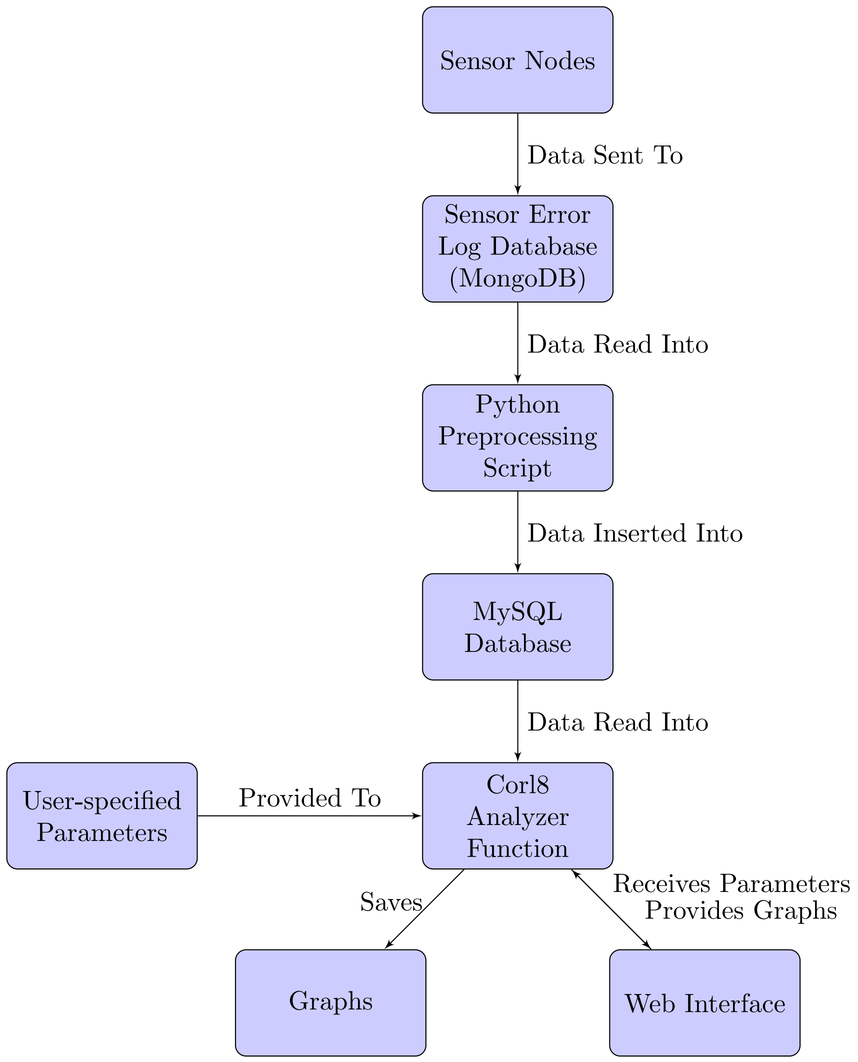


Figure 4.1: Diagnostic Data Flow

pendent diagnostic measure is a graph set. The graph set contains one or more graphs. The first graph represents the entire data set. If the parameters are set such that the data is also analyzed based on divisions (equal width segments), a graph will also be created for each division. The graph set is then saved as a single image. The final result of the analyzer function is a collection of graph sets.

We discuss the details of each parameter in Section 4.3. Here we briefly consider some of the most basic parameters. A specific node may be specified, or all nodes may be selected. The specific diagnostic measures to be analyzed may also be specified. Database connection information must be provided so that the analyzer can fetch the information from the MySQL database. Data may also be rounded to a specified number of decimal places, and points which are duplicated may be removed. A minimum number of points is specified for the system to proceed with data analysis. During data analysis, the system tests whether or not the points are correlated by using a linear regression test and comparing the R-squared test statistic with a user-specified minimum value for the data to be considered correlated. The parameters also specify if the data should be analyzed across a number of divisions (equal width segments) which helps to identify trends that exist only in segments of the collected data. Finally, the parameters specify where to save the output and how to format the graph set images.

The *R Analyzer Function* can also accept input from the *Web Interface*. In this case, the graph set resulting from the analyzer is sent back to the web interface for the user to view. The web interface is also written in R and is powered by Shiny Server [26]. The interface supports many of the same parameters, but the inputs are specified in an interactive manner.

4.2 MySQL Database

The MySQL database may contain diagnostic measures from any number of nodes and in any number of categories. The database must be formatted according to our specifications. To analyze an entire system, all diagnostic information must be placed into one table. To analyze one device at a time, the data may be split up to avoid creating large tables. When running the analyzer, the user specifies the username, password, and host for the database connection, as well as, the database and table names.

The table structure is shown in Table 4.1. The table contains eight columns. First, the

Column Name	***Column Type***	***Key***
id	int(11)	Primary Key
diagnostic	varchar(100)	Key
pmin	decimal(18,9)	
pradio	decimal(18,9)	
psample	decimal(18,9)	
raw	decimal(18,9)	
time	datetime	Key
device	varchar(100)	Key

Table 4.1: MySQL Table Structure

`id` column is the primary key for the database. The `diagnostic` column captures the name of a diagnostic measure. In our sensor system, we use names such as *appDiagnostics-sampleAttempts* and *gm862Diagnostics-wakeErrors*. The names may include spaces but should not include single quotes. The `device` column records the name of the device being observed. It is important to use a device name that is human readable since this will be included in the graphs. Also, no two devices should have the same name, or the system will combine their data. We use names such as *srb_1* for Savannah River Basin, device 1.

The data columns (`pmin`, `pradio`, `psample`, `raw`) contain the values of the diagnostic measures in several formats: errors per minute, errors per radio attempt, errors per sample attempt, and the raw count reported in the diagnostic message. The number of data columns may vary among applications. For example, a user may wish to create a data column containing errors per event detected. The data column to use for analysis is configurable in the user-specified parameters and via the web interface. We recommend that these be decimal columns with the appropriate precision to reduce the likelihood of rounding errors. In some cases, it may not make sense to use per minute, transmission, or sample error rates; the raw value may be more appropriate. For example, signal strength and battery voltage will only make sense when interpreted using the raw values. The `time` column records the data timestamp.

Diagnostic measures are reported from the sensor nodes at a specified period. In our deployment, devices record observation data every 5 minutes, but only record diagnostic data every 15 minutes. Thus, our interval is generally 15 minutes. However, device reboots, clock skew, and communication errors result in reports that do not come in with data that is precisely 15 minutes apart. As a result, when reporting data to MySQL, the time difference between this report and the previous report is used when computing per minute measures.

4.2.1 Populating MySQL

Our sensor nodes send cumulative diagnostics since the last reboot. These counts are then stored in a Mongo database [16]. Because the counts are set to zero after each reboot and then increase monotonically, the system must detect reboots. It must also exclude counts from prior reports before dividing the diagnostic measure in the current report to computer counts per minute, per sensor reading attempt, and per transmission attempt.

The sensor system sends back `cumulativeUp`, `sampleAttempts`, and `radioAttempts` counters for the node. The `cumulativeUp` count represents the number of seconds the node has been running. The `sampleAttempts` count is the total number of sample attempts made by the node since the last reboot. Similarly, the `radioAttempts` count is the number of transmission attempts made by the node since the last reboot. We use the `cumulativeUp` counter and the message timestamps to detect reboots. If the device has not rebooted, the difference in the `cumulativeUp` counter should match the number of seconds between the timestamps. (The system allows a 10 second grace period in case clocks have skewed.) The system cannot use only up-time since a node might run for 1000 seconds, report, reboot, and then run for 1100 seconds before its next report. The `cumulativeUp` counter is still important because the node might be down for a period of time before the reboot, and the system should consider only up-time when calculating per minute statistics.

When processing data, the system processes all the diagnostic reports from each device separately, in chronological order. This allows the system to track the details from the previous report to detect reboots and properly adjust the counts to compute the per minute, per sensor attempt, and per transmission attempt statistics. Once the statistics have been obtained, the system inserts one row for each diagnostic reported by the node. Thus, every row inserted for one report corresponds to the same device and time but contains statistics for a different diagnostic measure. After one report has been fully processed, the system proceeds to the next report from that device. After all reports for a device have been processed, the system advances to the next device. When the processing is complete, the database contains all the diagnostic data reported by the sensor network.

For reference, we include the Python preprocessing code that allows us to import data from our MongoDB diagnostic log database into the MySQL database used by Corl8 in Appendix D. It is expected that users of Corl8 will need to create custom tailored preprocessing code. The included code allows us to, at any time, retrieve the entire Mongo database and process it into the MySQL database. The code includes several lines which require user customizations: the MySQL connection details, the MongoDB database and collection, and the diagnostic measures to import.

4.3 R Analyzer Function

The R analyzer function is responsible for determining which data to display to the user for further analysis. The analyzer requires R [20] be installed, with the gridExtra [2], RMySQL [10], and ggplot2 [40] libraries (and their dependencies). The analyzer function is controlled by the parameters provided by the user and acts on the data in the MySQL table. A change in the parameters can make a significant difference in the false positive/negative rate and volume of data output for further analysis. As a result, it is important to understand what each of the parameters controls. An overview of the parameters is given in Table 4.2, but we discuss each in detail in later subsections. (The full code for the Analyzer appears in Appendix A.) Before discussing the parameters in detail, we provide an overview of the functions operation to give context for the parameters.

4.3.1 Behavior Overview

An example graph set is shown in Figure 4.2. The associated data set is for node 1 deployed at the Baruch Institute in Georgetown, SC and compares the number of TCP disconnect errors with the number of escape errors reported by the cellular board. The data set was analyzed using a minimum R-Squared value of 0.75 with 4 divisions. The upper left graph shows the complete data set, which has an R-Squared value of 0.798; this meets the requirement that it be at least 0.75 to be considered correlated. The next graph (upper right) shows the first division is not correlated. The following graph (middle left) is correlated and causes the system to output this graph set for further analysis. The next two divisions (middle right and lower left) are not shown because the data falling in those divisions did not meet the division threshold requirement. Looking at the graph of the complete data set, it is easy to see that only a few points are present in the top half[1] so these divisions would not meet the division threshold of 5. This correlation shows a problem in the system which is further discussed in Section 6.1.

[1]There is only one distinct point in the upper half, but it is darker which indicates that there are multiple equal points.

Parameter	***Instructions***
device	The device to consider. If “all”, all devices will be pulled from the database.
dbName	The database name.
dbUser	The username for the database.
dbPassword	The password for the database.
dbHost	The host for the database.
dbTable	The database table.
diagnostics	The vector of diagnostics to analyze or NULL to use all available diagnostics from the database.
diagColX	The column to use for the X diagnostic.
diagColY	The column to use for the Y diagnostic.
threshold	The minimum number of points required to run the correlation test.
round	The number of decimal places to allow or NULL to use data as is.
maxDuplicates	The maximum number of duplicates of a single point allowed in the test.
minRS	The minimum r-squared value from the correlation test required to output a graph.
minDate	The start date in the form (YYYY-MM-DD HH:MM:SS).
maxDate	The end date in the form (YYYY-MM-DD HH:MM:SS).
requireSubCor	Require that a division be correlated in order to output the graph set.
divisions	Number of segments to divide the data into.
divisionThreshold	The minimum number of points in a division to run the correlation test.
folder	The folder to use when saving the output graphs.
graphWidth	The graph width (per graph, image will be enlarged for each graph).
graphHeight	The graph height (per graph, image will be enlarged for each graph).
nCol	The number of columns in the graph image.
returnGraph	The first graph set will be returned, and the function will stop. No graphs will be saved. Return value is a list (graphlist, width, height).
noDeviceError	The error message given if the device parameter is blank (for use with the Shiny server app).
placeholderTitle	The title for placeholder graphs. (Placeholder graphs are generated when a division does not meet the requirements to be analyzed.)

