The Network Stack Trace: Performance diagnosis for networked systems

Justin McCann
jmccann@cs.umd.edu
University of Maryland

Michael Hicks
mwh@cs.umd.edu
University of Maryland

Abstract

Transient network stalls that degrade application performance are frustrating to users and developers alike. Software bugs, network congestion, and intermittent connectivity all have the same symptoms—low throughput, high latency, and user-level timeouts. In this paper, we show how an end host can identify the sources of network stalls using only simple counters from its local network stack. By viewing the network stack as a producer-consumer dependency graph and monitoring its activity as a whole, our rule-based expert system correctly identifies which modules are hampering performance over 99% of the time, with false positive rates under 3%. The result is a network stack trace—a lightweight snapshot of the end host's networking stack that describes the behavior of each application, socket, connection, and interface.

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 enterprise environments where all the data may be available. End users have an even more difficult time trying to diagnose system performance. When a user's video stream has problems, it could be for any number of reasons: the browser plugin may be buggy, the neighbors' wireless networks may be creating interference, their computer or the server may be overloaded, or there may be congestion along the Internet path. To the user, the symptoms are all the same: a stalled or stuttering application.

In this paper, we present Network Stack Trace (NeST), a system with which an end host can identify the source of short network stalls that lead to low throughput, high latency, and connection timeouts. Our goal is roughly equivalent to determining where messages are being blocked or dropped, e.g., in the application, in TCP's buffers, in the IP network, or at the physical layer. NeST treats a host's network stack as a dependency graph of modules (applications, sockets, connections, tunnels, interfaces) that provide service to each other. Higher-layer modules must produce messages for lower-layer modules to send, and consume messages as they are received. As described in Section 2, rather than trace specific messages through the stack to see where they are getting dropped or hung up, we monitor a few basic counters exported by each module, and accumulate evidence from them to make a diagnosis. In particular, diagnosis proceeds in three steps: (1) snapshot the packet counters, queue lengths, and error counters for each module; (2) make observations from those counters about the module's (in)activity; and (3) perform a dependency analysis, relating one module's state to that of its dependents and neighbors, to determine the likelihood that the module is misbehaving.

For the last step we employ a heuristic-based expert system, described in Section 3. Comparing a module's behavior with that of related modules allows us to resolve questions that cannot be answered by examining each module in isolation. For example, if a module M's counters are not increasing, the module could be stuck, or it might simply have nothing to do. To provide evidence in favor of one explanation over the other, we can examine the counters at M's predecessor P: if we find that P's counters have not increased either, then M is unlikely to be at fault; on the other hand, if P's counters have increased, then we can infer that messages are getting stuck at M and place more blame there. We can look at M's neighbors to make further inferences. In particular, a host generally has more than one open network connection, so if many flows are experiencing problems, then the culprit is likely a dependency held in common. All of these inferences are inexact, so our algorithm employs probabilistic *certainty factors* [5] to accumulate the weight of the evidence for or against a certain module, ul-

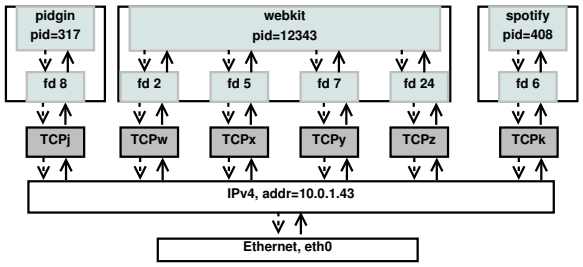


Figure 1: Example network stack graph. Application and socket modules are shown in light blue, TCP instances in grey, and IPv4 and Ethernet in white. Application modules are made up of component socket modules. The stack is conceptually two graphs; dotted arrows show the outbound dependency edges, and solid arrows the inbound dependencies.

timately producing a final, compact diagnosis of the most likely source (or sources) of the problem.

To evaluate NeST's effectiveness, we implemented a prototype data collection and diagnosis system in GNU/Linux, discussed in Section 4. In Section 5 we present results from experiments conducted by injecting network and application faults in a controlled environment, causing applications to stop reading and writing from sockets, dropping packets on specific TCP connections, and dropping all packets along a network path. These experiments indicate that our system is able to detect unresponsive modules over 99% of the time, with a false positive rate of less than 3%.

