ABSTRACT

Title of dissertation: AUTOMATING PERFORMANCE DIAGNOSIS IN NETWORKED SYSTEMS

Justin Neil 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.

Our 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.

We prove this thesis by implementing the idea and demonstrating its effectiveness in two distinct environments: an individual host's networking stack, and a distributed streams processing system.

Using real applications, we show that our approach correctly diagnoses 99% of networking-related stalls due to application, connection-specific, or network-wide performance problems, with a false positive rate under 3%. Our 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 Neil 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 Neil Spring
Professor James Reggia
Professor Peter Keleher
Professor Mark Shayman

Dedication

To be completed.

Acknowledgments

To be completed.

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 in which 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 it "has not performed well on mobile devices" [27]. Processing overhead and resource scheduling in monolithic web browsers caused Google researchers to redesign according to a multi-process architecture, leading to significant speedups in page load times [38].

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 [40], contending for shared resources [42], disk writes or database queries [16]. 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 [18] 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 [18].

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

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 [35].

Locating and correcting the causes of performance stalls and excessive latency can have a significant positive impact on revenue: When Shopzilla completely redesigned 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 [17].

Earlier studies have found that user's intent to keep using a website and performance in completing tasks started to decline after two seconds of delay [19], 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 [41]. 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 [22, 39] and track individual messages as they are sent throughout the system. While these can help developers and operators to track down subtle bugs and performance 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, 13, 14] 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, 13, 14, 28, 45] 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, 14, 28, 31, 45] 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 prove 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.

Ideally, this research 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 distinct applications of the 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

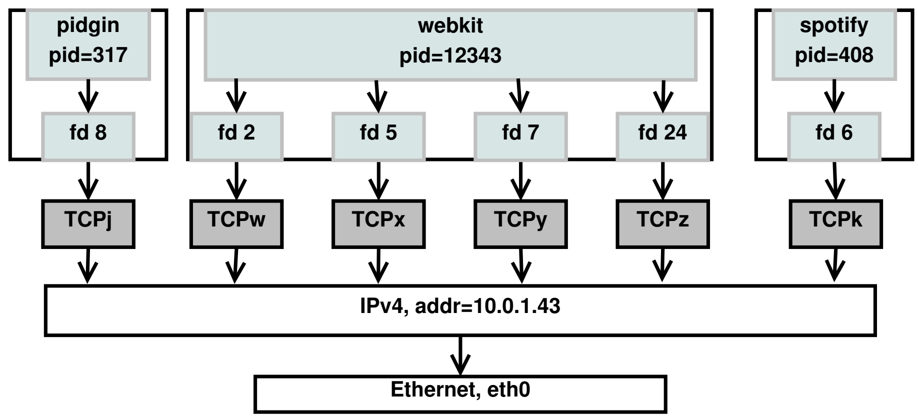


Figure 1.1: Example network stack dependency graph. Application and socket modules are shown in light grey, TCP instances in dark grey, and IPv4 and Ethernet in white. Application modules are made up of component socket modules.

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-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 [20]. 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.

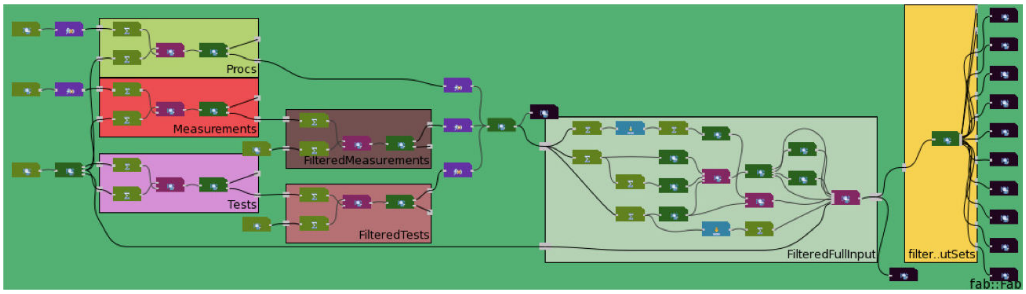


Figure 1.2: Example Streams application, as shown in Gedik and Andrade [20]. This is a logical view of the flow of messages between different pieces of code. A process-to-process view is also available.

Despite extensive tools for debugging [21] and visualizing performance information [15], 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

Chapter 2

Overview

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.

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).

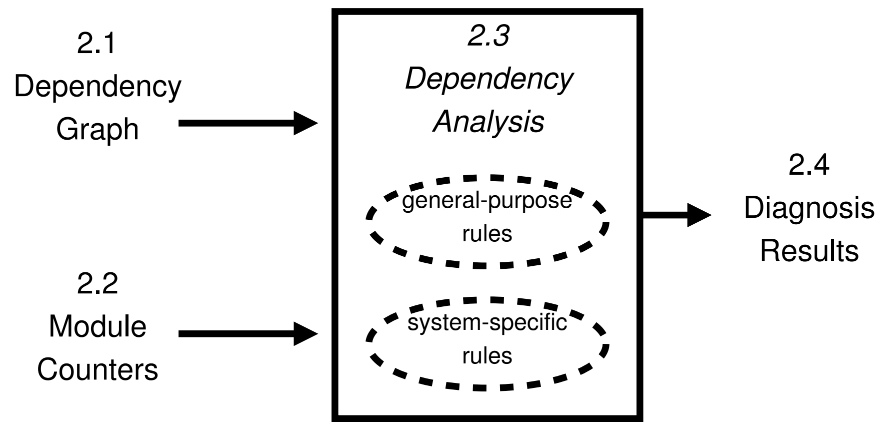


Figure 2.1: The diagnosis process. Module counters are used to describe each module's activity, and combined with the dependency graph to determine the effect of each module's behavior. The dependency analysis consists of general-purpose and system-specific rules. The resulting performance diagnosis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

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 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. For push-oriented systems, dependency corresponds to dataflow. Since dataflow originates at sources, if A sends to B, then A depends on

[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.

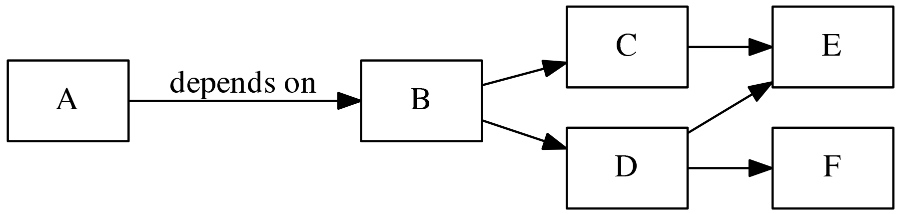


Figure 2.2: Example dependency graph. A depends on B for messaging service, which in turn depends on C and D.

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 module A, 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 sources. A module's *descendants* are its children, and recursively all of its childrens' children. Intuitively, a module M's ancestors include all nodes along paths from the sources 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

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.

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 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 was actively processing messages, and

2. whether a module *attempted* to process messages (or had 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 value of `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`. For the best diagnosis results, FlowDiagnoser requires the sources in a push-oriented system, or the sinks in a pull-oriented system, to implement at least one of the other two counters. This is the signal of whether the modules have work to do or are waiting on their dependents.

2.2.1 Snapshots

To be effective, FlowDiagnoser must take snapshots relatively frequently (from 50 ms to 10 seconds). 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 shorter than the longest stall the system designer is willing to tolerate. It is preferable for the interval to be at most half of the length of the maximum acceptable stall, to avoid snapshot-boundary conditions and provide two consecutive intervals for confirmation. For example, if a 200 ms stall is considered unacceptable, snapshots should be taken at least every 200 ms; a 100 ms interval is preferable. 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 frequency, the data collection 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 races conditions while reading a single module's counters. Chapter 3 and 4 discuss the implications of such anomalies.

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 a *diagnosis result*. Table 2.2 lists the diagnosis assigned to each module in the stack after every 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.

There are four possible diagnosis results:

- Healthy, which indicates that the module is actively processing messages;
- Stalled, which indicates that the module is faulty and is the cause of system performance problems;
- Blocked, which indicates that the module attempted to make progress, but was prevented from doing so by another faulty module; and

[2] The module in question does not support the `wait_time` or `queued_msgs` counters.

Module Diagnosis	Diagnosis Criteria
HEALTHY Module is active and processing messages	[($\Delta_{\texttt{total\_msgs}} > 0$)]$^{(a)}$
STALLED Module is not processing messages and is blocking others	[($\Delta_{\texttt{total\_msgs}} = 0$) *and* ($\Delta_{\texttt{wait\_time}} = 0$)]$^{(b)}$ *or* [($\Delta_{\texttt{total\_msgs}} = 0$)$^{(c)}$ *and* [($\texttt{queued\_msgs} > 0$) *or* (module is simple)^2]$^{(d)}$ *and* (no child is BLOCKED or STALLED)$^{(e)}$]
BLOCKED Module cannot process messages due to another's fault	[($\Delta_{\texttt{total\_msgs}} = 0$) *and* ($\Delta_{\texttt{wait\_time}} > 0$)]$^{(f)}$ *or* [($\Delta_{\texttt{total\_msgs}} = 0$)$^{(g)}$ *and* [($\texttt{queued\_msgs} > 0$) *or* (module is simple^2)]$^{(h)}$ *and* (a child is BLOCKED or STALLED)$^{(i)}$]
DONTCARE Module has completed its work, or ancestors are fine	[($\Delta_{\texttt{total\_msgs}} = 0$) *and* ($\texttt{queued\_msgs} = 0$)]$^{(j)}$ *or* [($\Delta_{\texttt{total\_msgs}} = 0$)$^{(k)}$ *and* (module is simple^2)$^{(\ell)}$ *and* (no parent is BLOCKED)$^{(m)}$]

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

- 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.

The following subsections discuss the criteria used to arrive at these diagnoses.

2.3.1 Active modules

FlowDiagnoser assumes that a module that has processed any messages ($\Delta_{\texttt{total\_msgs}} > 0$) is providing adequate service to its dependents, and labels it as HEALTHY (criterion *(a)*)—it is not the source of a stall.[3]

[3] 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)

In all other cases the module is *inactive* ($\Delta_{\texttt{total\_msgs}} = 0$), and the analysis must consider additional performance counters and the module's location in the dependency graph to determine if it is faulty.

2.3.2 Using the `wait_time` counter

A module may be inactive because it is waiting for another module to act—to produce a message for it, or to consume a message it has provided. As detailed in Chapter 3, some applications of FlowDiagnoser can directly track an application's read and write calls and determine whether it is attempting to send and receive messages. 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) in the call.

When the module supports the `wait_time` counter, the diagnosis process is simple. If the module has processed no messages ($\Delta_{\texttt{total\_msgs}} = 0$) and spent no time waiting ($\Delta_{\texttt{wait\_time}} = 0$), the module has not processed any messages because it is idle; FlowDiagnoser therefore marks it as STALLED (criterion *(b)*). On the other hand, if an inactive module did attempt to process messages but was blocked ($\Delta_{\texttt{wait\_time}} > 0$), it marks it as BLOCKED (criterion *(f)*).[4] Therefore, a module which supports the `wait_time` counter is always marked as HEALTHY, STALLED, or BLOCKED.

2.3.3 Using the `queued_msgs` counter

For some modules, FlowDiagnoser may be able to determine the size of the module's work queue (`queued_msgs`), i.e., the total number of messages that are available for the module to process. An inactive module that has messages in its work

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.

[4]There is another case, in which $\Delta_{\texttt{total\_msgs}} > 0$ and $\Delta_{\texttt{wait\_time}} > 0$. 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.

queue (`queued_msgs` > 0) is quite possibly blocking its parent. In this situation, the analysis must determine if the source of the blockage: either the module itself is stalled (meeting criteria *(d)* and *(e)*) or it is blocked on one of its children (meeting criteria *(h)* and *(i)*), as explained in Section 2.3.5.

An inactive module with an empty work queue (`queued_msgs` $= 0$) is inactive for a good reason: it had no work to do. Therefore, FlowDiagnoser marks the module as DontCare (criterion *(j)*) to indicate that its inactivity did not have an adverse affect on any of its parents.

2.3.4 Simple modules

Modules lacking both a `queued_msgs` and `wait_time` counter are called *simple modules*; these are considered explicitly in criteria *(d)*, *(h)*, and *(ℓ)* of Table 2.2. Such modules provide no clear signal as to whether they had any work to do (`queued_msgs`) or tried to do it (`wait_time`), so FlowDiagnoser must rely on their dependents to provide clues.

