ABSTRACT

Title of dissertation: AUTOMATING PERFORMANCE DIAGNOSIS IN NETWORKED SYSTEMS

Justin N. McCann, Doctor of Philosophy, 2012

Dissertation directed by: Professor Michael W. Hicks
Department of Computer Science

Diagnosing performance degradation in distributed systems is a complex and difficult task. Software that performs well in one environment may be unusably slow in another, and determining the root cause is time-consuming and error-prone, even in environments in which all the data may be available. End users have an even more difficult time trying to diagnose system performance, since both software and network problems have the same symptom: a stalled application.

The central thesis of this dissertation is that the source of performance stalls in a distributed system can be automatically detected and diagnosed with very limited information: the dependency graph of data flows through the system, and a few counters common to almost all data processing systems.

This dissertation presents FlowDiagnoser, an automated approach for diagnosing performance stalls in networked systems. FlowDiagnoser requires as little as two bits of information per module to make a diagnosis: one to indicate whether the module is actively processing data, and one to indicate whether the module is waiting on its dependents.

To support this thesis, FlowDiagnoser is implemented in two distinct environments: an individual host's networking stack, and a distributed streams processing system. In controlled experiments using real applications, FlowDiagnoser correctly diagnoses 99% of networking-related stalls due to application, connection-specific, or network-wide performance problems, with a false positive rate under 3%. The prototype system for diagnosing messaging stalls in a commercial streams processing system correctly finds 93% of message-processing stalls, with a false positive rate of 2%.

AUTOMATING PERFORMANCE DIAGNOSIS IN NETWORKED SYSTEMS

by

Justin N. McCann

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:
Professor Michael W. Hicks, Chair/Advisor
Professor Peter Keleher
Professor James Reggia
Professor Mark Shayman
Professor Neil Spring

Dedication

To Amber,
who always trusts,
always hopes,
always perseveres.

To Svea,
for never giving up,
patiently encouraging,
and writing even when it is no fun.

To Rory,
for reminding me of the joy
of playing outside
and laughing out loud.

To Tait,
for showing that love
comes in many forms,
often shaped like a monkey.

Acknowledgments

To my academic advisor, Prof. Michael Hicks: thank you for questioning assumptions, constantly pushing, and not being afraid to get rid of ideas that seemed important at the time, but turned out to be not so necessary in the end. Your insight and advice improved the research, the results, and the presentation all along the way.

Prof. Neil Spring encouraged me to stick with an idea that was a bit too ambitious from the start, provided great ideas to make it work, and constantly reminded me to focus on proving a falsifiable thesis. Prof. Jim Reggia's years of experience in diagnosis systems gave us all the confidence to try something different; thanks for reminding me that the Ph.D. process would not go on forever. Thank you to Prof. Pete Keleher and Prof. Mark Shayman for your insightful questions and direction.

Thanks to Dr. Thomas Dubois and Dr. Michael Marsh, who were instrumental in finding a way to break cycles in the dependency graph while preserving the important relationships.

Several people and groups helped in gathering and analyzing data from InfoSphere Streams and provided valuable feedback on StreamsDiagnoser. Special thanks belongs to Dr. Octavian Udrea of IBM's Thomas J. Watson Research Center, Dr. John May of Lawrence Livermore National Laboratory, Andrew Skene, and Roshan Punnoose for their work in this area, in spite of many competing demands for their time.

To my friends and colleagues at the University of Maryland and at work, thank you for the many times you listened when I was stuck, and provided the encouragement and ideas that kept me moving. And to all of my cycling buddies: thanks for getting me out on my bike. My wife thanks you as well.

Thank you to Professor Jim Skon, a true friend whose support and advice I treasure deeply. Thanks also to Mark Peterman and Ben Obrock, two of my oldest and dearest friends: it is good to know that time does not weaken a friendship, but only makes it stronger.

Special thanks goes to all of our friends and family at Bethany Community Church, who have been a constant source of encouragment and support over the years. I really cannot imagine doing this without you.

To my Mom and Dad, Mom and Dad Rhoton, Cavan, Jenn, Robin, Myron, Brody, Emily, and all my nieces and nephews: thanks for supporting us all in this dream of mine, even though it meant a lot of distraction and distance over these years. We have thought of you constantly, love you dearly, and cannot wait to spend more time with you.

Finally, all glory, honor, and praise be to God our Father and the Lord Jesus Christ, whose Spirit is the breath of creativity that gives order to the universe.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Diagnosing performance degradation in distributed systems is a complex and difficult task. Software that performs well in one environment may be unusably slow in another, and determining the root cause is time-consuming and error-prone, even in environments where all the data may be available. End users have an even more difficult time trying to diagnose system performance. When a user's video stream has problems, it could be for any number of reasons: the browser plugin may be buggy, the neighbors' wireless networks may be creating interference, the user's computer or the video server may be overloaded, or there may be congestion on the Internet path between the two. To the user of a distributed service or application, the symptoms are all the same: a stalled or stuttering application.

1.1 Sources of performance stalls

This dissertation focuses on detecting and finding the source of short performance stalls lasting a few hundred milliseconds to a few seconds. They can be caused by faulty or slow software; contention for shared server resources such as the CPU, disk I/O, backend database, or a shared lock; network congestion and retransmission timeouts; or other errors in system components.

One common source of performance problems is the software itself. One commonly suspected culprit are browser plugins such as Adobe Flash. When Apple famously refused to support Flash on their mobile platforms, Steve Jobs claimed, "Flash is the number one reason Macs crash," and "has not performed well on mobile devices" [31]. Processing overhead and resource scheduling in monolithic web browsers spurred Google researchers to redesign the browser to use a multi-process architecture, leading to significant speedups in page load times [44].

Even well-written software such as the Apache web server can experience sudden spikes in request latency due to head-of-line blocking for disk accesses [46], contending for shared resources [52], disk writes, or database queries [19]. Whether these stalls in progress are due to bugs, inefficient locking mechanisms, or calls to backend database servers, they prevent the application software from responding to requests in a timely manner.

Another common source of performance stalls is network congestion. A 2011 study of user-facing network traffic at two Google datacenters [21] found that packet loss and retransmissions are fairly common: 2.5–5.6% of all user-facing TCP connections retransmit packets. While fast retransmit and selective acknowledgments (SACK) can avoid complete throughput stalls when packet loss occurs, the study also showed that roughly 1% of all connections stalled for at least 200 ms due to a retransmission timeout (RTO). Even when TCP is able to avoid retransmission timeouts, any retransmissions are costly: short web requests take on average 7–10 times as long to complete when the TCP connection retransmits any packets [21].

In modern distributed applications, seemingly rare events can have a significant effect on response time. When applications depend on dozens or hundreds of separate services to respond in a timely manner, the outliers in the long tail of the latency distribution are not so uncommon. Amazon e-commerce applications can consult up to 150 services to respond to one request, with each transaction potentially experiencing low throughput or a transient stall due to network congestion, server processing, contention for disk I/O, a longer-than-usual database query, or a transient problem in the network [19]. Each service is required to meet a service-level agreement (SLA), typically to complete 99.9% of transactions in under 300 ms. Even assuming these 150 transactions are perfectly parallel, only 86% of requests will be completed within 300 ms;[1] serialized transactions increase delay.

1.2 Latency's impact on the bottom line

While these transient stalls may be brief, and affect only a small percentage of connections, their impact to users is disproportionate. No matter their source—stalled network connections, overloaded server software or database systems, or congested networks and normal processing time—user-visible delays directly affect the bottom line of large Internet companies. Even small reductions in search or page view volume add up to hundreds of millions of dollars in lost revenue [40].

Locating and correcting the causes of performance stalls and excessive latency can have a significant positive impact on revenue. When Shopzilla completely re-

[1] If $P_{ServiceTime>300ms} = 0.001$, then $P_{RequestTime\leq 300ms} = \prod_{i=1}^{150}(1 - 0.001) = 0.86$

designed their backend architecture in an effort to lower page load times, reducing them from 6–9 seconds to 1.2 seconds on average, they saw a 120% increase in search-engine referred traffic, a 7–12% increase in visitor conversion rates, and a 5–12% increase in gross revenue [20].

Earlier studies have found that user's intent to keep using a website and their performance in completing tasks starts to decline after two seconds of delay [23], but recent large-scale studies at AOL, Google, and Microsoft show that users are much more sensitive to latency than previously thought. In controlled experiments, engineers at Google and Bing randomly selected users to experience added delays in server processing, simulating a slower response from backend services. A 400 ms increase in server-side latency reduced per-user searches (and ad views) at Google by 0.76%; a 500 ms increase reduced per-user revenues by 1.2% at Bing. As latency increased, the results were even more dramatic: per-user revenue dropped by 4.3% when users were subjected to a two-second increase in delay [47]. AOL data show an inverse relationship between page load times and the number of pages viewed per visit [5].

1.3 Stalls are hard to diagnose

Many systems exist for monitoring and analyzing the performance of distributed applications. Some require invasive changes to instrument software source code [26, 45] and track individual messages as they are sent throughout the system. While this can help developers and operators to track down subtle bugs and perfor-

mance problems, the required code changes create a high barrier to entry, especially when monitoring a third-party system for which no source code is available.

Other approaches analyze per-packet network captures [6, 14, 15] to try to infer the states of important system elements. While packet captures can be taken without affecting the performance or source code of the monitored system, they are too expensive to run and analyze continuously, and by nature have little information when a system stops transmitting data. When traffic ceases, it could be that the software has stalled, every transport-layer (TCP) connection has detected network congestion and backed off its retransmissions, or the system has completed all of its current work. Without monitoring the end hosts, it is difficult to reliably distinguish between cause and effect.

Many sophisticated monitors aggregate data from throughout the network [2, 6, 14, 15, 32, 60] to detect systemic problems. These systems are able to locate and detect a wide range of network, software, and system misbehaviors, but mostly rely on complicated analyses that are difficult to recreate, and are of little use for a single host or end user.

Another common approach is to perform protocol-specific analysis [2, 15, 32, 35, 60] to detect performance problems exhibited by specific network technologies. These analyses can be invaluable for tracking down difficult and nuanced problems in modern systems of systems. However, applying these protocol-specific insights to new problem domains is not straightforward.

While these systems all provide sophisticated and detailed analyses, they are difficult to implement, expensive to run, and in many cases not generalizable.

1.4 Thesis

The central thesis of this dissertation is that the source of performance stalls in a distributed system can be automatically detected and diagnosed with very limited information: the dependency graph of data flows through the system, and a few counters common to almost all data processing systems.

The automated fault diagnosis system requires as little as two bits of information per module: one to indicate whether the module is actively processing data, and one to indicate whether the module is waiting on its dependents.

To support this thesis, the approach is implemented and evaluated in two distinct environments: an individual host's networking stack, and a distributed streams processing system.

1.5 Goals

The goal of this research is to create an approach to messaging performance diagnosis that is efficient enough to run constantly, can automatically detect and report performance stalls using as little information as possible, and is general enough to apply across application domains. It also will enable the following:

- An end user will be able to tell whether their web browser, network connection, or a single TCP stream is causing their performance problems.
- Individual hosts in a distributed system will be able to detect software, connection-specific, or more widespread network problems and report them in

a succinct manner to a monitoring service for cross-correlation and analysis. Such reports will also provide evidence to help pinpoint the root cause of the stall, such as a faulty network interface.

- System administrators and developers will be able to monitor the health of communication in a distributed system, to find which processes or subsystems are preventing progress overall.

- Subject matter experts will apply the basic principles of the FlowDiagnoser approach to finding performance stalls in their own systems.

1.6 Overview

Chapter 2 describes the FlowDiagnoser approach for locating the source of performance stalls in distributed systems. FlowDiagnoser first constructs a dependency graph, a directed graph that represents the movement of messages between modules of the system. Rather than trace specific messages to see where they are getting dropped or hung up, FlowDiagnoser periodically monitors a few basic counters exported by each node, and performs an abstract analysis of the modules' behavior to make a diagnosis.

Once FlowDiagnoser has constructed the dependency graph, diagnosis proceeds in three steps:

1. Periodically snapshot the message counters from each module.

2. Use the counters to infer the module's (in)activity state.

3. Perform a dependency analysis, relating one module's state to that of its dependents and neighbors, to determine whether the module is misbehaving.

The resulting diagnosis is a set of annotations applied to the original graph, with each module labeled to indicate whether it was healthy, blocked by another module, stalled and preventing other modules' progress, or its performance can safely be ignored.

In addition to the automated diagnosis, FlowDiagnoser provides several visualizations and summary reports which explain which modules were behaving well, which ones stalled progress, and show the changes in counter values over time. These reports and visualizations also help an expert user to determine if the diagnosis was correct, given the how the counters in the system change over time.

Two applications of FlowDiagnoser show its ability to accurately diagnose performance stalls lasting from hundreds of milliseconds to a few seconds in two distinct settings.

The Network Stack Trace (NeST) is the first application of the FlowDiagnoser approach, described in Chapter 3. NeST diagnoses the source of performance stalls that are caused by applications, are specific to particular network connections, or are due to network-level events, and does so using only the counters available at a single end host's networking stack. Figure 1.1 illustrates the NeST dependency graph: each higher-layer module depends on its lower-layer modules to forward messages provided to them, and to provide messages to read. Conversely, higher-

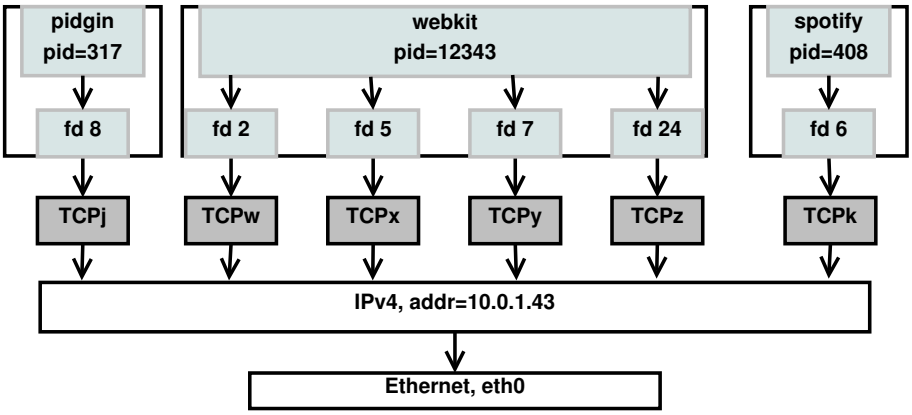


Figure 1.1: Example network stack dependency graph. Application and socket modules are shown in light grey, TCP instances in dark grey, and IPv4 and Ethernet in white. Application modules are made up of component socket modules.

layer modules must produce messages for lower-layer modules to send, and consume messages as they are received.

NeST is designed as an aid for system administrators and developers to debug and correct the causes of network-related performance stalls that affect end-user performance. A series of controlled experiments using real applications shows that NeST is 99% effective at diagnosing performance stalls due to the application (whether from bugs or resource contention), TCP retransmission timeouts, and excessive network congestion, with a false positive rate under 3%.

Chapter 4 describes the second application of the FlowDiagnoser approach. This tool, called StreamsDiagnoser, diagnoses the source of performance stalls in InfoSphere Streams, a distributed real-time stream processing engine created by IBM [24]. Stream processing engines are designed to continuously update query results, transform data, and make decisions based on information as it flows through a series of processing steps. An example Streams application is shown in Figure 1.2.

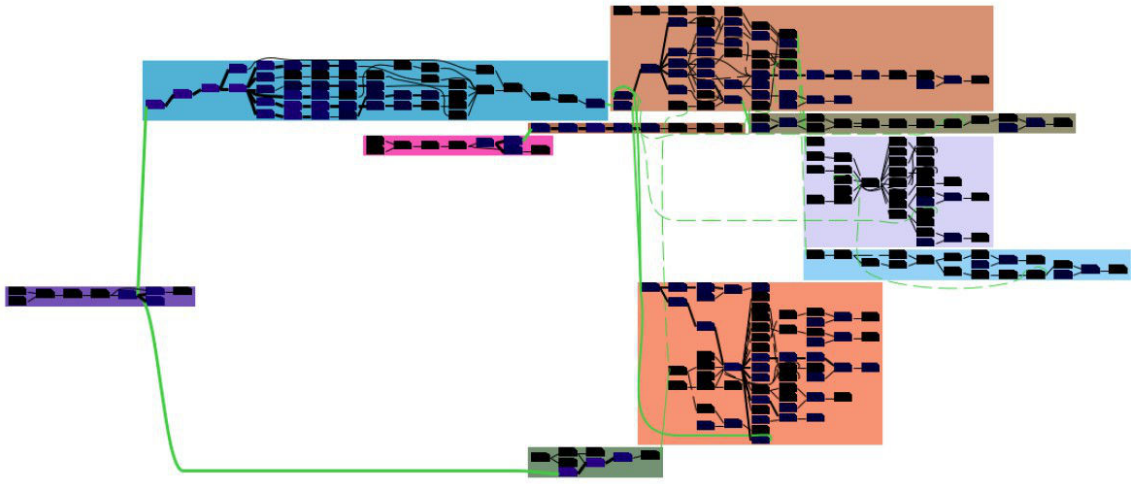


Figure 1.2: Example Streams Application showing the flow of messages between processes; a logical view is also available.

Despite extensive tools for debugging [25] and visualizing performance information [18], Streams users and researchers run into performance problems that are hard to diagnose, since problems in one part of the system can quickly propagate to others. Synthetic benchmarks and instrumentation of real applications show that StreamsDiagnoser is 93% accurate in attributing the source of performance stalls lasting more than two snapshot periods.

As FlowDiagnoser monitors a system over time, it develops a series of diagnosis results which are assigned to each module in the system. Chapter 5 presents an approach for analyzing these diagnosis results, using several summarization and visualizations that FlowDiagnoser outputs. Chapter 6 discusses other approaches for finding the source of performance problems in networked and distributed systems, and Chapter 7 concludes with directions for future work.

1.7 Contributions

This dissertation describes a low-cost, general approach for detecting and diagnosing transient performance stalls in networked and distributed applications. This approach is:

- *Automatic* and requires no user intervention
- *Efficient* as it relies only on commonly available counters, with little access to historical data.
- *Accurate* at diagnosing the source of transient performance stalls before they result in higher-level timeouts.
- *General*: it is useful for detecting performance stalls in both an end host's networking stack and modern streams-processing systems.

FlowDiagnoser is the first performance diagnosis system that provides a general, automated approach that applies to both network-related performance and distributed system messaging, and specifies the minimum amount of information required for diagnosis.

Chapter 2

The FlowDiagnoser Approach

This chapter describes the FlowDiagnoser approach to finding performance stalls in networked and distributed systems. It consists of three parts, illustrated in Figure 2.1:

1. Obtain the dependency graph which describes the movement of messages through the system, discussed in Section 2.1.

2. Periodically snapshot counters for each module in the graph to determine each module's behavior. This is described in Section 2.2.

3. After each snapshot, perform a dependency analysis over the graph and counters to diagnose performance problems as described in Section 2.3.

The output of the dependency analysis is an annotated graph, where each module is labeled with a diagnosis: it is healthy and performing well, blocked by one or more of its dependents, faulty and blamed for blocking other modules from making progress, or its performance is immaterial. The resulting output is described in Section 2.4.

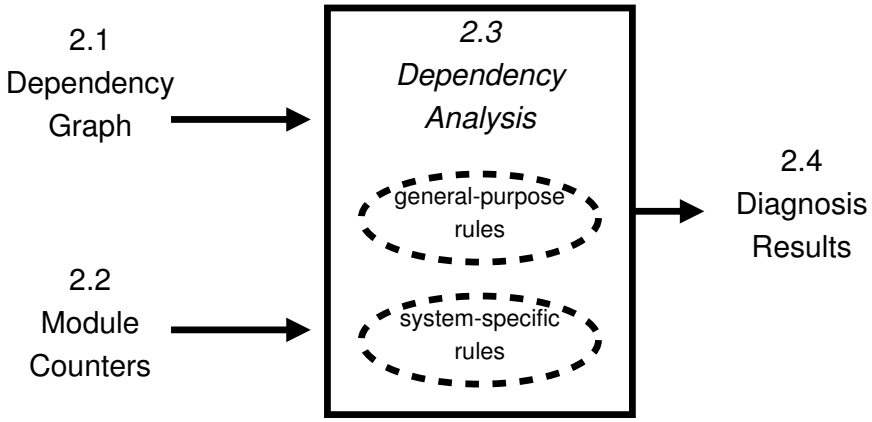


Figure 2.1: The diagnosis process. Module counters are used to describe each module's activity, and combined with the dependency graph to determine the effect of each module's behavior. The dependency analysis consists of general-purpose and system-specific rules. The resulting performance diagnosis is an annotated graph.

2.1 The Dataflow and Dependency Graphs

In the FlowDiagnoser model, a system can be viewed as a *dataflow graph*, where nodes represent *modules* that process messages, and directed edges identify flows of messages between modules. Each edge is a lossless, finite-capacity pipe with exactly one module at each end. Each module has a finite *work queue* of messages that it must process; during processing it may transmit messages to other modules. Messages enter the system via *sources* (which have no incoming edges) and leave the system via *sinks* (which have no outgoing edges).

A system may be either *push-oriented* or *pull-oriented*.[1] In the former, dataflow is driven by the source modules. A source module A connected to module B will produce messages and attempt to write them to the pipe that connects it to B. Since this pipe has finite capacity, A's write may block; in this case, B is required

[1]Individual flows within a system could be either push- or pull-oriented, but such generality is not required for the two applications of FlowDiagnoser, described in Chapters 3 and 4.

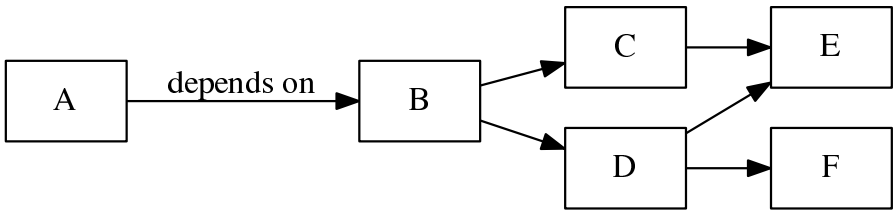


Figure 2.2: Example dependency graph. A depends on B for messaging service, which in turn depends on C and D.

to read messages from the pipe (depositing them into its own work queue) before A can write further messages. In a push-oriented system, if sources are not producing messages, then the system is idle, but this is not necessarily a problem.

In *pull-oriented* systems, dataflow is initiated by sinks. A sink module A connected to B will try to read messages from the pipe that connects the two. If B fails to produce data for A, then A will block. In a pull-oriented system, if sinks are not trying to read messages, then they have no need of data so it need not be provided.

For purposes of diagnosis FlowDiagnoser employs a system *dependency graph* which bears close relation to the dataflow graph. In this graph, nodes are modules, and edges identify dependencies: $A \rightarrow B$ indicates that A *depends on* B to provide it with messaging service. *Root* modules have no incoming dependency edges.

For push-oriented systems, dependency corresponds to dataflow. Since dataflow originates at sources, if A sends to B, then A depends on B to read messages from the connecting pipe so that A does not block. For pull-oriented systems, dependency is the reverse of dataflow. Here, a sink A depends on upstream node B to have data ready when A asks for it; if B has not produced data, then A cannot make progress when it wants to.

Figure 2.2 illustrates an example dependency graph. When capturing dependencies for a push-oriented system, all data in this system originates on the left side at the *root* module A, which is the message source. Messages flow from left to right through the graph to the message sinks on the right side (modules E and F). Module A produces messages for B to consume and pass on to C and D. A depends on B to consume the messages it has produced. If this graph were capturing dependencies for a pull-oriented system, then data would originate at E and F, flowing in the opposite direction of the edges in the graph, driven by sink A.

Given an edge $A \rightarrow B$ in the dependency graph, A is the *parent* of B, which is the *child* of A. A module's *ancestors* are its parents, and recursively all of its parents' parents, i.e., all modules along the paths leading from the module back to the roots of the dependency graph. A module's *descendants* are its children, and recursively all of its children's children. Intuitively, a module M's ancestors include all nodes along paths from the roots to M, and its descendants are all modules from M to the sinks. In Figure 2.2, module F's parent is D, and its ancestors are $\{D, B, A\}$. Module E has two parents (C and D) and four ancestors $\{A, B, C, D\}$. Neither module has children or descendants. Module B's descendants are $\{C, D, E, F\}$.

At a high level modules can represent whatever system elements the designer feels are important. A module in the graph could represent a piece of code that explicitly sends and receives messages, an end host, or even an abstraction of a complex system which is itself made up of many subelements. While in many cases the dataflow graph is a multi-rooted tree, or *directed acyclic graph* (DAG), FlowDiagnoser does not require this. The dataflow (and dependency) graph may

Counter	Description
`total_msgs`	Total messages emitted (required)
`wait_time`	Total time spent waiting for service.
`queued_msgs`	Current number of messages in the work queue

Table 2.1: Basic FlowDiagnoser module counters.

change over time, so the monitoring system observes and incorporates these updates to the graph.

2.2 Module Counters

Once FlowDiagnoser has derived the dependency graph, it diagnoses the system's behavior by periodically snapshotting (up to) three counters associated with each module, shown in Table 2.1:

- `total_msgs` counts the cumulative number of messages that a module has processed (and thus it increases monotonically);
- `wait_time` counts the cumulative time (increasing monotonically) a module has spent blocked waiting on its dependents to produce a message for it to read (in a pull-oriented system) or to consume messages it has produced (in a push-oriented system);
- `queued_msgs` tracks the length of the module's work queue.

FlowDiagnoser uses these counters to determine two pieces of information:

1. whether a module is actively processing messages, and

2. whether a module *attempted* to process messages (or has something to do)

For `total_msgs` and `wait_time` counters, FlowDiagnoser considers the *difference* between the current snapshot's value and the prior snapshot's value, denoted as $\Delta_{\texttt{total\_msgs}}$ and $\Delta_{\texttt{wait\_time}}$. These differences indicate whether the module was active (a nonzero $\Delta_{\texttt{total\_msgs}}$) or was blocked waiting for service (a nonzero $\Delta_{\texttt{wait\_time}}$). It uses the current snapshot value of the module's work queue ($\sigma_{\texttt{queued\_msgs}}$) to determine whether the module still had work to do when the period ended.

Each module in the system must implement `total_msgs`. In general, either `wait_time` or `queued_msgs` must be supported by the root modules (furthest ancestors) in the dependency graph; this indicates whether the system as a whole should be active. These two counters provide a signal of whether the modules have work to do or are waiting on their dependents, and is transitively propagated from parents to children as needed.

2.2.1 Snapshots

To be effective, FlowDiagnoser must take snapshots relatively frequently (from 50 ms to 10 seconds), but must not harm the performance of the monitored system. As such, FlowDiagnoser does *not* require a consistent stop-the-world snapshot of the entire graph: the counters can be read from each module one after the other while the system continues to run. Doing so is better for the monitored system,

but gives rise to potential inconsistencies: according to the counters FlowDiagnoser sees, a module could appear to have consumed more messages than its producers have ever provided to it.

FlowDiagnoser also does not require that each module in the system count messages in the same way: a single one-megabyte write by an application is counted as one message, no matter how many IP datagrams it is turned into. It may also experience race conditions while reading a single module's counters. Chapter 3 and 4 discuss the implications of such anomalies.

While the actual implementations are timescale agnostic—i.e., they process on a per-snapshot basis, no matter the frequency—snapshots must be frequent enough to find short performance stalls. However, they should not be so frequent that normal timing variations (e.g., inter-packet gaps) cause it to issue spurious warnings.

The following rule of thumb is useful: snapshot intervals should be longer than normal (acceptable) pauses in communication, but at most half of the duration of the longest stall the system designer is willing to tolerate. For example, if a 200 ms stall is considered unacceptable, snapshots should be taken at least every 100 ms: otherwise a 200 ms stall could straddle two subsequent snapshots and be missed entirely. However, if modules are normally quiet for 150 ms at a time, a 100 ms interval may generate false positives during normal operation.

Whatever the snapshot interval, reading each module's counters serially (as opposed to in one stop-the-world atomic transaction) can cause false positives. Assume that for a given snapshot, the child's counters are actually read at time t, and the parent's at time $t + \epsilon$. If all of the parent's messages were emitted between time

t and $t + \epsilon$, these messages cannot be counted in the child's counter (read at time t). Thus it may appear that the child did not process messages the parent provided, and the child may be blamed erroneously.

These problems can largely be avoided by ignoring transient one-snapshot stalls, and requiring a two-snapshot confirmation of any positive diagnosis. The diagnosis summary outputs described in Chapter 5 highlight transient and consecutive-snapshot stalls separately. The evaluation criteria used in Chapter 3 and Chapter 4 treat each diagnosis individually.

2.3 Dependency Analysis

The diagnosis algorithm takes the current dependency graph and counter values as inputs and assigns to each module in the graph one of four possible *diagnosis results*:

- HEALTHY, which indicates that the module is actively processing messages;
- DONTCARE, which indicates that the module was inactive because it had no work to do, and its performance did not adversely affect other modules in the system;
- BLOCKED, which indicates that the module attempted to make progress, but was prevented from doing so by another module; and
- STALLED, which indicates that the module is faulty and is the cause of system performance problems.

These diagnosis results cover the four stall-related performance categories: either a module is active (HEALTHY) or not. If it is inactive, it either has no work to do (DONTCARE) or it should be active. If should be active, its performance is either due to another module's fault (BLOCKED) or its own (STALLED). The following two subsections discuss the criteria used to arrive at these diagnoses, and the FlowDiagnoser algorithm itself.