Table 4.2: Analyzer Parameters

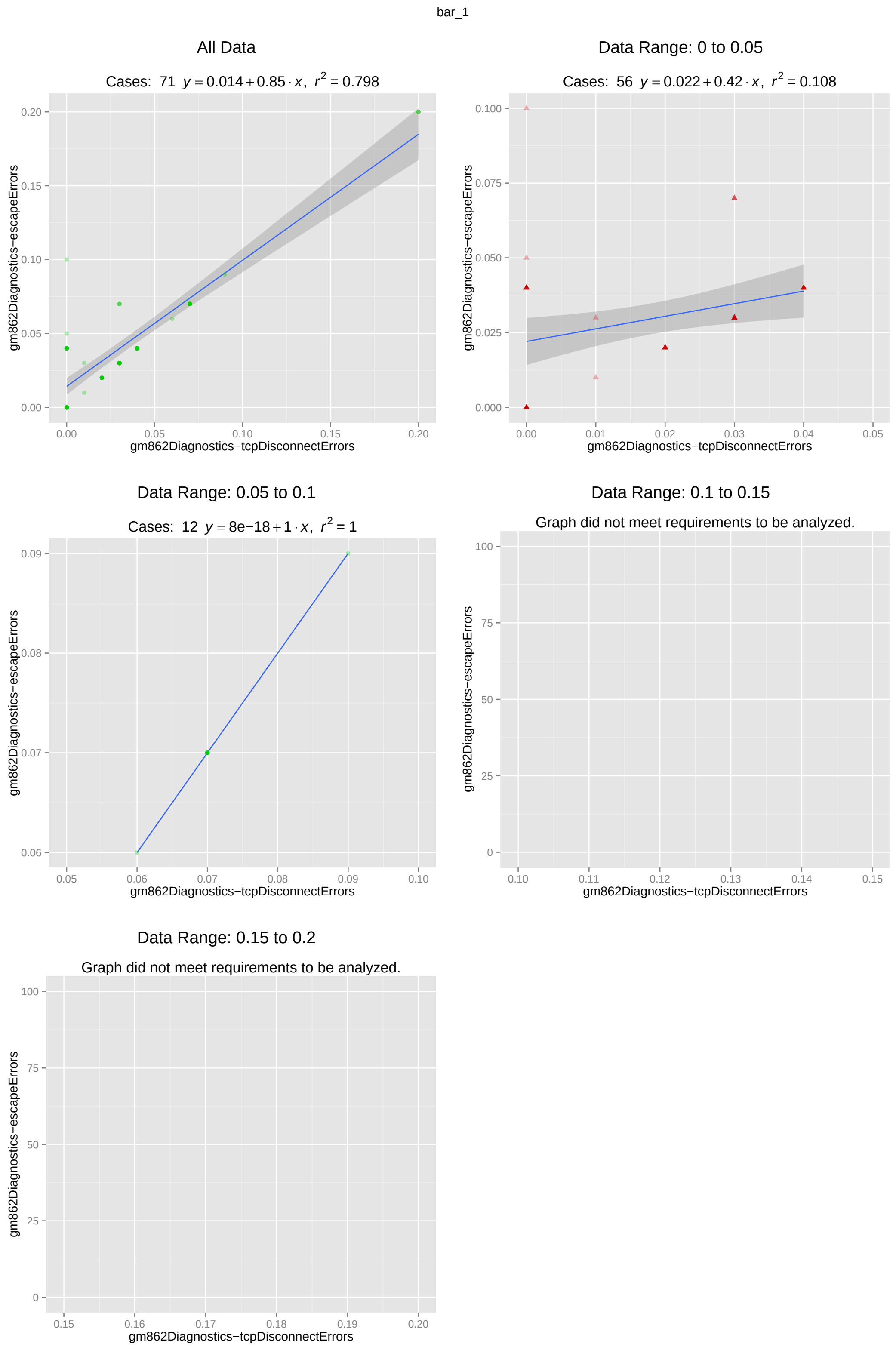


Figure 4.2: Baruch REC Node 1: TCP Disconnect Errors Versus Escape Errors

```
for (id in devices) {
  for (i = 1; i <= length(diagnostics); i++) { ## Arrays in R are 1 based
    ## Fetch data for i
    for (j = 1; j <= length(diagnostics); j++) {
      if (i == j)  { next; } ## Don't analyze a diagnostic against itself
      ## Fetch data for j
      ## Run analysis
    }
}
```

Listing 4.1: Analyzer Function Pseudocode

The analyzer function first connects to the database and fetches the distinct devices and associated diagnostics that are available. If a specific device has been selected, it will perform analysis for that device only rather than using all the available devices.

Next, the function loops over all the fetched devices. For each diagnostic, it fetches the available data for that device. Next, it pairs this diagnostic with every other diagnostic for analysis. The pseudocode for this is in Listing 4.1. We consider both (i, j) and (j, i) because divisions of data are analyzed only along the x-axis. If i and j are only correlated for high values of j, this would be found when analyzing (j, i) but not when analyzing (i, j).

The analysis involves several steps. First, the system selects the reported data measures for the two diagnostic measures being analyzed and merges the data sets based on matching timestamps. Specifically, this means that these two diagnostic measures were reported at the same time. Next, it removes any incomplete cases where data is only present for one of the two diagnostics. For example, if a new diagnostic measure was added to the system, it would not be present in older diagnostic reports. If the system was comparing the pre-existing diagnostic to the new diagnostic, it would need to discard all the data for the pre-existing diagnostic recorded prior to the implementation of the new diagnostic. Next, the system tests that the number of complete cases meets the user-specified threshold for analysis. If it does, the system continues.

The user may request that points which have been duplicated more than a certain number of times be reduced. For example, if the limit was set to 10 and the point (0.5,0.25) occurred 18 times in the data, the system would remove 8 of the points, leaving 10. Points that have been duplicated a large number of times can cause the R-squared result to appear higher than would be expected upon visual inspection of the data. For example, two diagnostic statistics that are frequently both 0 would have a large number of points at (0,0). This causes the model to have a high R-squared value because it is very good at predicting the dependent value (0) when the independent value is 0.

Unfortunately, the model may not be a good predictor for data that does not occur at (0,0), which is what is important. Removing all but a small number of these points results in an R-squared value which more accurately represents the predictive power of the model. The system does not remove all of the points because it is still important that the model be better at predicting repeated points than ones which only occur sporadically. Thus, this setting allow the user to balance the importance of predicting repeated points and the predictive power for the distinct points. Next, the system checks that the filtering process has not caused the number of remaining data points to drop below the threshold for analysis.

After passing all the checks, the data set is analyzed and a graph is generated and stored temporarily. The analysis approach is described in Section 4.3.2. If the user specifies more than one division to be analyzed, the data is divided into even segments between min and max, with each division being `(max-min)/numberOfDivisions` wide (equal width segments). Each division that meets the specified threshold is analyzed and graphed. Divisions not meeting the division threshold are not graphed; however, an empty placeholder graph is created so it is easy to identify that the division did not meet the threshold. Finally, if at least one correlated set of data was detected, all the temporary graphs are saved as a single image depicting the graph set.

4.3.2 Analysis Methodology

The statistical analysis is a standard linear regression test over the complete set of data, followed by repeated linear regression over each of the divided segments, if any. The R-Squared value is compared with the user-specified value to determine if the graph should be labeled as correlated. Whether or not to output the final graph set is determined by user settings which specify whether any one graph needs to be correlated or if the correlated graph must be from one of the divisions.

It may be of interest to the user to try different correlation tests or different methods of dividing the data. Each of these tests are readily integrated within the R code. Adjusting the correlation test would require altering the two lines of code which currently run the test and save the test statistic. The graph output will also need to be adjusted to show the new fit line and to display the correct formula for this line. When adjusting these, it is important to adjust them both for the full data set analysis and the analysis of the divided data sets as these are analyzed separately in the code. Adjusting the division method requires adjusting the bin numbers assigned to each data point. These are assigned in one line of code. Adjusting the graph label also requires changes to only one line of code to reflect the new division system. A few other lines of code will also need to be modified to provide the necessary support code. While changing the tests does require altering the R code, it allows extensions of the Corl8 project.

4.3.3 Device Parameter

The `device` parameter is used to limit the amount of data analyzed by Corl8. The default value of “all” results in all devices being analyzed. The alternative considered is to pass the name of a single device, in which case only that one device will be processed.

4.3.4 Database Parameters

The database parameters are used to specify the database host (`dbHost`), the database name (`dbName`), the database username (`dbUser`), the database password (`dbPassword`), and the table name (`dbTable`). A user may wish to use different tables to limit the amount of data contained in each table. For example, the user may wish to have a table for each deployment such that the system could be instructed to run on all the nodes in a particular deployment. Different tables may also be used to test different methods of data consolidation.

4.3.5 Diagnostic Parameters

The diagnostic parameters are used to control the diagnostic measures to be analyzed and the data columns to be used in analyzing them. The `diagnostics` parameter can be a vector of diagnostic categories to be analyzed, or `NULL`, resulting in all the available categories being analyzed. In either case, the system analyzes each diagnostic measure as the independent variable in comparison with every other measure as the dependent variable. Since there must be independent and dependent variables, the minimum length of the vector is two.

The `diagColX` and `diagColY` parameters determine the data columns from which diagnostic measures should be read. `diagColX` is used for the independent variable, and `diagColY` is used for the independent variable.

4.3.6 Filter Parameters

Several parameters control data filtering. The first is the `threshold` parameter, which specifies the number of complete cases that must be present in the data for any further tests to be run. (A *complete case* occurs when a node reports both diagnostic measures for the same time period.) The threshold is applied both to the incoming data and to the filtered data.

The `round` parameter is used to round data points to a specified number of decimal places. If set to `NULL`, the data will be used exactly as it appears in the MySQL table. Positive numbers set the number of decimal places. Negative values are also allowed, resulting in rounding to powers of ten. For example, a value of -1 would round 6 to 10. When rounding at 5, R rounds to the even digit. For example, when rounding to two decimal places, 3.225 rounds to 3.22 and 3.235 rounds to 3.24. This functionality is OS-dependent and subject to representation error. For example, the system might represent 3.235 as 3.23499999999998, which would cause it to round down to 3.23. For more details, see the R documentation on the round function [21].

The `maxDuplicates` parameter is used to remove data points that occur more than a parameter-specified number of times in the data set. This parameter is primarily used to ensure that the R-squared value is not significantly increased by a concentration of points at a certain value. Diagnostic measures are paired based on the timestamp of the report, meaning that the measures were reported at the same time. Commonly, points become concentrated at (0,0), meaning both measures reported no activity. Using a value of 10 allows the R-squared value to benefit from the concentrated points, but false correlations are reduced. It is important that all data is rounded for this parameter to work well, otherwise values such 0.9985 and 0.9983 will not be culled. The appropriate number of decimal places to round to should be based on the significance of those decimal places in data. A value which is frequently between 0 and 1 may need 3 decimal places where a value between 1 and 100 may need only 1 to be well-represented.

Next, the `minRS` parameter specifies the minimum R-squared value for data to be considered correlated. Data that is considered correlated is graphed using green circular points, while data that is considered un-correlated is graphed using red triangular points. There must also be at least one correlated graph for the analyzer to save the graph files for user inspection. Thus, a higher value will reduce the number of graphs the user must inspect but may not output an important graph, whereas a lower value increases the number of output graphs which must be inspected.

The `minDate` and `maxDate` parameters allow the data points to be filtered by when they were collected. This allows users to analyze data from a specific time period. These parameters must be strings in the form of YYYY-MM-DD HH:MM:SS. If values are not passed, or are set to `NULL`, the system will not filter on that date range. This allows, for example, year-to-date analysis by setting `minDate = 2014-01-01 00:00:00` and `maxDate = NULL`.

4.3.7 Division Parameters

The `divisions` parameter is used to specify the number of divisions applied during data analysis. Divisions are established by creating even width segments between the minimum and maximum data values, where each segment is `(maximum-minimum)/numberOfDivisions` wide. The `divisionThreshold` parameter allows users to specify a lower threshold for divisions since they only contain a portion of the original data set. For our work, we used an overall threshold of 25, with 4 divisions, and a division threshold of 5.

4.3.8 RequireSubCor Parameter

The `requireSubCor` parameter forces one of the data set divisions to be correlated in order for the system to save the graph set. If this parameter is *true* and only the full data set is correlated, the graph set will not be saved. If the parameter is *false*, a graph would be saved. In our tests, several graph sets demonstrated that when the complete data set passed the requirements to be considered correlated, but none of the divisions passed the requirements, the graphs did not actually demonstrate a conclusive trend.

4.3.9 Graph Parameters

The graph parameters specify how the graphs will be saved. The `folder` parameter specifies the save path. The `graphWidth` and `graphHeight` parameters specify the width and height of each graph in pixels, respectively. The `nCol` parameter specifies the number of columns that should be used when plotting multiple graphs. As the number of divisions is increased, the number of columns should also be increased to prevent an overly tall image. A graph set image which is 2 graphs wide and 3 graphs high would be a total size `graphWidth*2` by `graphHeight*3`.

4.3.10 Web Interface Parameters

The `returnGraph` parameter is used to avoid creating temporary files when the system returns graph sets to the web interface. When using the command-line interface, this parameter should be set to the default value of *false*.

The `noDeviceError` parameter is the error string displayed when no device is specified. R Shiny Server tries to fetch a graph immediately when the user loads the web application. As a result, this error is always shown in the location for a graph when the interface is first loaded. Thus, the web interface uses this string to inform the user to select the node and diagnostics they want to analyze, and to then click "Update Graph." When using the command-line interface, the default value should be used.

4.3.11 Placeholder Graph

The `placeholderTitle` parameter is used as the title for placeholder graphs. Placeholder graphs are produced when a data set division does not meet the minimum qualifications for analysis. For the web interface, a placeholder graph is also be produced when the complete data set does not meet the minimum qualifications for analysis.