Compared to related approaches, discussed in detail in Section 6, NeST offers several benefits, as it:

1. Is fully decentralized and requires no cooperation among peers, and can be implemented on any end host with immediate benefit

2. Relies on common event counters, and requires no per-packet information, improving efficiency

3. Uses a protocol-independent algorithm to detect unresponsive modules, ignoring protocol minutiae, and combining information from multiple sources to increase the quality of the diagnosis

We believe that NeST can form a key part of comprehensive framework for distributed system performance fault diagnosis, which is a goal for ongoing and future work.

2 Network Stack Trace

Network Stack Trace (NeST) diagnoses a system's network performance problems at the level of modules. Each module appears as a node in a *network stack graph*

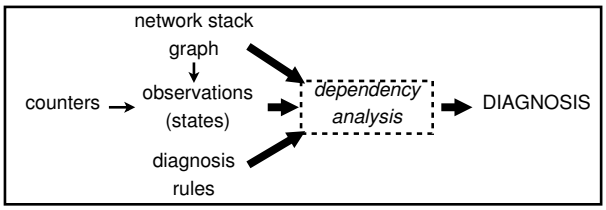


Figure 2: The diagnosis process. Module counters are used to determine each module's behavior. This is combined with the dependency graph and a set of heuristics to determine the impact of each module; the resulting performance diagnosis is an annotated graph.

(NS graph) that captures the elements of the host's network stack and the applications that (currently) use it. An example graph is given in Figure 1, which portrays three active applications (pidgin, webkit, and spotify), connected via sockets (with various file descriptors (FDs)) to one or more TCP connections, each of which is homed at the same IPv4 address and Ethernet interface. 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.

NeST diagnoses potential problems according to the steps illustrated in Figure 2:

1. Learn and maintain the structure of the graph (the monitored system's nodes and their connections)

2. Observe each module's *activity state* based on periodic snapshots of performance counters

3. Compare each module's state with that of nearby modules to diagnose whether it is unresponsive.

The resulting diagnosis is an annotated version of the NS graph, where each module has been assigned a pair of decimal numbers between -1.0 and +1.0, one for outbound traffic, and another for inbound traffic. Each number indicates *the level of confidence that the module is unresponsive* to traffic flowing in that direction. A +0.9 certainty indicates high confidence that the module is behaving poorly, and -0.9 indicates high confidence that the module is performing well. Values near zero indicate a high level of uncertainty.

Now we consider each of the three steps listed above.

2.1 The NS graph

In the first step we acquire the structure of the dependency graph. We do this by instrumenting an end host's network stack to track applications, sockets, connections, and interfaces and their dependencies as they come and go (details in Section 4). Each node in the graph depends on the *parent* module(s) above it in the protocol stack

Counter	Description
`total_msgs`	Total messages emitted (required)
`wait_msec`	Milliseconds spent waiting for service
`errors`	Messaging errors (collisions, drops, etc.)
`buffered_msgs`	Current number of messages in the buffer

Table 1: NeST module counters. Two sets of counters are kept for each module—one for outbound (sent) traffic, and another for inbound (received) traffic.

and *child* modules below. For outbound flows a module depends on its parents to provide messages, while for inbound flows a module depends on its children to provide messages. We can view the network stack as representing two dependency graphs, one for the inbound direction and one for the outbound direction, which are identical except for the direction of their edges. As applications and connections open and close, we track the changes to the graph and update it accordingly.

To provide for varying levels of abstraction, modules can be made up of *components*. These components are modules in their own right, but when combined together describe the behavior of a broader entity. For example, an application's network behavior is fully described by the traffic on all its sockets, and traffic from a guest virtual machine may be considered as a whole.

2.2 Counters and activity states

In the second step, we attempt to discern each module's *activity state*: whether it is active and processing messages, waiting for messages to serve or its dependents to handle the messages it has produced, or idle and neither waiting for messages nor doing any work. We infer a module's activity state by periodically snapshotting a small set of packet and performance counters and interpreting their values according the module's location in the NS graph. Each module maintains a separate set of counters for outbound flows and inbound flows, leading to independent observations of a module's outbound and inbound states.

The message counters we track are shown in Table 1. For the first three counters, we are interested in the *difference* between the current value and the value taken in the last snapshot to detect activity during the snapshot interval, while for the last we read just its current value. At the least, we require each module to count the number of messages processed (`total_msgs`); other counters, if supported, can be used to make more fine-grained determinations of the module's state. Table 2 illustrates the six activity states we infer from these counters.