2.3.1 Diagnosis Criteria

Table 2.2 lists the diagnosis assigned to each module in the stack after each snapshot. The first column lists the result and its description, while the second column lists the criteria FlowDiagnoser uses to confer that diagnosis. The criteria are labeled with superscript letters, which are referenced in the discussion. Each result is mutually exclusive; a module is assigned exactly one of these results per snapshot.

2.3.1.1 Active modules

FlowDiagnoser assumes that a module that has processed any messages ($\Delta^{M}_{\texttt{total\_msgs}} > 0$) is providing adequate service to its dependents, and labels it as HEALTHY (criterion *(a)*): it is not the source of a stall.[2]

[2]There are, of course, cases where a system designer would like to find the source of *bottlenecks* in which some modules are actively processing data, but at rates that limit (rather than prevent) the progress of other modules. In this case, it may make sense to apply some other diagnosis result, even though the module is active. We defer such extensions to future work.

Module Diagnosis	**Diagnosis Criteria**
HEALTHY Module is active and processing messages	$[\ (\Delta^{M}_{\texttt{total\_msgs}} > 0)\ ]^{(a)}$
DONTCARE Module has completed its work, or ancestors are fine	$[\ (\Delta^{M}_{\texttt{total\_msgs}} = 0)\ \textit{and}\ \neg\text{HasWork(M)}\ ]^{(b)}$
BLOCKED Module cannot process messages due to another's fault	$[\ (\Delta^{M}_{\texttt{total\_msgs}} = 0)\ \ \textit{and}\ \ \text{HasWork(M)}\ ]^{(c)}\ \ \textit{and}$ $[\ (\Delta^{M}_{\texttt{wait\_time}} > 0)^{(d)}\ \textit{or}$ $(\texttt{wait\_time}\ \text{unsupp.}\ \textit{and}\ \text{CanPassBlame(M)})^{(e)}$ $]$
STALLED Module is not processing messages and is blocking others	$[\ (\Delta^{M}_{\texttt{total\_msgs}} = 0)\ \ \textit{and}\ \ \text{HasWork(M)}\ ]^{(f)}\ \ \textit{and}$ $[\ (\Delta^{M}_{\texttt{wait\_time}} = 0)^{(g)}\ \ \textit{or}$ $(\texttt{wait\_time}\ \text{unsupp.}\ \textit{and}\ \neg\text{CanPassBlame(M)})^{(h)}$ $]$

Table 2.2: FlowDiagnoser Diagnosis Criteria. For each snapshot, each module M in the dependency graph is labeled with one of the diagnoses in the first column, according to the criteria in the second.

In all other cases the module is *inactive* ($\Delta^{M}_{\texttt{total\_msgs}} = 0$), and the analysis must determine if the module should be active.

2.3.1.2 Determining if a module has work to do

When a module M is inactive, FlowDiagnoser first determines if the module has work to do by checking the `queued_msgs` counter and M's location in the dependency graph. This is the HasWork(M) function used in criteria *(b)*, *(c)*, and *(f)*, which returns true if one of the following conditions is met:

1. M supports the `queued_msgs` counter, and $\sigma^{M}_{\texttt{queued\_msgs}} > 0$;

2. M does not support the `queued_msgs` counter, and has a parent that is BLOCKED; or

3. M does not support the `queued_msgs` counter, and is a root in the dependency graph.

If M supports the `queued_msgs` counter, FlowDiagnoser knows whether the module has work to do. If ($\sigma^{M}_{\texttt{queued\_msgs}} > 0$), there are still messages in M's work queue; otherwise M is inactive for a good reason: it had no work to.

If M does not support `queued_msgs`, FlowDiagnoser infers whether the module should be active by checking the module's parents. If one of its parents is BLOCKED, then FlowDiagnoser assumes that the inactive module had work to do. Note that this assumes that M's parents have already received a diagnosis, which implies a topological ordering on the graph.

Finally, if M does not support `queued_msgs` and is a root in the dependency graph, FlowDiagnoser assumes that M should be active. This means that an inactive root module will always be diagnosed as STALLED unless it can pass the blame to one of its descendants.

When an inactive module has no work to do (criterion *(b)*), FlowDiagnoser marks the module as DONTCARE. This indicates that M's inactivity did not have an adverse affect on any of its parents.

Otherwise, the analysis must determine why M did not complete its work: either the module is blocked by one of its children (meeting criteria *(d)* or *(e)*), or M itself is stalled (meeting criteria *(g)* or *(h)*).

2.3.1.3 Passing the blame

Since M depends on its children for service, M may be inactive because it is waiting for one of its child modules for service. In push-oriented flows it may be waiting for a child to consume a message it has provided, or in pull-oriented flows for the child to produce a message for it to read.

If M can pass blame to one of its children, FlowDiagnoser marks M as BLOCKED (criteria *(d)*, *(e)*). Otherwise, M itself is responsible and FlowDiagnoser marks M as STALLED (criteria *(g)*, *(h)*). FlowDiagnoser makes this determination by looking at the module's `wait_time` counter if it is supported; otherwise it checks the counters of M's children.

The `wait_time` counter. The total time the module has spent waiting in a messaging-related operation is accumulated in the `wait_time` counter, which increments as the module is blocked waiting to complete that operation. When the module supports the `wait_time` counter, the diagnosis process is simple. If the module has processed no messages ($\Delta^M_{\texttt{total\_msgs}} = 0$), but attempted to and was blocked by its child ($\Delta^M_{\texttt{wait\_time}} > 0$), FlowDiagnoser marks it as BLOCKED (criterion *(d)*).[3]. Otherwise, the module spent no time waiting ($\Delta^M_{\texttt{wait\_time}} = 0$), and has not processed any messages because it is idle; FlowDiagnoser therefore marks it as STALLED (criterion *(g)*).

[3]There is another case, in which $\Delta^M_{\texttt{total\_msgs}} > 0$ and $\Delta^M_{\texttt{wait\_time}} > 0$. FlowDiagnoser is concerned with stalls, so it marks the module as HEALTHY; a more fine-grained analysis might determine that the module's performance was limited by a bottleneck.

The CanPassBlame() function. If M does not support the `wait_time` counter, FlowDiagnoser can infer whether M is blocked by (one of) its children or is itself stalled; this is the CanPassBlame() function. It returns true if *any* child of M is inactive and has work to do. Since it is called only when M is either blocked or stalled, the function does not need to call the full HasWork() function for each child: it already knows that each child has a parent that is blocked (M). Therefore, CanPassBlame() returns true if any child C is inactive and does not have an empty work queue, that is, when $\Delta^{C}_{\texttt{total\_msgs}} = 0$ and either `queued_msgs` is unsupported or $\sigma^{C}_{\texttt{queued\_msgs}} > 0$.

While the simple pass-the-blame rule described here works in many cases, a monitored system may require a different set of criteria based on its communication semantics. Section 3.3.2 describes a slightly modified, system-specific rule employed when monitoring the network stack.

2.3.2 Diagnosis Algorithm

The diagnosis criteria described above assume that FlowDiagnoser can consult both a module's parents (and transitively, back to the roots of the dependency graph) to determine whether a module has work to do, and also consult a module's children to determine if they are the source of the module's performance problems. This process is easy enough when the graph is acyclic and there is a topological ordering that allows the diagnosis process to visit parents before their children, but is not entirely straightforward when the graph includes cycles.

When a module is part of a cycle in the graph (e.g., $A \rightarrow B \rightarrow C \rightarrow A$), checking its parents' states is problematic: is module C responsible for providing service to A (via B), or is A responsible for providing service to C? If all three modules are inactive, should they be marked as BLOCKED, STALLED, or DONTCARE? It is possible that this cycle has created a deadlock, and no module is able to act until its child has made progress.

In addition, the HasWork() function assumes that a module's parents are diagnosed before the module itself. This is necessary when a module is inactive to determine if the module should be active, by checking whether its parents are blocked, and recursively back to the roots of the graph. This implies a topological ordering on the graph so that ancestors can be visited before their descendants, which requires that the graph be acyclic.

Algorithm 1 FlowDiagnoser algorithm

Require: dgraph (original dependency graph), counters
1: wgraph ← BreakNonBlockedCycles(dgraph, counters)
2: dag ← MergeStronglyConnectedComponents(wgraph)
3: diagnosis ← DependencyAnalysis(dag, dgraph, counters)
4: **return** dag, diagnosis

The FlowDiagnoser diagnosis algorithm, shown in Algorithm 1, performs the following steps:

Step 1: Traverse the graph in random order, breaking any cycles that include modules known to be HEALTHY or DONTCARE. This step is shown in Algorithm 2.

Step 2: After this first pass, all modules that are still part of a cycle are known to be inactive. Since every strongly connected component in a directed graph

is either a single module or a cycle of modules, standard techniques (such as Tarjan's algorithm [50]) can be applied to merge these cycles of inactive modules into super-modules that represent the cycle [17].

Step 3: Use the resulting Directed Acyclic Graph (DAG) to perform the dependency analysis shown in Algorithm 3. The original dependency graph is needed only to determine if a module is a root in the original graph.

Step 4: Return the resulting DAG and diagnosis results.

2.3.2.1 Step 1: Break Cycles

Algorithm 2 BreakNonBlockedCycles

Require: dgraph (original dependency graph), counters
1: wgraph ← dgraph
2: **for all** $M \in$ wgraph **do**
3: **if** $\left(\Delta^{M}_{\texttt{total\_msgs}} > 0\right)$ **then**
4: ▷ Module is HEALTHY and does not propagate blame
5: remove $outEdges_M$ from wgraph
6: **else if** (M supports `queued_msgs`) $\wedge \left(\sigma^{M}_{\texttt{queued\_msgs}} = 0\right)$ **then**
7: ▷ Module has no work and does not propagate blame
8: remove $outEdges_M$ from wgraph
9: **end if**
10: **end for**
11: **return** wgraph

The first step of the diagnosis algorithm is to break any cycles that include modules that do not propagate a waiting indication. The BreakNonBlockedCycles() function shown in Algorithm 2 transforms the original dependency graph into an intermediate working graph (wgraph). This function visits each module in random order (Line 2) and removes all outbound edges from modules that are active (Lines 3–

5) or have no work to do (Lines 6–8). Note that this function *does not* access information from any module's parents or children. The resulting graph has no cycles that include modules that are known to be active or DontCare. Removing these edges does not adversely affect the dependency analysis, since any *parent* that is active or DontCare does not pass any blame to its children. The reasons for this are rather subtle, and are discussed further in Section 2.3.2.4.

2.3.2.2 Step 2: Merge Inactive Cycles

The second step of the extended diagnosis algorithm is to merge the remaining (inactive) cycles into super-modules [17]. Merging cycles of inactive modules is preferable to diagnosing them separately for two reasons:

a. A cycle of inactive modules indicates a potential deadlock, and there is no general way to determine which module in the cycle originally caused the problem. Diagnosing deadlocked modules as a group gives a clear indication that the whole group is mutually dependent.

b. All the modules in one cycle may be blocked by a separate module or cycle downstream. Merging the cycles preserves this relationship.

To indicate this relationship, both the cycle-free DAG and diagnosis results are returned from the diagnosis algorithm in Algorithm 1 (Line 4). The follow-on presentation can apply each cycle's diagnosis to the individual modules, and also indicate that the module was part of a deadlocked cycle.

The counters used for the resulting super-modules are the maximum of the component modules' $\Delta_{\texttt{total\_msgs}}$, $\Delta_{\texttt{wait\_time}}$, and $\sigma_{\texttt{queued\_msgs}}$. Since any active modules are by definition *not* part of one of these cycles, the cycle itself is inactive $\left(\Delta^{cycle}_{\texttt{total\_msgs}} = 0\right)$. Any $(\Delta_{\texttt{wait\_time}} > 0)$ or $(\sigma_{\texttt{queued\_msgs}} > 0)$ will cause the super-module's counter to be greater than zero. This gives an appropriate waiting indication (if any) for the cycle as a whole.

2.3.2.3 Step 3: Dependency Analysis

The third step of the algorithm is to perform the dependency analysis shown in Algorithm 3. Since all cycles in the dependency graph have been broken by removing modules' out-edges (step one) or merging cycles into super-modules (step two), the resulting graph is a directed acyclic graph, and FlowDiagnoser can obtain a topological ordering of the modules (Line 3). As stated in Table 2.2, an active module is HEALTHY (Line 7, criterion *(a)*).

For all inactive modules, the analysis then checks if the module has work to do (Line 9), and marks it as DONTCARE if not (Line 10, criterion *(b)*). Otherwise, the module should have been active.

If the module supports `wait_time`, FlowDiagnoser marks it as BLOCKED if it was waiting (Lines 13–14, criterion *(d)*) and STALLED if not (Lines 15–16, criterion *(g)*). Otherwise, if the module can pass blame to its descendants in the DAG, it is blocked (Lines 19–21, criterion *(e)*); if not it is STALLED (Lines 22–24, criterion *(h)*).

Algorithm 3 DependencyAnalysis

Require: dag, dgraph (original dependency graph), counters

▷ Diagnose each module in the DAG.
▷ Cycles of blocked/stalled modules will be diagnosed as a group.
for all $M \in$ TopologicalSort(dag) **do**
  ▷ TopologicalSort visits modules from ancestors to descendants.
  ▷ By this point, all of M's parents have been diagnosed.
  if $\left(\Delta^{M}_{\texttt{total\_msgs}} > 0\right)$ **then** ▷ Module is active.
    diagnosis[M] ← HEALTHY
  else ▷ Module is inactive.
    if ¬HasWork(M, dag, dgraph, diagnosis) **then** ▷ No work to do.
      diagnosis[M] ← DONTCARE
    else ▷ Module has work to do.
      if (M supports `wait_time`) **then**
        if $\left(\Delta^{M}_{\texttt{wait\_time}} > 0\right)$ **then** ▷ Module tried to read/write.
          diagnosis[M] ← BLOCKED
        else ▷ Module was idle.
          diagnosis[M] ← STALLED
        end if
      else ▷ M does not support `wait_time`
        if CanPassBlame(M, dag) **then**
          ▷ Pass the blame.
          diagnosis[M] ← BLOCKED
        else
          ▷ No child is blocked or stalled.
          diagnosis[M] ← STALLED
        end if
      end if
    end if
  end if
end for
return diagnosis

Algorithm 4 HasWork

Require: module M, dag, dgraph (original dependency graph), parents' diagnoses

```
1: ▷ Return true iff module M has work to do.
2: if (M supports queued_msgs) then
3:    if (σ^M_queued_msgs > 0) then
4:       return true
5:    else
6:       return false
7:    end if
8: else if (M is a root module in original dependency graph) then
9:    ▷ Assume root modules always have work.
10:   return true
11: else if ( ∃P ∈ dag.getParents(M) s.t. (diagnosis[P] = BLOCKED) ) then
12:   ▷ Assume there is work if a parent in the DAG is blocked.
13:   return true
14: else
15:   return false
16: end if
```

The HasWork() function The HasWork() function shown in Algorithm 4 takes four parameters: a module M, the DAG, the original dependency graph, and the diagnoses for M's parents, which have already been completed since the DAG is traversed in topological order. To determine if M has work to complete, it checks `queued_msgs` if it is supported and returns the appropriate response (Lines 3–7). Next, if the module is one of the root modules (furthest ancestors) in the original dependency graph (Lines 8–10), FlowDiagnoser assumes it always has work to do.[4].

If the test on Line 11 is reached, the module is a non-root module (Line 8 is false) that does not support `queued_msgs` (Line 2 is false). The HasWork() function then checks to see if any of M's parents are BLOCKED (Lines 11–13); if they are, it assumes that M has work to do. Otherwise, M's inactivity did not affect its parents

[4]Note that this check is made only if the `queued_msgs` counter is not supported by the (root) module

during this snapshot, so the module is assumed to have nothing to do (Line 15) which results in it being marked DontCare in the dependency analysis.

If M is not a root in the original dependency graph, but has no parents in the DAG, the condition in Line 11 of HasWork() is always false. This is appropriate, since any in-edges that were removed in the call to BreakNonBlockedCycles() were from parents that were either active or had no work to do, i.e., they are known not to be Blocked.

Algorithm 5 CanPassBlame

Require: dag, blocked/stalled module M, counters
1: ▷ Determine whether blame can be passed from M to some child.
2: **for all** child ∈ dag.getChildren(M) **do**
3: **if** $\left(\Delta^{\text{child}}_{\texttt{total\_msgs}} = 0\right)$ **then**
4: ▷ Child is inactive.
5: **if** (child supports `queued_msgs`) $\wedge \left(\sigma^{\text{child}}_{\texttt{queued\_msgs}} = 0\right)$ **then**
6: ▷ Child has no work to do.
7: **continue**
8: **else** ▷ Pass the blame to/through the inactive child.
9: **return true**
10: **end if**
11: **end if**
12: **end for**
13: ▷ Did not find any child to pass blame to.
14: **return false**

The CanPassBlame() function The final piece of the dependency analysis is the CanPassBlame() function shown in Algorithm 5. This function is called only when the module M is inactive (Algorithm 3, Line 19), and FlowDiagnoser needs to determine whether it can pass the blame to one of M's children. Therefore, it returns true if *any* of the children is *potentially* Blocked or Stalled. It is important to

note that this function *does not* need to determine exactly why the child is blocked, i.e. if the child can pass its blame on to one of the grandchildren.

If a child is inactive (Lines 3–11), the function checks whether the child has an empty work queue (Line 5), and if so, proceeds to the next child (Line 7). Otherwise, the inactive child either does not support `queued_msgs` or has something in its queue. In both cases, the parent can pass blame to the child since the child is inactive (Line 9).

Finally, if no child is blocked or stalled, the function returns false (Line 14). This occurs when there are no children to check (children = $\emptyset$), all of the children are active (Line 3 is false), or all of the inactive children have nothing in their work queue (Lines 5–7).

2.3.2.4 Discussion

To remove cycles from the original dependency graph, each module is first visited at random, and outgoing edges from certain modules are removed. This breaks any cycles that involve active or DontCare modules. Removing these particular edges that originally existed between the parent modules and their children is valid for two reasons:

1. The original parent modules are either active (diagnosed in Line 7 of Algorithm 3) or known to be DontCare (Line 10), and do not need to check their children to pass blame. Therefore, the out-edges add nothing to the parents' diagnosis.

2. The only time children need to check their parents' state is in Lines 11–13 of the HasWork() function (Algorithm 4). However, since none of the original parents were BLOCKED (they are either HEALTHY or DONTCARE), the original parents add nothing to the children's diagnosis. The missing in-edges are therefore unnecessary to the children.

Once the first pass is performed to break some cycles, any remaining cycles of inactive modules are merged together into super-modules. This grouping may indicate possible deadlock among the modules in the cycle. The remaining DAG is traversed in topological order (ancestors first), and each module is diagnosed before its descendants. By doing this, any information that needs to be propagated from the roots of the dependency graph to the descendants (namely, whether or not the module should be active) is available by the time each module is visited. The topological sort also ensures that each module is visited at least once during the DependencyAnalysis.

Overall, each module in the original dependency graph is visited at least twice, and at least three times if it is not part of a cycle:

1. Once in BreakNonBlockedCycles(), with time complexity $O(|modules|)$;

2. At least once in Tarjan's algorithm to find and merge strongly connected components, which has time complexity $O(|modules| + |wgraph\ edges|)$ [50];

3. Visiting each (DAG) module once in Line 3 of DependencyAnalysis(), with complexity $O(|modules|)$;

4. Potentially visiting each module's parents in Line 11 of HasWork(), with worst case complexity $O(|edges|)$; and

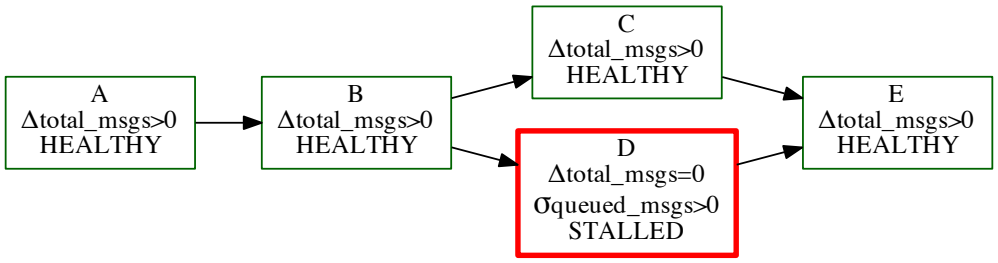
5. Potentially visiting each module's children in CanPassBlame(), with worst case complexity $O(|edges|)$.

Therefore, the overall worst-case time complexity of the FlowDiagnoser algorithm is $O(|modules| + |edges|)$. While the number of modules in the DAG may be less than in the original dependency graph, such a reduction cannot be expected.

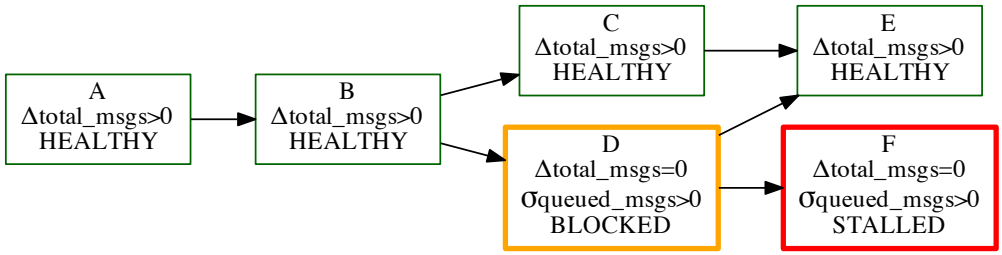
2.3.3 Correlating evidence

One beneficial feature of the FlowDiagnoser algorithm is that when determining whether an inactive module M is Blocked or Stalled, it takes into account correlations among multiple flows. To see this, consider the example dependency graphs in Figure 2.3. In both subfigures, A is able to pass messages along the path from $A \rightarrow B \rightarrow C \rightarrow E$, but module D is inactive ($\Delta^{D}_{\texttt{total\_msgs}} = 0$).

In Figure 2.3(a), FlowDiagnoser has a signal that D's parent is partially blocked: $\sigma^{D}_{\texttt{queued\_msgs}} > 0$. Since D is not processing messages sent to it, but its children are still active ($\Delta^{E}_{\texttt{total\_msgs}} > 0$), then D is at fault. In the network stack, several TCP connections may be doing fine (and thus providing messages for the IP module to forward) while one TCP connection is stalled due to congestion further out in the network. In this case, there is enough information to correctly locate the stall's source.



(a) Example dependency graph in which module D is STALLED and not forwarding traffic. This is identical to Figure 2.2 with module F removed.



(b) The dependency $D \rightarrow F$ indicates that D is in fact BLOCKED by its child F.

Figure 2.3: Example dependency graphs with performance stalls. Traffic is able to flow along the edges from $A \rightarrow B \rightarrow C \rightarrow E$, but is blocked from $B \rightarrow D$. In subfigure (a) (with module F removed), the activity at D's child module E indicates that D is at fault. In subfigure (b), D is blocked by its other child module, F.

Figure 2.3(b) has an additional dependency to consider, from $D \rightarrow F$. If module E or F stop processing altogether, their incoming work queues may fill and block their parents from sending new messages; this phenomenon is known as *backpressure*. In this case, D is BLOCKED by its child F. Therefore FlowDiagnoser marks F as STALLED.

2.4 Diagnosis Results

This diagnosis process is performed on the entire graph for each snapshot taken of the modules' counters. This means that over time, FlowDiagnoser creates a series of diagnoses that show the status of each module in the system. To summarize the diagnosis results over the entire monitoring period, FlowDiagnoser provides several visualization and reporting outputs:

- A per-module diagnosis summary which explains how often a module was diagnosed as HEALTHY, BLOCKED, STALLED, or DONTCARE, and the average and maximum duration of the periods it was STALLED, described in Chapter 5.
- A time series visualization that shows the diagnosis provided to each module over time, as explained in Section 3.5.4.
- A time series visualization of the module counters, with a separate heatmap provided for each counter, also explained in Section 3.5.4. This helps an expert user to see when a module was HEALTHY, BLOCKED, STALLED, or its performance did not matter (DONTCARE).

2.5 Summary

The FlowDiagnoser approach to diagnosing performance stalls in distributed systems consists of three parts: a dependency graph which describes the relationships between the modules of the system, counters used to make an initial assessment of module performance, and a dependency analysis performed to determine

each module's health. Modules are diagnosed as HEALTHY and processing messages, BLOCKED by another module, STALLED and preventing other modules from making progress, or labeled as DONTCARE since their inactivity does not affect overall system health.

The following two chapters describe the application of this approach in two different environments. The first, discussed in Chapter 3, is a system called the Network Stack Trace (NEST), which automatically diagnoses software and network-related performance stalls by instrumenting an end host's networking stack. The second, StreamsDiagnoser, is a prototype diagnosis engine for detecting and locating performance stalls in a multi-host, multi-process distributed stream-processing system; this is described in Chapter 4.

Chapter 3

Diagnosing Problems in the Network Stack

This chapter describes the *Network Stack Trace* (NEST), an instantiation of the FlowDiagnoser approach to the task of detecting the source of stalls in networked applications running on an end host.

In this setting, the modules are the layers of the network stack: the (whole) application, its sockets, and the protocol endpoints (TCP, IP, and physical device) that send/receive data over the network. Together they comprise the *network stack dependency graph* (NS graph). The process used to acquire and maintain this graph is described in Section 3.1.

NEST obtains module counters from the operating system and by specially instrumenting applications' read and write calls, as explained in Section 3.2. The dependency analysis, described in Section 3.3, extends FlowDiagnoser by applying a NEST-specific rule that distinguishes between network-wide and connection-specific performance stalls.

The prototype data collection and diagnosis engine described in Section 3.4 is used in a series of controlled experiments to instrument real applications. These results, presented in Section 3.5, show that NEST is 99% effective at detecting performance stalls due to the application (whether from bugs or resource contention), TCP retransmission timeouts, and excessive network congestion, with a false posi-

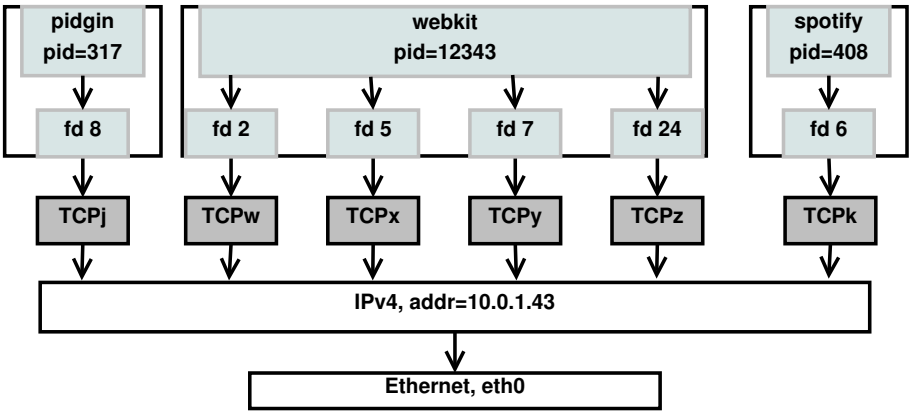
tive rate under 3%. Section 3.6 discusses potential extensions to the NEST model, and Section 3.7 provides a summary of the results.

The NEST diagnosis provides a succinct summary of the health of each module in the host's network stack, and if run online could enable applications and users to mitigate performance problems in real time. For example, the user or software could restart a TCP connection or select a different remote server to avoid connection-specific performance problems, change their wireless channel or transmit rate to achieve better performance, or restart their browser process if it has stalled.

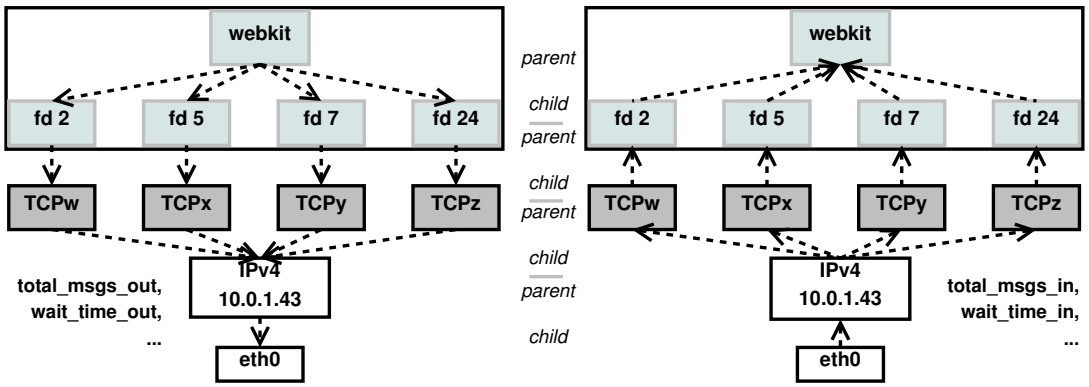
3.1 The Network Stack Dependency Graph

Each element of the host's network stack appears as a module in a *network stack dependency graph* (NS graph). In particular, modules consist of (1) running applications (one module per application); (2) each of an application's sockets (one module per socket); (3) the TCP state for each socket; (4) each IP source/destination an application is communicating with; and (5) each physical device through which communication is taking place.

Figure 3.1(a) shows an example NS graph. The three active applications (pidgin, webkit, and spotify) are connected via sockets with various file descriptors (FDs) to their respective transport-layer (TCP) connections. Each TCP connection is bound to the same local IPv4 address `10.0.1.43` and Ethernet interface `eth0`. If an application in this scenario appeared stalled—e.g., webkit was playing a video that has frozen—then NEST should diagnose the source of the problem as being lo-



(a) Entire host network stack dependency graph.



(b) Outbound dataflow graph for webkit.