4.4 Graphs

The graphs produced by our system always place the graph of the full data set in the upper-left corner. Division graphs are the plotted across and then down. (In a two column image with three divisions, the upper-left would be the full data set, the upper-right would be the division starting with the minimum data point, the lower-left would contain the graph of the middle division, and the lower-right would be the last division.)

Points are plotted with 30% opacity; duplicated points show darker than points occurring only one time. The least square regression line is shown on the graphs, along with a shaded 95% confidence area. Graphs with green, circular points met the minimum R-Squared value to be considered correlated. Graphs with red, triangular points do not meet the minimum value.

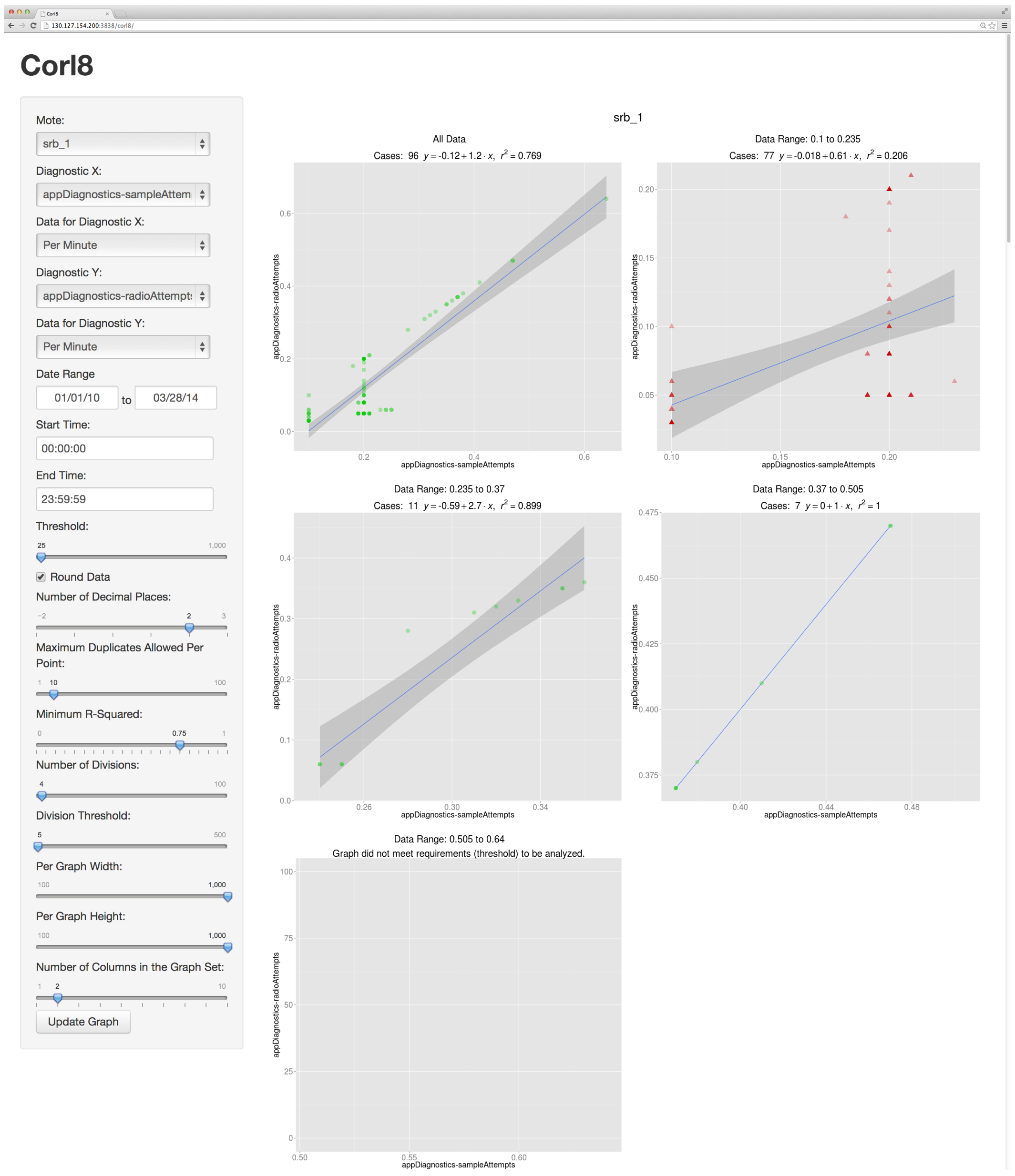
Graphs are titled with the appropriate device ID, and axes are labeled with the diagnostic measure. Each graph is titled with the division boundaries (if any) and the formula for the regression line. When graph files are saved, the files are named based on the node ID and the diagnostic measures in the graph. So for example, a graph saved as `bar_1_appDiagnostics-radioAttempts_gm862Diagnostics-cumulativeUp.png` is a graph for the bar_1 node[2] and graphs the `appDiagnostics-radioAttempts` diagnostic measure versus the `gm862Diagnostics-cumulativeUp` diagnostic measure. (When saving graphs, all non-alphanumeric characters except for dash and underscore are removed from the node ID and diagnostic measure name.)

4.5 Web Interface

The web interface is a key part of our system because it allows users to interactively view specific graph sets, often based on suspected correlations. Users can specify function parameters and select the node and diagnostics to analyze. Specifying the parameters in the web interface allows developers to interactively determine the best parameters to use for analysis. After inspecting graphs generated by Corl8's batch analysis mode, users may wish to reanalyze some of the data to help narrow the cause of unexpected diagnostic correlations. This can be done using different date ranges or comparing other diagnostic categories that may be involved in the trend. The web interface is designed for simplicity; it facilitates use of the system by field technicians who have no programming skills. Also, because it generates one graph set at a time, it allows the user to quickly pinpoint potential anomalies.

The web interface allows users to interactively set many of the parameters for Corl8's analysis function. (Some parameters are not available, including the database connection information, error messages, and the `returnGraph` parameter.) The interface is shown in Figure 4.3. When it is first opened, the area where the graph set would normally be displayed contains a message directing the user to select the desired settings and click the "Update Graph" button at the bottom of the sidebar. Once processing is complete, the graph set is updated and ready to be viewed. Depending on the size of the data set, it may take several seconds for the graph set to be updated, but in our tests, the time was always reasonable, even for very large data sets. For details on each of the fields, we refer readers to the appropriate function parameter, as described in Section 4.3. The *node* field corresponds to the `node` parameter. The *diagnostic x* and *diagnostic y* fields are placed in a vector

[2]In our deployment, this is node 1 at Baruch REC.


130.127.154.200:3838/corl8/
srb_1
All Data
Cases: 96 y = -0.12 + 1.2 · x, r² = 0.769
Data Range: 0.1 to 0.235
Cases: 77 y = -0.018 + 0.61 · x, r² = 0.206
Data Range: 0.235 to 0.37
Cases: 11 y = -0.59 + 2.7 · x, r² = 0.899
Data Range: 0.37 to 0.505
Cases: 7 y = 0 + 1 · x, r² = 1
Data Range: 0.505 to 0.64
Graph did not meet requirements (threshold) to be analyzed.
appDiagnostics-sampleAttempts
appDiagnostics-radioAttempts
Update Graph

Corl8

Mote: srb_1

Diagnostic X: appDiagnostics-sampleAttem

Data for Diagnostic X: Per Minute

Diagnostic Y: appDiagnostics-radioAttempts

Data for Diagnostic Y: Per Minute

Date Range: 01/01/10 to 03/28/14

Start Time: 00:00:00

End Time: 23:59:59

Threshold: 25 (25 – 1,000)

☑ Round Data

Number of Decimal Places: 2 (-2 – 3)

Maximum Duplicates Allowed Per Point: 10 (1 – 100)

Minimum R-Squared: 0.75 (0 – 1)

Number of Divisions: 4 (4 – 100)

Division Threshold: 5 (5 – 500)

Per Graph Width: 1,000 (100 – 1,000)

Per Graph Height: 1,000 (100 – 1,000)

Number of Columns in the Graph Set: 2 (1 – 10)

Figure 4.3: Corl8 Web User Interface

and correspond to the diagnostics parameter. *Data for diagnostic x* becomes diagColX. *Data for diagnostic y* becomes diagColY. The *date range*, *start time*, and *end time* become minDate (beginning of the date range and the start time) and maxDate (end of the date range and end time). The *threshold* field becomes threshold. If the *round data* checkbox is unchecked, NULL will be passed as the round parameter. Otherwise, the selected *number of decimal places* will be passed as the round parameter. The selected number of *maximum duplicates allowed per point* corresponds to the maxDuplicates parameter. The *minimum R-squared* corresponds to the minRS parameter. The selected *number of divisions* corresponds to the divisions parameter. The selected *division threshold* corresponds to the divisionThreshold parameter. The selected *per graph width* corresponds to the graphWidth parameter. The selected *per graph height* corresponds to the graphHeight parameter. Finally, the selected *number of columns in the graph set* corresponds to the nCol parameter.

4.5.1 User Interface Implementation and Configuration

The web interface is powered by Shiny Server [26]. We use the free, open source edition of the server, but if needed, the enterprise edition could also be used. Shiny Server provides a framework for developing web applications. The framework ensures that updates are optimized to occur infrequently through the use of caching. This means that if the user changes only one setting, only the values depending on that setting will be recomputed. It also implements many of the controls needed for collecting user input and passes the associated selections to the server script for processing.

For our system, we modified a few style details provided by Shiny Server, but otherwise the framework provided all the required user interface controls. This allowed the file which creates the user interface to be short and easy to understand. The details are given in Appendix B.

The Corl8 server-side system component automatically loads the available nodes and diagnostic measures from the database. The system also updates the available diagnostic measures each time the user selects a new node. This ensures that the options presented to the user are valid, even if different nodes collect different diagnostic measures. The person deploying Corl8 should configure the dataOptions variable, which is on line 2 of the user interface definition file. This variable contains a list of the data columns available in the MySQL table. In the list, a description is given of the data, followed by the column name. An example configuration is shown in Listing 4.2. In this

```
dataOptions <- list("Per Minute" = "pmin",
                    "Per Radio Attempt" = "pradio",
                    "Per Sensor Attempt" = "psample",
                    "Raw Data" = "raw")
```

Listing 4.2: Example Configuration for the `dataOptions` Variable

```
1 # Configuration Variables
2 dbName <- "research"
3 dbUser <- "research"
4 dbPassword <- "corl8"
5 dbHost <- "localhost"
6 dbTable <- "rates"
7 noDeviceError <- "Select the variables at right and click \"Update Graph.\""
8 placeholderTitle <- "Graph did not meet requirements (threshold) to be analyzed."
```

Listing 4.3: Example Server Configuration Variables

configuration, four data columns are available. The first column contains per minute data, and the column name is `pmin`. The second column contains per radio attempt data and is named `pradio`. The remaining two columns represent per sensor attempt and raw data and are named `psample` and `raw`, respectively.

4.5.2 Server Implementation and Configuration

The server-side implementation is also easy to understand. The person deploying Corl8 must configure the server file. An example configuration is shown in Listing 4.3. Lines 2-6 of the file configure the MySQL connection details. Lines 7 and 8 configure the `noDeviceError` and `placeholderTitle` variables, but these two variables do not have to be modified. All of the configuration variables are passed as parameters to the analyzer script; the details are described in Section 4.3. The rest of the file implements the necessary code to retrieve and display the available nodes, to update the available diagnostic measures when the user selects a different node, and to run analyses and display the resulting graph sets. The complete implementation of the server is given in Appendix C.

Chapter 5

Use-Case Scenarios

In this chapter, we present two use-cases to demonstrate the utility of Corl8. The first illustrates the use of batch-based analysis to discover possible errors. The second example illustrates Corl8's ability to support the investigation of potential flaws using the interactive web interface.

5.1 Batch Analysis

The batch analysis mode allows users to analyze all of the available diagnostic data to search for correlated error rates. Running the system in this manner results in the production of a collection of graph set images requiring further analysis by the user. In our example, we use data from deployed and test nodes from the Intelligent River® project. In Figure 5.1, we show the number of diagnostic measures reported from each of these nodes, measured in thousands. In total, we have over 3 million diagnostic measures from 36 nodes.

To remove duplicate data points, we use the default rounding setting to round our data to the nearest hundredth. To assist the reader in reproducing these results, the full function call is shown in Listing 5.1. The running time of Corl8 in batch mode on a single lab machine was about 9 hours. Our lab machine has 8 GB of RAM and an Intel Core2 Quad CPU at 2.66 GHz; however, because Corl8 uses a single thread, it is unable to take advantage of the 4 cores. By using multiple processes and instructing each instance of Corl8 to process a single device, this time could be significantly reduced. After the analysis completed, Corl8 flagged about 590 graph sets for further investigation. The collection of graph sets can be seen in Figure 5.2.