If a module has processed any messages it is considered active (state ACT). If the module is not active (`total_msgs` $= 0$) then either it is idle or it is blocked waiting for messages from dependent modules. For application modules we can directly distinguish between the two situations by observing whether the application attempted to read or write from its sockets. As such, NS-graph modules representing application sockets export the `wait_msec` counter, which counts the number of milliseconds spent in blocking system calls such as `read()`, `write()`, and `connect()`, and for non-blocking operation the amount of time spent in `poll()`, `select()`, and their equivalents. When the module has processed no messages and its `wait_msec` counter is nonzero, we say the module's state is WAIT to indicate that the module is completely blocked. On the other hand, if the module processed no messages and `wait_msec` = 0, then the module's state is IDLE: we know that it processed no messages because it did not attempt to read or write any.

For all other modules we have no such direct indication of intent, but we can derive some useful information based on the module's location in the graph. In particular, suppose that a socket module is in the WAIT state because the TCP connection serving it has provided it with no messages. It would be useful to know whether the TCP connection is itself waiting for messages from the IP module, or it is otherwise behaving badly. Since we cannot know what TCP is doing exactly, we mark it as ANW because at the least we know it has an ancestor that is waiting, which suggests that it could be waiting too. In general, we do this for any inactive module that does not itself export the `wait_msec` counter.

On the other hand, we do not care about the exact state of a non-leaf module if all of its furthest ancestors (the applications and sockets) are IDLE; in this case it is clear that any fault lies with the applications since they are not reading/writing messages. In this case we give the module state IGN. In all other cases the module could be at fault for a stall, but we cannot say for sure what it is doing, so we give it state NONE.

The last two counters, `buffered_msgs` and `errors`, provide additional information about a module's behavior, and we discuss them further in Section 3.

2.3 Diagnosing system performance

The goal of the diagnosis process is to estimate the likelihood that a module is unresponsive. While the activity state of a module speaks directly to this hypothesis, the activities of related modules in the NS graph provide additional clues. For example, each producer is expected to emit messages that its consumer(s) should process. The observation that a producer has emitted some messages (its state is ACT) but its consumer has not processed any

State	Description	Counters Involved
ACT	Module is active	`total_msgs` > 0
WAIT	Module is waiting (blocked by another)	`total_msgs` $= 0$ and `wait_msec` > 0
IDLE	Module is idle (not sending/receiving)	`total_msgs` $= 0$ and `wait_msec` $= 0$
ANW	One or more ancestors were waiting	`total_msgs` $= 0$, `wait_msec` unsupported, some ancestors waiting
IGN	Ignore this module	`total_msgs` $= 0$, `wait_msec` unsupported, all ancestors idle/ignore
NONE	Module is inactive	`total_msgs` $= 0$, `wait_msec` unsupported (otherwise)

Table 2: Module activity states, and how they are calculated. The ACT, IDLE, WAIT, and NONE states are observed directly. The ANW and IGN states are propagated to children from their parents, and overwrite the NONE state.

(its state is NONE) suggests the consumer, not the producer, is unresponsive. On the other hand, if a consumer is waiting to receive messages but a producer is not providing them, the producer is blamed and the consumer is absolved.

We use *certainty factors* [5] to accumulate evidence according to encodings of rules such as these. We describe our approach in detail in the next section.

3 The rule system

We perform a diagnosis by comparing the states of pairs of nearby nodes in the NS graph. With each comparison we accumulate evidence in favor of or against the hypothesis that a given module is the source of a stall. In particular, the diagnosis process assigns a *level of confidence* in the range $[-1.0, +1.0]$ to each hypothesis, "module X is unresponsive (idle)" where a strongly positive level indicates a high degree of certainty the hypothesis holds, a strongly negative level indicates a high degree of certainty that it *does not* hold, and a level around zero indicates maximal uncertainty. We implement the diagnosis using a well-known approach for heuristic systems involving the repeated application of *certainty factors* (CF). Certainty factors come from the original MYCIN research [5] and in our system represent the *change in belief* that the indicated module is unresponsive [11].

Our algorithm, given in Section 3.2, basically works in two phases. First, we consider each pair of neighboring nodes in the NS graph, with node P as the producer and node C as the consumer. Given the observations B_P and B_C of their activity states we index a two-dimensional *diagnosis table* DT to acquire a pair (σ_P, σ_C), where σ_P and σ_C are the certainty factors with which we update the hypotheses that P and C are at fault, respectively. (In other words, $DT[B_P][B_C] = (\sigma_P, \sigma_C)$.) This first pass assigns credit or blame to each module according to the behavior of their parents and children. In the second phase, we identify and absolve groups of modules all blocked on a common child, since the fact that they are all blocked implicates that child, and the first phase might have heaped two much blame on the parents.