(c) Inbound dataflow graph for webkit.

Figure 3.1: Example NS dependency graph, showing the individual modules. Application and socket modules are shown in light grey, TCP instances in dark grey, and IPv4 and Ethernet in white. Application modules are made up of component socket modules. Subfigures (b) and (c) show the dataflow graphs related to webkit only; the dataflow graph for outbound flows (left) and inbound flows (right) have the same shape, but the directed edges are reversed.

calized to one of the above modules, thus blaming the application, TCP's behavior, IP-level connectivity, or the physical network device.

In NEST, applications drive all traffic on the host: dataflow is push-oriented for outbound traffic and pull-oriented for inbound traffic. Therefore, NEST uses the same NS graph for both kinds of flows.

Figure 3.1(b) shows the dataflow graph for outbound (push-oriented) flows sent from the local host. For outbound flows, the diagnosis rules assume that children—the lower layers in the network stack—will attempt to process and emit all messages that are produced unless they detect problems such as congestion. On the other hand, for inbound (pull-oriented) flows received by local applications, shown in Figure 3.1(c), the rules assume that no messages need to be produced unless the sink (parent) is actively consuming them. The diagnosis implementation for both is the same: the focus on parents' and ancestors' (lack of) requests for service.

As detailed in Section 3.4, NeST instruments an end host's network stack to track applications, sockets, connections, and interfaces and their dependencies as they come and go. As applications and connections open and close, NeST tracks the changes to the graph and updates it accordingly. At configurable intervals, NeST snapshots the entire host's network counters and uses them to make a diagnosis.

3.2 Network Stack Counters

The counters used by NeST are given in Table 3.1. Each module in the NS graph has a counter for the number of outbound and inbound messages processed, called `total_msgs_out` (abbreviated `tmo` in the figures) and `total_msgs_in` (`tmi`), respectively. For the TCP, IP, and physical device modules, NeST populates these counters using operating system-provided information. For applications and sockets, it acquires the message counters using custom instrumentation.

Counter	Abbr.	Description
`total_msgs_out`	`tmo`	Total messages sent (required)
`wait_time_out`	`wto`	Total time spent blocked while writing to child
`total_msgs_in`	`tmi`	Total messages received (required)
`wait_time_in`	`wti`	Total time spent blocked while reading from child

Table 3.1: NeST module counters.

Since application socket modules are the sources in the NS graph, for them NeST keeps two additional counters, `wait_time_out` (`wto`) and `wait_time_in` (`wti`), to track time spent waiting to send outbound traffic, and to receive inbound traffic, respectively.

NeST does not track the queuing behavior of modules, so the `queued_msgs` counter from Table 2.1 is not used. Earlier implementations attempted to accurately track the number of bytes waiting in a TCP connection's queues, but this was unreliable [27,28] or too slow for frequent snapshots in the versions of the operating system used [29].

3.2.1 Application and socket modules

To track application behavior, NeST sums the counters of the application's sockets.

To track per-socket message counts, NeST increments the `total_msgs_out` counter for each call to message-sending system calls (e.g., `write`, `connect`, and `send`) using the socket, and the `total_msgs_in` counter for each call to message-

receiving calls (e.g., `read` and `recv`). It does this by intercepting calls using an LD_PRELOAD interceptor library, described in Section 3.4.

NeST implements `wait_time_out` and `wait_time_in` for sockets in the same interceptor library: it accumulates the amount of time (in milliseconds) spent blocking in the underlying call. To support non-blocking calls, it also tracks the time spent in `poll()`, `select()`, and their equivalents. Note that NeST does not increment the message counter until after a blocking call completes. In the expected case, every socket will accumulate a bit of waiting time during each snapshot, even when it sends messages successfully. For example, a single-gigabyte `write()` call may take most of a snapshot interval to complete, incrementing `wait_time_out` by 100s of milliseconds and `total_msgs_out` by 1.

3.3 Stack Dependency Analysis

NeST follows the diagnosis algorithm given in Section 2.3 to declare each module as either Healthy, Stalled, Blocked, or DontCare. NeST makes one customization to this algorithm, described shortly. The following subsection describes how the diagnosis results should be interpreted in NeST.

3.3.1 Interpreting the results

Since in practice each module in the NS graph is a piece of code, it seems intuitive to interpret a diagnosis result of Stalled as indicating a bug of some kind. However, with the exception of application or socket modules, it is unlikely

for stalls to be the result of bugs inside the operating system's network stack, which is generally well-tested.

When a particular socket is marked STALLED, this means that the application has not attempted to read (or write, depending on flow direction) on this particular socket. It is possible that the application keeps open long-lived sockets, but reads or writes on them only when a user clicks on a link or types a message, so a STALLED socket does not necessarily indicate a problem; it does mean that the underlying network modules are not expected to provide any service. A BLOCKED socket indicates that the application attempted to communicate over the socket, but either the outbound write buffer was full or the inbound receive buffer was empty.

Since an application's counters are the sum of its sockets' counters, if any socket is HEALTHY, then the application is said to be HEALTHY; a more conservative option that marks the application as HEALTHY only when *all* its sockets are healthy may be more useful in some situations, but this would require a change in how NEST accounts for application counters. If the application is not HEALTHY, then if any of its sockets are BLOCKED the application is BLOCKED. This means that the application did attempt to communicate on at least one of its sockets and was unable to. Finally, if an application does not attempt to communicate on *any* of its sockets, then the application itself is said to be STALLED. As with sockets, the usefulness of this result depends on the application semantics.

For transport-layer connections such as TCP, a diagnosis of DONTCARE means that its socket is idle and not attempting to read (or write, depending on flow direction). A STALLED diagnosis means that an application attempted to use the

connection, but was blocked and unable to make progress; it may be that the remote application did not read or write any data, or the path between the two hosts was congested.[1] A BLOCKED diagnosis means that desired progress was blocked by a lower-level or network-wide congestion event.

Either diagnosis could be interpreted in terms of TCP's estimate of the (virtual) queue available to it in the network. A TCP sender will stop transmitting when it thinks the network or receiver has a full buffer; its receive queue empties when the network or remote sender is unable to provide enough messages for it to process. When many connections detect congestion simultaneously, there is a broader problem: all the (virtual) queues appear to be blocked.

For lower-layer network modules such as IP, and Ethernet or wireless interfaces, when any transport-layer connection has attempted to write across the network (outbound) or has received any data (inbound), the IP endpoint and Ethernet interface modules are also active and are marked HEALTHY. This is not particularly precise, since *any* active connection automatically removes any blame from these low-level modules, even if the connections were to hosts in the local subnet. On the other hand, when no transport-layer connections are able to obtain service from an IP address endpoint and the interface is silent, both low-level modules are diagnosed as STALLED, regardless of how many connections were blocked. This assumes that some application was active or blocked; if all the applications are idle (or there

[1] A protocol-specific analysis may be able to distinguish between these two (for example, by looking at the TCP advertised window), but NEST does not.

are no applications running), these lower-layer modules are marked as DontCare. Possible improvements are discussed in Section 3.6.1.

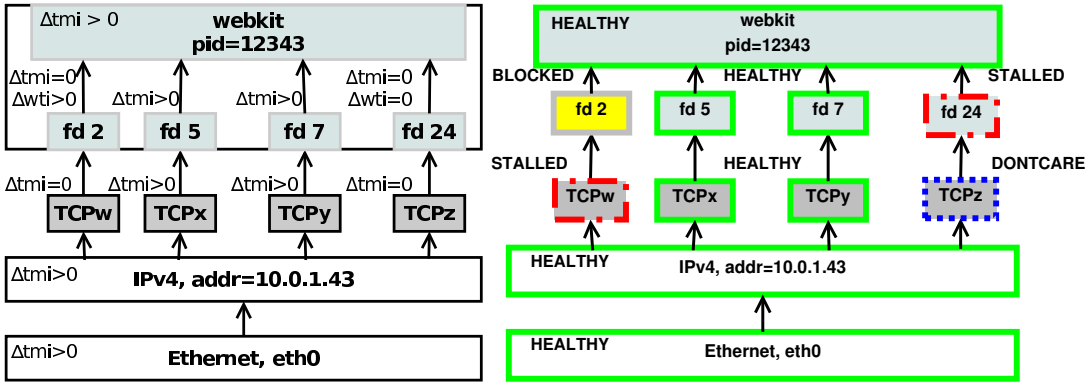
3.3.2 Distinguishing connection-specific and network-level faults

The dependency analysis NeST employs is essentially the same as that described in Section 2.3: a module is considered Healthy whenever it is active ($\Delta_{\texttt{total\_msgs}} > 0$), and NeST can directly diagnose application and socket modules since they export the `wait_time` counter. The diagnosis process therefore needs to consider only the cases where some other module is inactive.

An end host often has multiple network connections active at the same time: users may be listening to a music stream while browsing the web, uploading photos, or instant messaging. Each additional flow provides additional clues about the source of stall. The following sections consider how multiple flows aid in diagnosis, and the rule variant used to distinguish between connection-specific and network-wide performance stalls.

When a *single* connection independently experiences performance problems, as seen in Figure 3.2, NeST is able to easily locate the source of the problem using the standard algorithm given in the previous chapter.

In this example, the `webkit` application has four open connections. As shown in the inbound dataflow graph and counters for the `webkit` (Figure 3.2(a)), `webkit` is reading from three of its sockets (`fd 2`, `fd 5`, and `fd 7`), and its fourth is idle (`fd 24`, $\Delta_{\texttt{tmi}} = 0$ and $\Delta_{\texttt{wti}} = 0$). Note that two of the connections (`TCPx` and



(a) Webkit graph with (abbreviated) inbound counters.

(b) Inbound diagnosis results for webkit graph.

Figure 3.2: Example counters and diagnosis results for the `webkit` inbound dataflow graph. In subfigure (a), the inbound counters are shown to the upper left of each module. `total_msgs_in` is abbreviated as `tmi` and `wait_time_in` as `wti`. Subfigure (b) shows the resulting diagnosis: BLOCKED modules are yellow, HEALTHY modules are outlined with a solid green border, STALLED modules outlined with a red dash-dot border, and DONTCARE modules outlined with a blue dotted border.

`TCPy`) are actively receiving data and their `total_msgs_in` counters are increasing ($\Delta_{\texttt{tmi}} > 0$), and their descendants `10.0.1.43` and `eth0` are providing data to them.

However, `webkit` is unable to receive messages via `fd 2` ($\Delta^{\texttt{fd 2}}_{\texttt{tmi}} = 0$ and $\Delta^{\texttt{fd 2}}_{\texttt{wti}} > 0$), and `TCPw` is inactive ($\Delta^{\texttt{TCPw}}_{\texttt{tmi}} = 0$). Since `TCPw`'s children are active ($\Delta^{\texttt{10.0.1.43}}_{\texttt{tmi}} > 0$), the fault lies with `TCPw` and NeST marks it as STALLED.[2] The resulting diagnosis output is shown in Figure 3.2(b).

When *multiple* connections experience problems simultaneously and no traffic is being transmitted, the pass-the-blame rule described in Section 2.3.1.3 automatically absolves the parent modules (marking them as BLOCKED), and passes the blame to the lowest-level descendant that is inactive (marking it as STALLED). This is a good general-purpose rule, but can lead to some unsatisfying results.

[2]This is due to criteria *(f)* and *(h)* in Table 2.2.

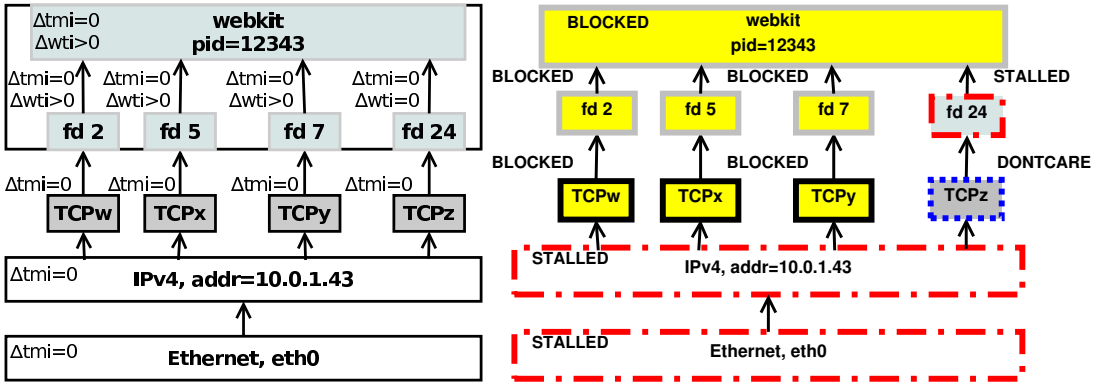
In the network stack, higher-layer modules will defer transmissions if they find the lower layers to be unreliable (i.e., they detect that the lower layers are not delivering their packets to the remote host). Hence, it seems reasonable to pass the blame to an inactive child module. However, lower layers of the stack can only pass on what has been provided to them by their parent, and blindly blaming the lower layer module(s) may cause NeST to misplace the fault.

When *any* transport-layer (TCP) connection is active, the IP module will be active as well, since IP is a best-effort forwarding service and performs no buffering. So, when IP is not sending (receiving) datagrams, it must be because *all* of the active connections are experiencing performance problems at the same time.[3]

In this case, NeST cannot tell with certainty whether the underlying IP network is experiencing a major congestion event or outage, or whether each individual TCP connection is experiencing its own unique, independent fault. However, it seems reasonable to assume that *many* TCP connections should not experience independent faults simultaneously.

To see how NeST takes advantage of this observation, consider Figure 3.3(a). This example is similar to the one in the previous figure, but in this case all of the TCP connections are inactive ($\Delta_{\texttt{tmi}} = 0$), and `fd 2`, `fd 5`, and `fd 7` are all waiting on service ($\Delta_{\texttt{tmi}} = 0$ and $\Delta_{\texttt{wti}} > 0$). The question is: is it more appropriate to blame the Ethernet module, IP module, TCP modules, or some combination? The simple pass-the-blame rule would mark only the `eth0` interface as Stalled, and the rest as Blocked.

[3]Any connections with idle readers/writers are marked as DontCare and ignored.



(a) Stalled traffic dataflow graph with counters.

(b) Diagnosis results graph, using $\Theta = 2$.

Figure 3.3: Sockets `fd 2`, `fd 5`, and `fd 7` are waiting on service, and NeST needs to determine whether the TCP connections or the lower-layer network is at fault. Note that `TCPz` is also inactive, but `fd 24` has not attempted to read from it. Since there are at least $\Theta = 2$ connections with ancestors waiting for service, and the lower layer is inactive ($\Delta_{\texttt{total\_msgs}} = 0$), NeST assumes that a lower layer networking problem is causing the TCP connections to block. Blocked modules are yellow, Stalled modules outlined with a red dash-dot border, and DontCare modules outlined with a blue dotted border.

A simple heuristic works well in practice. Select a threshold Θ number of connections; if the number of connections experiencing simultaneous stalls is greater than or equal to Θ, mark the TCP connections as Blocked, and the underlying network module(s) as Stalled. For more complicated graphs, this process continues as the diagnosis proceeds down through the graph. Otherwise, there is not enough evidence to blame only the network, so NeST marks *both* the TCP connections and the lower-layer modules as Stalled, since it is impossible to distinguish between the two possibilities given the evidence available; a reasonable alternative is to blame the higher-layer module(s) only. The resulting diagnosis is shown in Figure 3.3(b).

Note that when the network is quiet, but there is only one TCP connection waiting for service, NeST marks both the TCP connection module and the underlying modules (IP, Ethernet, et cetera) as Stalled.

Section 3.6.1 discusses possible improvements to the precision of the results.

3.4 Data Collection Prototype

The NeST Linux prototype for end-host performance monitoring consists of three subsystems:

- a *collector* daemon that constructs the NS graph, takes periodic snapshots of module counters, and stores them to a database for post-processing;
- an *interceptor* library that tracks application counters and provides them to the collector; and
- the *diagnosis engine* which analyzes the snapshots.[4]

The prototype architecture is shown in Figure 3.4.

To monitor the various network protocols, simple adapters read the implementation-specific counters and present them in the common counter format shown in Table 2.1. For TCP connections, NeST uses the counters exported by the Web100 kernel patch [39]. IPv4 counters are gathered by reading `/proc/net/snmp`,

[4] While the diagnosis engine can be run in real-time as part of the collector, the implementation has not been tuned to do this. All evaluation was performed using the diagnosis engine in offline mode to post-process the collected data.

Ethernet interface counters via `netlink` sockets [41], and wireless interfaces via device- and driver-specific interfaces.

To track application behavior, NEST intercepts socket-related calls to `libc` using an LD_PRELOAD interceptor library which records the number of calls made and the time spent in them in a shared memory segment. The library also notifies the collector of new and closing applications and sockets by sending one-way messages to the collector's Unix domain socket, which allows the collector to read socket statistics from the shared memory (shm) area without interfering with the application.[5]

The collector takes a snapshot every 50–100 ms. As discussed in Section 2.2.1, this data collection architecture does not block the application when a snapshot is being taken. Indeed, it is subject to race conditions, since the interceptor library may be updating a counter in the shared memory area while the collector daemon is reading it.

The main consistency requirement is that each module's `total_msgs` and `wait_time` counters be monotonically increasing. If they ever decrease (usually due to a race condition while reading the 64-bit value), the delta is invalid and NEST discards the second snapshot. If the third snapshot is also lower than the first, NEST assumes that the first snapshot was a spurious increase, discards it, and continues from there. In practice that these invalid snapshots are extremely rare,

[5]Applications could also be instrumented using a general-purpose tracing library such as LTTng [51] or DTrace [12], or a kernel-level Linux Security Module (LSM) [55]. Earlier experiments with PTrace [42] proved to be problematic, largely due to implementation complexity [16].

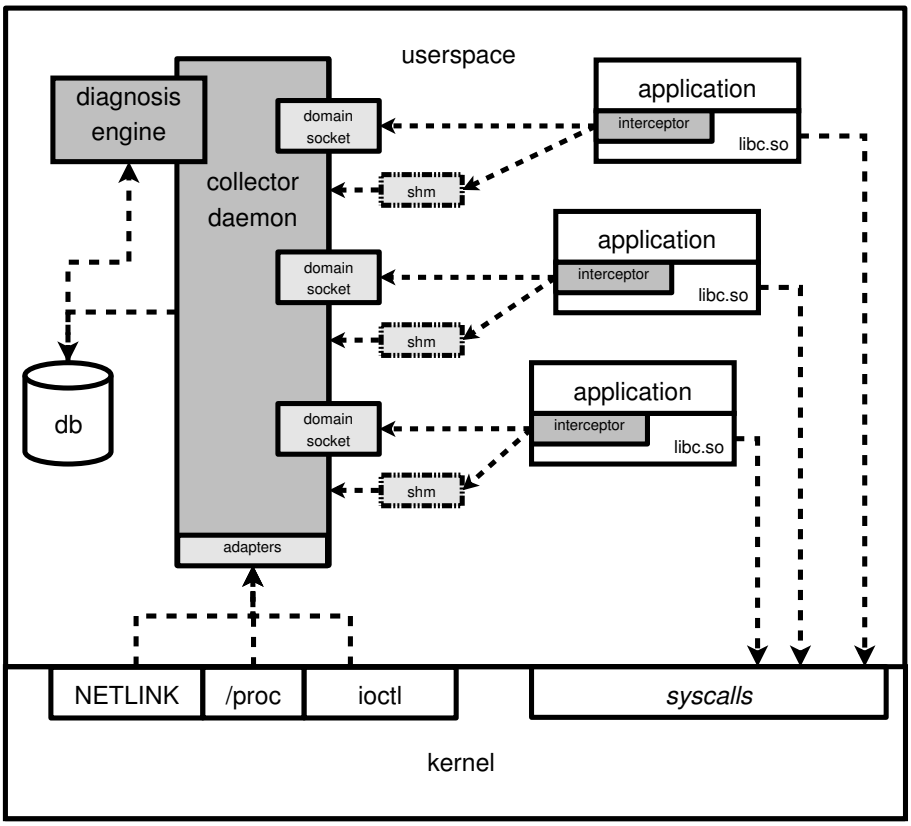


Figure 3.4: Network Stack Trace collection architecture

but the estimates may be optimistic since the transfer rates were low (less than 100 Mbps). Since NEST snapshots counters fairly frequently, a few discarded snapshots have little impact.

3.5 Experimental Results

To evaluate NEST's accuracy, we used it to diagnose faults injected into sample programs and network flows in a controlled setting. Out of the thousands of runs performed, this section presents results from a representative series of experiments. These experiments cover all of the fault scenarios and also control groups run without

faults. In the test scenarios, NEST correctly diagnosed the faulty module more than 99% of the time, while incorrectly blaming modules only 3% of the time.

Section 3.5.1 describes the experimental setup, and Section 3.5.2 presents results from the accuracy evaluation. Section 3.5.3 assesses the prototype efficiency and potential improvements, and Section 3.5.4 discusses results of some illustrative diagnosis scenarios.

3.5.1 Experimental setup

All experiments were performed on Emulab [53] using the topology shown in Figure 3.5, varying the types of applications running (download-only, upload-only, or simultaneous upload/download) and the number of simultaneous connections. In these experiments, NEST instruments the network stack on host `site1n1`, and faults are injected on `site1n1`'s network connections and applications.

For download tests, we used `wget` version 1.12 to download a 100MB file twice in succession from an Apache version 2.2.3 web server. For upload tests, we used `iperf` version 2.0.5, which uses a separate thread for each connection to the remote `iperf` server. Since `wget` downloads each request serially, we use multiple instances of `wget` and `iperf` to generate background download traffic when testing the effect of simultaneous connections.

Each series of tests included a control group running normally, plus experiments with randomly injected faults targeting the network and applications. These faults include:

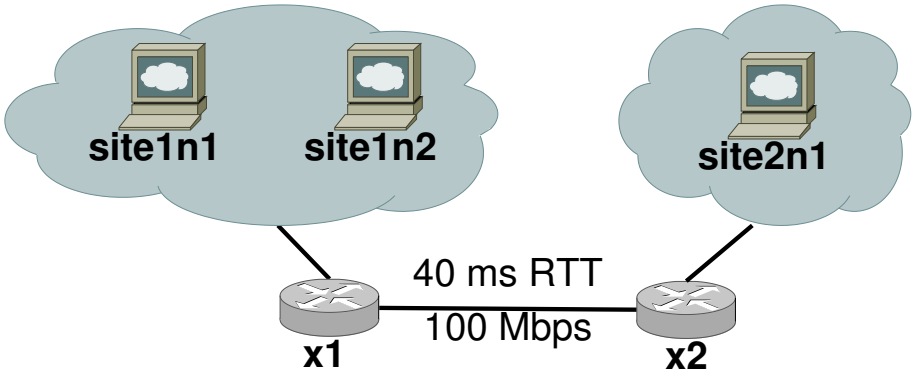


Figure 3.5: Emulab experiment topology. Two sites are connected by a single 100 Mbps bottleneck link between routers `x1` and `x2`. We inject faults by pausing applications, and dropping all packets on specific TCP connections and IP flows on router `x1`.

1. Pausing the application to force it to stop reading and writing from sockets

2. Dropping packets on certain TCP connections

3. Dropping packets on certain IP-to-IP flows

To inject application faults, a Unix signal is sent to the application process, which sets a global variable; the global variable is cleared when another signal arrives to end the fault period. Before and after each `read()` or `write()` call, each reader/writer thread checks this global variable in a loop; if it is set, the thread sleeps for 10 ms and then checks again. While in this loop, its socket should be marked STALLED. When all the threads stop reading/writing, the application should be marked STALLED as well.

To inject faults on TCP connections, we insert a firewall rule on the local gateway router `x1` to drop all packets matching the connection's five-tuple (source address, destination address, protocol=TCP, source port, destination port). We then remove the rule to let communication proceed normally.

To simulate network-wide faults, we insert a firewall rule on `x1` to drop all packets to or from `site1n1`'s local IP address. All of the host's traffic is crossing the bottleneck link in the experiments, so a single rule is sufficient.

Since we were unable to reliably inject layer-two faults using Emulab, the IP endpoint and Ethernet modules are combined together, labeled `ip+eth`. This combined module uses the counters from the Ethernet module, since only one IP address is assigned to the experimental interface. Thus any faults injected at the IP layer simulate a network-wide event.

3.5.2 Diagnosis accuracy

In the unloaded test network, a module should be marked STALLED (receive a positive diagnosis) only when it is targeted for a fault; this is the expected/correct positive result. Otherwise the correct diagnosis is negative (the null hypothesis). Application sockets are an exception to this rule, since their diagnosis depends on the type of traffic the application is performing. Since they send little or no outbound trafic, download sockets should always have a STALLED (positive) outbound diagnosis; conversely upload sockets should have a STALLED (positive) inbound diagnosis.[6] The same holds for the applications themselves.

The following section evaluates the ability of NeST to detect these injected faults. According to the evaluation criteria, the correct result is positive (STALLED) in two cases: (1) Upload-only or download-only applications and sockets are idle

[6] This ignores the initial request traffic (e.g. HTTP GET), but the number of relevant periods should be insignificant.

in the opposite direction of packet flow. (2) A module is unresponsive when it is *affected* by an injected fault. In all other cases, the correct result is negative (HEALTHY, BLOCKED, or DONTCARE); in general, each module should receive a clean bill of health.

3.5.2.1 Reading the diagnosis table

Table 3.2 lists the abbreviations used in the diagnosis accuracy table, which is shown in Table 3.3. In Table 3.3, NEST diagnosis results are grouped in rows by flow direction (inbound, outbound, and combined total) and then listed by module type: `iperf` application, `wget` application, sockets (broken down by application), TCP and `ip+eth`. There are four groups of columns:

- **Correct Answers**. This lists the Total number of diagnosis periods for each module type, along with the number of Actual Positive (AP) and Actual Negative (AN) periods expected according to the evaluation criteria. Note that the evaluation criteria ignore `wget`'s outbound request traffic, and always expect `wget` and its sockets to be marked as STALLED in the outbound direction (outbound AN column = 0).

- **NeST Diagnosis Results**. This lists the number of diagnoses that NEST assigned to each module, broken down by how well they matched the evaluation criteria ("Correct Answers"):

  - **True Positives (TP)**: NEST correctly marked the module as STALLED (i.e., a NEST positive was an Actual Positive).

  - **True Negatives (TN)**: NeST correctly did *not* mark the module as Stalled (i.e., a NeST negative was an Actual Negative).
  - **False Positives (FP)**: NeST incorrectly marked a module as Stalled, but the correct answer was negative (i.e., a NeST positive was an Actual Negative).
  - **False Negatives (FN)**: NeST erroneously marked a module as Healthy, Blocked, or DontCare while a fault was injected: the correct answer was positive, but NeST did not detect it properly (i.e., a NeST negative was an Actual Positive).

- **Positive Accuracy %**: The percentage of the time that a NeST positive diagnosis (Stalled) was correct:
  - **True Positive Rate (TPR)**: The percentage of the Actual Positive periods NeST correctly detected as Stalled (TP/AP).
  - **False Positive Rate (FPR)**: The percentage of the Actual Negative periods NeST incorrectly marked as Stalled (FP/AN).
  - **Positive Predictive Value (PPV)**: The percentage of NeST positive diagnoses that were actually correct ($TP/(TP+FP)$).

- **Negative Accuracy %**: The percentage of the time that a negative NeST diagnosis (Healthy, Blocked, or DontCare) was correct:
  - **True Negative Rate (TNR)**: The percentage of the Actual Negative periods NeST correctly detected (TN/AN).

Abbr.	Name	Explanation
Total	Total diagnoses possible	Count of snapshot deltas with valid measurements
AP	Actual Positive periods (known positives)	Number of measurement periods where a fault was active
AN	Actual Negative periods (known negatives)	Number of measurement periods where a fault was *not* active
TP	True Positive diagnoses	Count of our positive diagnoses that were also Actual Positives
TN	True Negative diagnoses	Count of our negative diagnoses that were also Actual Negatives
FP	False Positive diagnoses	Count of our positive diagnoses that were Actual Negatives
FN	False Negative diagnoses	Count of our negative diagnoses that were Actual Positives
TPR	True-Positive Rate (Sensitivity)	% of Actual Positives (AP) that were correctly diagnosed
FPR	False-Positive Rate	% of Actual Negatives (AN) that were False Positives (FP)
PPV	Positive Predictive Value (Precision)	When the diagnosis is positive, what % of the time is it correct?
TNR	True-Negative Rate (Specificity)	% of Actual Negatives (AN) that were correctly diagnosed
FNR	False-Negative Rate	% of Actual Positives (AP) that were False Negatives (FN)
NPV	Negative Predictive Value	When the diagnosis is negative, what % of the time is it correct?

Table 3.2: Abbreviations for evaluation tables.

– **False Negative Rate (FNR)**: The percentage of the Actual Positive periods NeST incorrectly marked as *not* Stalled (FN/AP).

– **Negative Predictive Value (PPV)**: The percentage of NeST negative diagnoses that were actually correct ($TN/(TN + FN)$).