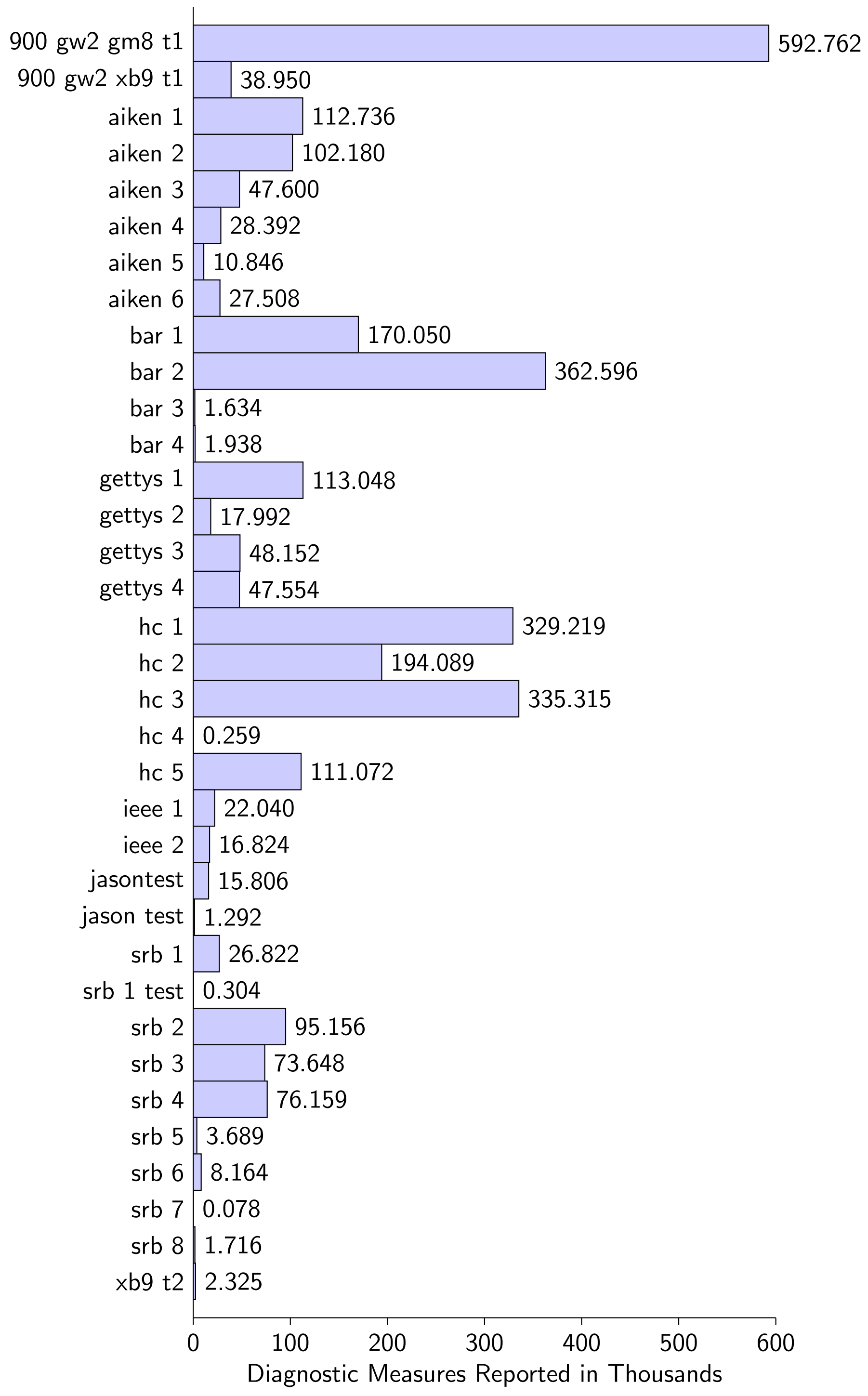
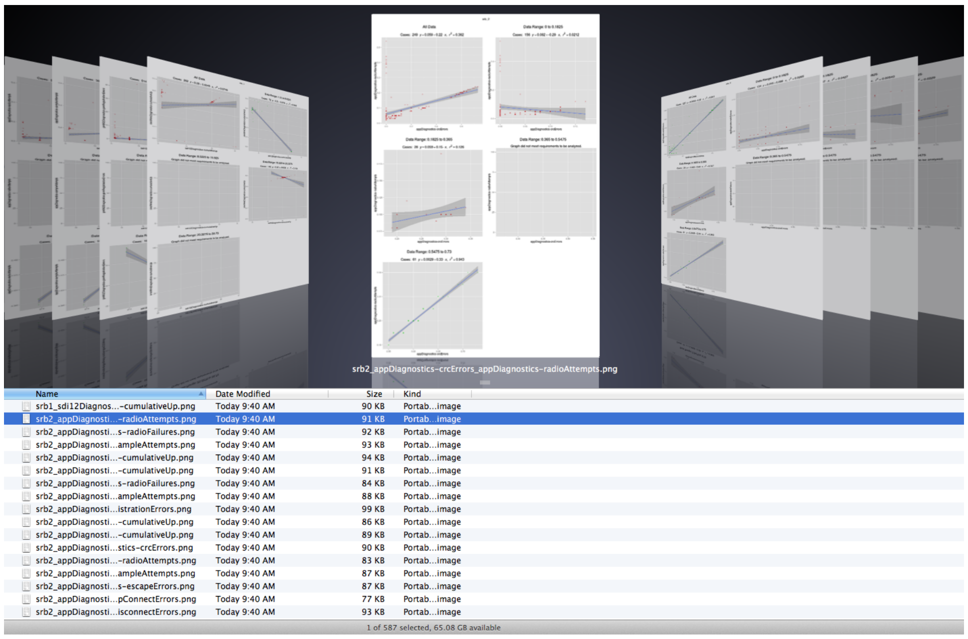


Figure 5.1: Nodes With Count of Diagnostic Measures

```r
# Function call with all the default parameters listed for reference
motecorr(device = "all",dbname="research", user="research", password="corl8",
         dbhost="localhost", table="rates",
         diagnostics = NULL, diagColX = "data", diagColY = "data",
         threshold = 25, round = 2, maxdup = 10, minrs = 0.75, minDate = NULL,
         maxDate = NULL, requireSubCor = TRUE,
         divisions = 4, divisionthreshold = 5,
         folder = "./graphs/", graphw = 500, graphh = 500, ncol = 2,
         returnGraph = FALSE, noDeviceError = "Blank device parameter",
         noGraphError = "Graph did not meet requirements to be analyzed.")

# Equivalent shortened function call
motecorr(dbname="research", user="research", password="corl8",
         dbhost="localhost", table="rates")
```

Listing 5.1: Batch Analysis Function Call


srb2_appDiagnostics-crcErrors_appDiagnostics-radioAttempts.png

Name	Date Modified	Size	Kind
srb1_sdi12Diagnos...-cumulativeUp.png	Today 9:40 AM	90 KB	Portab...image
srb2_appDiagnosti...-radioAttempts.png	Today 9:40 AM	91 KB	Portab...image
srb2_appDiagnosti...s-radioFailures.png	Today 9:40 AM	92 KB	Portab...image
srb2_appDiagnosti...ampleAttempts.png	Today 9:40 AM	93 KB	Portab...image
srb2_appDiagnosti...-cumulativeUp.png	Today 9:40 AM	94 KB	Portab...image
srb2_appDiagnosti...-cumulativeUp.png	Today 9:40 AM	91 KB	Portab...image
srb2_appDiagnosti...s-radioFailures.png	Today 9:40 AM	84 KB	Portab...image
srb2_appDiagnosti...ampleAttempts.png	Today 9:40 AM	88 KB	Portab...image
srb2_appDiagnosti...istrationErrors.png	Today 9:40 AM	99 KB	Portab...image
srb2_appDiagnosti...-cumulativeUp.png	Today 9:40 AM	86 KB	Portab...image
srb2_appDiagnosti...-cumulativeUp.png	Today 9:40 AM	89 KB	Portab...image
srb2_appDiagnosti...stics-crcErrors.png	Today 9:40 AM	90 KB	Portab...image
srb2_appDiagnosti...-radioAttempts.png	Today 9:40 AM	83 KB	Portab...image
srb2_appDiagnosti...ampleAttempts.png	Today 9:40 AM	87 KB	Portab...image
srb2_appDiagnosti...s-escapeErrors.png	Today 9:40 AM	87 KB	Portab...image
srb2_appDiagnosti...pConnectErrors.png	Today 9:40 AM	77 KB	Portab...image
srb2_appDiagnosti...isconnectErrors.png	Today 9:40 AM	93 KB	Portab...image

1 of 587 selected, 65.08 GB available

Figure 5.2: Resulting Graph Set Collection from Batch Analysis

Some of these graphs show expected correlations. An example is shown in Figure 6.2. This is graph of the number of sampling attempts versus the up-time for the sampling board used to collect the sample data. Graphs of this type show that the system is functioning properly. Still, users should review the graphs to ensure proper operation over time. Some graphs also give different views of the same data. For example, in our system, CRC errors were correlated to the number of sampling failures. They were also correlated to the number of sampling attempts and the up-time for the sampling boards since a sample failure would result in another sampling attempt and more up-time for the sampling system.

Other graphs may immediately be sources of concern. For example, Figure 6.1 is a graph showing that escape errors are correlated to TCP disconnect errors in our system, indicating that the cellular modem was sometimes indicating that a connection was established, when it was not. For more discussion on this graph, see Section 6.1.

The default analysis parameters have thus far met our goal of providing a reasonable number of useful graphs as output. However, we recommend evaluating different parameter selections to find the settings which work best for the user's particular deployment. One expected change is to increase or decrease the minimum R-squared value used to identify a correlated graph, which will further balance the number of graph sets with the false positive rate.

5.2 Interactive Analysis

The interactive analysis mode allows for faster results because the system produces only one graph set at a time in response to the user's request. Developers can use this interface to diagnose system issues. If a specific interaction is suspected, they can quickly view a graph of the two diagnostic measures in question. They can then modify the search parameters as needed to investigate whether the issue started or stopped at a specific time, and to see if viewing the data based on per minute, per radio attempt, or per sample attempt yields better information. For example, a developer might want to compare log errors per sample attempt versus log errors per radio attempt to see if one is more related than the other.

The web interface also allows the user to investigate what happens when different settings are used. Within the web interface, the user can interactively control every parameter of the analysis function, excluding the ones having to do with connecting to the database and saving the graphs.

Adjusting values interactively provides a quick way to increase and decrease the various thresholds and allowances, and to immediately see how a specific graph would be affected. This helps the user determine the optimal settings to use when running batch analysis.

Chapter 6

Results

Using Corl8 to analyze our Intelligent River® deployments throughout South Carolina, we were able to identify several interesting data trends, which helped to identify performance issues. In the following sections, we present three of the most interesting findings.

6.1 Cellular Modem Faults

The graph set in Figure 6.1 represents data from node 1, deployed at the Baruch Institute in Georgetown, SC. It shows TCP disconnect errors from our cellular board are correlated with escape errors from the same board. This correlation indicates an error in our sensor nodes. As background, to send data to the Intelligent River® server, the cellular modem must first create a connection to the server. After the connection is established, the modem automatically switches from "command mode" to "data mode." In command mode, each byte sent to the modem is interpreted as a command. In data mode, the bytes are instead relayed over the connection, which sends the data to the server. Once the connection is established, and the mote has sent its data, it needs to disconnect and shutdown. To complete this task, it must first escape back into command mode. This correlation data points to an issue where the cellular modem emits a connection response, but the connection is not fully established. As a result, the connection is immediately dropped, causing the device to fall back into command mode. From there, the escape command fails because the cellular mode is already in command mode, and the disconnect command also fails because it is no longer connected. This pattern of failure increases the up-time of the cellular chip, increasing energy