In the next subsection we describe the diagnosis table that forms the key part of the first phase of the algorithm, and in the following subsection we consider the second phase in more detail in the context of the full algorithm. We conclude this section with an example diagnosis.

3.1 Diagnosis tables

Table 3 depicts two diagnosis tables: table DT_{OUT} in Table 3(a) applies to outbound flows, and table DT_{IN} in Table 3(b) applies to inbound flows; each is indexed by observations made for a module's outbound and inbound traffic, respectively. (We write DT_* to refer to both tables.) The rows of the table identify the producer and the columns the consumer, where the cells contain the two certainty factors, the lower left to update the producer's diagnosis and the upper right to update the consumer's diagnosis. The first six rows and columns are indexed by activity states, while the remainder consider supporting evidence as explained below.

The symbols representing CFs in the table are defined in Table 4. Following the original certainty factor approach [5], we have four classes of CFs (Definite, Extreme, Strong, Weak) to represent the significance of each piece of evidence, either for or against. A module's current diagnosis is updated by each additional CF using the *parallel_combine* function ($\oplus$) [12] shown in Figure 3. This function balances the effect of supporting evidence, which pushes the diagnosis further in the same direction, against contradictory evidence, which pulls it back relative to the evidence accumulated so far. For example, if the accumulated evidence has a certainty of $+0.6$, an additional CF of $+0.4$ increases our confidence that the module is unresponsive to $+0.76$. Contradictory evidence with a -0.4 CF decreases the level of certainty, resulting in a weaker $+0.33$ confidence that the module is at fault. Definite factors overwhelm any other evidence, and consequently $1 \oplus -1$ (and $-1 \oplus 1$) are undefined.

σ_P \ σ_C	ACT	NONE	IDLE	WAIT	ANW	IGN	QZ	QNZ	ERR
Producer (parent) Observations									
ACT	▶◀	◀◁	▶○	◀●	◀◀	◀●	●●		○
NONE	◁●	○○	○	◁●	◀	●	○●		○
IDLE	○▶	○▶	○○	○●	◁◀	○●	▶	●	
WAIT	●○	●▷	●○	●●	●◁	●	▷	◁	●○
ANW	◁○	●◁	◀◁	●●	●◁	◀	◁	○	
IGN	●	●	●	●	●	●●			
QZ	○●	○					○●	●○	
QNZ	◁●	○							●○
ERR	○	○○							◁

(a) Diagnosis Table DT_{OUT} for OUTBOUND flows.

σ_P \ σ_C	ACT	NONE	IDLE	WAIT	ANW	IGN	QZ	QNZ	ERR
Producer (child) Observations									
ACT	◀◀	●○	▶○	○●	○◁	●			○○
NONE	○●	○○	◀○	▷●	◁●	●	○●		
IDLE	○	○◀	○○	○●	○◀	●	◁◀		
WAIT	●	●	●○	●●	●●	●			
ANW	◀	◀	◀◁	◁●	◁◀	●			
IGN	●	●	●	●	●◀	●●			
QZ		○	◀▷	◁	○		●		
QNZ		●◁	▶◁				●○		
ERR	◁	◁●		◁●	○				◁○

(b) Diagnosis Table DT_{IN} for INBOUND flows.

Table 3: Diagnosis Table, showing the evidence (certainty factors) attributed to each module based on the local observations. The presence of messages in the consumer's queue is ignored for inbound flows. The symbols are defined in Table 4.

Symb.	CF	Meaning
○	+1.0	Definite supporting factor
▷	+0.8	Extreme supporting factor
◁	+0.6	Strong supporting factor
○	+0.4	Weak supporting factor
●	−0.4	Weak contrary factor
◀	−0.6	Strong contrary factor
▶	−0.8	Extreme contrary factor
●	−1.0	Definite contrary factor

Table 4: Certainty Factors used in the diagnosis tables.

$$\sigma_1 \oplus \sigma_2 \stackrel{\text{def}}{=} \begin{cases} \sigma_1 + \sigma_2 - (\sigma_1 \times \sigma_2) \\ \qquad \text{when } \sigma_{i,j} \geq 0 \quad (i,j \in \{1,2\}) \\ \sigma_1 + \sigma_2 + (\sigma_1 \times \sigma_2) \\ \qquad \text{when } \sigma_{i,j} < 0 \\ \dfrac{\sigma_1 + \sigma_2}{1 - \min\{|\sigma_1|, |\sigma_2|\}} \\ \qquad \text{when} (-1 < \sigma_i < 0) \wedge (0 \leq \sigma_j < 1) \\ \sigma_i \quad \text{when } |\sigma_i| = 1 \wedge \sigma_j \neq -\sigma_i \end{cases}$$

Figure 3: The *parallel_combine* function ($\oplus$). Given the current level of certainty σ_1 and new information σ_2, return an updated level of certainty. Our version is modified to incorporate ± 1.0 (absolute) certainty factors.