	Module	Correct Answers Total	AP	AN	NeST Diagnosis Results TP	TN	FP	FN	Positive Accuracy % TPR	FPR	PPV	Negative Accuracy % TNR	FNR	NPV
Inbound	iperf	6367	3454	2913	3433	2898	15	21	99.3	0.5	99.5	99.4	0.6	99.2
	wget	19918	965	18953	965	18949	4	0	100.0	0.0	99.5	99.9	0.0	100.0
	socket	51687	22885	28802	22866	28777	25	19	99.9	0.0	99.8	99.9	0.0	99.9
	s:iperf	31769	21920	9849	21901	9828	21	19	99.9	0.2	99.9	99.7	0.0	99.8
	s:wget	19918	965	18953	965	18949	4	0	100.0	0.0	99.5	99.9	0.0	100.0
	TCP	65572	2017	63555	1807	61728	1827	210	89.5	2.8	49.7	97.1	10.4	99.6
	ip+eth	15174	1446	13728	1446	12716	1012	0	100.0	7.3	58.8	92.6	0.0	100.0
		158718	**30767**	**127951**	**30517**	**125068**	**2883**	**250**	**99.1**	**2.2**	**91.3**	**97.7**	**0.8**	**99.8**
Outbound	iperf	6367	138	6229	138	6229	0	0	100.0	0.0	100.0	100.0	0.0	100.0
	wget	19918	19918	0	19879	0	0	39	99.8		100.0		0.2	0.0
	socket	51687	31038	20649	30999	20649	0	39	99.8	0.0	100.0	100.0	0.1	99.8
	s:iperf	31769	11120	20649	11120	20649	0	0	100.0	0.0	100.0	100.0	0.0	100.0
	s:wget	19918	19918	0	19879	0	0	39	99.8		100.0		0.2	0.0
	TCP	65572	1165	64407	1165	63357	1050	0	100.0	1.6	52.6	98.3	0.0	100.0
	ip+eth	15174	879	14295	879	13774	521	0	100.0	3.6	62.7	96.3	0.0	100.0
		158718	**53138**	**105580**	**53060**	**104009**	**1571**	**78**	**99.8**	**1.4**	**97.1**	**98.5**	**0.1**	**99.9**
Total	iperf	12734	3592	9142	3571	9127	15	21	99.4	0.1	99.5	99.8	0.5	99.7
	wget	39836	20883	18953	20844	18949	4	39	99.8	0.0	99.9	99.9	0.1	99.7
	socket	103374	53923	49451	53865	49426	25	58	99.8	0.0	99.9	99.9	0.1	99.8
	s:iperf	63538	33040	30498	33021	30477	21	19	99.9	0.0	99.9	99.9	0.0	99.9
	s:wget	39836	20883	18953	20844	18949	4	39	99.8	0.0	99.9	99.9	0.1	99.7
	TCP	131144	3182	127962	2972	125085	2877	210	93.4	2.2	50.8	97.7	6.6	99.8
	ip+eth	30348	2325	28023	2325	26490	1533	0	100.0	5.4	60.2	94.5	0.0	100.0
		317436	**83905**	**233531**	**83577**	**229077**	**4454**	**328**	**99.6**	**1.9**	**94.9**	**98.0**	**0.3**	**99.8**

Table 3.3: Diagnosis accuracy from controlled NeST experiments; columns are defined in Table 3.2. Top section is for inbound flows, middle section for outbound flows, and combined results at the bottom. Values in the two rightmost sections are percentages. Statistical uncertainty is less than 0.3% for all measurements, except for outbound ip+eth (0.5%), inbound TCP (0.7%), and total TCP (0.4%).

3.5.2.2 NeST evaluation results

The TPR and TNR columns for the application and socket rows in Table 3.3 show that NeST is able to accurately detect faults in applications or individual sockets in well over 99% of the cases, with few False Positives. This is expected, since the `wait_time` counter allows NeST to observe their state directly.

At first glance, the `ip+eth` module and TCP modules appear to have much worse results: while the TPR column indicates that NeST accurately detects almost all of the Actual Positives, it has a significant False Positive Rate (FPR) for inbound `ip+eth` traffic (7.3%) where it appears to blame `ip+eth` incorrectly. It also misses 10.4% of the inbound TCP faults, as seen in the False Negative Rate (FNR) column for inbound TCP.

Likewise, while NeST detects 100% of outbound TCP connections' problems (TPR column), it has a significant number of False Positives (1050 in total, 1.6% False Positive Rate). This leads to a low Positive Predictive Value (PPV) for outbound TCP connections: according to the evaluation criteria, when the diagnosis blames an outbound TCP connection, it is correct only 62.7% of the time.

The main reason for this apparent lack of precision is the inherent ambiguity that arises when the whole network stack is silent, as discussed in Section 3.3.2. There are three main cases which appear in the experiments: when there is a single active flow, when multiple flows unexpectedly experience congestion, and when connection recovery is unexpectedly delayed.

3.5.2.3 One active flow

When only one TCP flow is active and the TCP or `ip+eth` modules are targeted by an injected fault, it is hard (if not impossible) to distinguish between an endemic IP-layer fault and a problem on the single TCP flow. As discussed in Section 3.3.2, in such single-flow situations NEST blames both the TCP and `ip+eth` modules. These are counted as `ip+eth` False Positives when a TCP-specific fault has been injected, and TCP False Positives when an `ip+eth` fault has been injected.

3.5.2.4 Multiple active flows experience congestion

In spite of efforts to limit network congestion, there are times in the experiments when multiple TCP flows experience congestion simultaneously. This can lead to counted False Positives against the `ip+eth` module when all of the TCP connections go silent simultaneously. When this occurs during a TCP fault injection period, it is counted as a TCP False Negative since TCP is absolved by its peers (using $\Theta = 2$). This is more indicative of a drawback in the evaluation criteria than the diagnosis algorithm.

3.5.2.5 Delayed connection recovery

When a network-level fault is injected against the `ip+eth` module, some TCP connections may take longer to recover than others. When other connections become active again, the `ip+eth` module is also active, and any TCP connections that do

not recover are marked as STALLED. Although the diagnosis is probably correct in these situations, these are counted as False Positives in the evaluation.

3.5.3 Prototype efficiency

While the goal is to create a data collection and diagnosis engine that is efficient, the prototype data collector daemon and diagnosis and analysis engine are written in Python, and not highly optimized. Nevertheless, they are efficient enough to use for experimental purposes, even though they record each module's counters in a database for post-processing; an engineered implementation in a low-level language would presumably perform much better.

During the controlled experiments described below, the data collection engine uses on average approximately 50% of one CPU on a 2.4 GHz quad-core Xeon machine. This is largely due to Python interpreter overhead, verbose logging for experimental purposes, and interaction with the backend database system that stores the counters for post-processing.

Similar systems for application and TCP connection logging using Event Tracing for Windows [33] found that application event tracing increased median CPU utilization by 1.6% CPU, and disk utilization by 1.2%. Yu, et al measured the CPU load required to read the full Windows TCP statistics table (similar to the Web100 counters NeST uses) every 50 ms to be 10% at 1000 connections and 30% at 5000 connections [60]. An engineered version of NeST should have similar overhead.

Creating the 158,718 total diagnoses in the 15-minute experiment runs described below takes just over three minutes (183 seconds). Of this time, 31 seconds is spent creating and initializing the diagnosis objects themselves, 27 seconds in filtering out dead modules from the super-graph containing all modules that ever exist, and 15-25 seconds due to inefficiencies in a custom enumerated type implementation. Clearly this can be improved upon.

3.5.4 Diagnosis scenarios

To provide an intuition for how the diagnosis engine works, and to explain some of the unexpected results, we now present three representative runs from the Emulab fault-injection experiments.

3.5.4.1 Network-level fault injection

Figure 3.6 shows two time series plots for a `wget` process (labeled with module #898) which downloads two 100 MB files in succession across the bottleneck link.

The top plot is a counter time series heatmap,[7] and shows the counters for all modules that are descendants of the `wget` process in question (i.e., the relevant subgraph). The counters for each module are plotted on individual rows; for example, `wget` #898 has four counters shown: 1. `total_msgs_in` (packets in), 2. `total_msgs_out` (packets out), 3. `wait_time_in` (read wait ms), and 4. `wait_time_out` (write wait ms).

[7] Section 5.3.2 includes additional discussion.

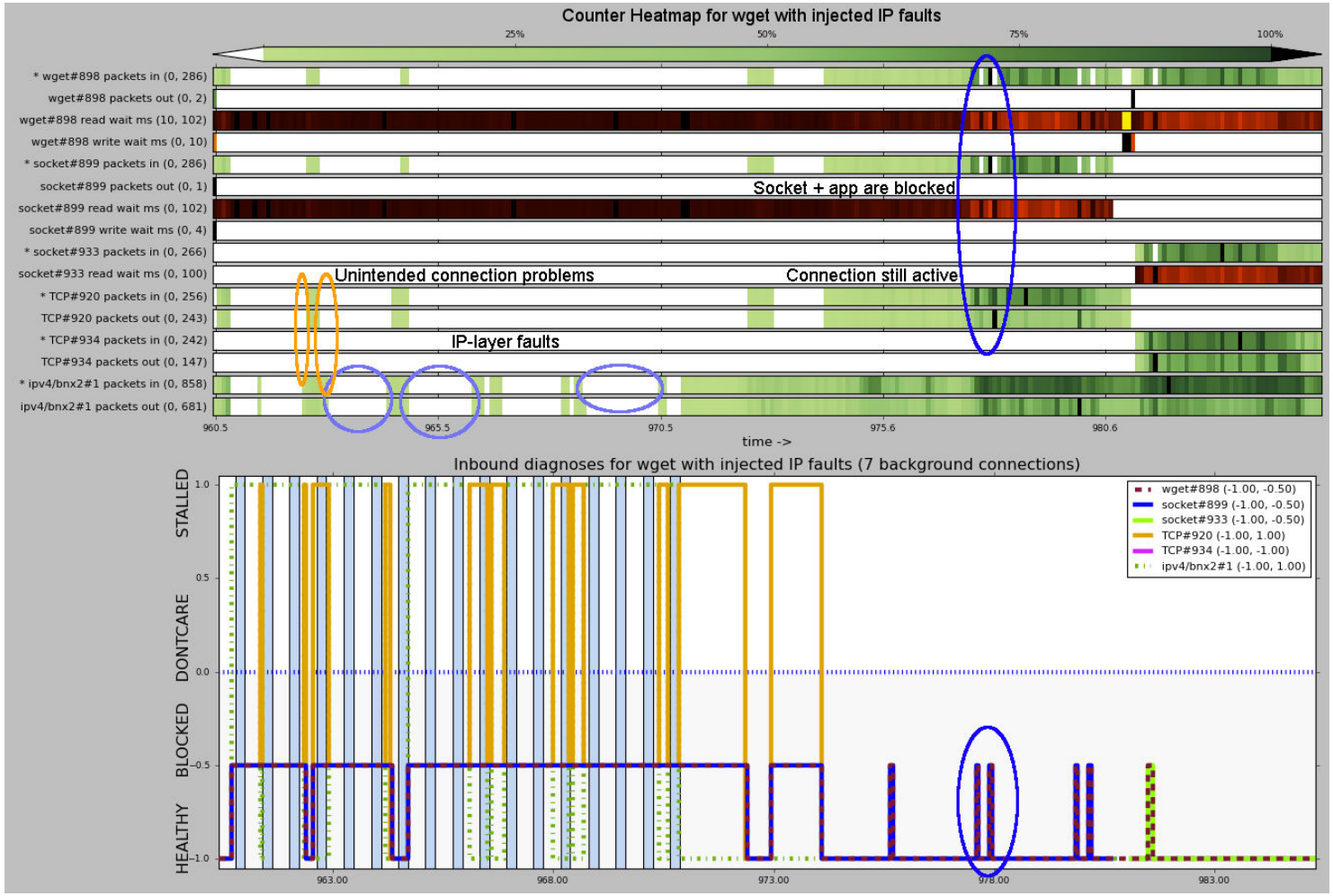


Figure 3.6: Inbound `wget` counters and diagnosis results plot. In addition to the `wget` download connection, there were 7 active background connections (not shown). The shaded blue areas in the bottom plot denote the periods when an IP fault was active. From the white (no activity) areas for the `ip+eth` counters, it is apparent that the fault caused all of the connections to back off for an extended period of time, even after the drop-all fault was removed (unshaded areas).

Each row of the heatmap shows the delta value of the counter during that snapshot. The color green indicates good behavior, and yellow/orange/red denote problems. A counter's row is white when the delta value is zero (good or bad), and black when it hits it maximum (delta) value.

For the `total_msgs_in` and `total_msgs_out` counters, where a higher value is better, the color ranges from white (zero) to light green, dark green, and to black (maximum snapshot delta for that counter). Thus, the darker the shade of green, the more messaging activity the module had during the snapshot.

For the `wait_time_in` and `wait_time_out` counters, where a lower value is better, the color varies from white (zero) through yellow (25% of max), orange (50%), red (75%) to black (maximum snapshot delta for that counter). Thus, a yellow section of the `wait_time_in` row indicates the module was waiting, and black indicates that the module waited as long as it ever did during a snapshot (generally black is the full snapshot duration). Note that the `wait_time_in` and `wait_time_out` rows appear only for application and socket modules.

Ancestor modules are plotted toward the top of the heatmap, and descendants toward the bottom. Although there are four counters possible for every application and socket module, no row is displayed for any counter whose value never changes. For example, the values of the `total_msgs_out` or `wait_time_out` counters never change for `socket#933`, so those rows are missing.

The bottom plot shows a time series of the diagnosis results, in this case the inbound diagnoses assigned to each module in the subgraph. The light blue shaded

sections in the bottom plot show the time periods when a fault was injected to drop *all* IP packets at the host's local gateway router.

In addition to the `wget` process shown, there were seven background connections which are not shown, since they were not strictly part of the `wget` dependency graph. Their traffic, however, is also passed via the `ip+eth` module, so this explains the green shaded areas in the `ip+eth` rows at the bottom of the heatmap, even while `wget` is inactive (e.g., between t965 and t970).

As can be seen from the light-green dash-dotted line in the bottom plot (labeled `ipv4/bnx2`), the `ip+eth` module is blamed whenever all of the connections are quiet. Even after the faults are removed, however, the remote hosts' TCP exponential backoff timers prevent the connections from recovering in many cases, since they have hit repeated retransmission timeouts (RTOs). This is evidenced by the lack of any traffic at the `ip+eth` module between the fault periods, which are marked with blue-shaded rectangles in the bottom plot.

However, there are many occasions when `TCP#920` (orange line in the bottom plot) does not recover after the fault is lifted, even though some of the other background connections recover quickly, as evidenced by the increased traffic on the `ip+eth` module (bottom two rows of the heatmap). As Section 3.5.2.5 discusses, these unintended connection-specific problems due to delayed recovery are counted as TCP False Positives in the accuracy results shown in Table 3.3; we manually confirmed the diagnosis to be correct.

Additionally, once the IP faults are finished, around time t978.0, there are occasional snapshots where `wget` is attempting to read, but it receives no data,

and is marked as BLOCKED (spikes on the right side of bottom plot). Since the underlying TCP connection is still active, however, the connection is marked as HEALTHY. This is not a bad result for NeST to apply; if the application had made a single large read request, it may take several seconds (or minutes) for it to be fulfilled, depending on the underlying speed of the network. So, the application is blocked, but the TCP connection is still providing it with service by filling the incoming buffer. Since no module is marked as STALLED in this case, and the expected result is a negative diagnosis for both modules, this result has no effect on the evaluation criteria.

3.5.4.2 Connection-specific fault injection

Figure 3.7 shows the counter heatmaps and diagnosis time series for an `iperf` upload process (labeled with module #698) which attempts to send ten seconds' worth of data on each of eight (8) simultaneous upload connections.

Faults are repeatedly injected on `TCP#714`, which is providing service to `socket#706`, by inserting a firewall rule at the local gateway `x1` in Figure 3.5 to drop all packets on that TCP connection. The connection-specific faults are injected during the green shaded portions of the bottom plot; `TCP#714` is marked as STALLED (red dashed line in the bottom plot) when it is not transmitting; two relevant periods are highlighted with blue circles in both plots.

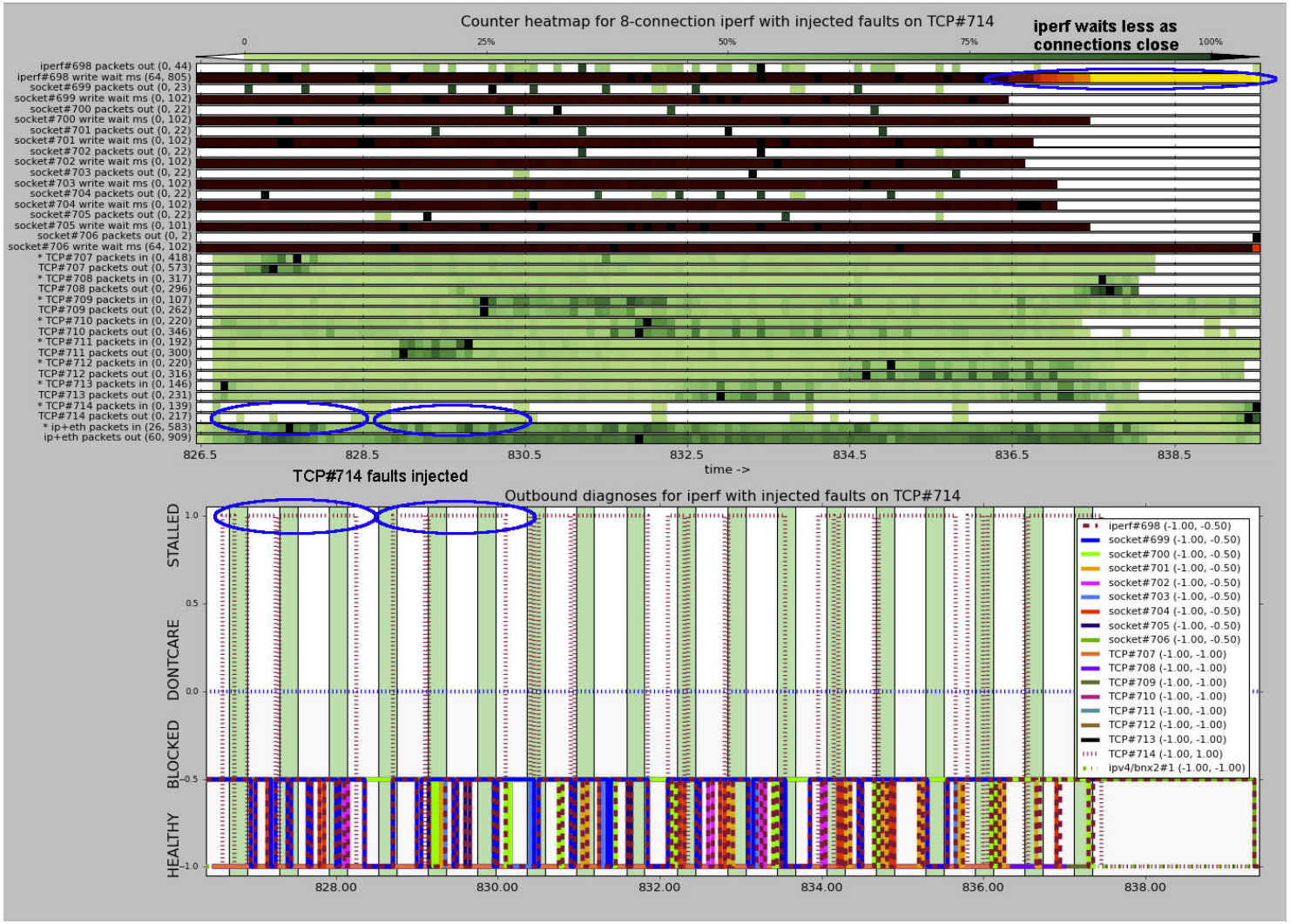


Figure 3.7: Outbound `iperf` counters and diagnosis results plot. The `iperf` process has 8 active outbound connections. One of them, `TCP#714` has faults injected (green shaded areas) and is marked as STALLED. Since the connection does not recover quickly, it continues to be inactive even after the fault is removed. Several of the fault periods are highlighted with blue ovals.

It should be noted that `TCP#714` occasionally recovers, and is marked as HEALTHY; this results in the vertical movement for the red dashed line. (A better visualization may be to incorporate the diagnosis results directly into the heatmap in the top half of the figure.)

Looking closely at the "packets out" rows in the heatmap, it is apparent that the `iperf` process is BLOCKED for much of the time; this is also indicated by the movement between HEALTHY (-1) and BLOCKED (-0.5) in the diagnosis time series at the bottom. When *all* of the sockets are BLOCKED, then the `iperf` application is also BLOCKED, so the maroon dashed line varies as well. Note that since all of the connections are active (with the exception of `TCP#714`), they remain marked HEALTHY. Hence a BLOCKED socket with a HEALTHY connection indicates a serious throughput bottleneck. This is a normal and expected result, and does not affect the evaluation.

Finally, as `iperf` closes its connections, its $\Delta_{\texttt{wait\_time}}$ value decreases over time, since the application counter is the sum of all the sockets' `wait_time` counters. This is shown by the change in coloration from dark red to orange to yellow at the end of the heatmap plot.

3.5.4.3 Application fault injection

An example of a faulty application is shown in Figure 3.8, which shows the two time series plots for an `iperf` process (labeled with module #122) which maintains two upload connections and two download connections. The upload connections are

socket#124 with TCP#126, and socket#125 with TCP#127. The download connections are socket#128 with TCP#130, and socket#129 with TCP#131.

To inject application faults, a Unix signal is sent to the application process, which sets a global variable; the global variable is cleared when another signal arrives to end the fault period. Before and after each `read()` or `write()` call, each reader/writer thread checks this global variable in a loop; if it is set, the thread sleeps for 10 ms and then checks again. While in this loop, its socket is marked as STALLED. When all the threads stop reading/writing, the application (maroon dashed line) is marked as STALLED as well.

Sometimes, the reader or writer threads are already blocked in a system call, and are delayed in seeing the global variable, as is seen by the staggered counters and diagnosis results around time t289. If the threads are blocked the entire time the variable is set, as the writer sockets #124 and #125 are around time t294, they can miss the fault period entirely. Neither of these situations is counted as a False Negative.

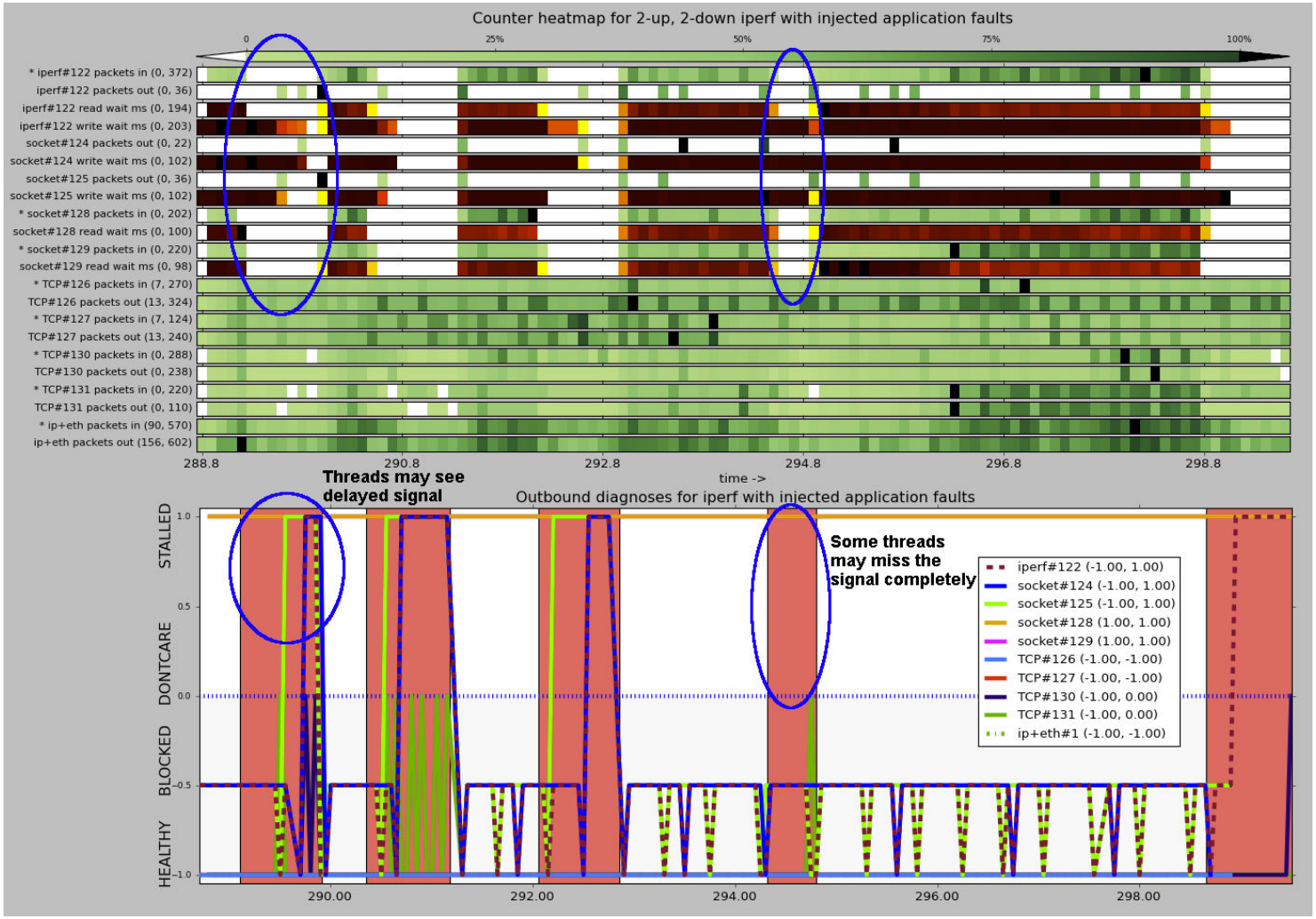


Figure 3.8: Outbound `iperf` counters and diagnosis results plot. The `iperf` process has 2 active outbound connections, and two active inbound connections. Faults are injected by sending a signal to the application to stop reading and writing from the network. As the `iperf` threads stop writing, the application and sockets are marked as STALLED in the outbound direction. Sometimes the threads are already blocked and either see the stop-writing signal late, or miss it altogether.

3.6 Potential Extensions

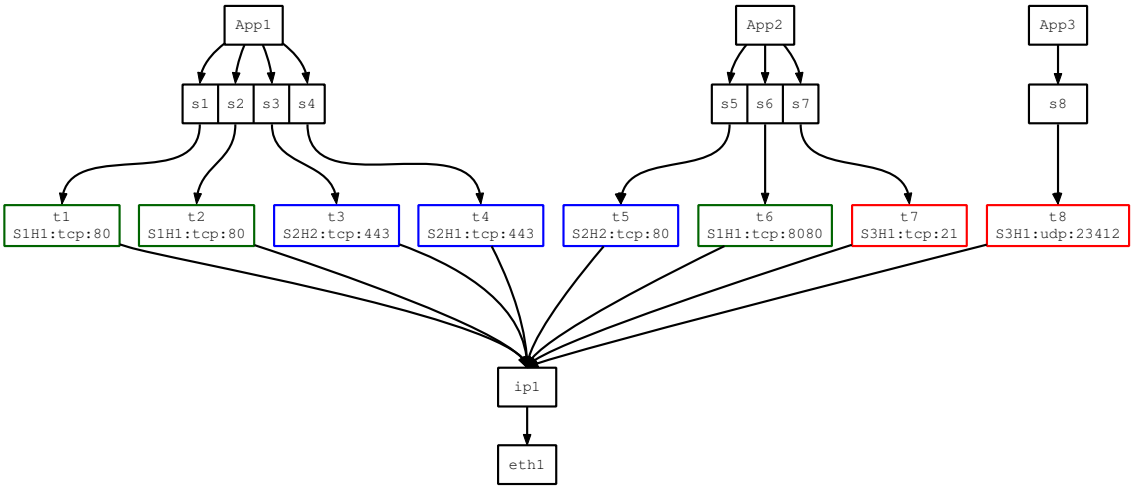
There are two main extensions to NeST that would improve upon the current results: extending the dependency graph to account for shared network dependencies, and including application-level work queues in the analysis.

3.6.1 Accounting for shared dependencies

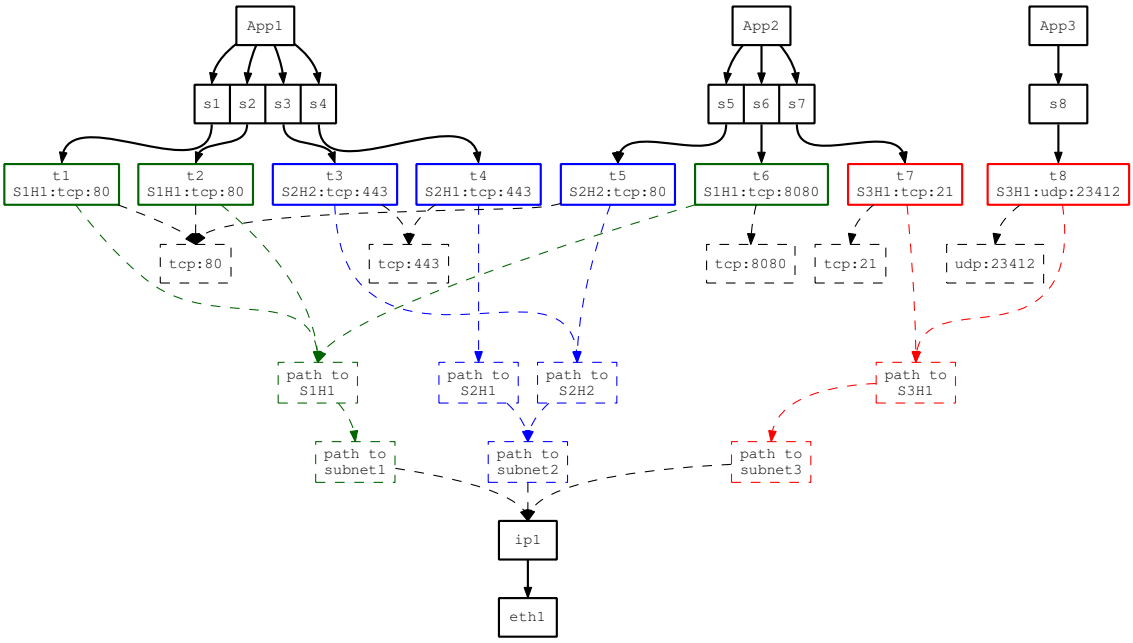
NeST's ability to distinguish between connection-specific and network-level events is not terribly precise. If any connection is active during a snapshot, then the network-level `ip+eth` module is declared Healthy. Similarly, if all of the connections are destined for a single remote host or subnet, a problem at the far side that hurts all of the connections will cause NeST to blame the `ip+eth` module as Stalled and mark the connections as Blocked, instead of indicating that the path to the remote host is the culprit.