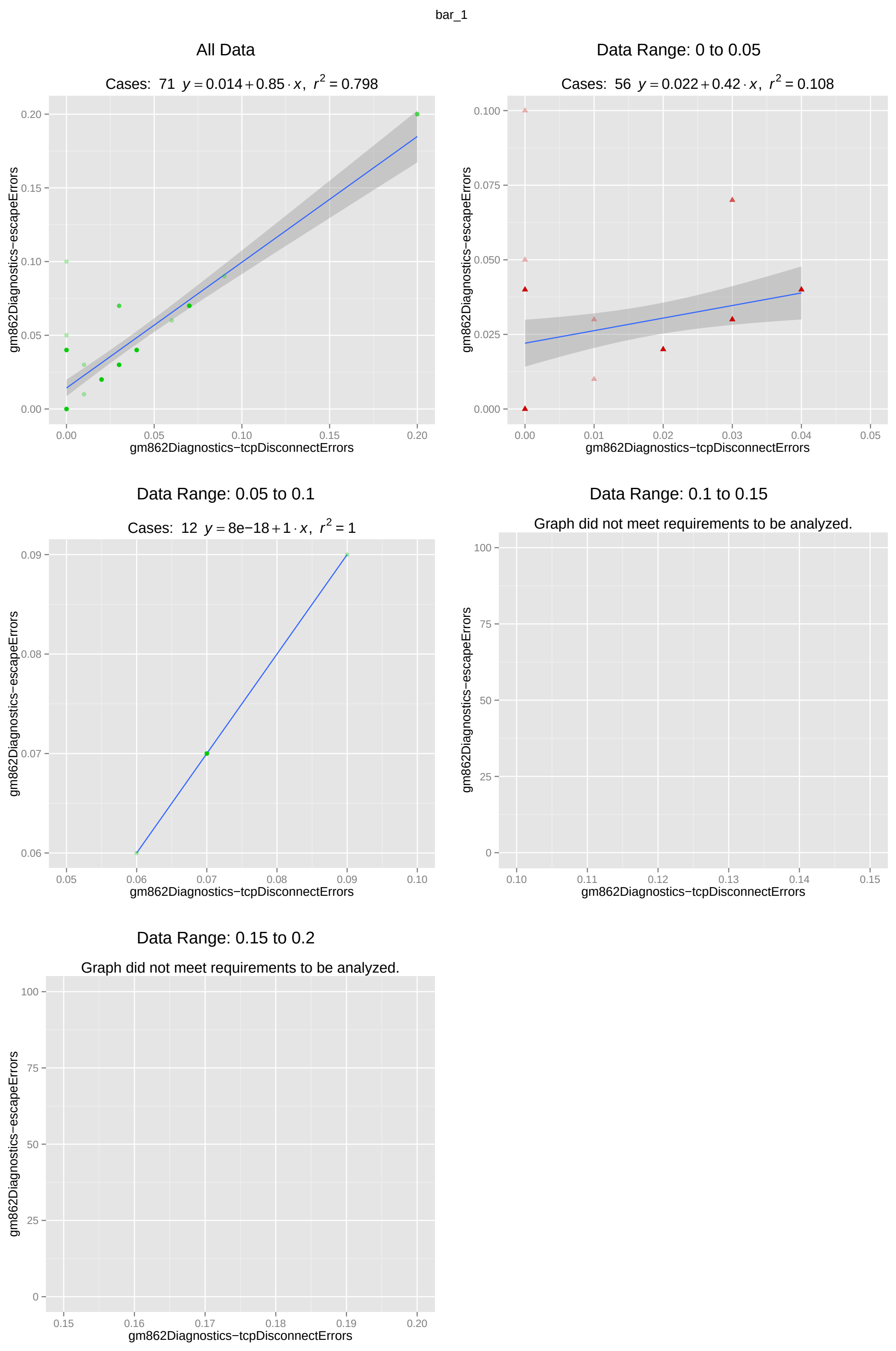


Figure 6.1: Baruch Node 1: TCP Disconnect Errors Versus Escape Errors

consumption, and decreasing device longevity. This correlation pattern points to this error chain.

6.2 Sample Attempts Versus Node Up-time

The graph set in Figure 6.2 shows the number of sample attempts performed versus the up-time of the sampling board in a device used at an IEEE conference demonstration. This graph set shows that the node was working properly. Looking closely at the graph, one outlier shows where 2 sampling attempts were made, and the ADS sampling board was up for 8.25 seconds, instead of the expected 8 seconds. If there were more outliers in the data set, it would be a cause for concern, but since there is only one outlier, it does not warrant concern.

6.3 Hunnicutt Creek Sampling Problems

While monitoring the nodes in the on-campus Hunnicutt Creek deployment, we noticed that two nodes were occasionally unresponsive for long periods. Looking at the application diagnostics, we observed that a large number of sample failures were occurring during these time periods. Since these sample failures were unexpected, we employed Corl8 to determine the cause of the failures. The observed failures began on February 24, 2014, so we ran a batch test using a start date (`minDate`) of `2014-02-24 00:00:00`. We also reduced the threshold to 10, and the division threshold to 3 because we had only a few reports showing these issues. Corl8 automatically flagged three graphs for node 1. These graphs showed sample failures as correlated to radio attempts, conversion errors on the sampling board, and up-time on the transmission board.

We were able to eliminate two of the graphs (radio attempts and up-time) because they showed that when a large number of sample failures occurred, the number of radio attempts (and thus up-time) was reduced. This is an expected correlation since the nodes only transmit when they have three successful samples. It did not, however, help to solve why the node experienced sample failures. The conversion errors from the SDI-12 sampling board provided more information. The graph produced by Corl8 is shown in Figure 6.3.

It was also interesting that Corl8 only flagged two graphs from node 3 —radio attempts and up-time on the transmission board. Thus, we turned to the web interface to allow us to view the graph of sample failures versus conversion errors. The graph is shown in Figure 6.4. From this

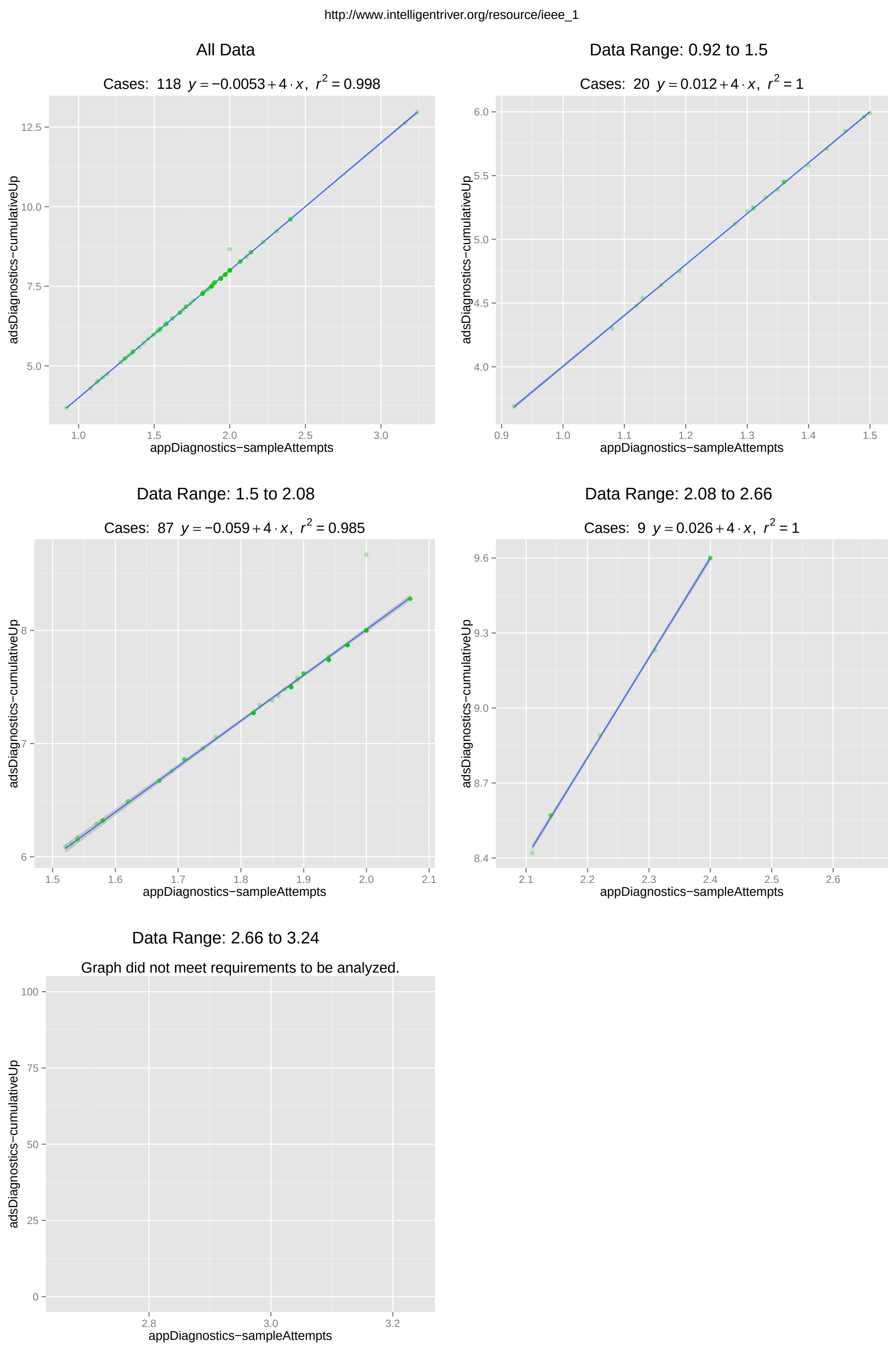


Figure 6.2: IEEE Demo Node 1: Sample Attempts Versus ADS Sampling Board Up-time

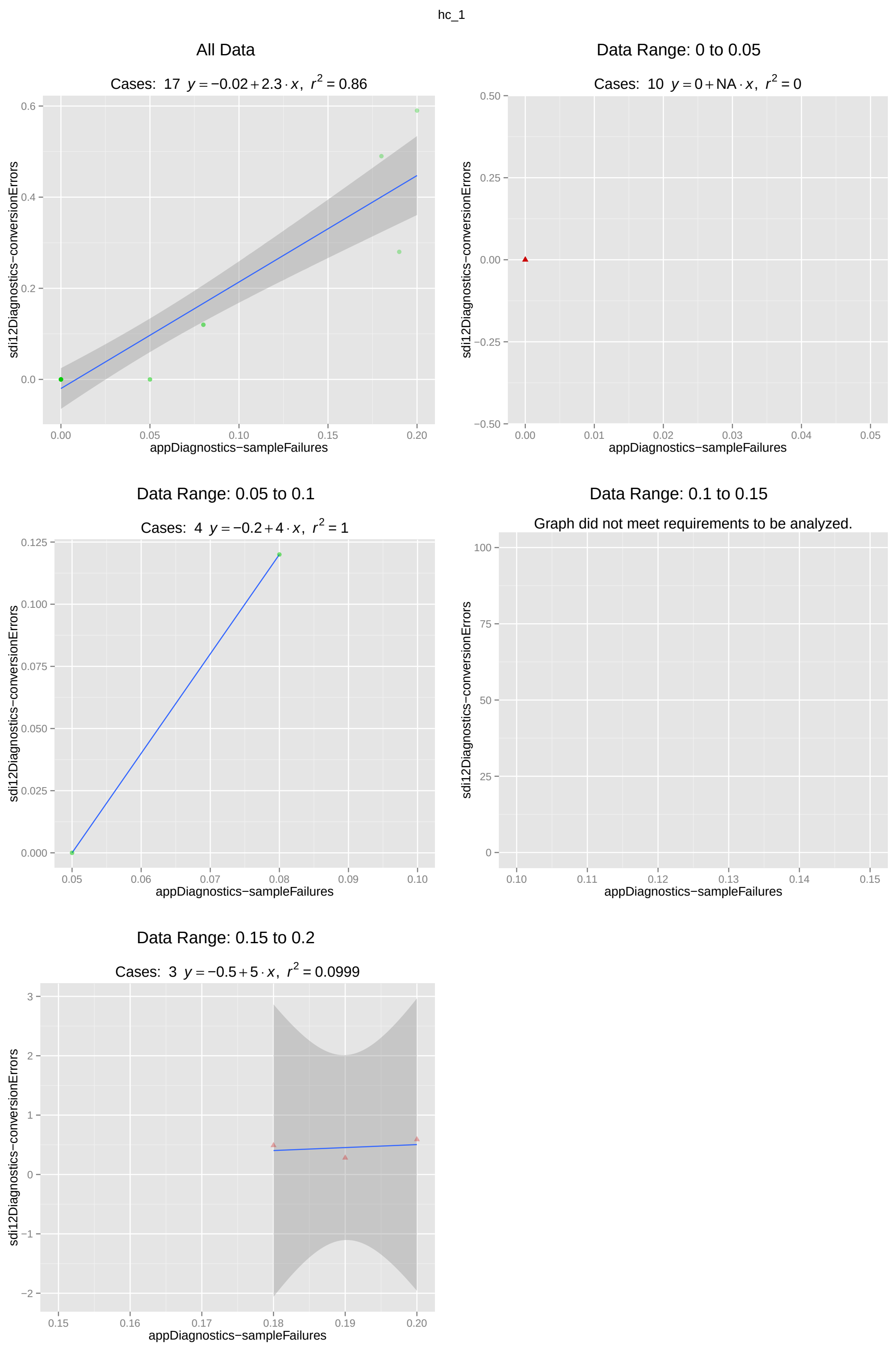


Figure 6.3: Hunnicutt Creek Node 1: Sample Failures Versus SDI-12 Board Conversion Errors