The analysis first considers the module's parents. As mentioned in Section 2.2, FlowDiagnoser requires that at least some of the modules in the system provide a waiting indication—this indicates whether the inactivity was expected (no module is waiting) or a problem (some modules are waiting, i.e. Blocked). If none of the module's parents are waiting for it to provide service, then the module was inactive because it had nothing to do, and FlowDiagnoser marks it as DontCare (criterion *(m)*).

On the other hand, if some parent is Blocked, the analysis must determine if the module is blocking its parent (criteria *(d)* and *(e)*), or the module is itself blocked by one of its children (criteria *(h)* and *(i)*).

2.3.5 Pass the blame

Whenever an inactive module M's ancestors have indicated that they want service from it (i.e., M's parents are BLOCKED), there are two possibilities left: either M itself is STALLED, or it is BLOCKED by its own descendants.

Since M depends on its children for service, if one of M's children is BLOCKED or STALLED, the analysis can assume that the child is the cause of M's performance problem. Therefore, FlowDiagnoser marks M as BLOCKED (criteria *(g)*, *(h)*, and *(i)*). If none of M's children are BLOCKED or STALLED, then M itself is responsible and FlowDiagnoser marks it as STALLED (*(c)*, *(d)*, and *(e)*).

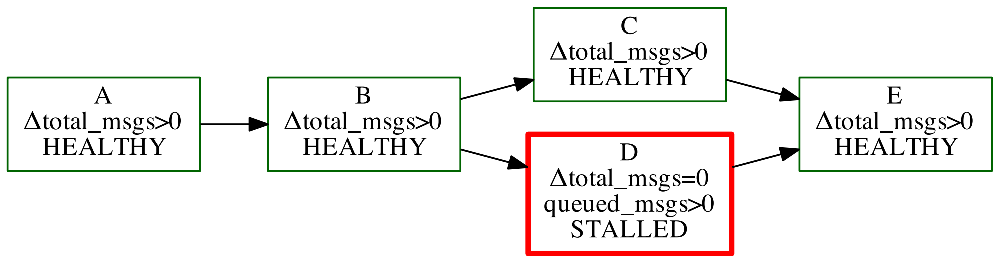
While the simple pass-the-blame rule described here works in many cases, a monitored system may require a different set of criteria based on its communication semantics. Section 3.3.2 describes a slightly modified, system-specific rule employed when monitoring the network stack.

2.3.6 Correlating evidence

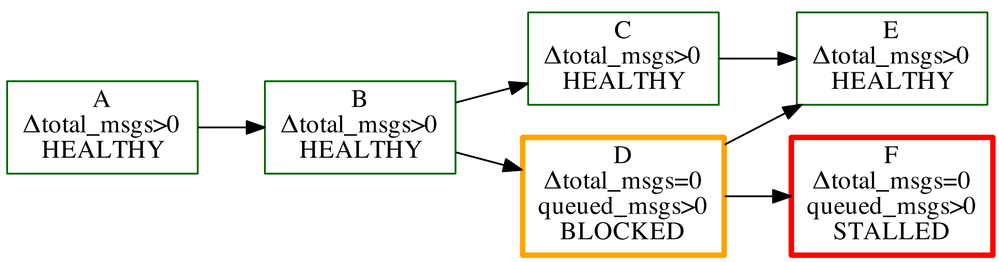
One beneficial feature of the FlowDiagnoser algorithm is that when determining whether an inactive module M is BLOCKED or STALLED, it takes into account correlations among multiple flows. To see this, consider the example dependency graphs in Figure 2.3. In both subfigures, A is able to pass messages 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:

$\texttt{queued\_msgs}^{D} > 0$. 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 timeseries visualization that shows the diagnosis provided to each module over time, as explained in Section 3.5.4.
- A timeseries 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; as such we call the dependency graph the *network stack dependency graph* (NS graph). We describe how we acquire and maintain the graph in Section 3.1.

NeST module counters are obtained 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 when distinguishing between network-wide and connection-specific performance stalls.

Our prototype data collection and diagnosis engine is described in Section 3.4, and experimental evaluation and results in Section 3.5. In a series of controlled experiments using real applications, we 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 positive rate under 3%.

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

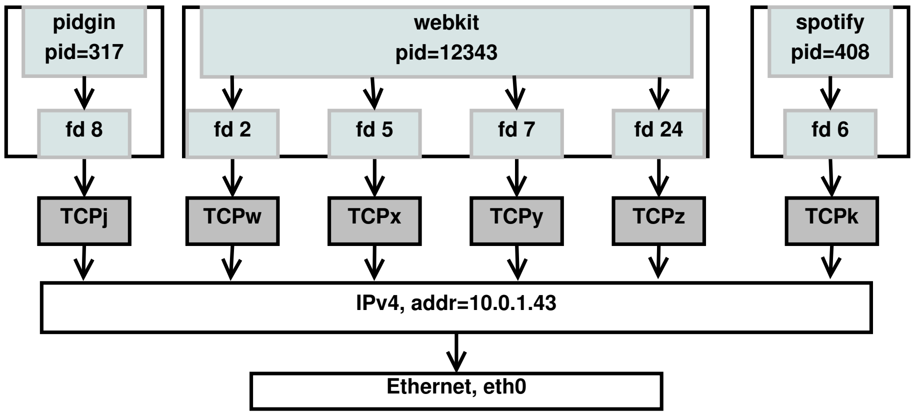
Each element of the host's network stack appears as a module in a *network stack dependency graph* (NS graph). In particular, modules consist of (1) running applications (one module per application); (2) each of an application's sockets (one module per socket); (3) the TCP state for each socket; (4)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

An example NS graph is given in Figure 3.1(a). 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 we would like NeST to diagnose the source of the problem as being localized to one of the above modules, thus blaming the application, TCP's behavior, IP-level connectivity, or the physical network device.

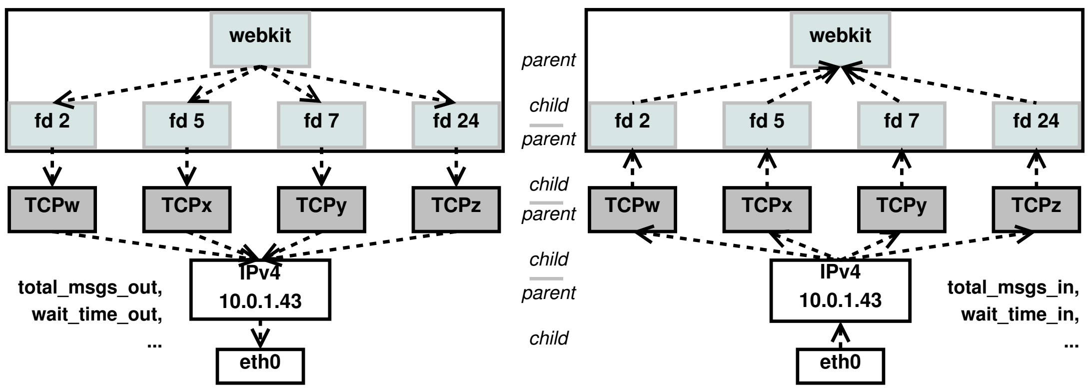
In NeST, applications drive all traffic on the host: dataflow is push-oriented for outbound traffic and pull-oriented for inbound traffic. Therefore, we can use the same NS graph for both kinds of flows.

For our example, the dataflow graph for outbound (push-oriented) flows sent from the local host is shown in Figure 3.1(b). 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), it is assumed 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 we detail in Section 3.4, we instrument an end host's network stack to track applications, sockets, connections, and interfaces and their dependencies as they



(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.

come and go. As applications and connections open and close, we track the changes to the graph and update it accordingly. At configurable intervals, we snapshot the entire host's network counters (described next), and use 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 inbound and outbound messages processed, called `total_msgs_out` (abbreviated `tmo` in the figures)

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.

and total_msgs_in (tmi), respectively. For the TCP, IP, and physical device modules, we populate these counters using operating system-provided information. For applications and sockets, we acquire the message counters using instrumentation.

Since application socket modules are the sources in the NS graph, for them we also keep 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. Once again we populate these counters by instrumenting the application, described below.

For NeST we do not track the queuing behavior of modules, so the queued_msgs counter from Table 2.1 is not used. In earlier implementations, we attempted to accurately track the number of bytes waiting in a TCP connection's queues, but found this to be unreliable [23, 24] or too slow in the versions of the operating system we used [25].

3.2.1 Application and socket modules

We implement each application counter as the sum of the counters of its respective sockets.

To track per-socket message counts, we increment 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`). We do this by intercepting calls using an LD_PRELOAD interceptor library, discussed in detail in Section 3.4.

We implement `wait_time_out` and `wait_time_in` for sockets in the same interceptor library: we accumulate the amount of time (in milliseconds) spent blocking in the underlying call. To support non-blocking calls, we also track the time spent in `poll()`, `select()`, and their equivalents. Note that we do 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. We make one customization to this algorithm, described shortly. In the meantime, we describe 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—but it does mean that the underyling 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 we account for application counters. If the application is not Healthy, then if any of its sockets are Blocked we say 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, we know 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 (out-

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

bound) 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 are no applications running), these lower-layer modules are marked as DONTCARE. Possible improvements are discussed in Section 3.6.

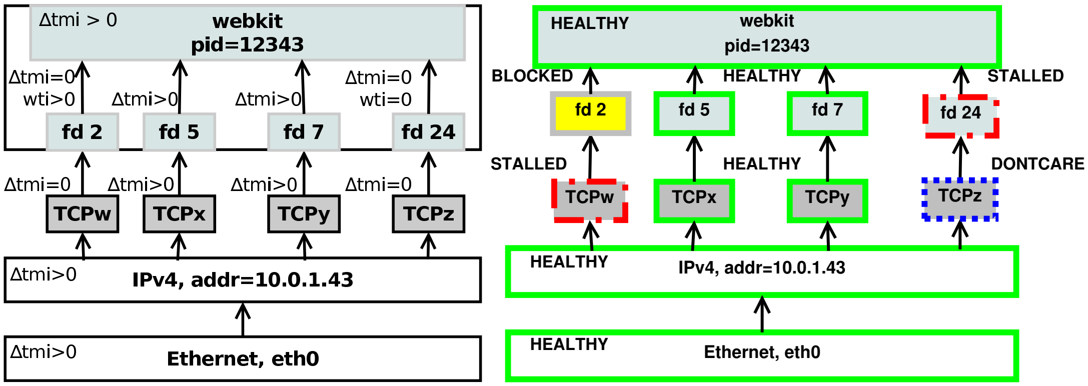
3.3.2 Distinguishing connection-specific and network-level faults

The dependency analysis we use in NEST 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 we can directly diagnose application and socket modules since they export the `wait_time` counter. We therefore need 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. We now consider how multiple flows aid in diagnosis, and the rule variant we use to distinguish between connection-specific and network-wide performance stalls.

3.3.2.1 Multiple active flows

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.



(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. We abbreviate `total_msgs_in` 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.

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 `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` ($\Delta_{\texttt{tmi}}^{\texttt{fd 2}} = 0$ and $\Delta_{\texttt{wti}}^{\texttt{fd 2}} > 0$), and `TCPw` is inactive ($\Delta_{\texttt{tmi}}^{\texttt{TCPw}} = 0$). Since `TCPw`'s children are active ($\Delta_{\texttt{tmi}}^{\texttt{10.0.1.43}} > 0$), we know the fault lies with `TCPw` and mark 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 2.3.5 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 *(c)*, *(d)*, and *(e)* 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 us to misplace the fault.