The standard NS graph shown in Figure 3.9(a) shows only the modules on the local host which process network messages. In this figure, each connection is colored based upon the remote IP subnet. The green connections (`t1`, `t2`, `t6`) are destined for the same host (`S1H1`) on `subnet1` (`S1`). The blue connections (`t3`, `t4`, `t5`) are destined for two different hosts (`S2H1`, `S2H2`) on `subnet2` (`S2`). The red connections (`t7`, `t8`) are destined for the same host (`S3H1`) on `subnet3` (`S3`).

In some sense, the diagnosis lacks precision because the heuristic described in Section 3.3.2 uses a rather crude independence assumption: in the ambiguous case



(a) Standard NS graph. Connections destined for the same remote subnet are colored similarly: green for `subnet1` (`S1`), blue for `subnet2` (`S2`), and red for `subnet3` (`S3`).



(b) Extending the NS graph to include shared dependencies

Figure 3.9: The standard network stack dataflow graph, shown in the top subfigure, includes only the modules that actually process data on the local host. As seen in the bottom subfigure, the dataflow graph could be extended to include the concept of *shared dependencies*, in which modules are also dependent on particular paths (colored dashed boxes labeled "path to X") or firewall rules (black dashed boxes labeled with `protocol:port`).

shown in Figure 3.3, if the number of waiting connections N exceeds the threshold Θ, then a lower-level network-wide problem must have occurred.

However, this independence assumption does not account for the possibility of external *shared dependencies*. For example, if all of the green connections to `S1H1` (`t1`, `t2`, `t6`) experience problems, the problem is more likely to be particular to the interaction with `S1H1` or to a link along that path than a network-wide issue (as is indicated when NeST marks the IP module as Stalled).

By ignoring these shared dependencies, the NeST diagnosis results are imprecise in two ways:

1. NeST marks more modules as Stalled than strictly necessary.

   Ideally, a diagnosis output will be *parsimonious* and flag as few modules as possible. For example, if a common problem blocked all of the connections to `S1H1`, it is preferable to blame a single module that represents the "path to `S1H1`" than to separately blame three TCP connections that hold a shared dependency in common. This occurs when other connections not sharing the same path are active while these other shared-dependency connections are waiting.

2. NeST blames a broader-scoped module as Stalled than necessary.

   In addition to marking as few modules as possible, the scope of blame should be as narrow as possible. When NeST marks the `ip+eth` module as Stalled, it is indicating that the entire network is experiencing problems, as far as it

can tell. If possible, NeST should narrow the scope of the diagnosis output and lay blame upon a specific path (e.g. "path to `S1H1`").

These shared dependencies could be introduced into the dataflow graph, as shown in Figure 3.9(b). Instead of each transport-layer connection having an edge directly to the IP module, a virtual module representing the "path to host H" is inserted (colored boxes with dashed outlines), which aggregates all connections to host `H`. In turn, these remote-host modules are connected to a virtual "path to subnet S" module (e.g., grouping by the /24 IP prefix), which aggregates all connections to subnet `S`.

In addition, NeST could attempt to detect problems due to firewall rules that may block connections to remote `protocol:port` pairs such as `TCP:8080`, shown in the dashed-line black boxes in Figure 3.9(b). Clearly these aggregation possibilities could be arbitrarily numerous and complicated, for example grouping by destination autonomous system (AS), remote host:protocol:port, or even the local IP address:protocol:port. Such complicated diagnosis may be best performed by post-processing the simple diagnosis results.

Rather than try to maintain counters for the aggregate (which is feasible but requires bookkeeping), NeST could use the following algorithm to approximate the aggregate's counters:

1. The aggregate is active if any connection to it is active (Healthy).
2. If not, the aggregate is potentially blocked if any connection to it is potentially blocked.

3. Otherwise, the aggregate is idle (DontCare).

This ordering of priorities on the counter approximation is similar to that for applications and sockets, and loosely tracks the order of priority for diagnosis in general: any activity is an indication of success. It then looks for attempts to make progress (via `wait_time` and `queued_msgs`), and then whether any connections were completely idle (DontCare).

With this extension to the dataflow graph and a slight change to the inactive-module heuristic (Section 3.3.2), NeST may be able to solve both of the problems mentioned above. Instead of initially blaming all the waiting modules, but absolving connections when count(potentiallyBlocked) $> \Theta$ (another way of interpreting the heuristic), when comparing sets of parents and children that are all potentially blocked, NeST could use the following rule:

$$\begin{cases} \text{parents} \leftarrow \textsc{Blocked} & \\ \text{children} \leftarrow \textsc{Stalled} & \text{if } |\text{parents}| \geq \Theta \\ & \\ \text{parents} \leftarrow \textsc{Stalled} & \\ \text{children} \leftarrow \textsc{DontCare} & \text{if } |\text{parents}| < \Theta \end{cases}$$

The first part of the rule limits the number of modules marked as Stalled, by blaming the aggregate if possible. The second part of the rule limits the scope of the diagnosis to as narrow a set of shared dependencies as possible. Thus, if connections to at least Θ different subnets fail at the same time, NeST marks the

broader `ip+eth` module as STALLED; otherwise it only marks the paths to those subnets or the individual connections themselves.

Instead of modifying the dependency graph and dependency analysis, the shared-dependency aggregation could be performed as part of a post-processing step. Modules marked as BLOCKED or STALLED could be reviewed for common dependencies (destination host, subnet, protocol:port, et cetera), and then checked to determine if any module also in that aggregation was HEALTHY.

3.6.2 Specifying expected application behavior

The `wget` and `iperf` programs used in the evaluation in Section 3.5 were chosen due to their simple performance specification. In the manner used in the experiments, `iperf` should always write to sockets at remote port 5001, and receive on sockets with local port 5001; `wget` should always read from its sockets, and is not expected to write to them. This makes it simple and straightforward to determine if the NeST diagnosis is correct.

However, other applications do not have such straightforward performance characteristics. For example, when streaming one video from Hulu, `webkit` maintains up to 40 open connections. Most of the sockets are BLOCKED for the duration of the video stream. Other sockets are BLOCKED in reads for several seconds at a time, then transmit and receive some data, and are then idle (STALLED) and neither read nor write for over a second. Whether this is a software bug or desired application behavior is completely dependent on the application.

To completely automate the diagnosis and evaluation of these more complex applications, NeST needs to know what the application (or socket) itself should be doing. The HasWork() function described in Chapter 2 assumes by default that any root module (application or socket) has work to do. If NeST were able to instrument the application or socket work queues, it could mark sockets as Stalled only when there was some work for the socket itself to perform.

There are several ways for NeST to track per-socket work queues:

- Provide an API that allows the application to specify when a given socket is expected to be (in)active for reads and/or writes. Using this mechanism, NeST would know at any moment what a socket should be doing, but this would require changes to application source code.

- Provide an API or application-specific adapter that reads an application's internal work queue and presents that counter to NeST.

- Allow a developer or systems administrator to provide a simple performance specification for each program, similar to the one used in the accuracy evaluation. This specification could include statements such as "Always read on connections to remote port 80," but this significantly limits its usefulness for applications that reuse connections.

- Create application-specific modules in the interceptor library that interpret the data payloads of `read()` and `write()` calls, e.g. by looking for HTTP GET, POST, or PUT requests. The maintenance and performance overhead would likely make this not worth the effort.

- Write adapters to monitor an application's log or standard output and determine which sockets should be active and in which flow directions. This assumes that the application has a useful per-socket logging capability.

3.7 Summary

Using the FlowDiagnoser approach described in Chapter 2, the Network Stack Trace (NeST) *automatically* finds the source of network-related performance stalls. NeST uses *efficient* performance counters that are local to an end host system, and is *accurate* enough to diagnose over 99% of performance stalls with a low false positive rate of around 3%.

Chapter 4

Diagnosing Problems in InfoSphere Streams

This chapter describes StreamsDiagnoser, a second instantiation of the FlowDiagnoser approach to the task of detecting performance stalls in InfoSphere Streams, a stream-processing engine sold commercially by IBM [24].

This chapter begins by describing the elements of the InfoSphere Streams programming model and some of the performance problems experienced by InfoSphere Streams applications in Section 4.1. Section 4.2 describes the process to construct the FlowDiagnoser dependency graph for streams applications. Section 4.3 discusses the performance counters StreamsDiagnoser collects, and how it uses them for performance diagnosis in Section 4.4.

Section 4.5 describes the prototype implementation, and Section 4.6 the results from controlled experiments. These experiments show that StreamsDiagnoser is able to detect 93% of injected faults, with a False Positive Rate of 2%; initial tests on real applications also provided promising results.

Section 4.7 describes experiences instrumenting live applications, and the chapter concludes in Section 4.8 with a brief summary of StreamsDiagnoser results.

4.1 Basic Streams Model

In its basic architecture, InfoSphere Streams is similar to research systems such as Aurora [1, 7] and Borealis [4]. Application developers write and compose *operators* which operate on sliding windows of data *tuples*;[1] these tuples flow between operators as data *streams*.

Streams applications are collections of operators composed into a *processing graph*. Each node in the graph is single Unix process called a *processing element* (PE); the discussion assumes that operators and PEs are equivalent, although this need not be the case.

Each PE has *input ports* from which it reads and *processes* incoming tuples, and *output ports* on which it *submits* tuples for downstream PEs to process. As in the general model described in Chapter 2, PEs without input ports are called *sources*, and PEs without output ports are called *sinks*. A *stream connection* is the logical connection between one PE's output port and another's input port over which tuples flow; these are the edges in the Streams processing graph.

The Streams processing framework is quite sophisticated, and provides for code and process distribution over multiple hosts. That is, the PEs in an application may reside on a single end host, or scattered across an entire cluster. A

[1] In addition to data tuples, Streams operators can transfer non-data window *punctuations* (puncts) across the stream connections to signal window or structure boundaries. While the Streams runtime counts data and punctuation separately, the StreamsDiagnoser counters include both. For conciseness, we refer to both data and punctuations as *tuples*.

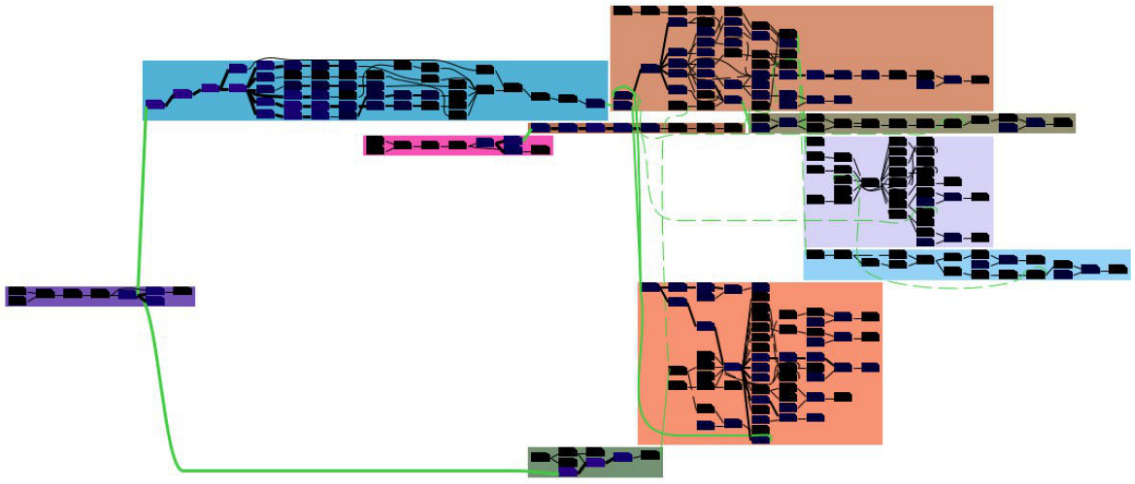


Figure 4.1: Streams Application. Each node in the Streams processing graph represents a single Processing Element (PE), which are connected via *stream connections* (black or green lines). PEs are grouped by their *job* (large colored rectangles). Stream connections that are exported/imported between jobs are shown as green lines; connections that have never transferred a tuple are shown as dashed lines. Note that PEs can have multiple connections to the same input port (fan-in) or from the same output port (fan out); this occurs in the middle of the light purple job to the right side of the figure.

Streams processing graph from a real application (discussed in Section 4.7) is shown in Figure 4.1; a simpler example is shown in Figure 4.2, discussed shortly.

4.1.1 Streams operators

InfoSphere Streams allows operators to be specified in many different languages (including C++, Perl, Python, Java, and other languages that use the JVM) or in the custom Streams Processing Language (SPL). SPL is a domain-specific language (DSL) that makes it easy to specify the windowing characteristics (last-n tuples, time-based, etc) and the operations to perform upon the arrival of each tuple on an input port.

There are a wide range of standard Streams operators available, which can be used to, for example: read from or write to files, directories, or network sockets; filter, sort, join, aggregate, or transform data tuples; and throttle, delay, de-duplicate, split, or pair up tuples as they traverse through the system [30].

While SPL provides a concise way of specifying operations on tuples and passing them between named streams, it does not have the extensive set of libraries available in Java or C++. For more complex operations, developers often write operators in a general-purpose language; these custom operators can block for any number of reasons: disk I/O, DNS or database queries, or writing to a network socket. SPL operators can also block due to disk I/O or complicated timing and synchronization mechanisms.

4.1.2 Example processing graph

Figure 4.2 provides an example Streams processing graph used throughout this discussion. Each node in the processing graph is a processing element (PE) labeled with a letter (`A`–`N`, `P`, `Q`, `R`). Each PE has input ports labeled `in[0-9]` and output ports labeled `out[0-9]`. *Source* PEs with no input ports are shown with dashed lines, and *sink* PEs with no output ports with dotted lines. *Stream connections* are labeled with numbers (`1`–`18`), referred to as $\texttt{sconn}_1$–$\texttt{sconn}_{18}$. Output ports may have *fan-out* and transmit to multiple input ports, as seen on `F` : `out0`. Input ports may have *fan-in* and receive from multiple output ports, as seen on `N` : `in0`.

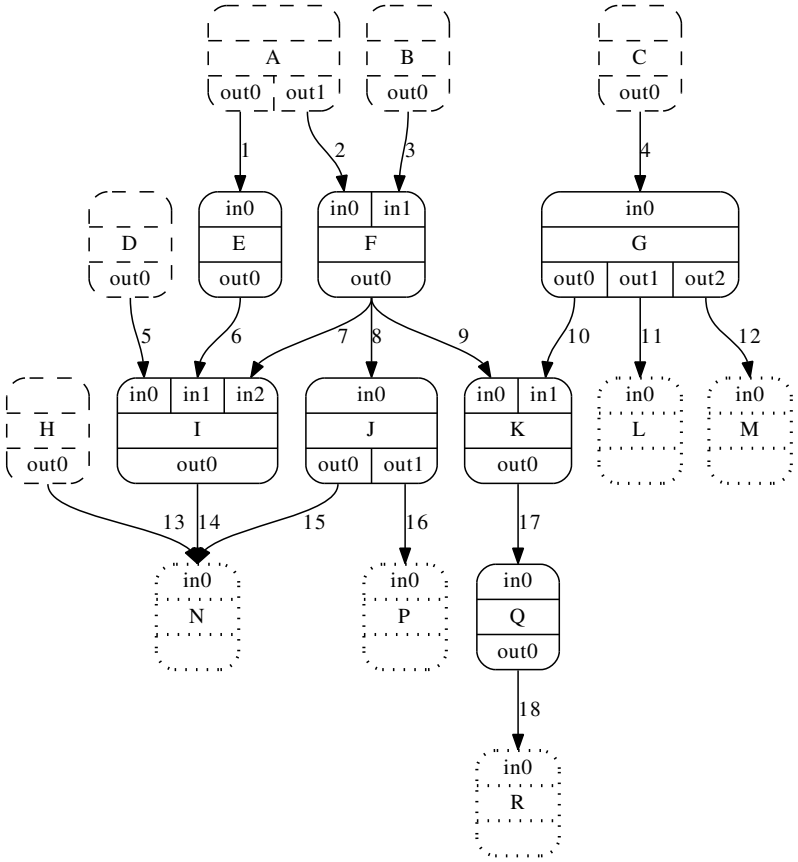


Figure 4.2: Example Streams processing graph. See the text for details.

4.1.3 Streams counters

In the Streams framework, counters are tracked by PEs at their input and output ports. Of the many counters available, two are required by StreamsDiagnoser:

- `nSubmitted`, which counts the total tuples submitted to an output port
- `nProcessed`, which counts the total tuples processed from an input port

In the example processing graph, PE `F` has three sets of counters:

- $\texttt{nProcessed}_{\texttt{F:in0}}$, the total number of tuples processed on input port `F` : `in0`
- $\texttt{nProcessed}_{\texttt{F:in1}}$, the number processed on input port `F` : `in1`

- $\texttt{nSubmitted}_{\texttt{F:out0}}$, the number submitted on output port `F : out0`

Since Streams counters are tracked by *port* (e.g. `N : in0`), and not by connection (e.g. on the connection `H : out0` → `N : in0`), the input port counter `nProcessed` is the total number of tuples processed on *all* incoming connections. Similarly, the output port counter `nSubmitted` is the total number of tuples transmitted across *all* outgoing connections from this port.

4.1.4 Streams Performance Problems

When a Streams PE stops processing messages, its incoming queue can quickly fill with tuples submitted to it by upstream PEs. This can have two effects:

1. Upstream PEs block when submitting tuples on their output port, and stop processing new tuples from their input ports, thus propagating the problem upstream. This behavior is known as *backpressure*, and naturally limits the rate of tuples that can flow through the system.[2]

2. Downstream PEs may be starved of tuples to process, and become inactive.

For example, in Figure 4.3, if PE `K` stopped processing tuples on input port `K : in0`, its input queue would eventually fill, causing `F` to block and stop processing tuples from PEs `A` and `B`. This backpressure may in turn cause `A` and `B` to block. Hence, the downstream PEs `E`, `J`, and `P` may have no tuples to process, and will

[2]Like other stream processing systems, InfoSphere Streams is designed to allow *load shedding* by dropping incoming tuples when backpressure occurs. Many developers prefer not to enable this feature, and to process all of the data at a delay instead.

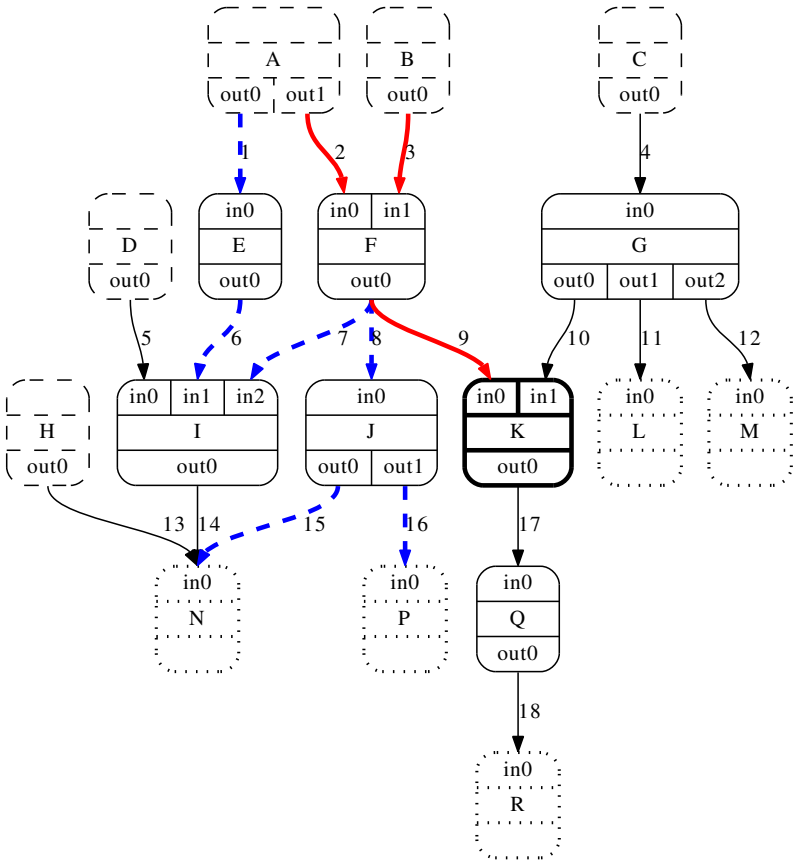


Figure 4.3: Example Streams processing graph with backpressure in effect. Heavy red lines indicate backpressure across stream connections. Dashed blue lines indicate where traffic has ceased due to blocked PEs. StreamsDiagnoser should mark PE `K` as causing the problem that effects the other PEs.

become inactive. If the sources `D` and `H` are also inactive, their downstream PEs `I` and `N` may become inactive as well.

Hence, a problem in one part of the Streams graph may quickly propagate to nodes that are seemingly unrelated. When this occurs, it is often difficult to determine which PE(s) are causing the problem, and which are merely affected by it. This is especially difficult when multiple PEs cause problems simultaneously. The StreamsDiagnoser system is designed to pinpoint which PEs are causing these problems.

4.2 The StreamsDiagnoser Dependency Graph

The first step in applying the FlowDiagnoser methodology to Streams is to generate and maintain a dependency graph that corresponds to a processing graph. The main challenge is that Streams counters are tracked by PE *ports*, but are attached to modules in the graph in the theoretical model described in Chapter 2. There are several options available to make this transformation.

4.2.1 Option 1: Each PE is a module

One reasonable approach would be to treat each PE in the Streams graph as a FlowDiagnoser module by summing its input or output counters.

If Streams PEs behaved like modules in the network stack, StreamsDiagnoser could simply look at a PE's total output (summing the per-port `nSubmitted` counters), and track which PEs in the graph are submitting tuples. Then, it might assume that if an upstream PE submitted tuples, all of its downstream processors must also submit tuples. However, this assumption is invalid for several reasons, as illustrated in Figure 4.4.

(a) Operators do not always forward (submit) tuples they receive. Some operators such as `filter` will discard any tuples that do not meet the filter criteria; blaming them in that case would be erroneous.

In Figure 4.4, PE `E` is processing tuples from `A` but is not submitting any on its output port because none of them meet its filter criteria.

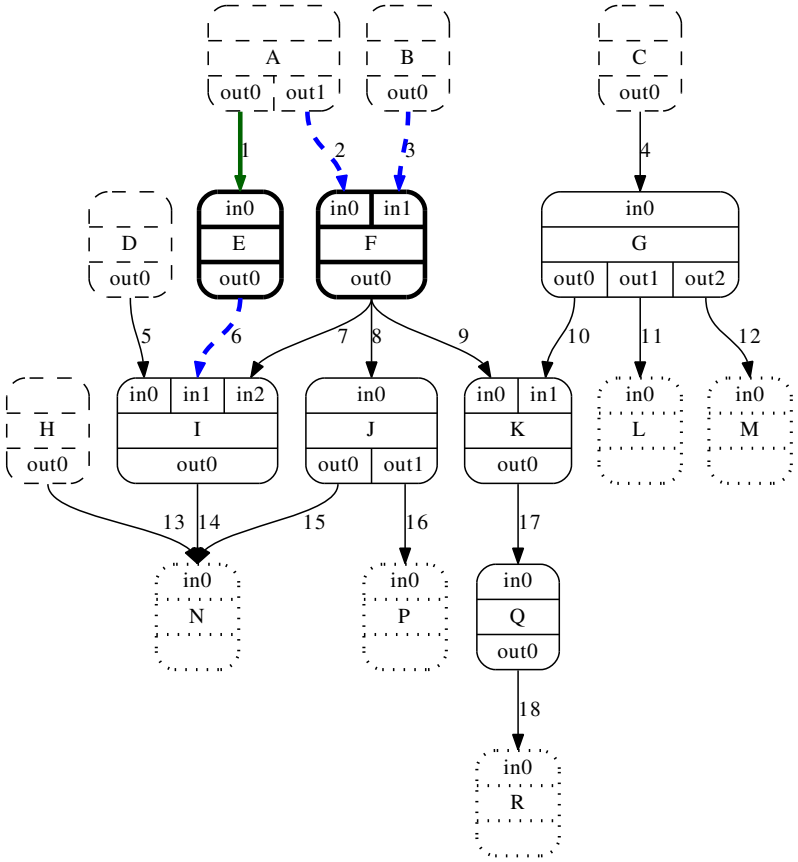


Figure 4.4: PE `A` has submitted tuples on $\texttt{sconn}_1$ (heavy green line), but not on $\texttt{sconn}_2$ (dashed blue line). `E` is filtering tuples that do not meet its criteria, and does not submit any to its downstream $\texttt{sconn}_6$. `F` is inactive since it has not received any tuples to process.

(b) Operators with multiple output ports may submit tuples to one port without submitting them to the others. Thus, StreamsDiagnoser would blame downstream processors for not submitting tuples when they have never received any to pass along.

In Figure 4.4, `F` has not received any tuples to process, since neither `A` nor `B` have submitted any to it (dashed blue edges). If StreamsDiagnoser were to sum `A`'s `nSubmitted` counters, it would lose this information and may provide an incorrect diagnosis.

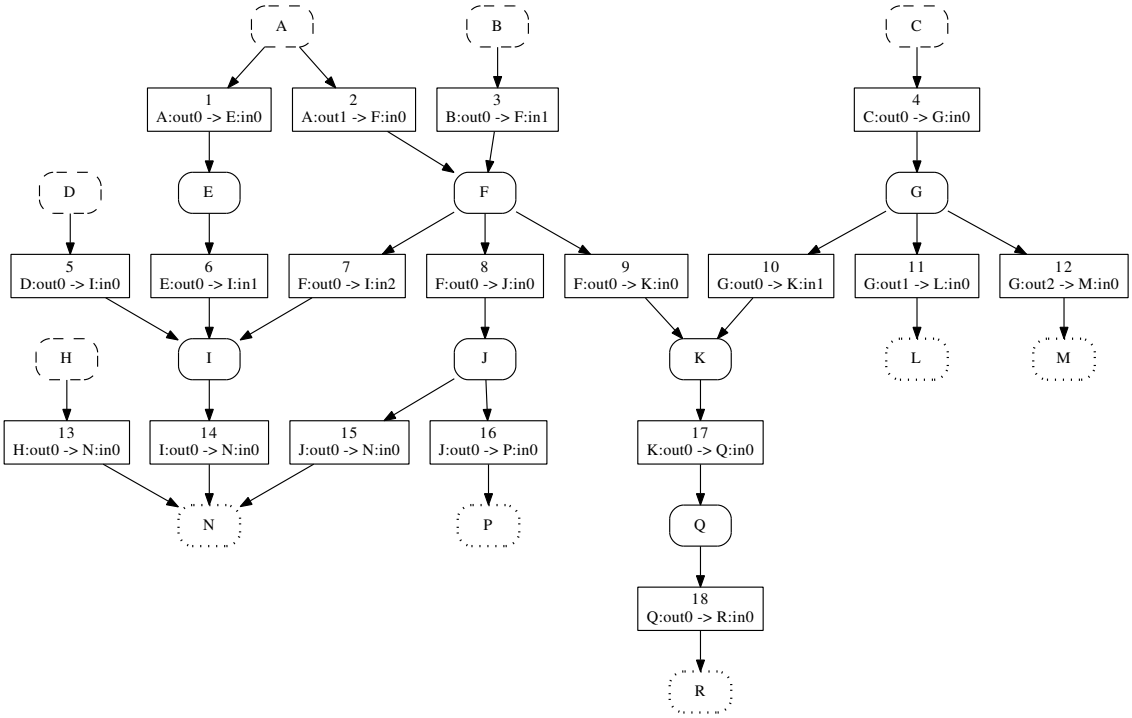
(c) Finally, when backpressure occurs, upstream modules are blocked from submitting any more tuples, and the `nSubmitted` counters no longer increase. Since StreamsDiagnoser does not have a direct signal of whether a PE is waiting as NeST does for application sockets in the network stack (Chapter 3), StreamsDiagnoser needs some other signal to determine whether a PE is blocked or merely idle.

4.2.2 Option 2: Ports as modules

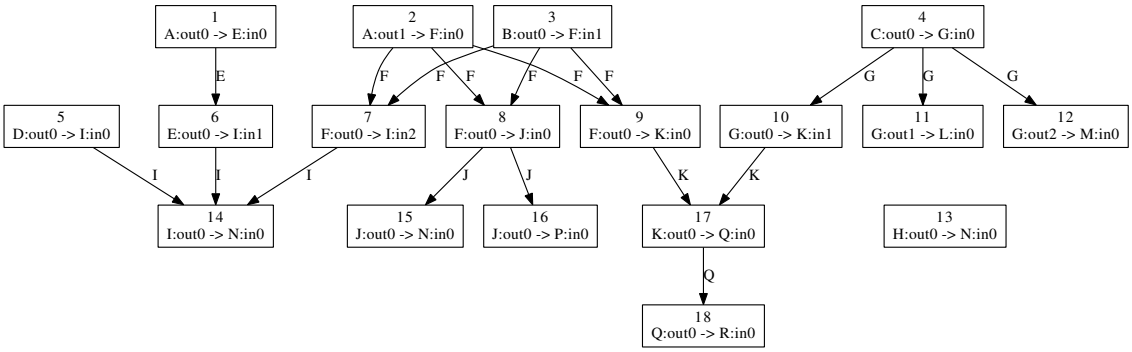
Rather than combining all of a PE's ports into one module representing the PE itself, StreamsDiagnoser could treat each *port* as a module in the StreamsDiagnoser dependency graph, using `nProcessed` as the signal of activity on input ports, and `nSubmitted` for output ports. However, this would have many of the same problems as described for Option 1: StreamsDiagnoser cannot expect that a PE receiving traffic on one input port will transmit traffic on all its output ports, and StreamsDiagnoser would still lack a signal to indicate if a module is waiting.