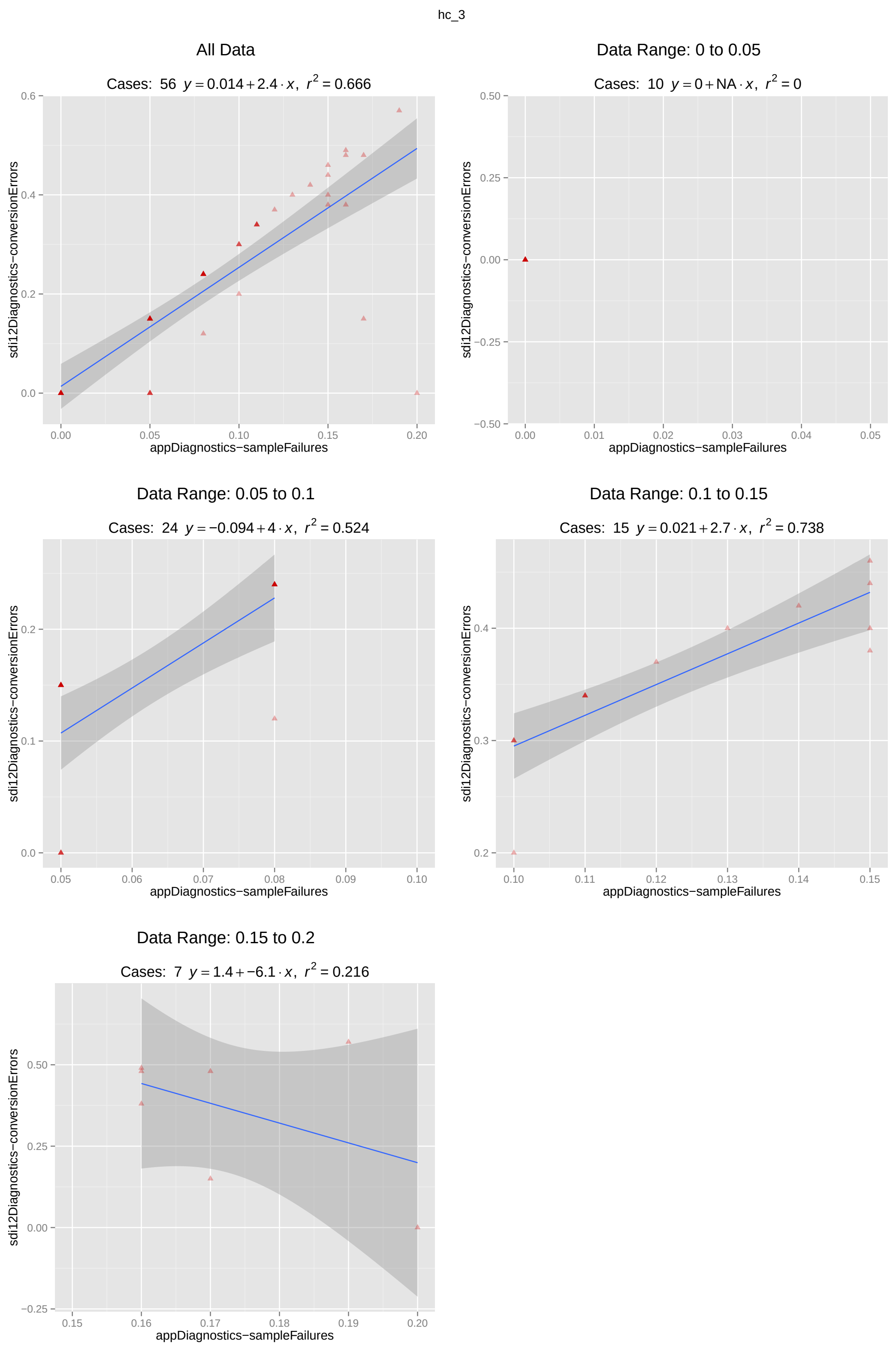


Figure 6.4: Hunnicutt Creek Node 3: Sample Failures Versus SDI-12 Board Conversion Errors

graph, we can observe that the general trend is still present, but the presence of several outliers caused Corl8 to not automatically flag the data set. At the time of writing, the development team had not yet identified the underlying cause of this error.

6.4 Conclusion of Results

We were able to find these errors through the use of Corl8's batch mode processing and further investigated the errors using the interactive mode. The time spent investigating the graphs was only a few hours to obtain useful information for addressing errors in our system and verifying that other components were working as expected. As a result, we are confident in the utility of our system to vastly decrease the time required to debug wireless sensor network systems.

Chapter 7

Conclusions and Discussion

Researchers are deploying wireless sensor networks more than ever before due to an increasing demand to monitor the physical world. Large numbers of sensors integrated with small, low-power, wireless transceivers compose these networks. The harsh, volatile locations in which these devices are often deployed increase their failure rate and decrease the rate at which packets can be successfully transmitted.

Corl8 helps developers determine causes of failure by analyzing correlated diagnostic measures. Since hardware resources such as memory and power are often scarce, our work seeks to minimally impact the amount of data that must be stored and transmitted. We also seek to avoid add-in network protocols, which consume additional resources and would be difficult to add to existing wireless sensor networks. The flexibility of our system allows researchers with sensor networks already collecting some amount of diagnostic data to implement our system with no changes to their network. They need only route received diagnostic data into our system.

Corl8 allows for batch mode analysis to help identify unknown faults in a system, and interactive analysis for investigating suspected faults. In our tests, Corl8 successfully found the correlated diagnostic measures to explain why nodes were seeing an unusually large number of cellular disconnect errors and assisted an exploration into why some nodes were experiencing an increased number of sampling errors. We believe Corl8 will help researchers more quickly and easily diagnose performance anomalies.

7.1 Recommendations for Further Research

It is our hope that the reach of the Corl8 system can be expanded to include observation data. We believe this will lead to more robust diagnostic analyses, particularly when environmental factors are suspected as a potential cause for failure. We also hope that this will lead to researchers in other disciplines using Corl8.

Appendices

Appendix A Analyzer motecorr.R Code

```r
library(gridExtra)
library(RMySQL)
library(ggplot2)

#' Node Diagnostic Analyzer
#'
#' This function will output graphs into the specified folder or return them.
#'
#' @param device The device to consider. If "all", all devices will
#'   be pulled from the database.
#' @param dbName The database name.
#' @param dbUser The username for the database.
#' @param dbPassword The password for the database.
#' @param dbHost The host for the database.
#' @param dbTable The database table.
#' @param diagnostics The vector of diagnostics to analyze or NULL to use all
#'   available diagnostics from the database.
#' @param diagColX The column to use for the X diagnostic.
#' @param diagColY The column to use for the Y diagnostic.
#' @param threshold The minimum number of points required to run the correlation test.
#' @param round The number of decimal places to allow or NULL to use data as is.
#' @param maxDuplicates The maximum number of duplicates of a single point allowed
#'   in the test.
#' @param minRS The minimum r-squared value from the correlation test required
#'   to output a graph.
#' @param minDate The start date in the form (YYYY-MM-DD HH:MM:SS)
#' @param maxDate The end date in the form (YYYY-MM-DD HH:MM:SS)
#' @param requireSubCor Require that a division be correlated in order to output the
#'   graph set.
#' @param divisions Number of segments to divide the data into.
#' @param divisionThreshold The minimum number of points in a division to run the
#'   correlation test.
#' @param folder The folder to use when saving the output graphs.
#' @param graphWidth The graph width (per graph, image will be enlarged for each graph).
#' @param graphHeight The graph height (per graph, image will be enlarged for each graph).
#' @param nCol The number of columns in the graph image.
#' @param returnGraph The first graph set will be returned, and the function will
#'   stop. No graphs will be saved. Return value is a list
#'   (graphlist, width, height).
#' @param noDeviceError The error message given if the device parameter is blank (for
#'   use with the Shiny server app).
#' @param placeholderTitle The title for placeholder graphs. (Placeholder graphs are
#'   generated when a division does not meet the requirements to be analyzed.)
#'
#' @keywords correlation, motes
#' @export
#' @examples
#' motecorr()
#' motecorr('srb_4')
#' motecorr(folder="./highTreshold", threshold=100)
#' To plot only the full graph with no divisions, don't forget to disable requireSubCor
#' motecorr(divisions = 1, requireSubCor = FALSE)
motecorr <- function(device = "all",dbName = "research", dbUser = "research",
      dbPassword = "corl8", dbHost = "localhost", dbTable = "rates",
      diagnostics = NULL, diagColX = "data", diagColY = "data",
      threshold = 25, round = 2, maxDuplicates = 10, minRS = 0.75,
      minDate = NULL, maxDate = NULL, requireSubCor = TRUE,
      divisions = 4, divisionThreshold = 5,
      folder = "./graphs/", graphWidth = 500, graphHeight = 500, nCol = 2,
      returnGraph = FALSE,
      noDeviceError = "Blank device parameter",
      placeholderTitle = "Graph did not meet requirements to be analyzed.") {
 con <- dbConnect(MySQL(),dbname=dbName, user=dbUser, password=dbPassword,
  host=dbHost)
 if (length(device) == 0) {
  dbDisconnect(con)
  stop(noDeviceError)
 }
 if (device == "all") {
  ids <- dbGetQuery(con, paste("select distinct(`device`) from",dbTable))
  if (is.null(diagnostics)) {
   diag <- dbGetQuery(con, paste("select distinct(`diagnostic`) from",dbTable))
  } else {
   diag <- data.frame(diagnostics)
   colnames(diag) <- c("diagnostic")
  }
 } else {
  ids <- data.frame(device)
  if (is.null(diagnostics)) {
   diag <- dbGetQuery(con, paste("select distinct(`diagnostic`) from",dbTable,
           paste("where `device`='", device,"'",sep="")))
  } else {
   diag <- data.frame(diagnostics)
   colnames(diag) <- c("diagnostic")
  }
 }
 dateWhere <- ""
 if (!is.null(minDate)) {
  minDate <- gsub("[^[(:num:)(:)(-)(/)] ]", "", minDate)
  dateWhere <- paste(dateWhere, " AND `time` >= '",minDate,"'",sep="")
 }
 if (!is.null(maxDate)) {
  minDate <- gsub("[^[(:num:)(:)(-)(/)] ]", "", minDate)
  dateWhere <- paste(dateWhere, "AND `time` <= '",maxDate,"'",sep="")
 }
 devcount <- 0

 for (id in ids$device) {
  devcount <- devcount + 1
  print(paste("Device",devcount,"of",nrow(ids),id))
  diagn <- length(diag$diagnostic)
  d1 <- 1
  while (d1 <= diagn) {
   print(diag$diagnostic[d1])
   data1 <- dbGetQuery(con,
           paste("SELECT `",diagColX,"` as `data`, `time` from ",dbTable,
           " WHERE `device` = '",id,"' AND `diagnostic`='",
           diag$diagnostic[d1],"'",dateWhere, sep=""))
   if (nrow(data1) > 1) { ## Need at least 2 rows to test correlation
    if (!is.null(round)) {
     # Round data as requested
     data1$data <- round(data1$data, round)
    }
    d2 <- 1
    while (d2 <= diagn) {
     if (d1 == d2) { ## Skip Identical Diagnostics
      d2 <- d2 + 1
      next
     }
     data2 <- dbGetQuery(con,
           paste("SELECT `",diagColY,"` as `data2`, `time` from ",dbTable,
           " WHERE `device` = '",id,"' AND `diagnostic`='",
           diag$diagnostic[d2],"'",dateWhere, sep=""))
     if (nrow(data2) > 1 && !is.null(round)) {
      # Round data as requested
      data2$data2 <- round(data2$data2, round)
     }
     result <- merge(data1, data2)
     result <- result[complete.cases(result), ];
     if (nrow(result) >= threshold && !is.null(round)) {
      result$data2 <- round(result$data2, round)
     }
     if (nrow(result) >= threshold && nrow(result) > maxDuplicates) {
      result <- result[with(result,order(data,data2)), ]
      remove <- rep(FALSE, maxDuplicates);
      ## Remove the Overly Duplicated Points
      for (t in (maxDuplicates+1):nrow(result)) {
       if (result[t, "data"] == result[(t-maxDuplicates), "data"]
         && result[t, "data2"] == result[(t-maxDuplicates), "data2"]) {
        ## Remove t becasue we have more than maxDuplicates
        remove <- append(remove, TRUE)
       } else {
        remove <- append(remove, FALSE)
       }
      }
      result <- result[!remove, ]
     }
     division <- c()
     graphlist <- list()
     if (nrow(result) >= threshold) {
      ## We need to retest since we may have removed points
      ##corr <- cor.test(result$data, result$data2)
      saveplots <- FALSE

      m = lm(result$data2 ~ result$data)
      rs <- summary(m)$r.squared

      color <- "#CC0000"
      shape <- 17
      if (!is.na(rs) && rs > minRS) {
       color <- "#00CC00"
       shape <- 16
       # If requireSubCor, this won't cause graphs to be saved
       if (!requireSubCor) { saveplots <- TRUE }
      }
      p <- ggplot(result, aes(x = data, y = data2)) + geom_smooth(method = "lm") +
          geom_point(alpha=0.3,colour=color,shape=shape)
      eq <- substitute(
          expression("Cases: "~c~~italic(y) == a + b %.% italic(x)*","~~italic(r)^2~"="~r2),
          list(
            a = format(coef(m)[1], digits = 2),
            b = format(coef(m)[2], digits = 2),
            c = nrow(result),
            r2 = format(rs, digits = 3)
          )
      )

      graphlist[[length(graphlist)+1]] <- p +
       scale_x_continuous(name=diag$diagnostic[d1]) +
       scale_y_continuous(name=diag$diagnostic[d2]) + ggtitle(eval(eq))
      ## Add Data Range
      graphlist[[length(graphlist)]] <- arrangeGrob(graphlist[[length(graphlist)]],
         main = textGrob("\n All Data",
         gp = gpar(fontsize = 16, face = "bold", col = "black")))


      if (divisions > 1) {
       ## Make our division graphs
       min <- result[1,"data"]
       max <- result[nrow(result),"data"]
       step <- (max-min)/divisions
       for (t in 1:nrow(result)) {
        div <- floor((result[t, "data"]-min)/step)
        if (!is.nan(div) && div > (divisions - 1)) {
         # Fixes a slight round off/boundary problem where the last point
         # can end up in its own division.
         div <- divisions - 1
        }
        division <- append(division, div)
       }
       divided <- split(result, as.factor(division))
       currentmin <- min
       currentdiv <- 1
       for (df in divided) {
        while (df$data[1] >= currentmin && currentmin < max) {
         # This catches divisions with no data
         title <- paste("\n Data Range:",currentmin,"to",(currentmin+step))
         if (df$data[1] < currentmin+step && nrow(df) >= divisionThreshold) {
          # Make sure we meet divisionThreshold and this data is for this division
          m = lm(df$data2 ~ df$data)
          rs <- summary(m)$r.squared

          color <- "#CC0000"
          shape <- 17
          if (!is.na(rs) && rs > minRS) {
           color <- "#00CC00"
           shape <- 16
           saveplots <- TRUE
          }
          p <- ggplot(df, aes(x = data, y = data2)) + geom_smooth(method = "lm") +
            geom_point(alpha=0.3,colour=color,shape=shape)
          subtitle <- substitute(
            expression("Cases: "~c~~italic(y) == a + b %.% italic(x)*","~~italic(r)^2~"="~r2),
            list(
             a = format(coef(m)[1], digits = 2),
             b = format(coef(m)[2], digits = 2),
             c = nrow(df),
             r2 = format(rs, digits = 3)
            )
          )

          graphlist[[length(graphlist)+1]] <- p +
           xlim(currentmin, currentmin+step) + xlab(diag$diagnostic[d1]) +
           scale_y_continuous(name=diag$diagnostic[d2]) + ggtitle(eval(subtitle))
         } else {
          # We didn't meet the threshold, plot a placeholder graph
          dfTemp <- data.frame()
          graphlist[[length(graphlist)+1]] <- ggplot(dfTemp) + geom_point() +
           xlim(currentmin, currentmin+step) + xlab(diag$diagnostic[d1]) +
           ylab(diag$diagnostic[d2]) + ylim(0, 100) + ggtitle(placeholderTitle)
         }
         graphlist[[length(graphlist)]] <- arrangeGrob(
            graphlist[[length(graphlist)]],
            main = textGrob(title,
              gp = gpar(fontsize = 16, face = "bold", col = "black"))
         )
         currentmin <- currentmin + step # Update the range
        }
       }
      }

      # Compute Graph Size
      if (length(graphlist) > nCol) {
       width <- graphWidth * nCol
       nCol2 <- nCol
      } else {
       nCol2 <- length(graphlist)
       width <- graphWidth * nCol2
      }
      height <- ceiling(length(graphlist)/nCol) * graphHeight
      # Add details to the graphlist
      graphlist[["main"]] <- paste("\n",id)
      graphlist[["ncol"]] <- nCol2

      if (returnGraph) {
       # If the graphs are to be returned (likely to shiny)
       dbDisconnect(con)
       return(list(graphlist,width,height))
      }
      if (saveplots) {
       # If graphs are to be saved
       png(
         file = paste(folder,
           gsub("[^[:alnum:]\\_\\-]", "",
              paste(id, diag$diagnostic[d1], diag$diagnostic[d2], sep="_")
           ),
           '.png', sep=""),
         width = width,
         height = height
       )
       do.call(grid.arrange, graphlist)
       dev.off()
      }
     } else if (returnGraph) {
      # If the graphs are to be returned, but we didn't make a graph.
      # (likely this is going to shiny server)
      dbDisconnect(con)
      df <- data.frame()
      graphlist[[length(graphlist)+1]] <- ggplot(df) + geom_point() +
         xlim(0, 10) + ylim(0, 100) + ggtitle(placeholderTitle)
      graphlist[["main"]] <- paste("\n",id)
      graphlist[["ncol"]] <- 1
      return(list(graphlist,graphWidth,graphHeight))
     }
     d2 <- d2 + 1
    }
   }
   d1 <- d1 + 1
  }
  if (returnGraph) {
    # If the graphs are to be returned, but we didn't make a graph.
    # (likely this is going to shiny)
   dbDisconnect(con)
    df <- data.frame()
   graphlist <- list()
    graphlist[[length(graphlist)+1]] <- ggplot(df) + geom_point() +
       xlim(0, 10) + ylim(0, 100) + ggtitle(placeholderTitle)
   graphlist[["main"]] <- paste("\n",id)
    graphlist[["ncol"]] <- 1
   return(list(graphlist,graphWidth,graphHeight))
  }
 }
 dbDisconnect(con)
 }
```