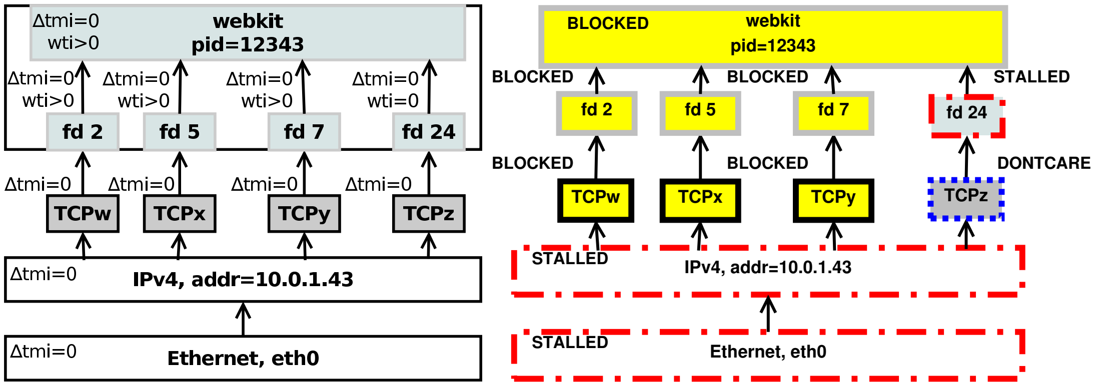
When *any* transport-layer (TCP) connection is active, the IP module will be active as well, since IP is a best-effort forwarding service and performs no buffering. So, when IP is not sending (receiving) datagrams, it must be because *all* of the active connections are experiencing performance problems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

In this case, we 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 we take 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.

We find a simple heuristic to work 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 we proceed down through the graph. Otherwise, there is not enough evidence to blame only the network, so we mark *both* the TCP connections and the lower-layer modules as STALLED, since it is impossible to distinguish between the

[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 we need 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 ($\Delta_{\texttt{total\_msgs}} = 0$), we assume 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.

two possibilities given the evidence we have; 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, we mark both the TCP connection module and the underlying modules (IP, Ethernet, et cetera) as Stalled.

We discuss possible improvements to the precision of our results in Section 3.6.

3.4 Data Collection Prototype

We have implemented a Linux prototype for end-host performance monitoring, which consists of three subsystems:

- a *collector* daemon that constructs the NS graph, takes periodic snaphots 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, we wrote simple adapters that read the implementation-specific counters and present them in the common counter format shown in Table 2.1. For TCP connections, we use the counters exported by the Web100 kernel patch [34]. IPv4 counters are gathered by reading `/proc/net/snmp`, Ethernet interface counters via `netlink` sockets [36], and wireless interfaces via device- and driver-specific interfaces.

To track application behavior, we intercept 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.

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.

Our 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), our delta is invalid and we discard the second snapshot. If the third snapshot is also lower than the first, we assume that the first snapshot was a spurious increase, and discard it and continue from there. We find in practice that these invalid snapshots are extremely rare, but that may be optimistic since the transfer rates were low (less than 100 Mbps). Since we snapshot counters fairly frequently, a few discarded snapshots have little impact.

[4] While the diagnosis engine can be run in real-time as part of the collector, our 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.

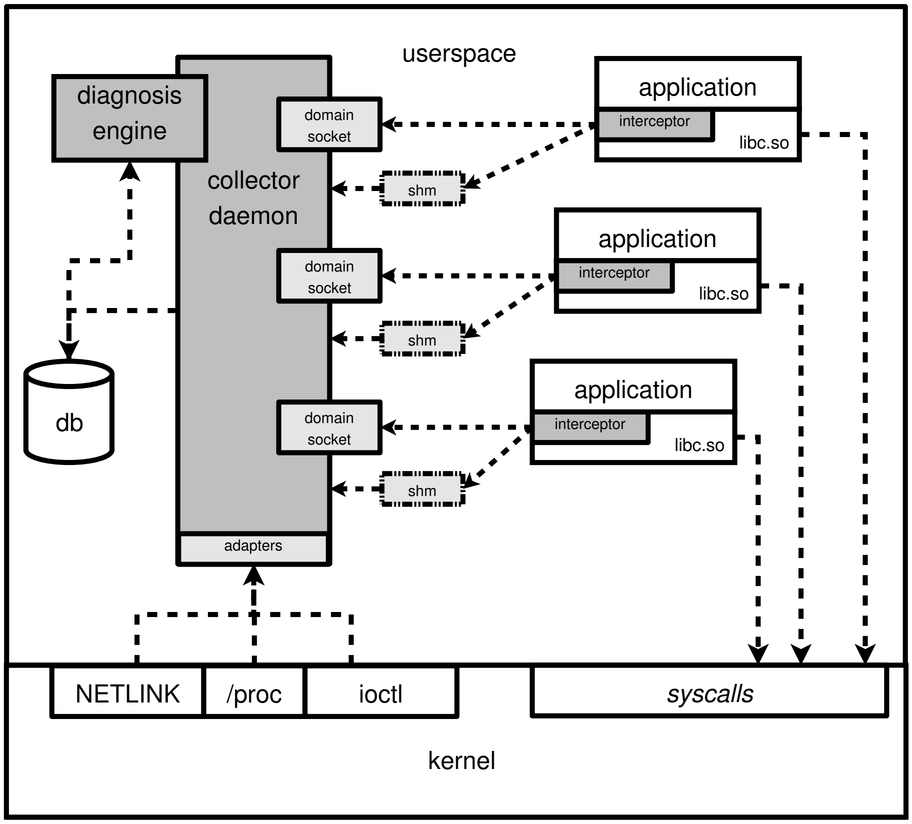


Figure 3.4: Network Stack Trace collection architecture

3.5 Experimental Results

To evaluate NeST's accuracy, we used it to diagnose injected faults into sample programs and network flows in a controlled setting. Out of the thousands of runs we have performed, we present the results from a representative series of experiments which cover all the fault scenarios and control groups run without faults. In our test scenarios, we found that NeST correctly diagnosed the faulty module more than 99% of the time, while incorrectly blaming modules only 3% of the time.

We describe our experimental setup in Section 3.5.1, and results from our accuracy evaluation in Section 3.5.2. We assess our prototype efficiency and potential improvements in Section 3.5.3, and explain some example diagnosis scenarios in Section 3.5.4.

3.5.1 Experimental setup

We ran our experiments on Emulab [43] 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, we instrument the network stack on host $site1n1$ and inject faults on its 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:

1. Pausing the application to force it to stop reading and writing from network 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 as STALLED. When all the threads stop reading/writing, the application should be marked as 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 connections five-tuple (source

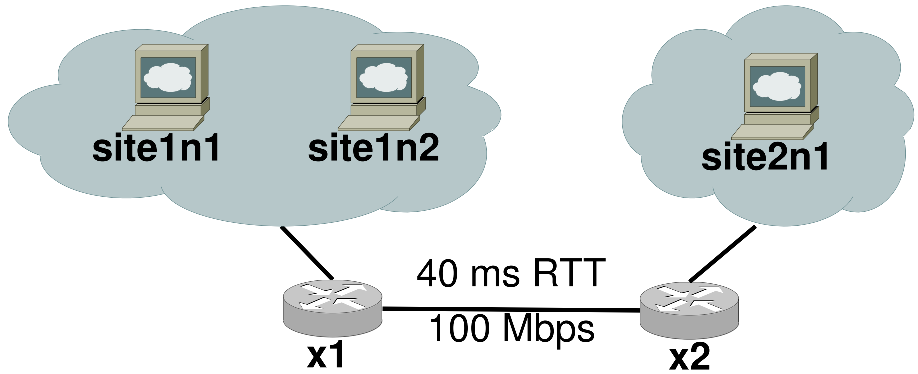


Figure 3.5: Emulab experiment topology. Two sites are connected by a single 100 Mbps bottleneck link between routers `x1` and `x2`. We inject faults by pausing applications, and dropping all packets on specific TCP connections and IP flows on router `x1`.

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 our experiments, so a single rule is sufficient.

Since we were unable to reliably inject layer-two faults using Emulab, we combine the IP endpoint and Ethernet modules together, labeled `ip+eth`, and use the counters provided by the Ethernet module (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 our 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.[5] The same holds for the applications themselves.

We now consider the ability of our diagnosis system to detect these injected faults. According to our evaluation criteria, the correct result is positive (STALLED) in two cases: (1) Upload-only or download-only applications and sockets are idle 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 our 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 we expect according to our evaluation criteria. Note that we always expect to mark `wget` and its sockets as STALLED in the outbound direction (outbound AN column = 0) since we ignore its outbound request traffic in our evaluation criteria.

- **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).

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

  - **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 we 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 we correctly detected as Stalled (TP/AP).
  - **False Positive Rate (FPR)**: The percentage of the Actual Negative periods we 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 we correctly detected (TN/AN).
  - **False Negative Rate (FNR)**: The percentage of the Actual Positive periods we incorrectly marked as *not* Stalled (FN/AP).
  - **Negative Predictive Value (PPV)**: The percentage of NeST negative diagnoses that were actually correct ($TN/(TN + FN)$).

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.

3.5.2.2 NeST evaluation results

As we can see from the TPR and TNR columns for the application and socket rows in Table 3.3, 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 us 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 we accurately detect almost all of the Actual Positives, we have a significant False Positive Rate (FPR) for inbound `ip+eth` traffic (7.3%) where we appear to blame `ip+eth` incorrectly. We also miss

		Correct Answers			NeST Diagnosis Results				Positive Accuracy %			Negative Accuracy %		
	Module	**Total**	**AP**	**AN**	**TP**	**TN**	**FP**	**FN**	**TPR**	**FPR**	**PPV**	**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%).

10.4% of the inbound TCP faults, as seen in the False Negative Rate (FNR) column for inbound TCP.

Likewise, while we detect 100% of outbound TCP connections' problems (TPR column), we have 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 our evaluation criteria, when our 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 we discussed in Section 3.3.2.1. There are three main cases which appear in our 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 we discussed in Section 3.3.2.1, in such single-flow situations our diagnosis 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 our efforts to limit network congestion, there are times in our 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 our evaluation criteria than our 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 our diagnosis is probably correct in these situations, these are counted as False Positives in our evaluation.

3.5.3 Prototype efficiency

While our goal is to create a data collection and diagnosis engine that is efficient, our prototype data collector daemon and diagnosis and analysis engine are written in Python, and not highly optimized. Nevertheless, we find them to be efficient enough to use for experimental purposes, even though we 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 our controlled experiments described below, our 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 we use to store the counters for post-processing.

Similar systems for application and TCP connection logging using Event Tracing for Windows [29] found that application event tracing increased median CPU utilization by 1.6% CPU, and disk utilization by 1.2%. Yu, et al measured the cost of reading the full Windows TCP statistics table (similar to the Web100 counters we use) every 50 ms to be 10% at 1000 connections and 30% at 5000 connections [45]. We expect an engineered version of our system would have similar overhead.

Creating the 158,718 total diagnoses in our 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 timeseries 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 timeseries heatmap,[6] 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).

Each row of the heatmap shows the delta value of the counter during that snapshot. We use green to indicate good behavior, and yellow/orange/red to 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

[6] Section 5.3.2 includes additional discussion.

(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, we vary the color 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 we have the `wait_time_in` and `wait_time_out` rows 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, we do not add a row 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 timeseries 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 we discussed in Section 3.5.2.5, 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 have manually confirmed our diagnosis to be correct.