4.2.3 Option 3: Stream connections as modules

The approach that StreamsDiagnoser uses to transform the PEs-with-ports Streams processing graph into the modules-and-edges StreamsDiagnoser dependency graph is to invert the streams processing graph, and turn each *stream connection* ($\texttt{sconn}_1$, $\texttt{sconn}_2$, ...) into a module in the dependency graph, connected by edges which represent the PEs.



(a) Step 1 of the graph transformation: creating a module for each (numbered) stream connection, and labeling it with the output and input ports that it connects (e.g. `A : out0` → `E : in0`). After this step, the PEs remain as dummy nodes with no ports.



(b) Step 2 of the graph transformation: removing the dummy nodes. Now all the modules are stream connections, which are logically connected by PEs. Note that $\texttt{sconn}_{13}$ from Source `H` to Sink `N` is isolated by itself, since it has no upstream or downstream connections.

Figure 4.5: To transform a Streams processing graph to a StreamsDiagnoser dependency graph, StreamsDiagnoser first converts each numbered streams connection into its own module in the graph (top subfigure). It then removes the PEs from the graph entirely (bottom subfigure). What remains is a graph of Stream Connections joined by PEs (labeled edges).

This can be viewed as a two-step process, as illustrated in Figure 4.5. StreamsDiagnoser first creates nodes for each numbered stream connection by pulling the output and input ports into a new node, and placing them in the graph between the PEs (Figure 4.5(a)). It then removes the PE nodes and labels the remaining edges; what remains is a dependency graph of streams connections (modules) that is used in the dependency analysis (Figure 4.5(b)). Note that when StreamsDiagnoser does this, the source and sink PEs disappear from the dependency graph; they exist only as the upstream side (for sources) or downstream side (for sinks) of the modules. Note as well that connection #13 from source `H` to sink `N` (labeled `H` : `out0` $\rightarrow$ `N` : `in0`) is isolated from the other streams, since no other streams feed into PE `H` or out of PE `N`.

Like the stream processing graph, the resulting dependency graph is *push-oriented*: the direction of the edges in the dependency graph matches the flow of data through the system. The following sections describe how StreamsDiagnoser assigns the counters to the modules (stream connections) in the dependency graph.

4.3 Stream Connection Counters

Once StreamsDiagnoser has obtained the dependency graph, it must define the counters for each module in that graph.

Recall that each output port has an `nSubmitted` counter that tracks the number of tuples emitted on that output port, and each input port has an `nProcessed` counter that tracks all tuples processed from the input port. StreamsDiagnoser as-

Counter	Description
`total_msgs`	Total messages processed from the connection (`nProcessed`)
`queued_msgs`	Total messages still in the connection [`nSubmitted` − `nProcessed`]

Table 4.1: StreamsDiagnoser module counters.

signs these to the module that represents each stream connection, and derives the FlowDiagnoser counters shown in Table 4.1 as follows:

- `total_msgs` ⇐ `nProcessed`. This is the number of tuples processed *out of* the stream (from the input port).
- `queued_msgs` ⇐ (`nSubmitted` − `nProcessed`). This is the number of tuples in the downstream PE's work queue.

By calculating `queued_msgs` from the output and input port counters, StreamsDiagnoser has the required signal that indicates whether the upstream PE is waiting on the downstream to complete its work, as described in Section 2.3. While deriving the `queued_msgs` counter is simple in theory, in practice it is a bit more complicated.

4.3.1 Recovering per-connection counters

As described in Section 4.1.3, the per-port counters account for the *total* number of tuples submitted or processed across all the connections from/to that port. For example, in the processing graph in Figure 4.2, PE `F`'s output port `out0` connects to input ports of three downstream PEs. This port's `nSubmitted` counter will

increment by *three* when `F` sends *one* message via that port, since the port connects to three destination input ports.

Since the dependency graph is based on individual stream connections (`F : out0` $\rightarrow$ `I : in2`), StreamsDiagnoser needs counters that track the number of tuples submitted into the connection, and the number of tuples processed out of the connection. In other words, for the example, StreamsDiagnoser would prefer that `out0`'s message counter be one, rather than three. As such, it needs to normalize the counters from each input and output port. This is easily achieved by dividing the per-port counter's increase among the current connections $\left(\Delta^{\text{sconn}}_{\texttt{nSubmitted}} = \frac{\Delta^{\text{port}}_{\texttt{nSubmitted}}}{\text{nConnections}}\right)$.

Unfortunately, doing so is problematic when the number of connections changes during the snapshot period. Additional work is necessary to obtain the required counters without the normalization and bookkeeping procedures, which are described in Appendix A.

4.3.2 Invariant violations

In addition to the counter normalization, another complication arises when monitoring Streams applications. As discussed in Section 2.2.1, the FlowDiagnoser approach does not require a stop-the-world snapshot of all of the counters. Even if StreamsDiagnoser could obtain such a consistent snapshot, it would be too expensive and unreliable in a multi-process, multi-host distributed system such as InfoSphere Streams.

One common occurrence in data from live systems is a violation of the one invariant that StreamsDiagnoser counters must hold:

- An input port can never process more tuples than have been submitted to it.

This violation is due to the time-delayed nature of the StreamsDiagnoser snapshots. During a single snapshot, StreamsDiagnoser can read `nSubmitted` on the output port, and later read `nProcessed` on an input port. When this happens,

$$(\sigma_{\texttt{nSubmitted}} < \sigma_{\texttt{nProcessed}}) \Rightarrow (\sigma_{\texttt{queued\_msgs}} < 0)$$

which implies that there is a negative queue between the output port and input port, an obvious impossibility.

When StreamsDiagnoser detects a negative-length queue, it assumes that some of the tuples processed have been *pre-counted*; that is, the downstream counter includes some processing that really belongs in the *next* snapshot. So, to avoid false positives (in which StreamsDiagnoser blames the downstream for not processing anything at all in the next snapshot), instead of ignoring the negative queue StreamsDiagnoser adds $abs(\sigma_{\texttt{queued\_msgs}})$ from this snapshot to $\Delta_{\texttt{total\_msgs}}$ in the next snapshot. This means for any snapshot where StreamsDiagnoser calculates a negative queue, the downstream will always appear active (HEALTHY) in the next measurement period.

4.4 Streams Dependency Analysis

Once StreamsDiagnoser has the counters for each stream connection (module in the dependency graph), it can apply the dependency analysis as described in

Section 2.3. With the available counters, the dependency analysis reduces to the following:

$$diagnosis = \begin{cases} \textsc{Healthy} & \text{if } \Delta_{\texttt{total\_msgs}} > 0 \\ & \ldots \text{else } \Delta_{\texttt{total\_msgs}} = 0 : \\ \textsc{DontCare} & \text{if } \sigma_{\texttt{queued\_msgs}} \leq 0 \\ \textsc{Blocked} \text{ or } \textsc{Stalled} & \text{if } \sigma_{\texttt{queued\_msgs}} > 0 \end{cases}$$

A change in the number of tuples processed ($\Delta_{\texttt{total\_msgs}}$) indicates whether or not the downstream PE (with the input port) was active; if so, the connection is Healthy. If it was inactive, StreamsDiagnoser checks `queued_msgs` to determine whether or not the downstream PE had work to do; if $\sigma_{\texttt{queued\_msgs}} \leq 0$, the stream connection was empty and is diagnosed as DontCare to indicate that it had completed all its work. Otherwise, the downstream PE did not process any tuples, but there were some stuck in the queue ($\sigma_{\texttt{queued\_msgs}} > 0$), and StreamsDiagnoser needs to determine why.

4.4.1 Detecting backpressure and inactive streams

As mentioned in Section 4.1.4, Streams applications can experience two major types of performance limitations:

1. An upstream PE may not produce (submit) enough tuples for its downstream PEs to consume (process), which leaves the downstream PEs idle.

2. Backpressure that occurs when a downstream PE does not consume (process) tuples as fast as its upstreams produce (submit) them.

The first performance limitation is easily detected by the StreamsDiagnoser: a diagnosis of DontCare indicates that an upstream PE has not produced enough modules for the downstream PE to process.

In the second case, performance problems caused by descendant modules propagate upstream through the dependency graph. Thus, the blame-passing diagnosis rule described in Section 2.3.1.3 and Table 2.2 (criteria *(e)* and *(h)*) helps to locate the module that is causing the problem.

Recall that using the blame-passing rule, when an inactive module M has tuples in its work queue, StreamsDiagnoser marks it as Stalled and blames it for blocking progress unless one or more of its children are also inactive with work to do. If its children are also inactive, StreamsDiagnoser marks M as Blocked and blames its children instead. This propagates the blame away from the sources (ancestors) and toward the sinks (descendants), and continues until it reaches a descendant that is either active (Healthy) or has nothing to do (DontCare).

4.4.2 Interpreting the results

Diagnosis results are attached to the stream connection (modules) between processing elements (PEs). The upstream PE submits tuples into the connection, and the downstream PE processes them. Since the dependency analysis looks at the

number of tuples submitted but still stuck in the queue, StreamsDiagnoser detects stalls caused by the downstream PE.

That is, for every stream connection module `A` : `outx` → `B` : `iny`, the following interpretation applies:

- HEALTHY: PE `B` was actively processing tuples on input port y.
- DONTCARE: PE `B` did not have any tuples to process on port y because PE `A` did not supply them.
- BLOCKED: PE `B` was unable to process tuples available on port y due to backpressure from one or more of its downstream PEs.
- STALLED: PE `B` was inactive and did not process tuples available on port y, and none of its downstream PEs were BLOCKED or STALLED.

Hence when StreamsDiagnoser blames a module, it is really blaming its downstream PE for not reading from its input port. Note that because a StreamsDiagnoser module is a *stream connection*, its parents and children are themselves other connections. Thus, StreamsDiagnoser passes the blame to other connections (and their downstream PEs) only if they also have tuples stuck in their work queue.

4.5 Data collection prototype

This section describes the prototype StreamsDiagnoser implementation and experimental results.

As a commercial product, InfoSphere Streams provides several interfaces for monitoring running applications, including: a web service and graphical user interface, an Eclipse plugin that allows a user to monitor and visualize the Streams processing graph in real time, and a command-line tool that can report the processing graph topology and PE (and operator) metrics via Streams-specific XML files.

When using the command-line tool, a snapshot request can take several seconds to complete from tool invocation to result. This is largely because upon each invocation, the command-line tool must first learn the topology and distribution of the PEs and then request the metrics from each PE. It also generates a large (up to 2 MB) XML file for the metrics output. This is how our colleagues collected the data for the live applications described in Section 4.7; the snapshot intervals varied between 13–15 seconds.

The controlled experiments use the same internal API provided to the Eclipse plugin. In this case, the monitoring and data collection runs as a separate Streams job, which receives a notification whenever the topology changes. It is also able to gather the metrics as frequently as once per second.

In both cases, for each snapshot the data collection prototype exports the Streams processing graph in GraphML format [11], attaching the per-port counters to the edges within the graph. Once an experiment run has completed, the GraphML files are loaded into the diagnosis engine, which converts the processing graph into the internal StreamsDiagnoser dependency graph and stores the accu-

mulated snapshots for each module (stream connection). The diagnosis algorithm is performed for each snapshot, and the results output and stored.

4.6 Experimental Results

To validate the data collection infrastructure and diagnosis algorithm, we performed controlled experiments on several basic Streams topologies. As with the NeST Emulab experiments, we inject several different kinds of performance anomalies at various places in the sample topologies, and evaluate StreamsDiagnoser's ability to correctly locate the source of the performance fault. The results show that StreamsDiagnoser is able to detect 93% of injected faults in controlled experiments, with a False Positive Rate of 2%; initial tests on real applications also provide promising results.

4.6.1 Basic topologies

The six basic topologies used in the experiments are depicted in Figure 4.6. These six include most of the basic forms that Streams applications are composed of [24]. The Tree forms include fan-out (multiple outgoing connections from one output port) and fan-in (multiple incoming connections to one input port). The TreeMux forms are similar to the Tree topologies, but each stream connection is attached to unique output and input ports (and thus counter normalization becomes unnecessary).

Considering each topology in detail:

(a) ComplexTree. Nineteen PEs with several levels of output port fan-out.

(b) ComplexTreeMux. Twenty-two PEs with several levels of fan-out at the PE level, but each outgoing connection is attached to a unique output port.

(c) MergeTree. Eight PEs with both 3:1 fan-out and 1:3 input port fan-in.

(d) MergeTreeMux. Nine PEs with fan-in and fan-out at the PE level, but each connection is attached to a unique output port and input port.

(e) MergeTreeBarrier. Like the MergeTree, but replacing the fan-in PE with a 3-port Barrier (labeled with a `B`), which reads one tuple from each of its inputs before submitting one tuple downstream. If any incoming port has no tuples, it blocks the remaining ports.

(f) MergeTreeBarrierMux. Like the MergeTreeBarrier (labeled with a `B`), but the fan-out PE has one output port for each stream connection.

These last two topologies should cause problems for StreamsDiagnoser, since the Barrier PE will stop processing tuples when one of its upstreams has stopped providing tuples to it. While this *is* a performance stall according to the diagnosis algorithm, it is correct behavior and is not counted as an Actual Positive (AP) in the evaluation.

The source operator in each topology is a DynamicBeacon that emits tuples at a fixed rate. For the controlled experiments, it emits tuples containing a single integer at rates of 100 tuples/sec and 1000 tuples/sec; a second batch of experiments runs the source at full rate to simulate system overload. All other operators (aside

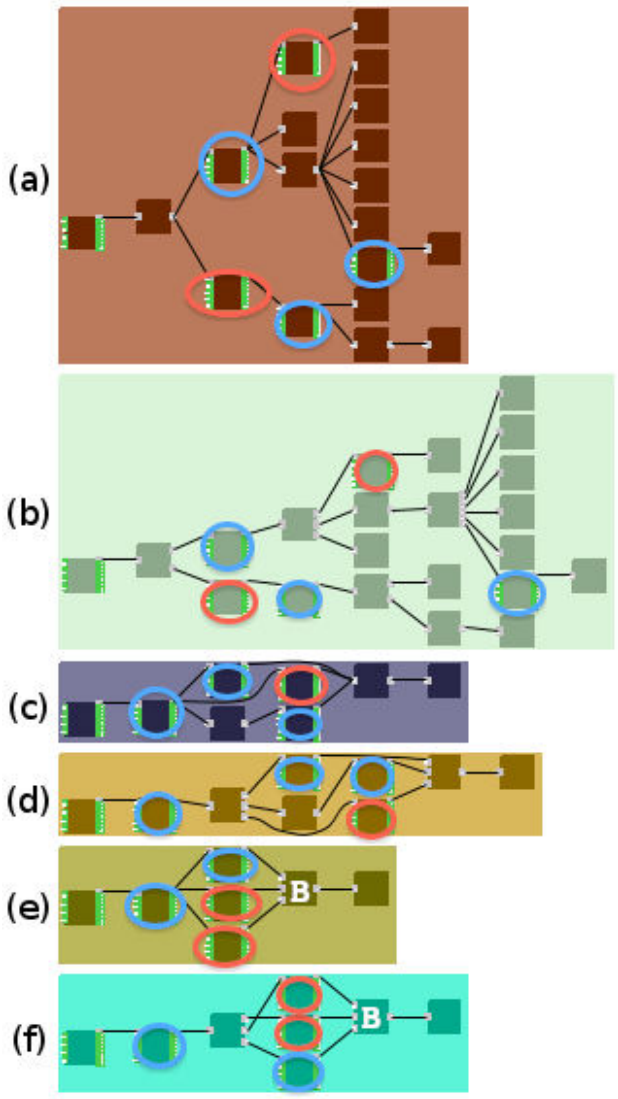


Figure 4.6: Basic Streams processing graphs used in the accuracy experiments. DynamicThrottle PEs are circled in red, and DynamicDropper (filter) PEs in blue; Barrier PEs are labeled with a `B`. Each processing graph is converted into a Streams-Diagnoser dependency graph according to the procedure outlined in Section 4.2.3.

from the circled ones, described below) perform joins (read from many inputs, send one output tuple), aggregations (read many input tuples then emit their sum), and simple pass-through.

To perform a true controlled experiment, and to isolate the experiments from each other while limiting experiment run time, each experiment is run on its own host. We have also conducted extensive experiments using multi-host deployment strategies.

4.6.2 Injected faults

We inject faults in the example topologies using two types of operators (not developed by us), which are circled in Figure 4.6: the DynamicDropper (blue) and DynamicThrottle (red):

- DynamicDropper. This operator emulates an on/off filter that reads (processes) and drops all tuples when instructed; otherwise it simply forwards them downstream as quickly as it can read them. It can also buffer tuples for a period of time, then submit them to its downstream port all at once. As Section 4.2.1 discusses, this could be considered normal behavior and should not be considered a performance stall.

- DynamicThrottle. This operator is used to control the *maximum* rate of tuples flowing through the system. The throttle varies between 0, 1, 100, and 1000 tuples/sec in the experiments, returning to full-rate after each throttle-test period. When the throttle is at 0 tuples/sec, it does not read from its input

port and causes backpressure. This is the main fault StreamsDiagnoser is trying to diagnose.

These dynamic operators are located at strategic points throughout the test topologies. Specifically, they are located:

- At the sinks, to test StreamsDiagnoser's ability to detect problems at the edges of the graph.
- At, before, and after fan-out (branch) points, to test the ability to find problems on either side of a branch.
- At, before, and after fan-in (merge) points, to test the ability to find problems on either side of a merge.
- Immediately upstream from the Barrier PE, to evaluate how StreamsDiagnoser handles a Barrier which is not provided with enough data on one of its ports.

These injected faults simulate common performance anomalies in running Streams applications: PEs which do not produce data and starve their downstream PEs of data to process (DynamicDropper), PEs which cannot process data at fast enough rates (low DynamicThrottle rate), and PEs which occasionally stop processing altogether (DynamicThrottle rate = 0) [43].

4.6.3 Diagnosis accuracy

Since StreamsDiagnoser is looking for performance *stalls*, the only time the "correct" diagnosis is positive (STALLED) is when the DynamicThrottle is set to 0,

that is, when a PE does not process any tuples on its input port. Since diagnoses are assigned to the stream connection (modules), *all* connections leading to the DynamicThrottle's input port should be marked as STALLED when the throttle is set to 0.

When the DynamicDropper is engaged, it processes tuples on its input port, but does not submit (forward) them to its downstream output port. This behavior simulates a filter module with tuples that do match the filter criteria; this should not necessarily be considered a problem, as explained in Section 4.2.1.

For all other modules, the evaluation always expects a negative diagnosis: HEALTHY, BLOCKED, or DONTCARE. Any diagnosis of STALLED that is *not* assigned to a DynamicThrottle PE when throttleRate = 0 is a False Positive.

In the controlled experiments, the topologies run on a single host, varying the source rates between 100 and 1000 tuples/sec. Snapshots are taken every five (5) seconds. Running on one host and limiting the source rate ensures that the experiment does not overload the monitored system, so only the injected faults should be present. Each change to the DynamicThrottle or DynamicDropper lasts for 5, 10, or 20 seconds to ensure that it covers an entire snapshot.

Table 4.3 presents the accuracy results for each topology. The abbreviations table is repeated here in Table 4.2. Please refer to Section 3.5.2.1 for an explanation of the accuracy evaluation table.

In the fixed-rate source experiments, shown in the first grouping, StreamsDiagnoser did very well on the COMPLEXTREE, COMPLEXTREEMUX, MERGETREE,

Abbr.	Name	Explanation
Total	Total diagnoses possible	Count of snapshot deltas with valid measurements
AP	Actual Positive periods (known positives)	Number of measurement periods where a fault was active
AN	Actual Negative periods (known negatives)	Number of measurement periods where a fault was *not* active
TP	True Positive diagnoses	Count of our positive diagnoses that were also Actual Positives
TN	True Negative diagnoses	Count of our negative diagnoses that were also Actual Negatives
FP	False Positive diagnoses	Count of our positive diagnoses that were Actual Negatives
FN	False Negative diagnoses	Count of our negative diagnoses that were Actual Positives
TPR	True-Positive Rate (Sensitivity)	% of Actual Positives (AP) that were correctly diagnosed
FPR	False-Positive Rate	% of Actual Negatives (AN) that were False Positives (FP)
PPV	Positive Predictive Value (Precision)	When the diagnosis is positive, what % of the time is it correct?
TNR	True-Negative Rate (Specificity)	% of Actual Negatives (AN) that were correctly diagnosed
FNR	False-Negative Rate	% of Actual Positives (AP) that were False Negatives (FN)
NPV	Negative Predictive Value	When the diagnosis is negative, what % of the time is it correct?

Table 4.2: Abbreviations for evaluation tables.

and MergeTreeMux experiments, finding almost all of the Actual Positive (AP) periods with only two False Positives (FP) and zero False Negatives (FN).

4.6.3.1 Barrier results

However, StreamsDiagnoser did not do as well on the MergeTreeBarrier and MergeTreeBarrierMux experiments, with a 7.7% False Positive Rate (FPR) and 17.9% False Negative Rate (FNR). In fact, when StreamsDiagnoser

Module	Correct Answers			Diagnosis Results				Positive Accuracy %			Negative Accuracy %		
	Total	**AP**	**AN**	**TP**	**TN**	**FP**	**FN**	**TPR**	**FPR**	**PPV**	**TNR**	**FNR**	**NPV**
ComplexTree	6138	114	6024	114	6024	0	0	100.0	0.0	100.0	100.0	0.0	100.0
ComplexTreeMux	7182	126	7056	126	7054	2	0	100.0	0.0	98.4	100.0	0.0	100.0
MergeTree	1035	35	1000	35	1000	0	0	100.0	0.0	100.0	100.0	0.0	100.0
MergeTreeMux	1150	28	1122	28	1121	1	0	100.0	0.1	96.6	99.9	0.0	100.0
MergeTreeBarrier	2736	123	2613	101	2413	200	22	82.1	7.7	33.6	92.3	17.9	99.1
MergeTreeBarrierMux	3078	123	2955	105	2745	210	18	85.4	7.1	33.3	92.9	14.6	99.3
Totals	**21319**	**549**	**20770**	**509**	**20357**	**413**	**40**	**92.7**	**2.0**	**55.2**	**98.0**	**7.3**	**99.8**

Table 4.3: Diagnosis accuracy from controlled StreamsDiagnoser experiments with a fixed-rate source; columns are defined in Table 4.2 and discussed in Section 3.5.2.1. Statistical uncertainty is less than 0.6% for all measurements, except for MergeTree-Barrier (3.5%) and MergeTreeBarrierMux (3.2%).

blamed a module in the MergeTreeBarrierMux experiment, it was correct only 33% of the time (PPV). This led to a very low 55.2% overall Positive Predictive Value (PPV, Total) in the fixed-rate experiments.

This is actually the result that should be expected: two of the DynamicThrottle PEs (circled in red in Figure 4.6(e) and (f)) are directly upstream from the Barrier PE (2nd PE from the right in both graphs).

When Throttle1 is set to throttleRate = 0, the Barrier cannot proceed since one of its inputs is empty. So, the Barrier stops reading on its *other* input ports, and those stream connections are marked as Stalled. They are, in fact, stalled according to the StreamsDiagnoser diagnosis criteria, but those criteria do not account for a Barrier's intended behavior.

When Throttle2 is *also* set to throttleRate = 0, since the Barrier has already stopped reading the tuples on the connection from Throttle2 → Barrier, instead of blaming the stream coming *into* Throttle2 (as the evaluation expects), StreamsDiagnoser again blames the Barrier. This causes the False Negative (FN) results.

Chapter 7 describes a potential approach that incorporates additional per-module information (introspection) to allow StreamsDiagnoser to diagnose these situations more appropriately.

4.6.3.2 Full-rate results

In addition to the rate-limited experiments, we ran a batch of full-rate tests on a single machine. Since there were more PEs than processor cores, this simulates an

overloaded system. Both batches included the same set of fault injections described in Section 4.6.2.

In the full-rate experiments, StreamsDiagnoser often marks modules other than the DynamicThrottle as STALLED. The calculated False Positive Rate (FPR) is as high as 7.5% for the full-rate COMPLEXTREE, unexpectedly blaming modules in 5,410 of 72,503 of the Actual Negative (AN) periods. A manual analysis using the visualizations indicates that StreamsDiagnoser is actually correct in the vast majority of these situations, as the system appears to move tuples through the graph in batches. The resulting backpressure causes the blame to oscillate between various modules, depending on which ones are active at a given time. We are working with the third-party developers to determine the root cause of this behavior.

4.7 Live application results

In addition to the controlled experiments performed on the six basic topologies, we evaluated StreamsDiagnoser on several real applications from a national laboratory and IBM research:

- App1: A highly tuned application designed for real-time data processing, with many operators combined into a few (thirteen) PEs. We analyzed 7748 snapshots taken over a two hour time period.

- App2: A complex and dynamic application, which varies the number of PEs and their connections over time based on demand and other factors. A snapshot of the processing graph is shown in Figure 4.1. We processed several runs

from this application, including over 20,000 snapshots from 963 modules in one run, and over 40,000 snapshots from 493 modules in another run.

The developers of App1 have done significant work optimizing their system for high throughput, including the use of a source-to-source translator that inserts code to wrap each tuple with pre- and post-processing timestamps. As a result, the experimental data provided for analysis did not show any significant performance issues. App1 did provide a useful baseline for the StreamsDiagnoser data collection, transformation, and diagnosis engine. Working with the data from this application helped to identify and fix several data collection and transformation bugs, and underscored the importance of the normalization steps described in Section 4.3 and Appendix A.

One of the runs for App2 included a crashed PE, which was marked as failed by the Streams runtime system. Because it had not processed all of the tuples in its queue when it failed, but remained in the graph, StreamsDiagnoser was able to correctly identify the crashed PE as the cause of backpressure-related performance stalls reaching up to 13 PEs upstream in a chain.

Overall, the experience with App2 was mixed due to the variability in the number of connections to each PE. This identified two undesirable aspects of the current StreamsDiagnoser prototype: the potential accumulation of errors from one snapshot to the next, and the possibility of an errant assignment of blame due to errors.

In practice the StreamsDiagnoser prototype has a much stricter correctness requirement on the counter values than NeST. The NeST prototype has two desirable and related features. First, counters are required only to be monotonically increasing and do not have to be absolutely correct over time. Second, the results from each snapshot are largely memory-free, i.e., any errors in earlier snapshot values are quickly discarded, since the diagnosis at each snapshot depends on a binary test on the change since the last snapshot: Is the current delta greater than zero?

Since StreamsDiagnoser calculates the $\sigma_{\texttt{queued\_msgs}}$ value at each snapshot as $(\sigma_{\texttt{nSubmitted}} - \sigma_{\texttt{nProcessed}})$, any error in obtaining or calculating the `nSubmitted` or `nProcessed` counters accumulates over time. That is, an off-by-one (or off-by-many) error in either counter can make the difference in whether a module is marked as Blocked, Stalled, or DontCare.

These potential errors may be compounded by the blame-passing rule described in Chapter 2, which transfers blame from an inactive parent to its inactive children if they both have work to do. If the child's queue erroneously appears to have tuples left in it ($\sigma_{\texttt{nSubmitted}} > \sigma_{\texttt{nProcessed}}$), then blame may be incorrectly passed from the parent to child, resulting in a false positive diagnosis for the child, and a false negative for the parent. On the other hand, if the child's queue incorrectly appears to be empty ($\sigma_{\texttt{nSubmitted}} \leq \sigma_{\texttt{nProcessed}}$), blame may not be passed to the child when it should, resulting in a false negative diagnosis for the child, and a false positive for the parent.

These problems occur only if the values provided by the monitored system are an unreliable representation of the actual number of messages sent on a given

connection. Since the number of connections changed frequently during each run of App2, it was difficult to determine in some cases if StreamsDiagnoser was flagging actual problems in the application, or merely interpreting bad data. Additional work is necessary to obtain the true per-connection submitted and processed counters. The most plausible solution is to ignore any transition snapshots where the number of connections to a PE port changed, and assume that the overall number of tuples stuck in the queue did not change during the transition snapshot. Another option would be to obtain per-connection counters directly from the underlying InfoSphere Streams framework, but this would require changes to the underlying system that may be non-trivial to implement.

Both development groups confirmed that capturing the processing graphs and snapshots had no measurable impact on their application performance, and have requested further collaboration to apply StreamsDiagnoser in their environments.

4.8 Summary

This chapter presents StreamsDiagnoser, an application of the FlowDiagnoser approach to finding performance stalls in InfoSphere Streams. Experimental evaluation shows that StreamsDiagnoser is accurate enough to detect 93% of injected faults, with a False Positive Rate of 2%, and efficient enough to run on real Streams applications.

Chapter 5

From Diagnosis to Fix

FlowDiagnoser analyzes each module's performance for each snapshot taken of the modules' counters, thus creating a series of per-module diagnoses over time. The expert user or administrator uses this information to understand the causes of observed performance problems so they can be fixed.