Listing 1: Analyzer Function motecorr.R

Appendix B Shiny Server ui.R Code

```r
# Configure the data options
dataOptions <- list("Per Minute" = "data",
                    "Per Radio Attempt" = "pradio",
                    "Per Sensor Attempt" = "psample",
                    "Raw Data" = "raw")

library(shiny)

# Define UI for corl8 application
shinyUI(pageWithSidebar(
  # Application title
  headerPanel("Corl8"),

  sidebarPanel(
    tags$head(
      tags$style(type='text/css',
        paste("@media (min-width: 1200px) { .row-fluid .span4 { width: 23%; }",
        ".row-fluid .span8 { width: 74%; } }",
        "@media (min-width: 768px) and (max-width:900px) { select { width: 190px; }",
        " input, textarea { width: 180px; }}")
      )
    ),
    uiOutput("mote"),
    uiOutput("diagX"),
    selectInput("diagColX", "Data for Diagnostic X:", dataOptions),
    uiOutput("diagY"),

    selectInput("diagColY", "Data for Diagnostic Y:", dataOptions),
    dateRangeInput("daterange", "Date Range", start = "2010-01-01", end = Sys.Date(),
                   min = "2000-01-01", max = Sys.Date(), format = "mm/dd/yy",
                   startview = "month", weekstart = 0, language = "en",
                   separator = " to "),
    textInput("mintime","Start Time:", value="00:00:00"),
    textInput("maxtime","End Time:", value="23:59:59"),
    sliderInput("threshold", "Threshold:", min=0, max=1000, value=25),
    checkboxInput("roundData", "Round Data", value=TRUE),
    conditionalPanel(
      condition = "input.roundData == true",
      sliderInput("round", "Number of Decimal Places:", min=-2, max=3, value=2)
    ),
    sliderInput("maxdup", "Maximum Duplicates Allowed Per Point:", min=1,
                                        max=100, value=10),
    sliderInput("minrs", "Minimum R-Squared:", min=0, max=1,
                                value=0.75, step= 0.05),
    sliderInput("divisions", "Number of Divisions:", min=1, max=100, value=4),
    sliderInput("divisionthreshold", "Division Threshold:", min=1, max=500, value=5),
    sliderInput("graphWidth", "Per Graph Width:", min=100, max=1000, value=500),
    sliderInput("graphHeight", "Per Graph Height:", min=100, max=1000, value=500),
    sliderInput("ncol", "Number of Columns in the Graph Set:", min=1, max=10, value=2),

    actionButton("graphButton", "Update Graph")
  ),

  mainPanel(
    h3(textOutput("caption")),
    plotOutput("corl8Plot")
  )
))
```

Listing 2: User Interface ui.R

Appendix C Shiny Server server.R Code

```r
# Configuration Variables
dbName <- "research"
dbUser <- "research"
dbPassword <- "corl8"
dbHost <- "localhost"
dbTable <- "rates"
noDeviceError <- "Select the variables at right and click \"Update Graph.\""
placeholderTitle <- "Graph did not meet requirements (threshold) to be analyzed."
# End Configuration

library(shiny)
library(gridExtra)
library(RMySQL)
library(ggplot2)
source("../motecorr.R")

con <- dbConnect(MySQL(),dbname=dbName, user=dbUser, password=dbPassword, host=dbHost)
ids <- dbGetQuery(con, paste("select distinct(`device`) from",dbTable))
dbDisconnect(con)
#ids <- dbGetQuery(con, paste("select distinct(`device`) from",dbTable))

# Define server logic required for corl8
shinyServer(function(input, output) {

  # Output the available devices
  output$mote <- renderUI({
    selectInput("mote", "Mote:", ids$device)
  })

  # Fetch the diagnostics
  diagOpt <- reactive({
    if (is.null(input$mote)) {
      return(c("Select a Mote"))
    }
    con <- dbConnect(MySQL(),dbname=dbName, user=dbUser, password=dbPassword, host=dbHost)
    diag <- dbGetQuery(con, paste("select distinct(`diagnostic`) from",dbTable,
        paste("where `device`='", input$mote,"'",sep="")))
    dbDisconnect(con)
    diag$diagnostic
  })

  # Render the diagnostic selects
  output$diagX <- renderUI({
    selectInput("diagX", "Diagnostic X:", diagOpt())
  })
  output$diagY <- renderUI({
    selectInput("diagY", "Diagnostic Y:", diagOpt())
  })

  # Run the processing as needed when the button is clicked
  graphlist <- reactive({
      input$graphButton
      isolate(
        if (input$roundData) {
          round <- input$round
        } else {
          round <- NULL
        }
      )
      minDate <- isolate(paste(as.character(input$daterange[1]),input$mintime))
      maxDate <- isolate(paste(as.character(input$daterange[2]),input$maxtime))

    isolate(motecorr(device = input$mote, dbName=dbName, dbUser=dbUser,
                     dbPassword=dbPassword, dbHost=dbHost, dbTable=dbTable,
                     diagnostics = c(input$diagX, input$diagY), diagColX = input$diagColX,
                     diagColY = input$diagColY,
                     threshold = input$threshold, round = round, maxDuplicates = input$maxdup,
                     minRS = input$minrs, minDate = minDate, maxDate = maxDate,
                     divisions = input$divisions, divisionThreshold = input$divisionthreshold,
                     graphWidth = input$graphWidth, graphHeight = input$graphHeight,
                     nCol = input$ncol, returnGraph=TRUE, noDeviceError = noDeviceError,
                     placeholderTitle = placeholderTitle))
  })
  # Helper Functions
  graphWidth <- function() {
    graphlist()[[2]]
  }
  graphHeight <- function() {
    graphlist()[[3]]
  }
  # Generate a plot
  output$corl8Plot <- renderPlot({
    do.call(grid.arrange, graphlist()[[1]])
  }, height=graphHeight, width=graphWidth)
})
```