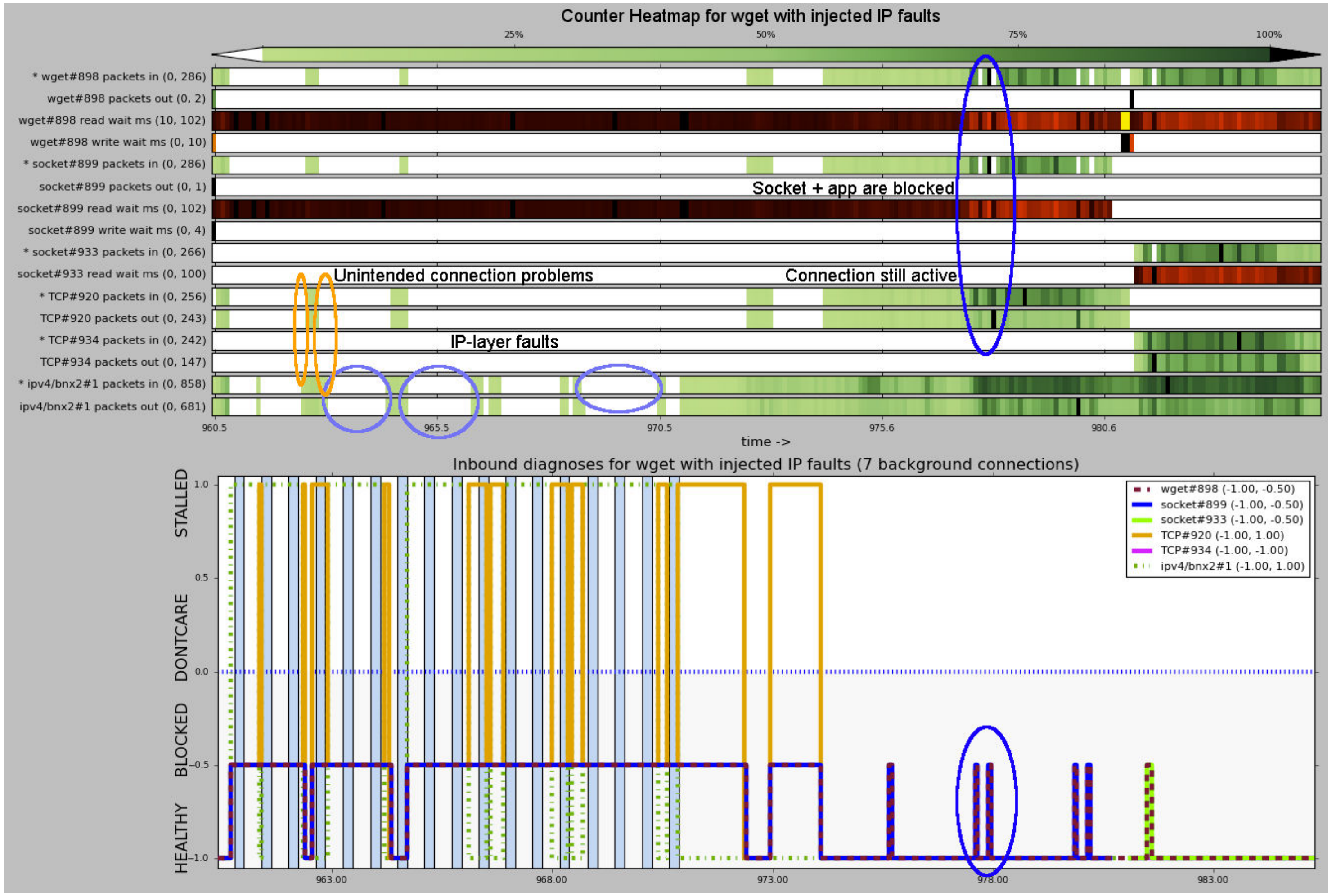


Figure 3.6: Inbound `wget` counters and diagnosis results plot. In addition to the `wget` download connection, there were 7 active background connections (not shown). The shaded blue areas in the bottom plot denote the periods when an IP fault was active. Looking at the `ip+eth` counters, it is apparent 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).

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 us 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 we expected a negative diagnosis for both modules, this result has no effect on our evaluation criteria.

3.5.4.2 Connection-specific fault injection

Figure 3.7 shows the counter heatmaps and diagnosis timeseries 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.

It should be noted that `TCP#714` occasionally recovers, and is marked as HEALTHY; this results in the vertical movment 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.)

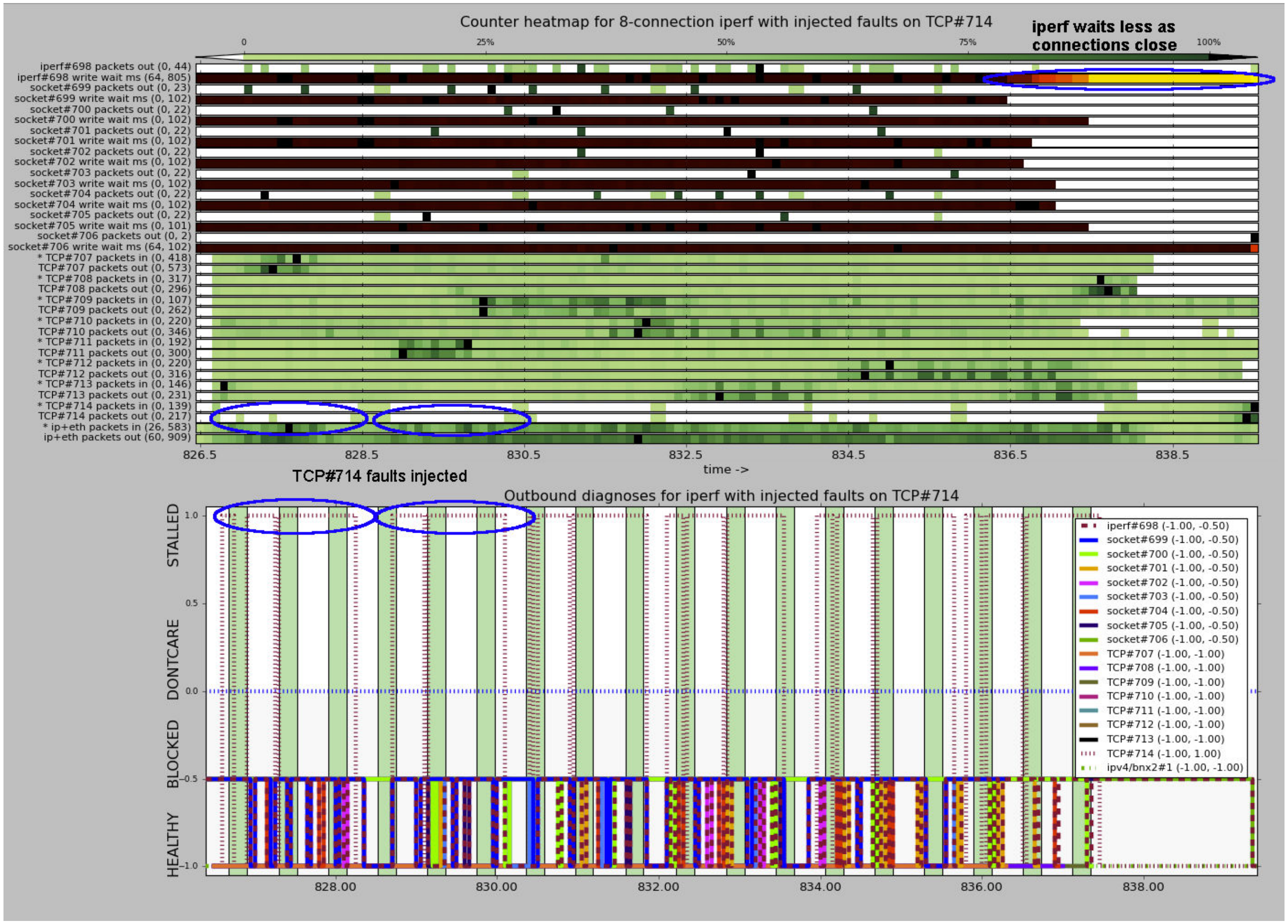


Figure 3.7: Outbound `iperf` counters and diagnosis results plot. The `iperf` process has 8 active outbound connections. One of them, `TCP#714` has faults injected (green shaded areas) and is marked as Stalled. Since the connection does not recover quickly, it continues to be inactive even after the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

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 timeseries 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 our 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 timeseries 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.

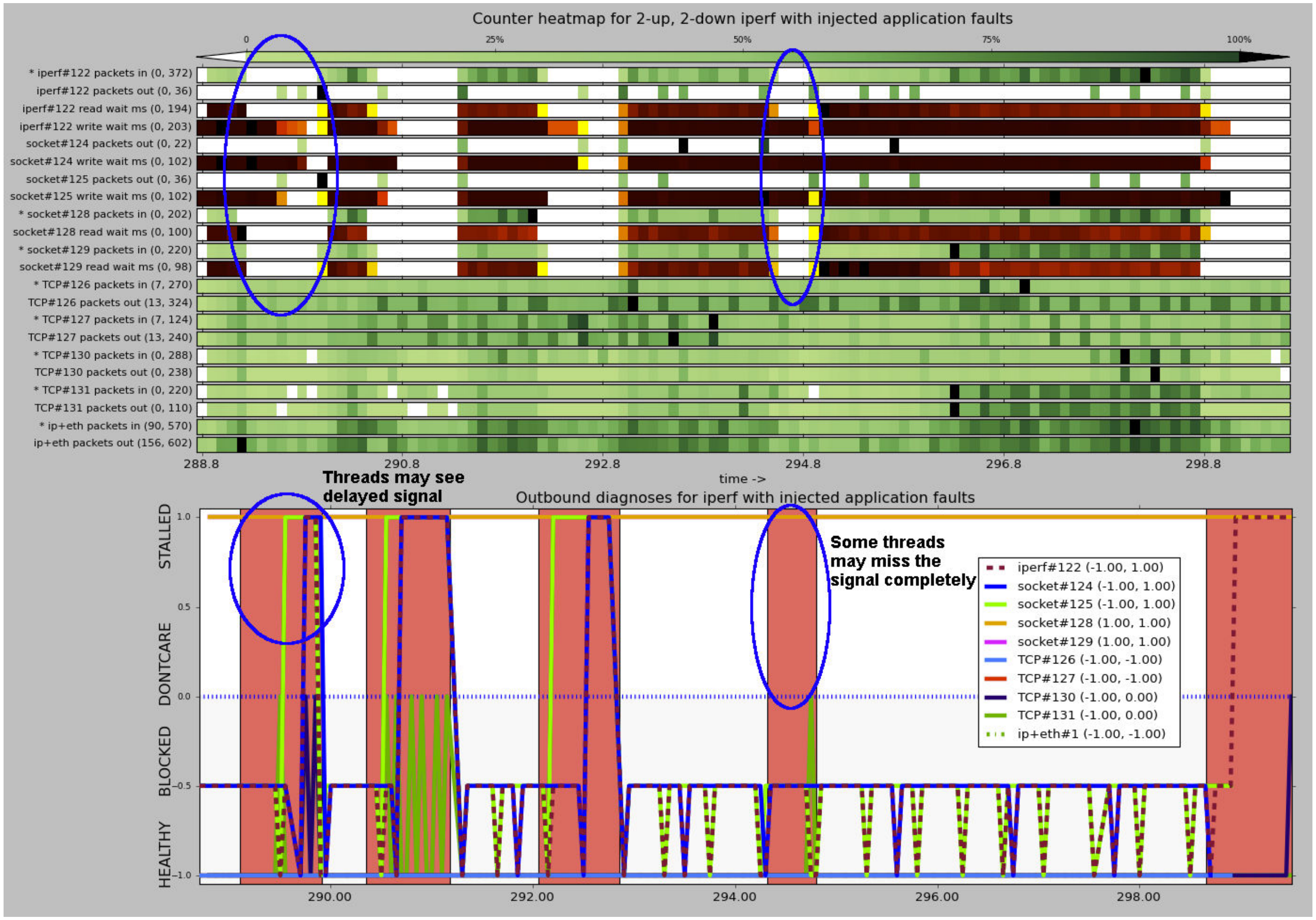


Figure 3.8: Outbound `iperf` counters and diagnosis results plot. The `iperf` process has 2 active outbound connections, and two active inbound connections. Faults are injected by sending a signal to the application to stop reading and writing from the network. As the `iperf` threads stop writing, the 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

Our 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 us 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, we have colored each connection 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`).

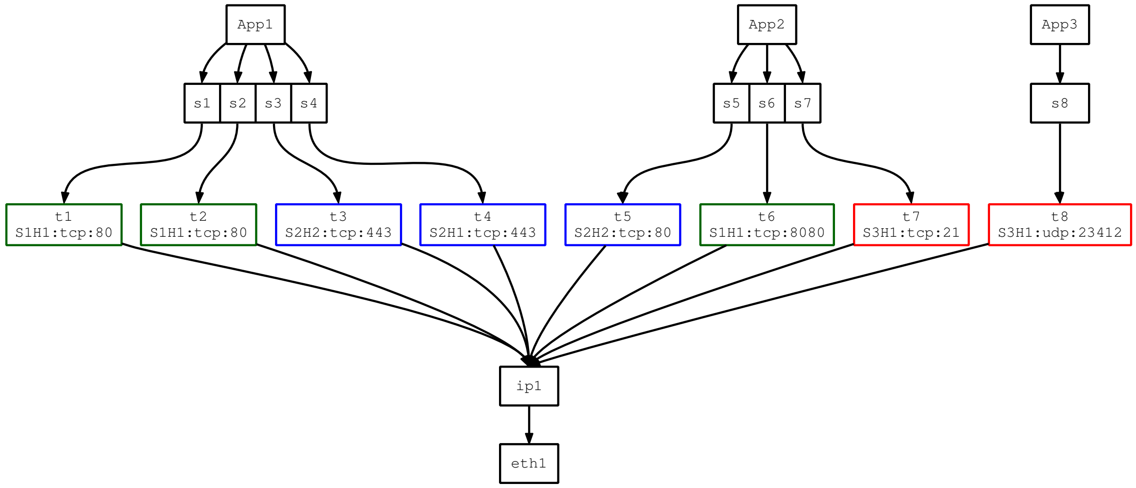
In some sense, our diagnosis lacks precision because the heuristic we described in Section 3.3.2 uses a rather crude independence assumption: in the ambiguous case shown in Figure 3.3, if the number of waiting connections N exceeds our threshold Θ, then a lower-level network-wide problem must have occurred.