This chapter describes the discovery process one can follow, using the FlowDiagnoser output, to investigate the causes of performance stalls, and determine which areas of their system need further investigation. It also describes the summarization and visualization features made possible by the FlowDiagnoser approach; these features represent a sketch of what is possible, not finished products. After briefly describing the discovery process and summarization features, the chapter presents a case study of one of the StreamsDiagnoser experiments from Chapter 4.

5.1 The Discovery Process

To investigate the performance problems diagnosed by FlowDiagnoser, a user follows three basic steps:

1. Get an *overview* of the system

2. See a *summary* of the results

3. Perform an in-depth *analysis* of the modules' performance

Before determining what parts of the system need attention, the user needs a broad *overview* of the system. While an expert user may already know the basic components of the system, unless the system is static, it is unlikely the user knows what it looked like at any given point in time. That is, while she may know generally which modules should be included in the dependency graph, she may not know which ones actually existed at a given point in time, or when they started and stopped. To aid in this process, FlowDiagnoser is able to automatically create two overview visualizations: a timeline showing each module's lifespan, and a time-agnostic version of the dependency graph that includes all modules that ever existed. Of course, given a recorded monitoring trace, FlowDiagnoser is also able to recreate the dependency graph as it existed at any point in time.

Once the user has a broad overview, she will likely want to focus her attention on the modules that were blocked or stalled most frequently. To do this, FlowDiagnoser can output a *summary* table that lists the modules with the most STALLED diagnoses. It can also group the list of stalled modules by important features, e.g., all TCP connections to a certain remote IP:port, or all PEs that implement a certain operator.

Once the user has the list of STALLED modules, she can perform the *analysis* required to determine which modules had the greatest impact on the system. FlowDiagnoser provides two time series visualizations that allow the user to see each module's diagnosis and counters *in relation to other modules in the graph*. In com-

Output	Purpose	Description
Timeline	overview	Displays modules' arrival and departure over time
All-modules graph	overview, summary	The dependency graph of all modules that existed during the monitored session, colored by overall diagnosis results. Includes diagnosis counts, STALLED timespan summary, and optionally the min/max value of each counter.
Stalled modules list	summary	List of all modules that were ever diagnosed as STALLED, ordered by number of STALLED diagnoses
Counters heatmap	analysis	Displays the change in each module's counters over time, in context with the rest of the dependency graph
Diagnosis timeseries	analysis	Displays each module's diagnosis results over time

Table 5.1: FlowDiagnoser Summaries.

bination with the dependency graph summary, the user can determine how often each module blocked its parents or was itself blocked by its own children.

5.2 Summarization and Visualization Outputs

Table 5.1 lists the overview, summary, and analysis outputs automatically created by FlowDiagnoser.

Although not presented in detail here, Visty, an early prototype visualization for the network stack, provides several overview features that should factor into any FlowDiagnoser visualization interface [54]. Most important are the timeline feature that shows the active modules over time, and a visualization of the dependency graph that fits the user's intuition. The timeline enables the user to see which modules were alive at any point in time, and to narrow their analysis to time periods (and

modules) of interest. Figure 5.1 shows an example NeST timeline, and displays which instrumented applications were running at each point in time.

The second overview visualization is a representation of the FlowDiagnoser dependency graph itself. Figure 5.2 shows an early visualization of the network stack (which lacks the diagnosis summarization described in Section 5.1). The following case study includes an example of the all-modules graph from the StreamsDiagnoser prototype, and explains how the summarization aids in the discovery process. The case study also includes detailed examples of the next three outputs.

The stalled modules list is a simple summary of all the modules that FlowDiagnoser marked as stalled, ranked in descending order of number of Stalled diagnoses.[1] This provides the user with a prioritized list of modules to investigate during the analysis phase.

Finally, for in-depth analysis, FlowDiagnoser provides two visualization time series: the counters heatmap and diagnosis timeline plot. Section 3.5.4 provides a walkthrough of several counter heatmaps and diagnosis plots from the NeST experiments. The counter heatmap shows the counters for each module in the dependency graph (or subgraph); viewing the counters for dependent modules together makes it possible for the user to see how the modules interact. When used in conjunction with the diagnosis plot, which shows each module's diagnosis result over each snapshot, this can direct the user's attention to significant time periods, and the counter heatmaps aid in understanding what triggered the diagnosis.

[1] A similar list of Blocked modules can be implemented easily.

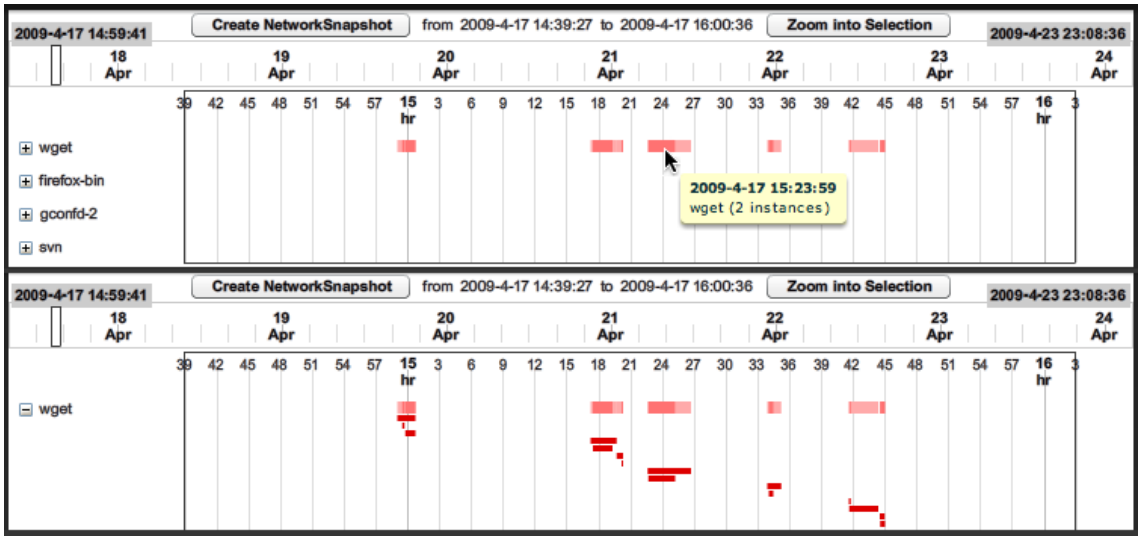


Figure 5.1: Network Stack Trace module timeline. The top pane shows the lifespans for the various monitored processes. The bottom pane shows an expanded timeline for the `wget` module. It includes the number of times `wget` was started and stopped (light pink bars), and the underlying socket and TCP connections (darker red bars underneath).

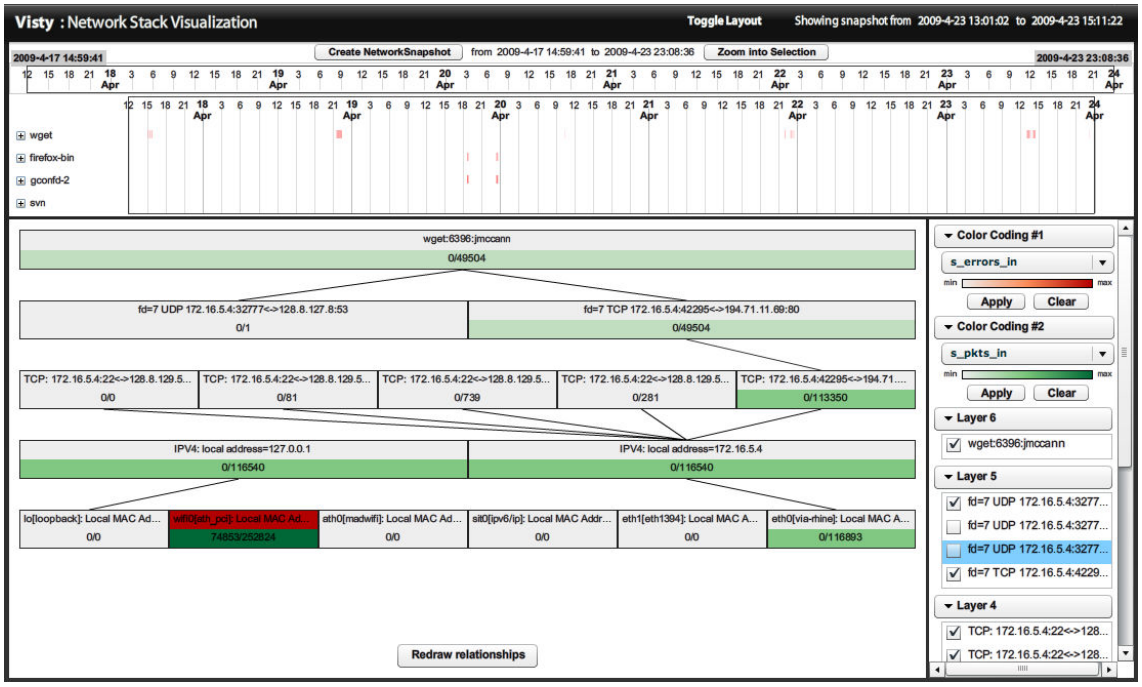


Figure 5.2: Network Stack Trace visualization prototype. It consists of an overview timeline in the top pane, and a depiction of the network stack in the bottom left pane. In the right pane, the user can select particular metrics for coloring nodes according to number of messages sent or received, or show/hide particular modules.

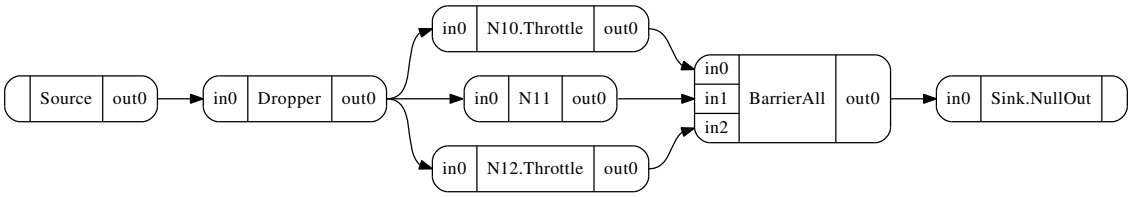
5.3 Case Study: MergeTreeBarrier

The case study concerns the MergeTreeBarrier experiment presented in Section 4.6, which includes a Barrier operator that reads one tuple from each input port before submitting any tuples on its output port. If any input port does not have tuples to process, then the barrier blocks the other ports (i.e. stops processing on other ports) until a tuple arrives on the empty queue.

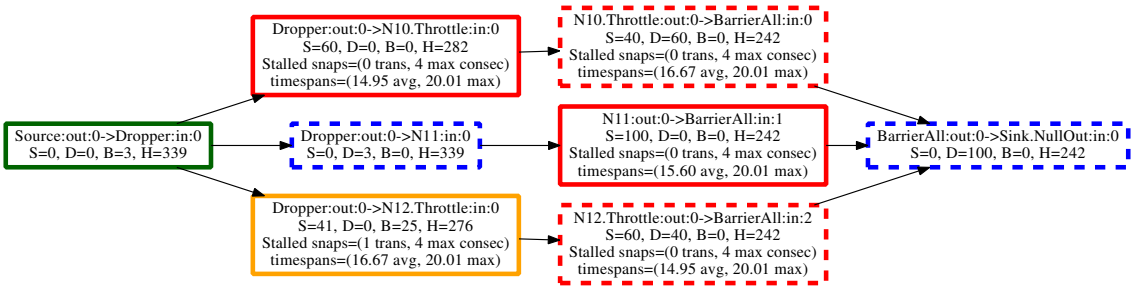
In this case study, the developer has been monitoring the MergeTreeBarrier job, and notices that the output rate is very low at times, and sometimes stops completely. Her goal is to identify which modules are causing the performance stalls, determine how they affect the rest of the system, and work toward a fix. Since the system is being monitored by StreamsDiagnoser, she consults the diagnosis output it provides.

5.3.1 Steps 1 and 2: Overview and Summary

The developer's first step is to gain an overview of the system itself. Figure 5.3(a) shows the original Streams processing graph for the MergeTreeBarrier topology (from Figure 4.6(e)). Connected to the BarrierAll operator are two DynamicThrottle operators, N10 and N12, and a simple pass-through (N11) which connect to the operator.



(a) MergeTreeBarrier processing graph. Nodes in this graph are PEs/operators.



(b) MergeTreeBarrier dependency graph. Nodes in this graph are the connections between PEs/operators. Dashed lines indicate DontCare (module was starved of data at some point). Red lines indicate some Stalled. Orange lines indicate some Stalled and Blocked. Blue lines indicate Healthy and DontCare only. Dark green lines indicate Healthy and Blocked only.

Figure 5.3: MergeTreeBarrier processing and connection dependency graphs.

Stream Connection		Diagnosis Counts				queued_msgs	
Upstream Op.	**Downstream Op.**	**Stall**	**DCare**	**Block**	**Healthy**	**min**	**max**
N11:out:0	BarrierAll:in:1	100	0	0	242	-7	8300
N12.Throttle:out:0	BarrierAll:in:2	60	40	0	242	-6	7677
Dropper:out:0	N10.Throttle:in:0	60	0	0	282	0	8148
Dropper:out:0	N12.Throttle:in:0	41	0	25	276	0	8304
N10.Throttle:out:0	BarrierAll:in:0	40	60	0	242	-5	8297

Table 5.2: MergeTreeBarrier stalled connections, ordered by total number of Stalled periods. The rightmost section lists the minimum and maximum values calculated for the `queued_msgs` counter.

5.3.1.1 All-modules graph

Figure 5.3(b) shows the resulting all-modules graph. This graph provides an overview of the system, in the form of the basic StreamsDiagnoser dependency graph, and a partial summary of the performance results, in the form of labels and colors added to this graph. Any modules that were ever marked as DontCare have dashed outlines. Modules that were Stalled are red if they were never Blocked, otherwise they are orange. The source module is dark green since was Healthy overall, although it was Blocked during three snapshots.

Looking at Figure 5.3(b), the user sees that the connections to both the Sink and to N11 show dashed blue lines, indicating that they have been mostly Healthy, but have been starved of data at times (DontCare). She also knows that the Barrier's three inputs have each been Stalled, as well as the N10.Throttle and N12.Throttle. The N12.Throttle is orange, so she is aware that it was occasionally Blocked.

The first line of each module is the stream connection name, followed by a count of the diagnoses assigned to each module on the second line (S=Stalled, D=DontCare, B=Blocked, H=Healthy).

For any modules that were Stalled, the summary displays two additional lines of text. The first is a summary of the number of Stalled results that were transient ("trans"), followed by the maximum number of consecutive Stalled snapshots. The final line gives the average and maximum amount of time that the mod-

ule was consecutively STALLED, excluding the transient stalls, which are roughly 5 seconds each (the snapshot interval).

Looking more closely, the user sees that the orange-colored module Dropper:out:0 → N12.Throttle:in:0 was stalled 41 times (S=41), with only one transient stall (1 trans). Of the remaining 40 stalled snapshots, the average timespan was 16.67 seconds, and maximum 20.01 seconds.

She also sees that the Source operator on the left has been HEALTHY overall (H=339), but was blocked 3 times. The Sink on the right was HEALTHY for 242 snapshots (H=242), but had nothing to do (DONTCARE) for 100 snapshots (D=100).

5.3.1.2 Stalled modules list

With this overview of system health, the user knows that five of the modules have been STALLED at some point. She then reviews Table 5.2. This table contains a ranked list of all modules that were STALLED during MERGETREEBARRIER experiment, ordered by total number of STALLED periods. Since the StreamsDiagnoser analysis focuses on which input ports are not processing the data provided to them, the focus is on the second column which lists the downstream operator (PE) and port.[2]

[2]Since one Streams operator can be used multiple times in a single application, assigned to different PEs, StreamsDiagnoser provides two views: one that looks simply at the PE-to-PE connections, and another grouped by the downstream operator name. The PE list shows which processes were STALLED, and the operator list which source code operators were STALLED.

The `BarrierAll` operator has been STALLED on all three of its input ports, most often on `input:1`. This indicates that `BarrierAll` often stopped reading on `input:1`, causing a maximum `queued_msgs` of 8300.[3] For a normal SPL operator, the user might focus her attention on how the code processes tuples from `input:1`. However, she knows that BarrierAll treats its inputs specially: it intentionally *stops* reading from inputs with tuples in them if any *other* input is empty. So, the user knows that `N11:output:0` is providing data that sometimes does not get processed immediately, because the barrier is waiting for data on its other inputs.

Because of BarrierAll's design, the user focuses her attention on its other ports. She wants to see how often they are marked as DONTCARE, indicating that the upstream did not provide enough data to process. She notices that N12.Throttle:out:0 → BarrierAll:in:2 was diagnosed as DONTCARE 40 times, and N10.Throttle:out:0 → BarrierAll:in:0 60 times. Looking further, she sees that Dropper:out:0 → N10.Throttle:in:0 has 60 STALLED diagnoses: apparently the stalls by the throttle operator are blocking flow to the barrier. Once the connection queue fills, this can cause upstream operators such as `N11` to be blocked from sending data on their outputs.

5.3.2 Step 3: Analysis

The summary information helps the user understand where more detailed investigation is needed. To support such investigation, FlowDiagnoser provides two

[3]Section 4.3.2 explains why the minimum queue size is sometimes less than zero: timing variations between when the upstream and downstream counters are recorded.

timeline visualizations. Figure 5.4 includes the timeline plots for the MergeTree-Barrier experiment.

The top plot is a counter heatmap, which shows the rate of change for each module's counters over time, and in relation to each other.[4] Each module has three rows assigned to it:

- Number of tuples `nSubmitted` into the connection during the snapshot
- Total `queued_msgs` sitting in the connection at the end of the snapshot
- Number of tuples `nProcessed` out of the connection during the snapshot (`total_msgs`)

The abbreviation S, Q, or P in each row label identifies the counter shown. The label also includes the minimum and maximum delta (Δ) for each counter in parentheses. Ancestor modules are listed toward the top of the heatmap plot, and descendants toward the bottom.[5]

To direct the user's attention, the diagnosis timeline at the bottom shows which modules were Stalled ($y = +1.0$), DontCare ($y = 0.0$), Blocked ($y = -0.5$), and Healthy ($y = -1.0$).

The following pattern for reading the heatmap and diagnosis timeline is useful:

[4] Section 3.5.4 includes additional discussion.

[5] The user must be careful to determine which groups contribute to each path through the graph; further research in this area is needed.

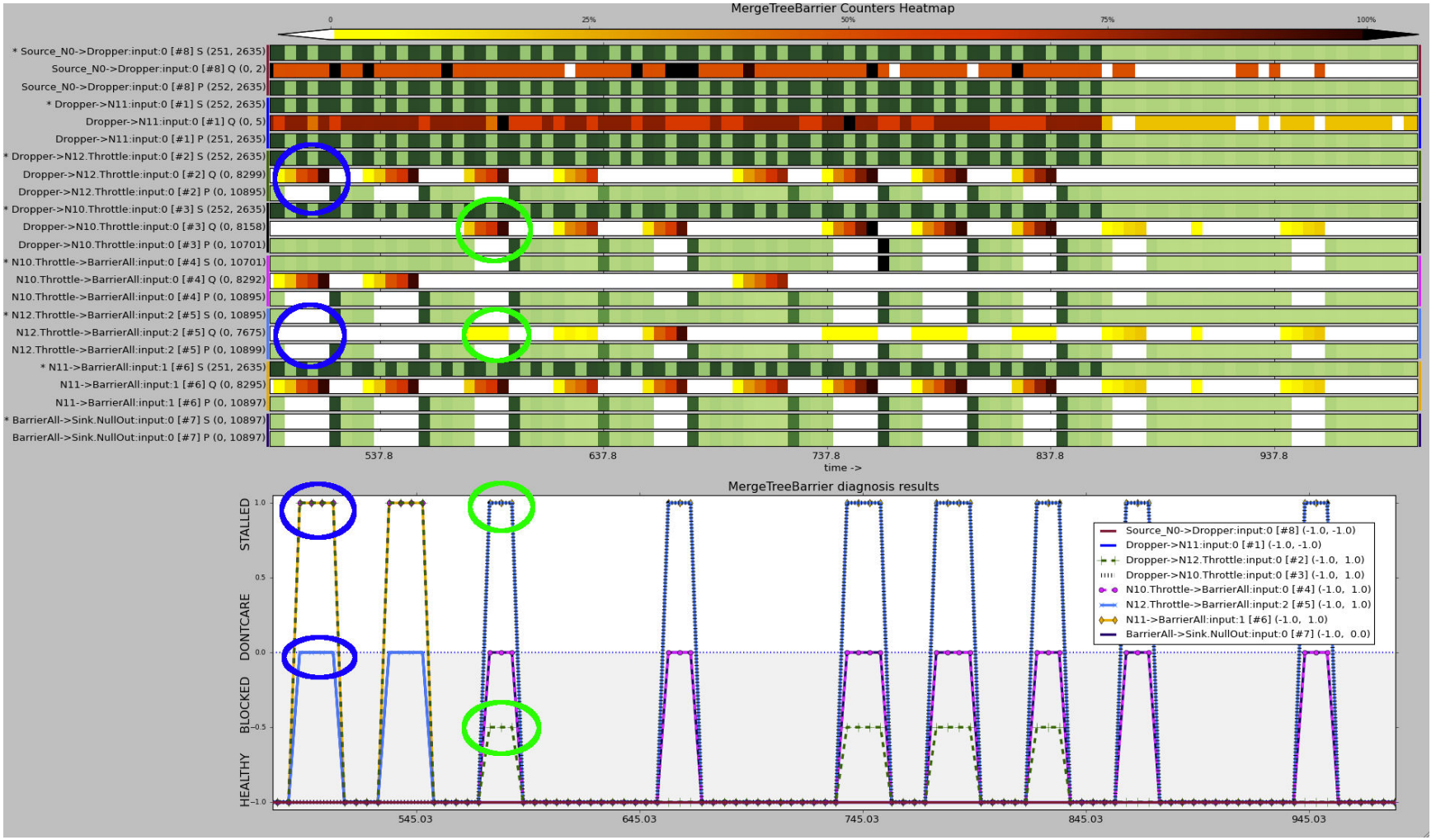


Figure 5.4: MergeTreeBarrier diagnosis timeline. The top figure shows the counters heatmap, and the bottom figure the StreamsDiagnoser results.

1. Green is good, and is a sign of messaging activity. In the `total_msgs` rows, black is also good, and marks the maximum number of messages processed in any period.

2. White denotes inactivity in all rows: `total_msgs` is 0, `queued_msgs` is empty, `wait_time` is 0.

3. Yellow, orange, red, and black are bad, and denote increased waiting or queue size.

4. Look to the diagnosis timeline for relevant events, especially transitions from HEALTHY to STALLED, or from HEALTHY to BLOCKED.

5. When a module is STALLED or BLOCKED, it is inactive, so its `total_msgs` counter heatmap will be white. The user then looks to that area of the heatmap.

6. When a module is STALLED, either the `wait_time` or `queued_msgs` is usually increasing, so the user looks for warning signals (yellow, orange, red, black) in those counter rows. For StreamsDiagnoser, the user looks to the streams-connection module itself, and then its parents if they are BLOCKED. For NeST, the user looks to the application and sockets.

7. Some modules may be active (green counter rows) while waiting or queueing (warning colors). This may be due to normal buffer fluctuations, but an increasingly darker shade of red indicates a processing bottleneck.

Figure 5.4 highlights two representative STALLED periods, with the salient features circled in blue (near time 500 on the x-axis), and in green (near time 580 on the x-axis).

5.3.2.1 N12.Throttle starves the barrier

In the first highlighted period, with blue circles, the connection labeled as Dropper→N12.Throttle:input:0 is STALLED since the N12.Throttle operator stopped reading from `input:0` (top blue circle). The Dropper PE continues to submit tuples into the connection, as seen in the continued green section in the top row circled, but none are being processed (white section in the bottom row circled). Since more tuples are being submitted than processed, the queue begins to build (yellow-orange-red-black progression in the middle row).

Since N12.Throttle has stopped processing tuples, it has none to submit downstream. This starves its downstream (child) connection N12.Throttle→BarrierAll:input:2 of data, and no tuples are submitted onto it (2nd blue circle). Thus all three rows in the 2nd circle are white, and the module is marked as DONTCARE (4th blue circle).

Since no data is coming into BarrierAll:input:2, the barrier stops reading on BarrierAll:input:0 and BarrierAll:input:1. Therefore, three connections are marked as STALLED (3rd blue circle, some lines occluded):[6]

- Dropper→N12.Throttle:input:0

[6]The evaluation in Section 4.6.3 counts the second two faults as False Positives.

- N10.Throttle→BarrierAll:input:0
- N11→BarrierAll:input:1

By cross-referencing the dependency graph from Figure 5.3 and the timeline plot, the user now knows that the Barrier has stopped reading from the N10.Throttle and N11 operators because the N12.Throttle stopped processing from its input:0.

5.3.2.2 N10.Throttle starves the barrier

In the second highlighted period, with green circles, both the N10.Throttle and N12.Throttle stop processing (reading) from their input ports. In this case, the N10.Throttle PE stopped processing first, so its queue began to build (top green circle). This starves N10.Throttle→BarrierAll:input:0 of data, so BarrierAll immediately stops reading from input:1 and input:2.

The 3rd green circle shows that StreamsDiagnoser marks the following connections as STALLED:

- Dropper→N10.Throttle:input:0
- N11→BarrierAll:input:1
- N12.Throttle→BarrierAll:input:2

The 2nd green circle highlights the tuples stuck in the connection N12.Throttle→BarrierAll:input:2, since BarrierAll has stopped reading from that input. Notice that the Dropper→N12.Throttle:input:0 connection *also* has a queue building (yellow-orange-red-black progression above the 1st green circle), since the

N10.Throttle and N12.Throttle have stopped processing almost simultaneously. However, since StreamsDiagnoser has already marked the N12.Throttle's child (downstream) connection as STALLED, the dependency analysis assumes that N12.Throttle cannot submit any more tuples to its connection, and marks the connection Dropper→N12.Throttle:input:0 as BLOCKED (4th green circle).

Again, by viewing the diagnosis timeline alongside the dependency graph, the user has identified that the BarrierAll operator stops processing from its inputs (potentially affecting N11 and N12.Throttle), because the N10.Throttle stopped processing from input:0.

The developer's original goal was to determine why the Sink.NullOut operator was not sending any data to the Sink. She now knows that processing stalls on N10.Throttle and N12.Throttle are causing the performance problems, and looks to the source code and system configuration to fix it.

5.4 Conclusion

By tracking the messaging performance of a system over time, FlowDiagnoser provides an expert user with the tools necessary to determine the cause of messaging performance stalls. In addition to the time series of diagnosis results, FlowDiagnoser provides three main types of system analysis tools: an overview of system communication, summaries of the automatic diagnosis results, and visualizations for in-depth analysis.

Chapter 6

Related Work

FlowDiagnoser describes a general approach to diagnosing the source of performance stalls in dataflow-oriented systems. It requires no protocol-specific knowledge other than a basic understanding of the forwarding semantics of the underlying system. It also requires little information: one counter per module, plus some waiting signal, in contrast to other approaches which require extensive instrumentation or message and event traces. FlowDiagnoser is also able to reliably diagnose the source of network performance stalls using only the information available at the local host.

Many systems have been proposed for diagnosing some aspect of network or software performance [2, 6, 15, 26, 37, 39, 45, 60]. Some of them are geared toward software analysis [26, 45] or specific network protocols [13, 15, 37, 39, 60], and have often relied on intimate familiarity with protocol design and implementation details. Few of these systems are useful if employed independently on a single host; its limited perspective makes diagnosis difficult [13]. Cooperative diagnosis, however, is of little use to a host that cannot reliably establish or complete network connections: diagnosis fails at the time it is needed most.

6.1 Protocol-Specific Analysis

Many systems perform protocol-specific analysis to detect performance problems exhibited by specific network or application technologies [2,15,32,35,60]. These analyses can identify very complicated protocol- and domain-specific performance problems, but applying their insights to new problem domains is not straightforward.

NeST is perhaps most inspired by the Web100 project [39], which thoroughly instrumented the Linux TCP stack to provide insight into network and TCP implementation behaviors. By monitoring the sender-side behavior of TCP's flow control and congestion avoidance mechanisms, Web100 can reliably distinguish between sender-, receiver-, and congestion-limited periods for transmissions from the local host.

Pathdiag is an active measurement tool built on Web100 that injects traffic into the network, and uses the Web100 instrumentation to detect common network performance problems along a path [37]. The mechanisms built in to Web100 and Pathdiag are limited to diagnosing behavior from the TCP sender side, since they use signals such as an increasing round trip time (RTT) and particular loss patterns to detect host misconfigurations, large router queue sizes, and excessive loss. NeST applies a more general approach but gives less-specific feedback: for example, Pathdiag might warn about a poorly sized router queue, while NeST simply blames the TCP connection itself when it stalls. NeST, however, is able diagnose performance stalls from both the sender's and receiver's point of view.

The TCP Rate Analysis (T-RAT) tool [61] uses packet captures to determine performance limitations of TCP/IP flows across a network backbone. T-RAT classifies each flow according to seven separate application, network, and TCP-specific performance limitations. It does so by grouping series of packets in each flow into flights of consecutive packets, and comparing the performance of successive flights of packets to determine the limitation of each flight. To do this, it makes use of several TCP-specific performance heuristics.