Listing 3: Shiny Server server.R

Appendix D   Python MongoDB to MySQL Processing Script

```python
import collections
import time
import pymongo as mo
import _mysql
import MySQLdb as mdb
import sys
from decimal import Decimal

## Connect to MySQL
## Configure the connection as follows:
## MySQL Host, Username, Password, Database
con = mdb.connect('localhost', 'root','','research');
cur = con.cursor()

## If not localhost, configure this connection as:
## MongoDB Host, Port
moc = mo.MongoClient()
## Please configure the proper MongoDB Database.Collection
ird = moc.ir_diagnostics.messages;

## Load in the diagnostics we want
## Configure this with the diagnostic measure categories and
## individual measures that should be imported.
diagnostics = {
   "appDiagnostics": [
      "sampleAttempts",
      "sampleFailures",
      "sampleLosses",
      "radioAttempts",
      "radioFailures",
      "crcErrors",
      "cumulativeUp"
   ],
   "sdi12Diagnostics": [
      "activationErrors",
      "conversionErrors",
      "collectionErrors",
      "cumulativeUp"
   ],
   "oneWireDiagnostics": [
      "searchErrors",
      "conversionErrors",
      "collectionErrors",
      "cumulativeUp"
   ],
   "adsDiagnostics": [
      "wakeErrors",
      "configErrors",
      "conversionErrors",
      "sampleErrors",
      "cumulativeUp"
   ],
   "gm862Diagnostics": [
      "rssi",
      "wakeErrors",
      "sleepErrors",
      "setProfileErrors",
      "gsmRegistrationErrors",
      "gprsRegistrationErrors",
      "rssiErrors",
      "timeErrors",
      "smsStartErrors",
      "smsEndErrors",
      "getIpErrors",
      "dropIpErrors",
      "emailStartErrors",
      "emailEndErrors",
      "tcpConnectErrors",
      "tcpDisconnectErrors",
      "escapeErrors",
      "cumulativeUp"
   ],
   "xbee900Diagnostics": [
      "wakeErrors",
      "cmdErrors",
      "sendErrors",
      "frameErrors",
      "baseRSSI",
      "cumulativeUp"
   ],
   "logDiagnostics": [
      "formatErrors",
      "headerReadErrors",
      "headerWriteErrors",
      "headerCorruptionErrors",
      "recordCorruptionErrors",
      "validationErrors",
      "enqueueErrors",
      "overflowErrors",
      "dequeueErrors",
      "underflowErrors"
   ],
};

ds = ird.distinct("deviceID"); ## Get the unique devices
dsids = [];
dscount = 0;
for id in ds:
   dscount = dscount + 1
   if id is None:
      print "No device id, skipping"
      continue
   elif id.find('#') >= 0:
      devid = id.split('#',1)[1]
   else:
      devid = id
   devstr = "Device " + devid + " " + str(dscount) + " of " + str(len(ds));
   print devstr
   dsids.append(devid)  ## Add it to our list so we know we got everything
   ## Get the sorted messages
   mes = ird.find(
         {"deviceID": id},
         ["deviceID", "datestamp", "unpackedMessage"]
      ).sort("datestamp",1);
   ## Init datalast, This will force it to look like a reboot happened
   dataLast = {
      "timestamp" : 0,
      "unpackedMessage":
         {
            "appDiagnostics": {
               "cumulativeUp": -100,
               "sampleAttempts": 0,
               "radioAttempts": 0
            }
         }
```

```
}

totalm = str(mes.count());
mcount = 0;
scount = 0;
for m in mes:
   if mcount % 1000 == 0:
      print devstr + " Processed "+ str(mcount) + " of " + \
         totalm + " Skipped "+str(scount);
   mcount = mcount + 1
   if "datestamp" not in m.keys():
      print "No datestamp, skipping"
      continue
   if ("unpackedMessage" not in m.keys() or
      "appDiagnostics" not in m["unpackedMessage"].keys()):
      scount = scount + 1
      continue ## Can't process without this, must be old data

   appkeys = m["unpackedMessage"]["appDiagnostics"].keys();
   if ("cumulativeUp" not in appkeys or "radioAttempts" not in appkeys or
      "sampleAttempts" not in appkeys):
      scount = scount + 1
      continue ## Can't process without this, must be old data
   m["timestamp"] = time.mktime(m["datestamp"].timetuple());
   if dataLast["timestamp"] > m["timestamp"]:
      print "Out of order data!"
      exit();

   # Fix data to be ints
   m["unpackedMessage"]["appDiagnostics"]["cumulativeUp"] = \
      Decimal(m["unpackedMessage"]["appDiagnostics"]["cumulativeUp"])
   m["unpackedMessage"]["appDiagnostics"]["radioAttempts"] = \
      Decimal(m["unpackedMessage"]["appDiagnostics"]["radioAttempts"])
   m["unpackedMessage"]["appDiagnostics"]["sampleAttempts"] = \
      Decimal(m["unpackedMessage"]["appDiagnostics"]["sampleAttempts"])

   ## Try to detect reboots
   time1 = Decimal(m["timestamp"] - dataLast["timestamp"])
   time2 = (m["unpackedMessage"]["appDiagnostics"]["cumulativeUp"] -
      dataLast["unpackedMessage"]["appDiagnostics"]["cumulativeUp"])
   radioa = (m["unpackedMessage"]["appDiagnostics"]["radioAttempts"] -
      dataLast["unpackedMessage"]["appDiagnostics"]["radioAttempts"])
   samplea = (m["unpackedMessage"]["appDiagnostics"]["sampleAttempts"] -
      dataLast["unpackedMessage"]["appDiagnostics"]["sampleAttempts"])

   ## Cache the keys
   munkeys = m["unpackedMessage"].keys()
   reboot = False;
   ## Compute Errors Per Minute
   if (abs(time1-time2) > 10): ## Reboot happened, 10 second grace
      reboot = True
      print "Reboot"
      if m["unpackedMessage"]["appDiagnostics"]["cumulativeUp"] > 1500:
         print "Possible error: " + \
            str(m["unpackedMessage"]["appDiagnostics"]["cumulativeUp"])

   if not reboot and (time2 == 0 or radioa == 0 or samplea == 0):
      print "ERROR: Duplicate data?"
      print m
      print dataLast
      scount = scount + 1
      continue
   for section in diagnostics:
      if section not in munkeys : ## Does this exist
```

```python
            continue
         for part in diagnostics[section]:
            if (part not in m["unpackedMessage"][section].keys() or
               (not reboot and
               part not in dataLast["unpackedMessage"][section].keys())):
               continue
            diag = section + "-" + part
            if m["unpackedMessage"][section][part] is str:
               m["unpackedMessage"][section][part] = \
                  Decimal(m["unpackedMessage"][section][part])
            if reboot:
               perminute = (m["unpackedMessage"][section][part] /
                  m["unpackedMessage"]["appDiagnostics"]["cumulativeUp"] * 60)
               persample = (m["unpackedMessage"][section][part] /
                  m["unpackedMessage"]["appDiagnostics"]["sampleAttempts"])
               perradio = (m["unpackedMessage"][section][part] /
                  m["unpackedMessage"]["appDiagnostics"]["radioAttempts"])
            else: ## Take out the last readings
               perminute = (m["unpackedMessage"][section][part] -
                  dataLast["unpackedMessage"][section][part]) / time2 * 60
               persample = (m["unpackedMessage"][section][part] -
                  dataLast["unpackedMessage"][section][part]) / samplea
               perradio = (m["unpackedMessage"][section][part] -
                  dataLast["unpackedMessage"][section][part]) / radioa
            ## Insert into MySQL
            cur.execute("INSERT INTO `research`.`rates` (`id`, " + \
               "`diagnostic`, `pmin`, `pradio`, `psample`, `raw`, " + \
               "`time`, `device`) VALUES (NULL, '" + diag + "', '" + \
               str(perminute) + "', '" + str(perradio) + "', '" + \
               str(persample)+"', '" + str(m["unpackedMessage"][section][part]) + \
               "', '"+str(m["datestamp"])+"', '"+devid+"');")
      dataLast = m;
   ## This device is done.  Print statistics and continue.
   print devstr + " Done. Processed "+ str(mcount) + " of " + totalm + \
      " Skipped " + str(scount);
print "Processing Complete."
du = collections.OrderedDict.fromkeys(dsids)
if (len(du) != dscount):
   print "WARNING: Errors may have occured during shortening.  Only output " + \
      str(len(du)) + " devices.";
```

Listing 4: Python MongoDB to MySQL Processing Script

Bibliography

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] Baptiste Auguie. *gridExtra: functions in Grid graphics.* http://CRAN.R-project.org/package=gridExtra, 2012. R package version 0.9.1.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] Elizabeth A. Basha, Sai Ravela, and Daniela Rus. Model-based monitoring for early warning flood detection. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 295–308, New York, NY, USA, 2008. ACM.

[5] P. Bonnet, M. Leopold, and K. Madsen. Hogthrob: Towards a sensor network infrastructure for sow monitoring (wireless sensor network special day). In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, DATE '06, pages 1109–1109, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[6] Octav Chipara, Chenyang Lu, Thomas C. Bailey, and Gruia-Catalin Roman. Reliable clinical monitoring using wireless sensor networks: Experiences in a step-down hospital unit. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 155–168, New York, NY, USA, 2010. ACM.

[7] CRAN Team. Cran - contributed packages, http://cran.rstudio.com/web/packages/, March 21, 2014 (*last access*).

[8] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. Emstar: A software environment for developing and deploying wireless sensor networks. In *USENIX 2004 Annual Technical Conference*, pages 283–296, 2004.

[9] Lewis Girod, Nithya Ramanathan, Jeremy Elson, Thanos Stathopoulos, Martin Lukac, and Deborah Estrin. Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. *ACM Trans. Sen. Netw.*, 3(3), August 2007.

[10] David A. James and Saikat DebRoy. *RMySQL: R interface to the MySQL database.* http://CRAN.R-project.org/package=RMySQL, 2012. R package version 0.9-3.

[11] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 99–112, New York, NY, USA, 2008. ACM.

[12] Ted Tsung-Te Lai, Wei-Ju Chen, Kuei-Han Li, Polly Huang, and Hao-Hua Chu. Triopusnet: Automating wireless sensor network deployment and replacement in pipeline monitoring. In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks*, IPSN '12, pages 61–72, New York, NY, USA, 2012. ACM.

[13] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[14] Mo Li and Yunhao Liu. Underground coal mine monitoring with wireless sensor networks. *ACM Trans. Sen. Netw.*, 5(2):10:1–10:29, April 2009.

[15] Kebin Liu, Mo Li, Xiaohui Yang, and Mingxing Jiang. Passive diagnosis for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 371–372, New York, NY, USA, 2008. ACM.

[16] MongoDB, Inc. MongoDB, https://www.mongodb.org/, March 20, 2014 (*last access*).

[17] R Special Interest Group on Databases. *DBI: R Database Interface*. http://CRAN.R-project.org/package=DBI, 2013. R package version 0.2-7.

[18] Lilia Paradis and Qi Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15(2):171–190, 2007.

[19] R Core Team. What is r? http://www.r-project.org/about.html, February 7, 2014 (*last access*).

[20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, http://www.R-project.org/, 2013.

[21] R Core Team. *R: A Language and Environment for Statistical Computing*, pages 412–413. R Foundation for Statistical Computing, http://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf, 2014.

[22] R Core Team and Motor Trend. R: Motor trend car road tests, https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html, March 21, 2014 (*last access*).

[23] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 255–267, New York, NY, USA, 2005. ACM.

[24] Nithya Ramanathan, Eddie Kohler, and Deborah Estrin. Towards a debugging system for sensor networks. *Int. J. Netw. Manag.*, 15(4):223–234, July 2005.

[25] RStudio, Inc. Rstudio, http://www.rstudio.com/, March 21, 2014 (*last access*).

[26] RStudio, Inc. *shiny: Web Application Framework for R*. http://CRAN.R-project.org/package=shiny, 2013. R package version 0.8.0.

[27] L.B. Ruiz, J.M. Nogueira, and A. A F Loureiro. Manna: a management architecture for wireless sensor networks. *Communications Magazine, IEEE*, 41(2):116–125, 2003.

[28] Linnyer Beatrys Ruiz, Isabela G. Siqueira, Leonardo B. e Oliveira, Hao Chi Wong, José Marcos S. Nogueira, and Antonio A. F. Loureiro. Fault management in event-driven wireless sensor networks. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '04, pages 149–156, New York, NY, USA, 2004. ACM.

[29] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 188–200, New York, NY, USA, 2004. ACM.

[30] Philipp Sommer and Branislav Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 12:1–12:14, New York, NY, USA, 2013. ACM.

[31] Ben Steffens. Radar for facebook, http://www.discoverradar.com/, December 31, 2013 (*last access*).

[32] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Lightweight tracing for wireless sensor networks debugging. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 13–18, New York, NY, USA, 2009. ACM.

[33] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 169–182, New York, NY, USA, 2010. ACM.

[34] Vinaitheerthan Sundaram, Patrick Eugster, Xiangyu Zhang, and Vamsidhar Addanki. Diagnostic tracing for wireless sensor networks. *ACM Trans. Sen. Netw.*, 9(4):38:1–38:41, July 2013.

[35] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, June 2004.

[36] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a sensor network expedition. In Holger Karl, Adam Wolisz, and Andreas Willig, editors, *Wireless Sensor Networks*, volume 2920 of *Lecture Notes in Computer Science*, pages 307–322. Springer Berlin Heidelberg, 2004.

[37] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2006.

[38] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 51–63, New York, NY, USA, 2005. ACM.

[39] Wei Wang, Vikram Srinivasan, Kee-Chaing Chua, and Bang Wang. Energy-efficient coverage for target detection in wireless sensor networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 313–322, New York, NY, USA, 2007. ACM.

[40] Hadley Wickham. *ggplot2: elegant graphics for data analysis.* Springer New York, http://had.co.nz/ggplot2/book, 2009.

[41] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 189–203, New York, NY, USA, 2007. ACM.