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 we mark the IP module as STALLED).

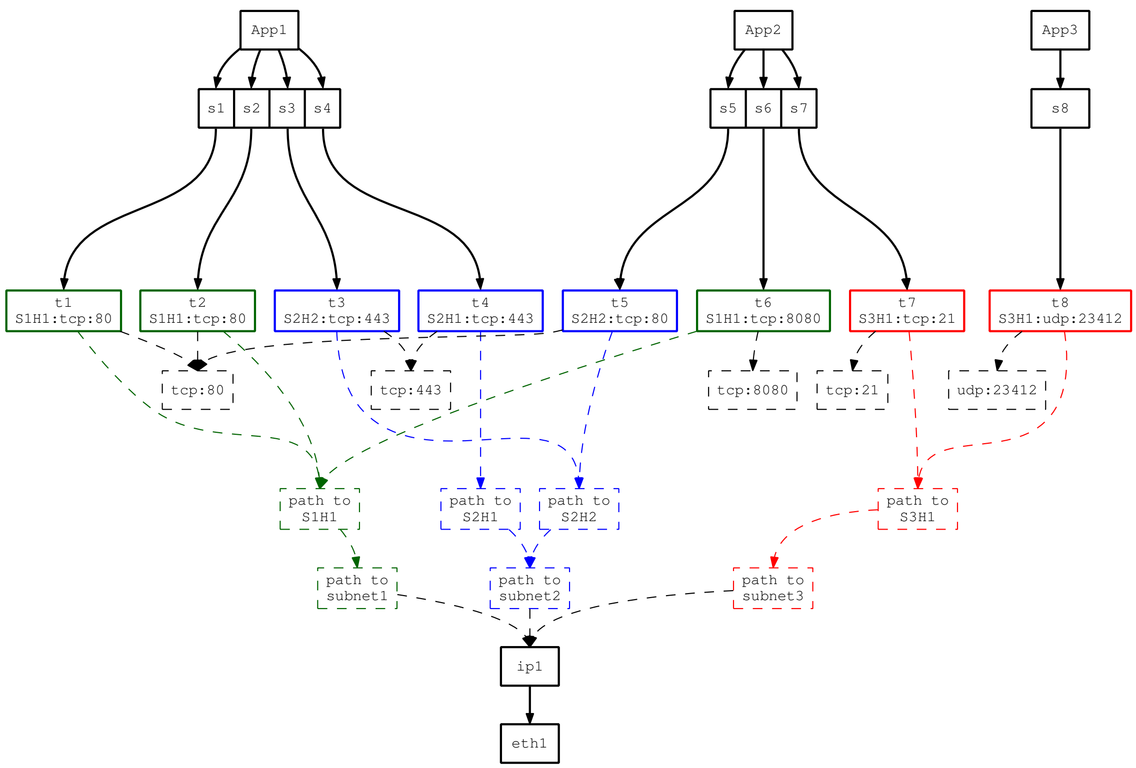
By ignoring these shared dependencies, our diagnosis results are imprecise in two ways:

1. We mark 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`, we would prefer to blame a single module that represents the "path



(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`).

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. We blame a broader-scoped module as STALLED than necessary.

   In addition to marking as few modules as possible, we would like to keep the scope of our blame narrow. When we mark the `ip+eth` module as STALLED, we are indicating that the entire network is experiencing problems, as far as we can tell. If possible, we would like to 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, we could attempt to detect problems due to potential firewall rules, e.g. those that 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. We expect that such complicated diagnosis would 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), we 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. We then look for attempts to make progress (via `wait_time` and `queued_msgs`), then whether any connections were completely idle (DontCare).

With this extension to the dataflow graph and a slight change to our inactive-module heuristic (Section 3.3.2), we 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 our heuristic), when comparing sets of parents and children that are all potentially blocked, we could use the following rule:

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

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, we mark the broader `ip+eth` module as Stalled; otherwise we only mark 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.

3.7 Summary

Using the FlowDiagnoser approach described in Chapter 2, we created the Network Stack Trace (NeST) system for *automatically* finding 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 [20].

We begin by describing the elements of the InfoSphere Streams programming model and the performance problems that InfoSphere Streams suffers (Section 4.1). Next we describe we construct the FlowDiagnoser dependency graph for streams applications (Section 4.2), which performance counters we collect and how (Section 4.3), and how we perform performance diagnosis (Section 4.4). Then we describe our prototype implementation (Section 4.5) and the results of controlled experiments (Section 4.6). We find that StreamsDiagnoser is able to detect 93% of injected faults in these experiments, with a False Positive Rate of 2%; our initial tests on real applications also provided promising results.

We conclude by discussing possible extensions to our approach.

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*

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

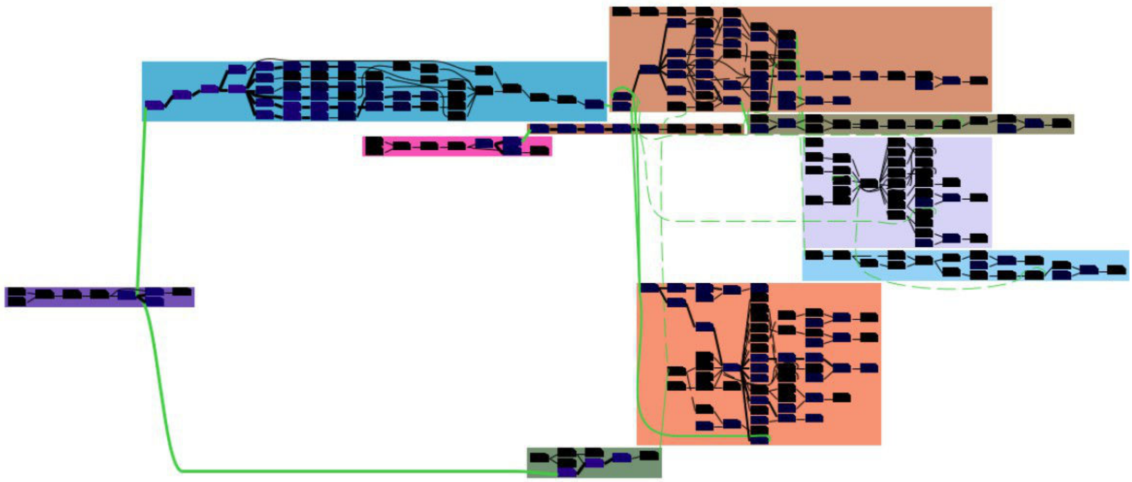


Figure 4.1: Streams Application. Each node in the Streams processing graph represents a single Processing Element (PE), which are connected via *stream connections* (black or green lines). PEs are grouped by their *job* (large colored rectangles). Stream connections that are exported/imported between 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.

(PE); for sake of discussion we will assume 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 our general model described in Section 2.1, 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 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, which we discuss 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 [26].

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

To facilitate our discussion of Streams applications, we will refer to the example Streams processing graph shown in Figure 4.2. 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`); we refer to them as $\mathtt{sconn}_1$–$\mathtt{sconn}_{18}$. 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`.

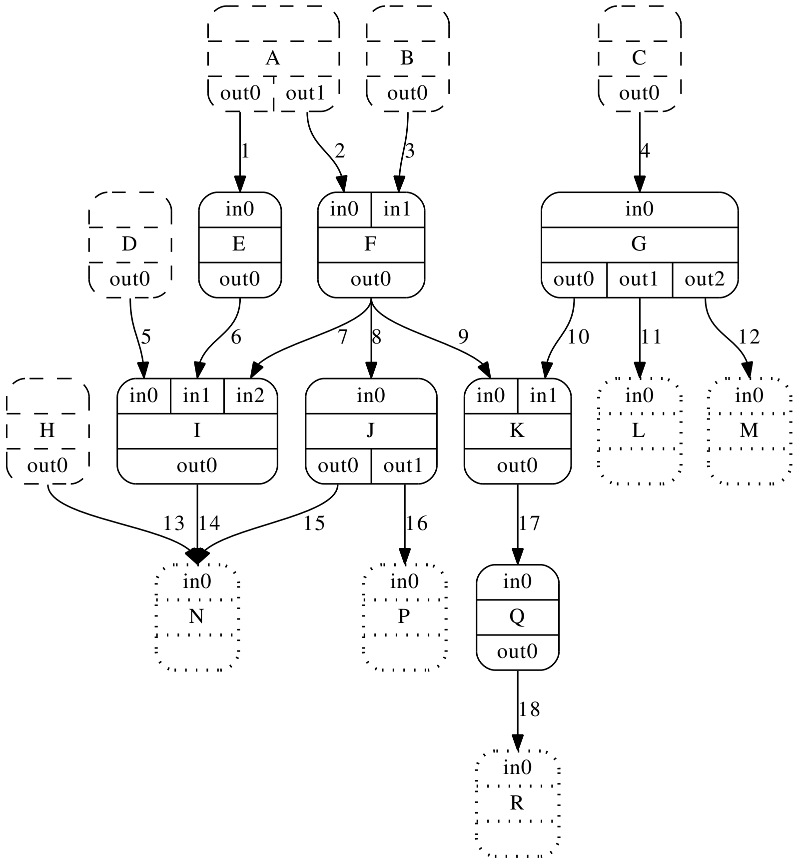


Figure 4.2: Example Streams processing graph. See the text for details.

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, we are interested in two of them:

- `nSubmitted`, which counts the total tuples submitted to an output port

- `nProcessed`, which counts the total tuples processed from an input port

In our 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, consider 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 become inactive. If the sources `D` and `H` are also inactive, their downstreams `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. Our StreamsDiagnoser system is designed to pinpoint which PEs are causing these problems.

[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.

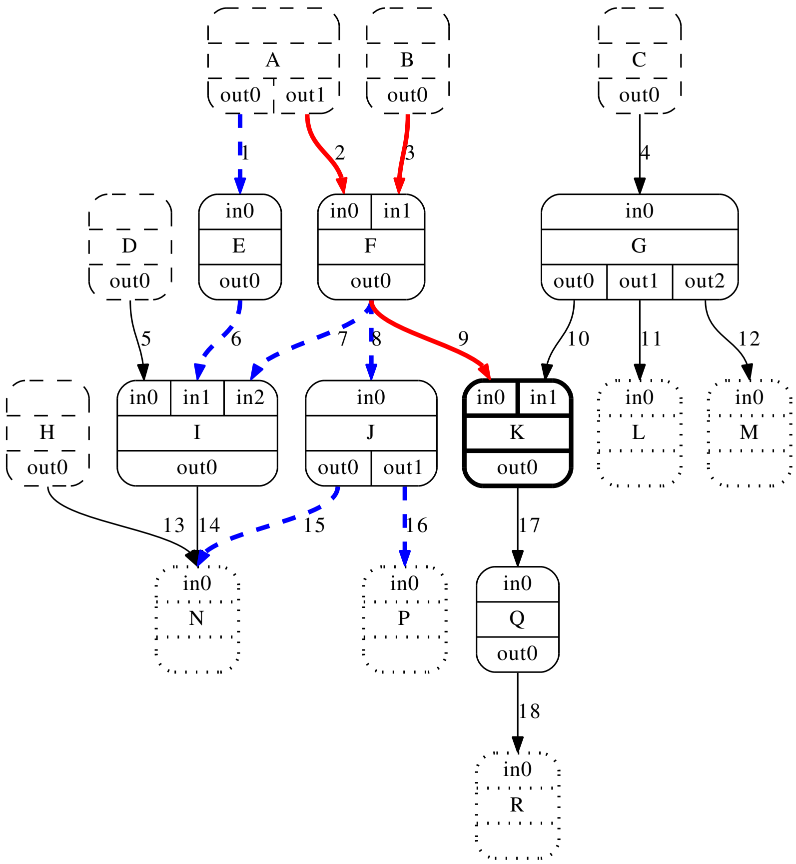


Figure 4.3: Example Streams processing graph with backpressure in effect. Heavy red lines indicate backpressure across stream connections. Dashed blue lines indicate where traffic has ceased due to blocked PEs. We would like to mark PE `K` as causing the problem that effects the other PEs.