A follow-on approach described by Siekkinen, et al, [48] improves upon T-RAT's flight-based approach, which they show to be unreliable in practice, to classify each 500 millisecond window of packets as application-, TCP-endpoint-, or congestion-limited. To do this, they also employ several TCP-specific heuristics that require parameter tuning, such as the ratio of packets marked with the PUSH flag, and number of time periods the flow was above and below this threshold.

Both of these tools provide interesting insights into the performance of TCP flows, but their use of packet captures requires them to use some rather unreliable inference techniques, while NeST is able to detect many of these performance limitations directly. The FlowDiagnoser approach is also applicable to other network protocols and messaging systems, and not strictly limited to TCP analysis.

The Scalable Network-Application Profiler (SNAP) [60] uses application event logs and Web100-like TCP instrumentation [38] to identify the causes of network-related performance problems in datacenter applications. SNAP employs a sender-side, TCP-specific protocol analysis, and correlates connection-specific problems across the data center and applications.

The SNAP paper describes fifteen significant performance bugs that were found as a result of their analysis. The SNAP approach is very similar to NeST in both its high-level goals and basic structure: continuous monitoring of protocol counters and application events. The FlowDiagnoser approach, however, is more general and not tied to the nuances of the TCP protocol stack, although with less specific results. For example, three of the bugs found by using SNAP related to the interaction between delayed ACKs and the TCP buffer sizes requested by the application. NeST would correctly identify the TCP connections and applications that were stalled as a result, but because of the TCP-specific delayed ACK rule used in SNAP, it is able to identify the specific TCP feature causing the stall.

By implementing the SNAP techniques as a module-specific diagnosis inside of NeST (Section 7.2), NeST may be able to increase the level of detail provided in its diagnosis results. FlowDiagnoser is also applicable to higher-level application components, as demonstrated in this thesis's applications of it to the network stack and streams.

NetPrints [2] is a system that performs automated network diagnostics to detect and correct home network device *misconfigurations* that prevent specific applications from functioning. It does so by employing a decision-tree algorithm to compare a user's faulty configuration against configurations supplied by other users. It also looks for certain features in the network behavior—including specific SYN/RST patterns in TCP connection attempts, one-way traffic without responses, and lack of any inbound/outbound traffic—that may indicate certain types of network faults that prevent communication. These network features and configurations are com-

bined to detect and potentially repair misconfigurations on the end host, firewall, or router.

While both NeST and NetPrints perform automated network diagnostics, their goals are orthogonal: NeST diagnoses performance stalls that occur during normal operation, and NetPrints detects misconfigurations that prevent throughput altogether. For example, when a user signals a problem with their application, NetPrints can determine that the application is attempting to connect remotely to their FTP server at home, and recommend enabling forwarding for port 21 on the home gateway. NetPrints relies on a centralized diagnostic server to collect and analyze information provided by many end hosts; this reliance on a centralized service is problematic during periods of extremely poor network performance.

TcpEval [10] is a tool designed to determine where delays in HTTP responses are introduced. It analyzes packet traces to construct a packet dependence graph to determine which TCP SYNs, data, and ACKs must happen before each other. By evaluating the timing over this happens-before graph [34], TcpEval is able to distinguish client delays, server delays, and various network behaviors. TcpEval's analysis is specific to the type of TCP congestion algorithm used, and does not readily generalize to other protocol types. Since it works over packet traces, it is also less efficient to implement than NeST's counter analysis.

Some software packages include application-specific monitoring and diagnosis frameworks. The Firebug [22] and YSlow [57] web browser add-ons and the Network Panel in Google Chrome [56] each provide a timeline view of the various elements (HTML, Javascript, CSS, images, embedded videos, etc.) loaded by a web browser

when displaying a web page. By analyzing the sequence of objects and the amount of time each takes, a web developer can determine which page elements cause slow page load and rendering times, which are a significant cause of website user dissatisfaction (Chapter 1). YSlow works with Firebug to rank each web page according to a series of performance rules that encode current best practices [58,59], such as "Gzip Components" and "Make Fewer HTTP Requests" [49].

WebProphet [35] is an automated tool for predicting web page load times, related in spirit to Firebug and YSlow, but quite different in approach. WebProphet creates a dependency graph of all of the objects (HTML, CSS, Javascript, images, etc) on a given web page by delaying the loading of particular objects; dependent objects are delayed by the same amount. By including TCP and DNS round-trip times, WebProphet is able to estimate the impact of changing various parameters (for example, the use of longer DNS or object cache times, or lowering the RTT by using a CDN).

Each of these web-specific tools provides useful feedback for web developers and site administrators to evaluate their web pages and sites according to current best practices. However, when an element takes an inordinate amount of time to load, these tools do not provide any insight into what was occurring in the application or network at the time, only the duration of the request. NeST provides an additional level of detail about the network performance of the application, and could make it clear that the object was delayed due to network issues, a stalled TCP connection, or a browser issue. Integrating the NeST diagnosis results with the application-

specific knowledge available in Firebug or the Chrome Network Panel would help to overcome the limitations described in Section 3.6.2.

6.2 Required Information

FlowDiagnoser requires very little information to make an accurate diagnosis, while other systems require much more. While these systems model performance-relevant behavior at a much finer level of detail, their extensive overhead makes it difficult to justify running them continuously, or analyzing all of the data.

6.2.1 Extensive instrumentation

NetLogger [26] and Pip [45] are examples of application-based instrumentation systems, which involve inserting specially tagged logging messages before and after function calls at critical places in application code. By carrying along unique request identifiers throughout the code and messaging flow, it is possible to identify where delays are caused across an entire distributed system.

Both systems provide insight into application and connection delays down to the function level, which is very useful for pinpointing the source of problems. The volume of collected data can be expensive to process, and while instrumenting fewer functions could reduce this volume, it would also reduce the utility of the approach. Although NeST does instrument an application's read and write calls through a `libc` interceptor library, it does not track individual messages; both NeST and StreamsDiagnoser require only per-module counters, which greatly re-

duces the required information. For both NetLogger and Pip, the requirement to add instrumentation to application source code is a high barrier to entry, especially for third-party applications where the source code is not available.

6.2.2 Packet or event traces

Some diagnosis systems use expensive traces of network packets [6, 14, 15] or system-level messages [3,18,25,26,45] to build the communication dependency graph, diagnose performance anomalies, and detect violations of system invariants.

Sherlock [6] is designed for large enterprise use, and employs end-host packet captures to assemble dependencies between network services (IP 3-tuples) at the host level. They then combine this service dependency graph with an externally-generated network topology to create an *inference graph*, which models physical components (machines, routers, links) and services (IP + port) as "root cause" nodes, clients as "observation nodes" which can collect response-time metrics, and "meta-nodes" to connect clients with the root causes. The observed service response times are used to place nodes in up, down, or troubled states; Sherlock then identifies the source of performance problems by searching for the highest-probability set of up to k causes that might explain the observed troubled/down states.

Orion [14] improves upon Sherlock's service discovery mechanisms by tracking delay distributions across the various services, learned from packet traces captured within the network: spikes in delay are likely to be correlated among services that depend on each other. Orion's associations are more fine-grained than Sherlock's,

able to track multi-process and multi-host applications, and permit far fewer false-positive edges in the dependency graph. Orion does not directly deal with performance and fault diagnosis, but provides more reliable dependency graphs to use for diagnosis.

Both Sherlock and Orion have a different goal than FlowDiagnoser: discovery of inter-system dependencies across an enterprise network, and in the case of Sherlock diagnosis of service and network delay spikes and failures. Due to its instrumentation approach, FlowDiagnoser does not need to infer the dependencies between processes and services (even across the network); it can learn and report them directly. Of course, this comes at the cost of some flexibility: both Sherlock and Orion can build their dependency graphs from packet traces. Their reliance on packet traces also provides them with less information about what the application is actually trying to do: it can be difficult to tell whether an application is blocked or stalled just from a packet trace, since in both cases no traffic is available.

Aguilera et al. [3] describe an approach for diagnosing high-latency paths in distributed systems of black boxes which communicate via request-response RPC or generic message passing. Nodes in their model may be hosts, multi-process applications, processes, or particular subsystems. By calculating send/receive timestamps on every message, they infer the dataflow graph of messages through the system, and the causal relationships between messages [34]. They then ascribe messaging delays to nodes in the system. While their approach requires no direct interaction with the monitored system (like that used in StreamsDiagnoser to provide the processing graph and counters), the messaging graph is inherently less reliable since it

is inferred rather than determined directly. Tracing each message is also expensive to collect and post-process.

A visual debugger for analyzing and debugging Streams applications is described by de Pauw et al. [18]. This tool relies on tracing individual tuples through the Streams processing graph. To limit overhead, this tracing is activated for limited periods of time on particular subpaths through the processing graph. Their visualization uses per-tuple timelines to show how long each tuple took to move between operators in the processing graph. This timeline visualization of the flow between operators and the ability to examine the fields of individual tuples makes this technique effective for finding subtle timing and correctness bugs. For example, it can help a developer to identify a join operator that is not provided with the right join criteria, or the wrong incoming data. However, it is too expensive to run continually and provides no automated diagnosis, so the systems are largely complementary.

6.3 System-wide Instrumentation

CONMan [8, 9] is similar in spirit to NeST, making use of a generic network module abstraction and tracking dependencies between modules. The main goal of CONMan is to simplify configuration and management of computer networks by using this generic module abstraction, but it also enables diagnosis of some network *faults* in which connectivity between devices is severed. CONMan uses a centralized *Configuration Manager* to track modules throughout the network, and

tracks counters across all the modules in the network to determine which modules have stopped forwarding completely [9].

NeST is able to reliably diagnose temporary network performance *degradation* from a single end host, a much more difficult goal. Specifically, NeST diagnoses transient (as opposed to permanent) performance problems without instrumenting the entire network. In contrast to the centralized, cooperative scheme described in CONMan, NeST is host-based and operates in a completely autonomous and decentralized manner: any end system can implement and use NeST to diagnose network performance problems independently. If the modules of the operating system (or distributed system) implemented the CONMan abstraction, a combination of the CONMan and FlowDiagnoser diagnosis rules could potentially provide more precise and accurate results, without CONMan's requirement to instrument the entire network.

NetMedic [32] models processes, configurations, devices, and machines in its dependency graph and uses the past history of their interactions to determine which component is the cause of misbehavior. NetMedic goes farther than FlowDiagnoser in its use of counters and metrics, applying CPU, disk IO, and application-specific counters to the diagnosis. Like CONMan and Sherlock, NetMedic requires centralized analysis of all the analyzed counters, but treats the counters as a black box; no semantics are required. NetMedic diagnoses problems by tracing abnormal counter states in one node to abnormal states in other nodes, using the weighted dependencies determined from the provided history of counters.

The Network-wide Information Correlation and Exploration (NICE) system [36] uses data collected from throughout a service provider network to find correlations between performance problems and potential root causes. Symptoms of performance problems include router CPU spikes and measured end-to-end packet loss, which may be attributed to invocation of particular router commands or link failure events. NICE transforms router syslogs, CPU and packet loss readings, active measurements, and other data sources into event vectors. Symptoms and potential causes that are strongly correlated over time are presented as a ranked list of possible causes (e.g., end-to-end packet loss was caused by a router CPU spike and a router configuration command). Such a system is useful for detecting and debugging chronic, long-term issues and the causes that have a strong statistical co-occurrence, but does not provide information about what is happening at a particular point in time as NeST does.

In all of these systems, the diagnosis process depends upon system-wide data for useful diagnosis, whereas NeST can diagnose performance problems from the point of view of a single host. With StreamsDiagnoser, the need for system-wide data is limited: it requires only a few counters from each communicating process. Of course, the StreamsDiagnoser output is limited to identifying which PE (operator) is not behaving correctly, and is unable to assign faults to underlying network elements (which are not modeled).

6.4 Summary

In comparison with other performance diagnosis systems that perform sophisticated analysis on extensive amounts of data, FlowDiagnoser occupies one end of the research spectrum: it uses a very small amount of information to identify modules that are the cause of messaging-related performance stalls. These diagnosis results could be used to triage which modules need further investigation, and perform more in-depth protocol- or system-specific analysis. FlowDiagnoser uses two signals—one to indicate node activity, and another to indicate waiting—to automatically diagnose performance problems in an approach that is accurate, efficient, and general.

Chapter 7

Conclusion

This dissertation presents the FlowDiagnoser approach to automatically diagnosing performance stalls in networked and distributed systems. Motivated by real-world experience, academic research, and examples from industry, these performance stalls have a significant affect on system performance, and affect users' attitudes about the usability of a system. The FlowDiagnoser approach is applied in two different problem domains: NeST, which finds the source of network-related stalls in a host's network stack, and StreamsDiagnoser which diagnoses processing stalls in the InfoSphere Streams stream processing system. In comparison with other approaches, FlowDiagnoser is simple and applies across problem domains. This chapter concludes the dissertation with directions for future work.

7.1 Bottleneck detection

In discussing the StreamsDiagnoser approach with Streams researchers and users, one repeated request is to find the source of backpressure-related *bottlenecks*, in which a module is still processing messages, but limiting its parents' output rate. FlowDiagnoser can probably be extended to detect slow modules in both Streams and the network stack by looking at module's `queued_msgs` or `wait_time` counter when a module is active (i.e., not only when `total_msgs` = 0). This may indicate

that child module is not processing as quickly as it should be. However, to do this reliably, the current binary diagnosis criteria (> 0 or $= 0$) may need to be relaxed, since relative processing rates often fluctuate during normal operation. The most promising approaches may be to require some minimum threshold number for the number of messages in the queue before FlowDiagnoser signals that a module is a bottleneck (since a few messages may always be in the buffer), or require a number of consecutive snapshots with a stable or growing queue before it marks the module as a bottleneck.

7.2 Module-specific diagnosis

One option that has garnered interest among Streams researchers is the possibility of adding introspection to the Streams runtime, and providing StreamsDiagnoser with more fine-grained information about what a module *should* be doing. For example, while a filter operator should not be expected to pass every message, a pass-through or message-modifying operator is misbehaving if it does not submit a message for every one processed. In addition, StreamsDiagnoser could treat a Barrier specially, to account for the issues described in Chapter 4.

This same approach would work for modules in the network stack; instead of treating each module generically, NEST could allow each module to specify its own implementation-specific diagnosis. For example, a specially instrumented application could specify which sockets should be *expected* to read or write at any given time; NEST would then verify whether its actual behavior meets expecta-

tion. NeST could also incorporate TCP-specific or interface-specific diagnosis criteria [15,37,39,60]. The general-purpose dependency analysis described in Chapter 3 would still be used for modules that have no implementation-specific criteria, or to handle conflicting results.

7.3 Online diagnosis

Streams researchers and developers have expressed interest in using StreamsDiagnoser online to detect performance stalls and kick off a reconfiguration of the Streams processing graph, either to share load or restart stalled PEs. NeST could also monitor an end user's system in real time, and either provide feedback and recommendations, or automatically take action to improve the user's experience.

7.4 Additional systems

Finally, FlowDiagnoser is designed to work at many levels of abstraction. While this dissertation evaluates process-to-process communication in StreamsDiagnoser, it should work equally well (but with higher processing overhead) when looking at each operator inside a Streams PE. It may be possible to apply the same diagnosis approach and implementation to tracking performance flows between Streams jobs (which are themselves comprised of many PEs), and even to track messaging problems in systems-of-systems processing graphs. However, this will likely require additional data processing and improvement, since even the loose

counter semantics employed by FlowDiagnoser may be too stringent when collecting data across many systems.

7.5 Conclusion

The thesis of this dissertation is that the source of performance stalls in a distributed system can be automatically diagnosed with very limited information: the dependency graph of data flows through the system, and a few counters common to almost all data processing systems.

The FlowDiagnoser automated fault detection system requires as little as two bits of information per module: one to indicate whether the module is actively processing data, and one to indicate whether the module is waiting on its dependents.

Prototype implementations and controlled experiments in two distinct environments—an individual host's networking stack, and a distributed streams processing system—support this thesis, and show that the automatic FlowDiagnoser approach is general, efficient, and accurate.

Appendix A

Normalization Procedure for Streams Counters

As discussed in Section 4.3.1 and Section 4.3, in InfoSphere Streams, counters are assigned to PEs' and operators' input and output ports. These per-port counters track the *total* number of tuples submitted or processed across all the connections from/to that port. This is similar to accounting for a busy network server's traffic by aggregating on the `IP:port` the server is bound to. The InfoSphere Streams runtime does not track the number of tuples submitted or processed on each stream connection.

Since the current model transformation is based on individual stream connections (`F : out0` $\rightarrow$ `I : in2`), StreamsDiagnoser expects counters that track the number of tuples submitted into the connection, and the number of tuples processed out of the connection. Therefore, it needs to normalize the counters from each input and output port.

A.1 Connection-counter normalization cases

Using the PEs and connections from Figure 4.2 as examples, there are four cases of concern, listed in Table A.1.

Case	Summary	Normalization Step
$1 \to 1$	One input port, one output port	None needed.
$1 \to N$	Multiple downstream subscribers	$\sigma^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}^{\texttt{NORM}}} += \frac{\Delta^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}}}{\text{nConns}_{\texttt{PE:OUT:i}}}$
$N \to 1$	Multiple upstream publishers	$\sigma^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}^{\texttt{NORM}}} += \sum_{PE_x:OUT:p \in \text{ upstream ports}} \Delta^{\texttt{PE}_x\texttt{:OUT:p}}_{\texttt{nSubmitted}^{\texttt{NORM}}}$
$M \to N$	Both directions multiple-subscription	Apply the normalized submitted values $\Delta^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}^{\texttt{NORM}}}$ before summing across upstream PEs.

Table A.1

A.1.1 [Case 1] $1 \to 1$ connections

This is the trivial case, and the counter needs no normalization.

A.1.2 [Case 2] $1 \to N$ fan-out connections

Whenever an output port's tuples are subscribed to by multiple downstream consumers, there is a $1 \to N$ fan-out, as seen at $\texttt{F} : \texttt{out0} \to \{\texttt{I} : \texttt{in2}, \texttt{J} : \texttt{in0}, \texttt{K} : \texttt{in0}\}$ in Figure 4.2.

For every call to `F : out0.submit()`, the $\sigma^{\texttt{F:out0}}_{\texttt{nSubmitted}}$ counter increases by the number of open connections (three), but as each downstream processes the submitted tuple, the downstream's `nProcessed` counter (e.g. $\sigma^{\texttt{I:in2}}_{\texttt{nProcessed}}$) increases by one.

Without normalization, this can lead to False Positives, where StreamsDiagnoser blames `I : in2` even though it has processed all the tuples submitted to it, since `nSubmitted` increases at three times the rate of `nProcessed`.

The normalization step at each snapshot is to divide the *increase* in the output port's `nSubmitted` counter by $\texttt{nConns}_{\texttt{PE:OUT:i}}$, the number of connections currently attached to output port i. This results in the normalized total value

$$\sigma^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}^{\texttt{NORM}}} \mathrel{+}= \frac{\Delta^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}}}{\texttt{nConns}_{\texttt{PE:OUT:i}}}$$

which is the number of tuples submitted to each connection on the port.

A.1.3 [Case 3] $N \rightarrow 1$ fan-in connections

This is the inverse of Case 2, where an input port subscribes to streams that originate from multiple output ports. PE `N`'s input port 0 is an example of this: $\{\texttt{H : out0}, \texttt{I : out0}, \texttt{J : out0}\} \rightarrow \texttt{N : in0}$ is an $N \rightarrow 1$ fan-in.

The value of $\sigma^{\texttt{N:in0}}_{\texttt{nProcessed}}$ accounts for *all* of the tuples processed on that input port, regardless of the sender. That is, `N : in0`'s `nProcessed` counter is the sum of all the tuples processed from `H : out0`, `I : out0`, and `J : out0`. It would be convenient to normalize the `nProcessed` counter in the same way that `nSubmitted` is normalized, but this cannot be done reliably. It is impossible to determine how many tuples have been processed from each individual incoming stream; StreamsDiagnoser knows only how many were processed in total.

Without normalization, it will usually appear that the downstream input port has processed all of the tuples from each connection. This may cause False Negatives

on `N : in0`, since the total number of tuples submitted on *one* incoming connection will almost always be less than the total number of tuples processed from *all* the incoming connections.

To account for the $N \rightarrow 1$ normalization, StreamsDiagnoser keeps a shadow counter `nSubmittedToInputPort`. After individually normalizing all of the output port counters as described in Case 2, StreamsDiagnoser sums $\Delta^{\mathtt{PE_x:OUT:p}}_{\mathtt{nSubmitted}^{\mathtt{NORM}}}$ across all of the output ports connected to this input port. This sum is the total number of tuples submitted to the input port in the last snapshot period. StreamsDiagnoser then increases `nSubmittedToInputPort` by this sum.

A.1.4 [Case 4] $M \rightarrow N$ multi-way connections

If an output port with multiple outgoing connections is connected to an input port that has multiple incoming connections, Once StreamsDiagnoser has correctly normalized the counters on the output ports (Case 2), it can apply the summation procedure from Case 3, and the counters work as expected.

A.2 Counter normalization algorithm

The algorithm to recover these per-connection counters is:

1. Keep track of the original counter values from the first snapshot, and use this as a baseline. This is necessary since StreamsDiagnoser does not know the number of connections that arrived and departed before it obtained the

first snapshot. Subtract out this initial baseline value from each subsequent snapshot.

2. Re-baseline at every topology change where the number of connections changed, since StreamsDiagnoser does not know the number of tuples were submitted on the first N connections, and how many were submitted on all $N \pm 1$ connections during the snapshot period.

3. At each snapshot, divide any increase by the current number of connections for each port.

In practice, StreamsDiagnoser creates two shadow counters:

- `nTuplesSubmittedToConnectionX`, which is the value of $\sigma^{\texttt{PE:OUT:i}}_{\texttt{nSubmitted}^{\texttt{NORM}}}$ from Case 2.

- `nSubmittedToInputPort`, which counts messages submitted into an input port by *all* of its incoming connections in Case 3.

The information required for this normalization includes:

1. The first-count baseline (to factor out changes StreamsDiagnoser never saw),

2. The last-connection-change baseline (to ignore earlier changes in `nConnections`),

3. The current number of connections (to assign the current increase in the counter to each connection), and

4. The total number of tuples submitted on an output port, and the total number of tuples processed on an input port, as provided by the Streams runtime.

Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] Bhavish Aggarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*. USENIX, Apr 2009.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, New York, NY, USA, 2003. ACM.

[4] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 882–884, New York, NY, USA, 2005. ACM.

[5] Dave Artz. The secret weapons of the AOL optimization team. `http://velocityconf.com/velocity2009/public/schedule/detail/7579`, 2009.

[6] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, pages 13–24, New York, NY, USA, 2007. ACM.

[7] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eduardo F. Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stanley B. Zdonik. Retrospective on aurora. *VLDB J.*, 13(4):370–383, 2004.

[8] Hitesh Ballani and Paul Francis. CONMan: a step towards network manageability. In *SIGCOMM*, pages 205–216, New York, NY, USA, 2007. ACM.

[9] Hitesh Ballani and Paul Francis. Fault management using the CONMan abstraction. In *INFOCOM*, 2009.

[10] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. *IEEE/ACM Transactions on Networking*, 9:238–248, June 2001.

[11] Brandes, M Eiglsperger, I Herman, M Himsolt, and MS Marshall. GraphML progress report: Structural layer proposal. In P Mutzel, M Junger, and S Leipert, editors, *Graph Drawing - 9th International Symposium, GD 2001 Vienna Austria,*, pages 501–512, Heidelberg, 2001. Springer Verlag.

[12] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[13] Ranveer Chandra, Venkata N. Padmanabhan, and Ming Zhang. WiFiProfiler: cooperative diagnosis in wireless LANs. In *MobiSys*, pages 205–219, New York, NY, USA, 2006. ACM.

[14] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, pages 117–130. USENIX Association, 2008.

[15] Yu-Chung Cheng, Mikhail Afanasyev, Patrick Verkaik, Péter Benkö, Jennifer Chiang, Alex C. Snoeren, Stefan Savage, and Geoffrey M. Voelker. Automating cross-layer diagnosis of enterprise wireless networks. In *SIGCOMM*, pages 25–36, 2007.

[16] Jonathan Corbet. Replacing ptrace(). `http://lwn.net/Articles/371501`, Jan 2010.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009.

[18] Wim De Pauw, Mihai Leţia, Buğra Gedik, Henrique Andrade, Andy Frenkiel, Michael Pfeifer, and Daby Sow. Visual debugging for stream processing applications. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 18–35, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[20] Philip Dixon. Shopzilla's site redo: You get what you measure. `http://velocityconf.com/velocity2009/public/schedule/detail/7709`, 2009.

[21] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for TCP. In *IMC*, IMC '11, pages 155–170, New York, NY, USA, 2011. ACM.

[22] Firebug. `http://getfirebug.com`.

[23] Dennis F. Galletta, Raymond M. Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *J. AIS*, 5(1):0–, 2004.

[24] Buğra Gedik and Henrique Andrade. A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM infosphere streams. *Softw: Pract. Exper.*, 2011.

[25] Buğra Gedik, Henrique Andrade, Andy Frenkiel, Wim De Pauw, Michael Pfeifer, Paul Allen, Norman Cohen, and Kun-Lung Wu. Tools and strategies for debugging distributed stream processing applications. *Softw. Pract. Exper.*, 39(16):1347–1376, Nov. 2009.

[26] Dan Gunter, Brian Tierney, Brian Crowley, Mason Holding, and Jason Lee. Netlogger: A toolkit for distributed system performance analysis. In *IEEE MASCOTS*, page 267, Washington, DC, USA, 2000. IEEE Computer Society.

[27] John Heffner. CurAppWQueue non-zero at end of connection. Post to web100-discussion mailing list, March 2011.

[28] John Heffner. Rcvbuf/sndbuf vs. queue sizes. Post to web100-discussion mailing list, February 2011.

[29] Tom Herbert. [patch] tcp: Fix slowness in read /proc/net/tcp. Post to linux-netdev mailing list, June 2010.

[30] IBM streams processing language (SPL) online documentation. `http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp`, March 2012.

[31] Steve Jobs. Thoughts on Flash. `http://www.apple.com/hotnews/thoughts-on-flash`, April 2010.

[32] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM*, pages 243–254, New York, NY, USA, 2009. ACM.

[33] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM.

[34] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[35] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. WebProphet: automating performance prediction for web services. In *NSDI*, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[36] Ajay Mahimkar, Jennifer Yates, Yin Zhang, Aman Shaikh, Jia Wang, Zihui Ge, and Cheng Tien Ee. Troubleshooting chronic conditions in large IP networks. In *CoNEXT*, CoNEXT '08, pages 2:1–2:12, New York, NY, USA, 2008. ACM.

[37] Matt Mathis, John Heffner, Peter O'Neil, and Pete Siemsen. Pathdiag: Automated TCP diagnosis. In *PAM*, Apr 2008.

[38] Matt Mathis, John Heffner, and Rajiv Raghunarayan. TCP Extended Statistics MIB. RFC 4898, Internet Engineering Task Force, May 2007.

[39] Matt Mathis, John Heffner, and Raghu Reddy. Web100: Extended TCP instrumentation for research, education and diagnosis. In *SIGCOMM*, pages 69–79, New York, NY, USA, 2003. ACM Press.

[40] Marissa Mayer. In search of... a better, faster, stronger web. `http://velocityconf.com/velocity2009/public/schedule/detail/8913`, 2009.

[41] Pablo Neira Ayuso, Rafael M. Gasca, Laurent Lefèvre. Communicating between the kernel and user-space in Linux using Netlink sockets. *Software: Practice and Experience*, 40(9), Aug. 2010.

[42] ptrace - process trace. `http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html`.

[43] Roshan Punnoose, Andrew Skene, Octavian Udrea, and Brian Williams. personal correspondence, 2011.

[44] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 219–232, New York, NY, USA, 2009. ACM.

[45] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, pages 115–128. USENIX, 2006.

[46] Yaoping Ruan and Vivek Pai. Understanding and addressing blocking-induced network server latency. In *USENIX ATC*, ATEC '06, pages 14–14, Berkeley, CA, USA, 2006. USENIX Association.

[47] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. `http://velocityconf.com/velocity2009/public/schedule/detail/8523`, 2009.

[48] M. Siekkinen, G. Urvoy-Keller, E. W. Biersack, and T. En-Najjary. Root cause analysis for long-lived TCP connections. In *CoNEXT*, pages 200–210, New York, NY, USA, 2005. ACM Press.

[49] Steve Souders. High-performance web sites. *Communications of the ACM*, 51(12):36–41, Dec. 2008.

[50] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[51] Dominique Toupin. Using tracing to diagnose or monitor systems. *IEEE Softw.*, 28(1):87–91, Jan. 2011.

[52] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *USITS*, USITS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[53] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, pages 255–270, Boston, MA, Dec. 2002. USENIX ATC.

[54] Krist Wongsuphasawat, Pornpat Artornsombudh, Bao Nguyen, and Justin McCann. Network stack diagnosis and visualization tool. In *Proceedings of the Symposium on Computer Human Interaction for the Management of Information Technology*, CHiMiT '09, pages 4:29–4:37, New York, NY, USA, 2009. ACM.

[55] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. *Foundations of Intrusion Tolerant Systems*, 0:213, 2003.

[56] Chrome Developer Tools: Network Panel. `https://developers.google.com/chrome-developer-tools/docs/network`.

[57] YSlow. `http://yslow.org`.

[58] Best practices for speeding up your web site. `http://developer.yahoo.com/performance/rules.html`.

[59] YSlow ruleset matrix. `https://github.com/marcelduran/yslow/wiki/Ruleset-Matrix`.

[60] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, NSDI'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[61] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *SIGCOMM*, pages 309–322, New York, NY, USA, 2002. ACM Press.