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 our theoretical model (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.

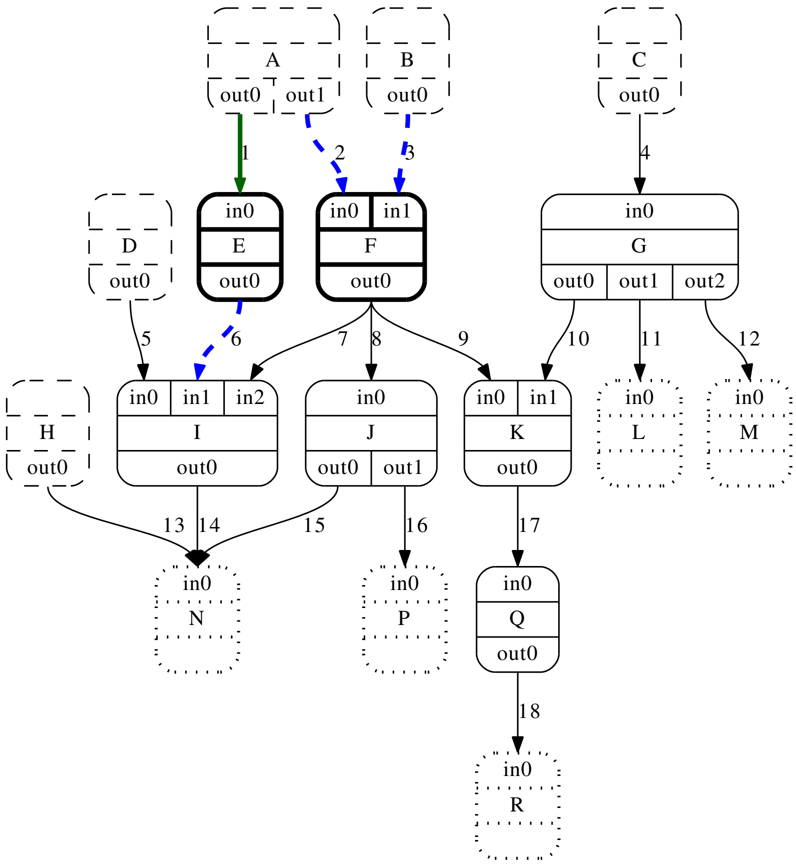


Figure 4.4: PE `A` has submitted tuples on $\texttt{sconn}_1$ (heavy green line), but not on $\texttt{sconn}_2$ (dashed blue line). `E` is filtering tuples that do not meet its criteria, and does not submit any to its downstream $\texttt{sconn}_6$. `F` is inactive since it has not received any tuples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

If Streams PEs behaved like modules in the network stack, we 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, we 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.

(b) Operators with multiple output ports may submit tuples to one port without submitting them to the others. Thus, we 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 we were to sum `A`'s `nSubmitted` counters, we would lose this information and may provide an incorrect diagnosis.

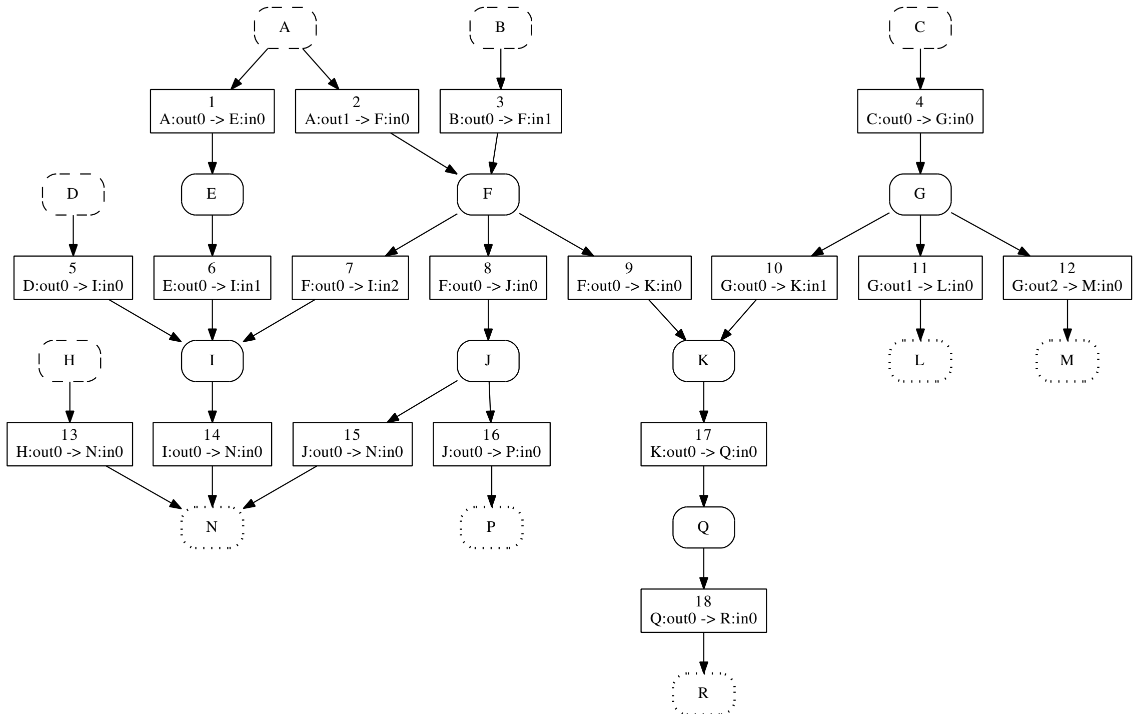
(c) Finally, when backpressure occurs, upstream modules are blocked from submitting any more tuples, and the `nSubmitted` counters no longer increase. Since we do not have a direct signal of whether a PE is waiting as we do for application sockets in the network stack (Chapter 3), we need some other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

4.2.2 Option 2: Ports as modules

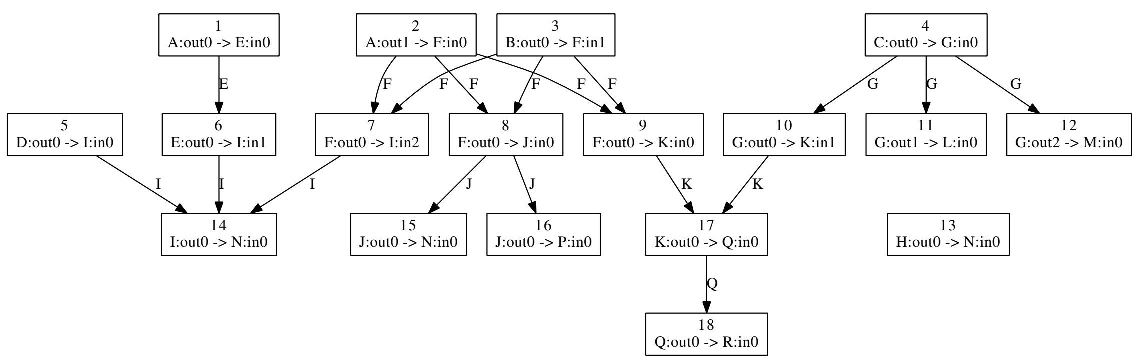
Rather than combining all of a PE's ports into one module representing the PE itself, we could treat each *port* as a module in our StreamsDiagnoser dependency graph, using `nProcessed` as our signal of activity on input ports, and `nSubmitted` for output ports. However, this would have many of the same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

4.2.3 Option 3: Stream connections as modules

The approach we settled on for transforming the PEs-with-ports Streams processing graph into our modules-and-edges StreamsDiagnoser dependency graph is to invert the streams processing graph, and turn each *stream connection* ($\mathtt{sconn}_1$, $\mathtt{sconn}_2$, …) into a module in our 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` $\rightarrow$ `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, we first convert each numbered streams connection into its own module in the graph (top subfigure). We then remove 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. We first create nodes for each numbered stream connection by pulling the output and input ports into a new node, and place them in the graph between the PEs (Figure 4.5(a)). We then remove the PE nodes and label the remaining edges; what remains is a dependency graph of streams connections (modules) that we use in our analysis (Figure 4.5(b)). Note that when we do 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`.

Note that the resulting graph is *push-oriented*: the direction of the edges in the dependency graph matches the flow of data through the system. We now describe how we assign the counters to the modules (stream connections) in the dependency graph.

4.3 Stream Connection Counters

Once we have obtained our StreamsDiagnoser dependency graph, we 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. We assign these to the StreamsDiagnoser module that represents each stream connection, and derive the desired FlowDiagnoser counters 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.

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.

These counters are shown in Table 4.1.

By calculating `queued_msgs` from the output and input port counters, we have our required signal that indicates whether the upstream PE is waiting on the downstream to complete its work, as we described in Section 2.3.3. 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 we 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, notice that 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 our dependency graph is based on individual stream connections (`F:out0` → `I:in2`), we expect 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 our example, we would prefer that `out0`'s message counter be one, rather than three. As such, we need 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 (i.e. $\Delta\texttt{nSubmitted}_{sconn} = \frac{\Delta\texttt{nSubmitted}_{port}}{nConnections}$).

Unfortunately, doing so is problematic when the number of connections changes during the snapshot period. We are currently working with InfoSphere Streams researchers and developers to obtain the counters we need without the normalization and bookkeeping procedures, which we describe in Appendix A.

4.3.2 Invariant violations

In addition to the counter normalization, there is an additional complication that arises when monitoring Streams applications. As we discussed in Section 2.2.1, the FlowDiagnoser approach does not require a stop-the-world snapshot of all of the counters. Even if we 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 we have observed while processing data from live systems is the violation of the one invariant that we hold on the StreamsDiagnoser counters:

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

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

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

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

When we detect a negative-length queue, we assume that some of the tuples processed have been *pre-counted*—that is, we have counted some processing that really belongs in the *next* snapshot. So, to avoid false positives (in which we blame the downstream for not processing anything at all in the next snapshot), instead of ignoring the negative queue we add $abs(\texttt{queued\_msgs})$ from this snapshot to $\Delta_{\texttt{total\_msgs}}$ in the next snapshot. This means for any snapshot that we calculate a negative queue, the downstream will always appear active (HEALTHY) in the next measurement period.

4.4 Streams Dependency Analysis

Once we have the counters for each stream connection (module in our dependency graph), we can apply the dependency analysis as described in Section 2.3. With the counters we have, it reduces to the following:

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

A change in the number of tuples processed ($\Delta_{\texttt{total\_msgs}}$) tells us whether or not the downstream PE (with the input port) was active; if so, the connection is HEALTHY. If it was inactive, we check `queued_msgs` to determine whether or not the downstream PE had work to do; if `queued_msgs` ≤ 0, the stream connection was empty and we diagnose it 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 (`queued_msgs` > 0), and we need to figure out why.

4.4.1 Detecting backpressure and inactive streams

As we 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 downstreams to consume (process), which leaves the downstreams 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 2.3.5 and Table 2.2 (criteria *(e)* and *(i)*) help us 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, we mark it as Stalled and blame it for blocking progress unless one or more of its children are also inactive with work to do. If its children are also inactive, we mark M as Blocked and blame 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 our dependency analysis looks at the number of tuples processed and still stuck in the queue, we are actually detecting 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.
- Stalled: PE `B` was inactive did not process tuples available on port y, and none of its downstream PEs were Blocked or Stalled.
- Blocked: PE `B` was unable to process tuples available on port y because one or more of its downstream PEs were Blocked or Stalled.

- DontCare: PE `B` did not have any tuples to process on port y because PE `A` did not supply them.

Hence when we blame a module, we are 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, we pass 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

We now describe our 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 you 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 we describe in Section 4.7; the snapshot intervals varied between 13–15 seconds.

For our controlled experiments, we used the internal API used by 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 our 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, we load the GraphML files into our diagnosis engine, which converts the processing graph into our internal StreamsDiagnoser dependency graph and stores the accumulated snapshots for each module (stream connection). We then run the diagnosis process for each snapshot, and store and output the results.

4.6 Experimental Results

To validate our 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. We find that StreamsDiagnoser is able to detect 93% of injected faults in controlled experiments, with a False Positive Rate of 2%; our initial tests on real applications also provide promising results.

4.6.1 Experimental setup

4.6.1.1 Basic topologies

There are six basic topologies we used in our experiments; they are depicted in Figure 4.6. These six include most of the basic forms that Streams applications are composed of [20]. 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 we replace 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.

We expect these last two topologies to 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 our diagnosis criteria, it is correct behavior and is not counted as an Actual Positive (AP) in our evaluation.

The source operator in each topology is a DynamicBeacon that emits tuples at a fixed rate. For our 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 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.

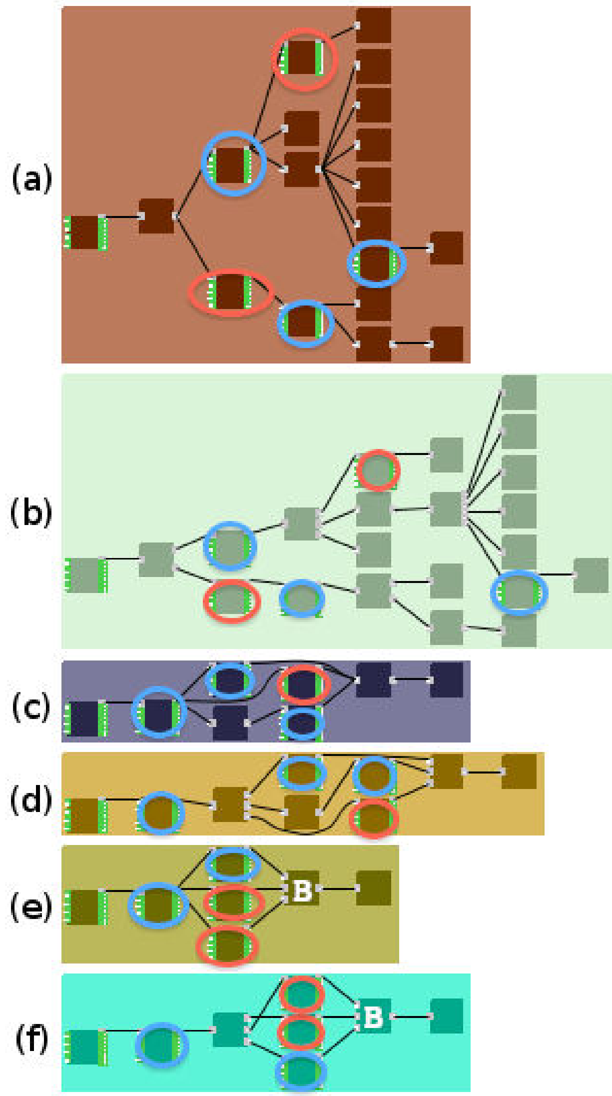


Figure 4.6: Basic Streams processing graphs used in the accuracy experiments. DynamicThrottle PEs are circled in red, and DynamicDropper (filter) PEs in blue; Barrier PEs are labeled with a `B`. Each processing graph is converted into a Streams-Diagnoser dependency graph according to the procedure outlined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

4.6.1.2 Injected faults

We inject faults in our 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 we discussed in Section 4.2.1, 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. In our experiments, the throttle varies between 0, 1, 100, and 1000 tuples/sec, 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 we are trying to diagnose.

These dynamic operators are located at strategic points throughout our test topologies. Specifically, we place them:

- At the sinks, to test our ability to detect problems at the edges of the graph.

- At, before, and after fan-out (branch) points, to test our ability to find problems on either side of a branch.

- At, before, and after fan-in (merge) points, to test our 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 downstreams 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) [37].

4.6.2 Diagnosis accuracy

Since we are looking for performance *stalls*, the only time the "correct" diagnosis is positive (STALLED) is when the DynamicThrottle is set to 0—that is, when it does not process any tuples on its input port. Since diagnoses are assigned to the stream connection (modules), we expect that *all* connections leading to the DynamicThrottle's input port will 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 match the filter criteria; this should not necessarily be considered a problem, as we explained in Section 4.2.1.

For all other modules, we always expect 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 our controlled experiments, we run the topologies on a single host, vary the source rates between 100 and 1000 tuples/sec, and take snapshots every five (5) seconds. By running on one host and limiting the source rate, we ensure that we do not overload the monitored system, so only our 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.

We present the accuracy results for each topology in Table 4.3. 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).

4.6.2.1 Barrier results

However, we 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 blamed a module in the MergeTreeBarrierMux experiment, it was correct only 33% of the time

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 MergeTreeBarrier (3.5%) and MergeTreeBarrierMux (3.2%).

(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 we should expect—two of our 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 our 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 $Throttle2 \rightarrow Barrier$, instead of blaming the stream coming *into* Throttle2 (as our evaluation expects), we again blame 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.

4.6.2.2 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 sytem. Both batches included the same set of fault injections described in Section 4.6.1.2.

In our full-rate experiments, we often mark modules other than the DynamicThrottle as Stalled. Our 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 our 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 we performed on the six basic topologies, we have evaluated StreamsDiagnoser on several real applications from a national laboratory and IBM research:

- App1: A highly tuned application that 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.

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.

Working with live data has helped us to validate and fix bugs in our data collection, transformation, and diagnosis engine, and identify the importance of the normalization steps described in Section 4.3. 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 the StreamsDiagnoser approach in their environments.

4.8 Summary

In this chapter, we presented StreamsDiagnoser, our application of the Flow-Diagnoser approach to finding performance stalls in InfoSphere Streams. Through experimental evaluation, we find that our automatic 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 timeseries visualizations that allow the user to see each module's diagnosis and counters *in relation to other modules in the graph*. In combination 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 [44]. 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

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.

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 timeseries: 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

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

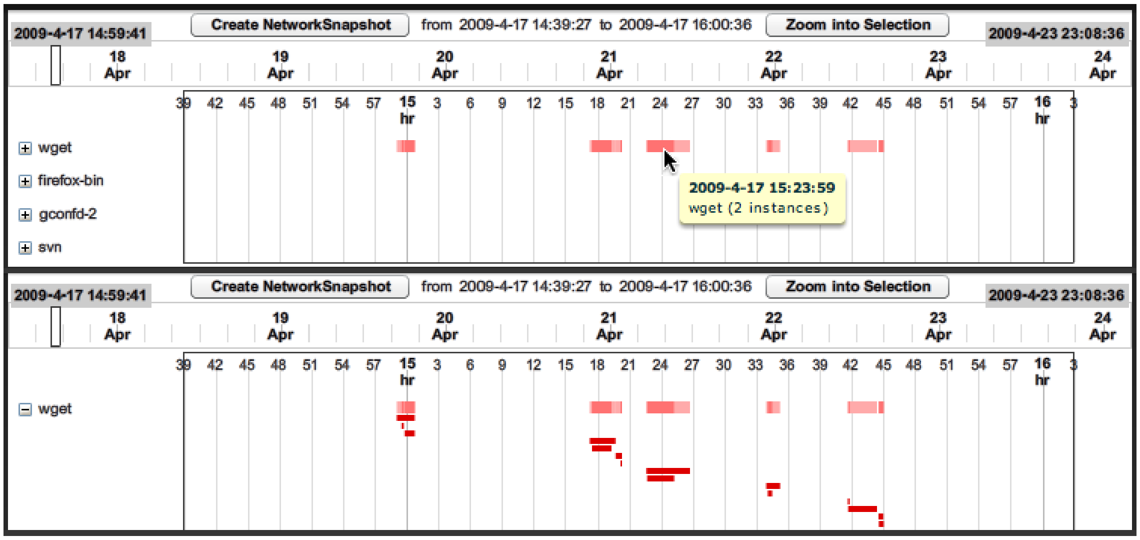


Figure 5.1: Network Stack Trace module timeline. The top pane shows the lifespans for the various monitored processes. The bottom pane shows an expanded timeline for the `wget` module. It includes the number of times `wget` was started and stopped (light pink bars), and the underlying socket and TCP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

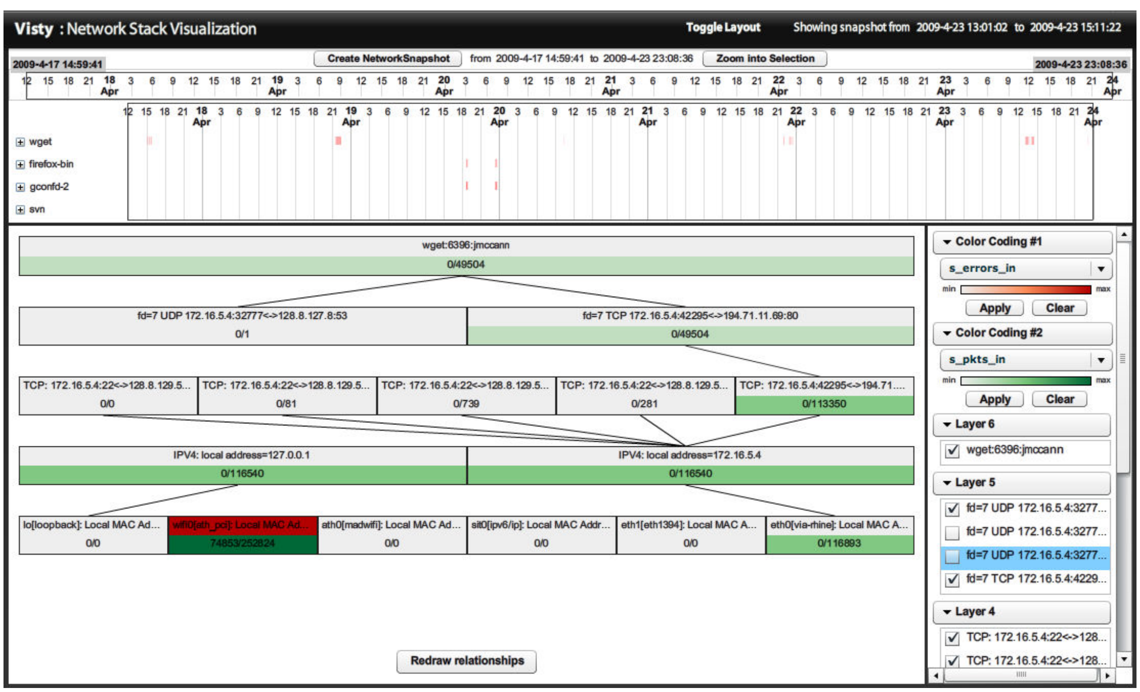


Figure 5.2: Network Stack Trace visualization prototype. It consists of an overview timeline in the top pane, and a depiction of the network stack in the bottom left pane. In the right pane, the user can select particular metrics for coloring nodes according to number of messages sent or received, or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

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.

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.

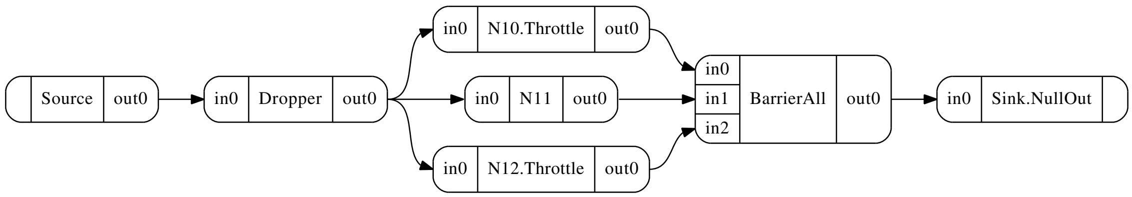
In this case study, the developer has been monitoring the MergeTreeBarrier job, and notices that the output rate is very low at times, and sometimes stops completely. Her goal is to identify which modules are causing the performance stalls, determine how they affect the rest of the system, and work toward 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

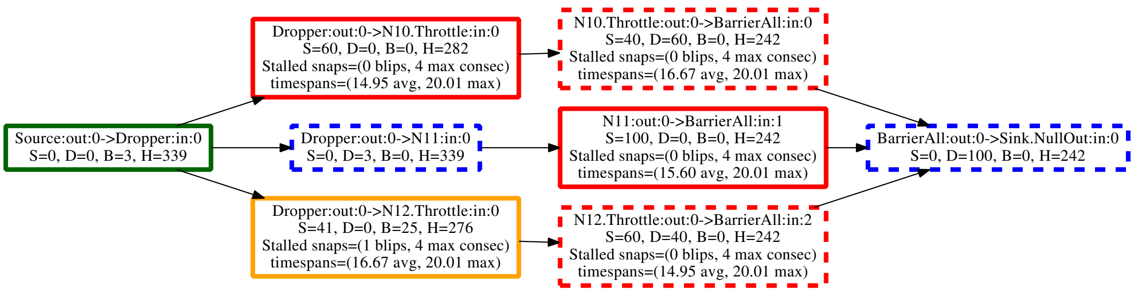
The developer's first step is to gain an overview of the system itself. Figure 5.3(a) shows the original Streams processing graph for the MergeTreeBarrier topology (from Figure 4.6(e)). Connected to the BarrierAll operator are two DynamicThrottle operators, N10 and N12, and a simple pass-through (N11)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

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



(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.

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 module 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 the 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]

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.

[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.

[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.

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 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 ($y = 0$), Blocked ($y = -0.5$), and Healthy ($y = -1.0$).

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

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.

[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.

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 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 Dropper→N12.Throttle:input:0 connection 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).

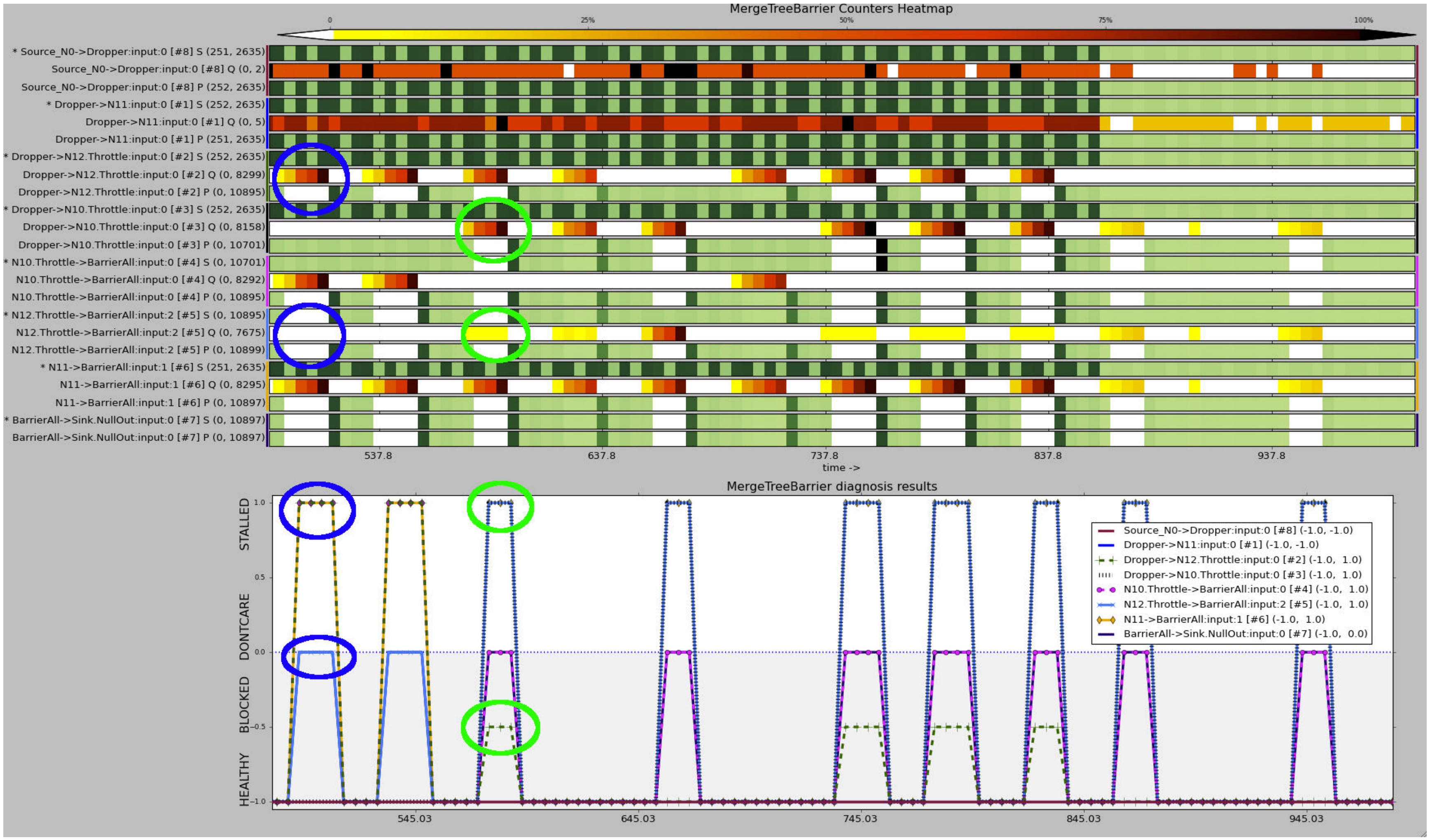


Figure 5.4: MergeTreeBarrier diagnosis timeline. The top figure shows the counters heatmap, and the bottom figure the StreamsDiagnoser results.

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
- 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

[6]The evaluation in Section 4.6.2 counts the second two faults as False Positives.

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 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 timeseries 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, 14, 22, 32, 34, 39, 45]. Some of them are geared toward software analysis [22, 39] or specific network protocols [12, 14, 32, 34, 45], 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 [12]. 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 [2,14,28,31,45]. 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 [34], 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 [32]. The mechanisms built in to Web100 and Pathdiag are limited to diagnosing behavior from the TCP sender side, since they use signals such as 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 Scalable Network-Application Profiler (SNAP) [45] uses application event logs and Web100-like TCP instrumentation [33] 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 7.1.2), we may be able to increase the level of detail provided in our 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 combined 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 [30], 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.

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 [22] and Pip [39] 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 reduces 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, 13, 14] or system-level messages [3,15,21,22,39] 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 [13] 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 [30]. 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. [15]. 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 our 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 communication ceases altogether. 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 altogether [9].

NeST is able to reliably diagnose temporary network performance *degradation* from a single end host, a much more difficult goal. Specifically, NeST is able to diagnose transient 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.

NetMedic [28] 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.

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, our need for system-wide data is limited: we need only a few counters from each communicating process. Of course, our StreamsDiagnoser output is limited to identifying which PE (operator) is not behaving correctly, and is unable to assign faults to underlying network elements (which we do not model).

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 diagnose the cause of messaging-related performance stalls. Using two signals— one to indicate node activity, and another to indicate waiting—the automatic FlowDiagnoser diagnosis system 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, we show that these performance stalls have a significant affect on system performance, and affect users' attitudes about the usability of a system. Our theoretical 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. We also evaluated FlowDiagnoser in comparison with similar approaches. We now conclude with directions for future work.

7.1 Potential extensions

We now discuss some potential extensions to the FlowDiagnoser approach to expand the breadth of the performance problems it can diagnose, or the specificity of its results.

7.1.1 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. We believe FlowDiagnoser can 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, we may need to relax our diagnosis criteria in some way, since relative processing rates will 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 we signal 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 we mark the module as a bottleneck.

7.1.2 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. We could also 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, we 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—our analysis would then verify whether its actual behavior meets expectation. We could also incorporate TCP-specific or interface-specific diagnosis criteria [14, 32, 34, 45]. The general-purpose dependency analysis described in Chapter 3 could be used for modules that have no implementation-specific criteria, or to handle conflicting results.

7.1.3 Online diagnosis

Several groups are interested in using StreamsDiagnoser in an online fashion to detect performance stalls and kick off a reconfiguration of the Streams processing graph, either to share load or restart stalled PEs. We would also like to use NeST to 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.

7.1.4 Additional systems

Finally, we have designed FlowDiagnoser to work at many levels of abstraction. While we look at process-to-process communication in StreamsDiagnoser, we expect it will work equally well (but with higher processing overhead) when looking at each operator inside a Streams PE. We hope to be able 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, we expect this will require additional data processing and improvement since even our loose counter semantics may be too stringent when collecting data across many systems.

7.2 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.

Our 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.

Using prototype implementations and controlled experiments in two distinct environments—an individual host's networking stack, and a distributed streams processing system—we prove 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 our current model transformation is based on individual stream connections (`F:out0` $\rightarrow$ `I:in2`), we expect counters that track the number of tuples submitted into the connection, and the number of tuples processed out of the connection. Therefore, we need 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.

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

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

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

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

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

Table A.1

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

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

Our normalization step at each snapshot is to divide the *increase* in the output port's `nSubmitted` by $nConns_{OUT}$, the number of connections currently attached to that port. This gives us the normalized total value

$$\texttt{nSubmitted}^{NORM} += \frac{\Delta_{\texttt{nSubmitted}}}{nConns_{OUT}}$$

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

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

The value of $\texttt{nProcessed}_{\texttt{N:in0}}$ accounts for all of the tuples processed on that input port, regardless of the sender. That is, it is the sum of all the inputs from `H:out0`, `I:out0`, and `J:out0`. While we may want to normalize the `nProcessed` counter itself like we do for `nSubmitted`, we cannot do this reliably—on an $N \to 1$ input port, it is impossible to know how many tuples have been processed from an individual incoming stream. We only know 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, and cause False Negatives on `N:in0` as it will appear to have read all of the tuples from a single connection even when it may not.

To account for the $N \to 1$ normalization, we keep a shadow counter `nSubmittedToInputPort`. After we have normalized all of the output port counters (as described in Case 2), we then sum $|Delta_{\texttt{nSubmitted}^{NORM}}$ across all of the output ports connected to this input port. This sum is the total number of tuples submitted to this input port in the last snapshot period. We then increase `nSubmittedToInputPort` by this sum.

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

Once we have correctly normalized the counters on the output ports (Case 2), we can apply the summation procedure from Case 3, and the counters work as expected.

A.2 Counter normalization algorithm

Our algorithm to recover these per-connection counters is:

1. Baseline all counters from the first snapshot, since we do not know the number of connections that arrived and departed before we obtained the first snapshot, to determine the rate of increase for each port's counters.

2. Re-baseline at every topology change where the number of connections changed.

3. Divide any increase by the current number of connections for each port.

In practice, this means that we create two shadow counters. For `nTuplesProcessed` as reported by InfoSphere Streams, we have the `nSubmittedToInputPort` shadow counter to track messages *submitted* into this input port by all of its incoming connections.

For `nTuplesSubmitted` as reported by InfoSphere Streams, we have the shadow counter `nTuplesSubmittedToConnectionX` which is the $\texttt{nSubmitted}_{\texttt{OUT:i}}^{NORM}$ value referenced above.

The information we require to do this normalization includes:

1. A first-count baseline (to factor out changes we never saw)

2. A last-change baseline (to factor out the earlier changes in `nConnections`)

3. Current `nConnections` (to divide the current increase)

4. The `nTuplesProcessedThisStream` [`nSubmitted`] counter that we actually use

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 et al. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, 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*. ACM, 2003.

[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 et al. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*. ACM, 2007.

[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*, 2007.

[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, 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] Ranveer Chandra, Venkata N. Padmanabhan, and Ming Zhang. WiFiProfiler: cooperative diagnosis in wireless LANs. In *MobiSys*. ACM, 2006.

[13] Xu Chen et al. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*. USENIX Association, 2008.

[14] Yu-Chung Cheng et al. Automating cross-layer diagnosis of enterprise wireless networks. In *SIGCOMM*, 2007.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] Dan Gunter et al. Netlogger: A toolkit for distributed system performance analysis. In *IEEE MASCOTS*, 2000.

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

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

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

[26] IBM streams processing language (SPL) online documentation. `http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1`, March 2012.

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

[28] Srikanth Kandula et al. Detailed diagnosis in enterprise networks. In *SIGCOMM*, 2009.

[29] 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.

[30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21, July 1978.

[31] Zhichun Li et al. WebProphet: automating performance prediction for web services. In *NSDI*. USENIX Association, 2010.

[32] Matt Mathis et al. Pathdiag: Automated TCP diagnosis. In *PAM*, Apr 2008.

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

[34] Matt Mathis, John Heffner, and Raghu Reddy. Web100: Extended TCP instrumentation for research, education and diagnosis. In *SIGCOMM*, 2003.

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

[36] 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.

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

[38] 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.

[39] Patrick Reynolds et al. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[40] 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.

[41] 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.

[42] 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.

[43] Brian White et al. An integrated experimental environment for distributed systems and networks. In *OSDI*. USENIX ATC, Dec. 2002.

[44] 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, New York, NY, USA, 2009. ACM.

[45] 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.