Our use of two tables and our design of the contents of those tables follows several high level principles that generally apply to all modules. We now consider these principles in turn. In this discussion, we sometimes write $DT[B_P][B_C]\#P$ to refer to the producer's factor σ_P and $DT[B_P][B_C]\#C$ to refer to the consumer's factor σ_C. We also write $*$ or a list x,y,z as indices to refer to multiple certainty factor pairs; e.g., we write $DT[\text{WAIT}, \text{NONE}][*]$ to refer to all certainty factor pairs in the WAIT and NONE rows of the table.

Principle 1: Push- vs. Pull-based flows Since the vast majority of network traffic on a host is ultimately driven by the applications that send and receive it, separate tables for outbound and inbound flows are needed. In particular, outbound flows are *push-oriented*: activity originates at the application and travels out of the interface from source-to-sink. Therefore, diagnosis table DT_{OUT} assumes that if the application *produces* no messages, no *downstream* nodes should be blamed. On the other hand, inbound flows are *pull-oriented*, with the inbound data dependent on the application asking for it. Therefore, diagnosis table DT_{IN} assumes that if the application *requests* no messages, no *upstream* nodes should be blamed. This duality requires that we encode these rules in two separate tables.

Principle 2: Producers must produce If a module is not producing messages, it is generally blamed by its consumer(s), as indicated by the positive CFs (○,◁,▷,◯)

in $DT_*[\text{IDLE},\text{NONE}][*]\#P$. Conversely, a module is performing well while it is emitting messages, so the rules absolve producers with negative CFs (●,◀,▶) in $DT_*[\text{ACT}][*]\#P$.

An exception is when an active module's consumer is still waiting, in spite of the messages produced. This is a weak indication that the producer module is unresponsive, as seen by the blame assigned to the producer

$$DT_{IN}[\text{ACT}][\text{WAIT},\text{ANW}]\#P = \circ$$

in the inbound table.

Principle 3: Consumers must consume When the producer emits messages, the consumer must pass them along. As such, $DT_*[*][\text{ACT}]\#C$ absolves the consumer unless its producer is waiting (WAIT,ANW). A careful inspection of this column highlights a subtle difference in the two tables. Sockets are the only type of *child* that can receive the IDLE and WAIT observations, and applications cannot be ACT unless at least one of their sockets is ACT; any additional idle or waiting sockets should not give additional credit to the application so these cells are empty. Additionally, a parent can only *receive* messages if (1) at least one of its children is active, or (2) a child has messages in its receive buffer; we do not credit the parent in this second case.

One the other hand, if the consumer is inactive, it is blamed in $DT_*[*][\text{IDLE},\text{NONE}]\#C$. The exception to this rule is when the producer is IDLE and the consumer's state is ambiguous (NONE), which partially absolves the consumer ($DT_*[\text{IDLE}][\text{NONE}]\#C$).

Principle 4: Waiting implicates lower layers When a module is WAIT or ANW, the module and/or one of its ancestors is blocked, waiting for service. This is a significant indication of performance problems, and absolves the waiting module, as seen by the negative CFs (●,◀,●) in $DT_*[*][\text{WAIT},\text{ANW}]\#C$ and $DT_*[\text{WAIT},\text{ANW}][*]\#P$. Since in our implementation only (top-level) applications and their sockets are explicitly waiting, we also pass along blame to lower layers when they are idle (IDLE, NONE, or ANW).

Principle 5: Idle hands are the devil's workshop If we know a module is idle ($DT_*[\text{IDLE}][*]\#P$ row, and $DT_*[*][\text{IDLE}]\#C$ column), we blame it directly (○). We also strongly absolve its dependents with the negative CFs (◀,▶,●) in $DT_{OUT}[\text{IDLE}][*]\#C$ and $DT_{IN}[*][\text{IDLE}]\#P$, which absolve the child in both cases. This is an example of why we require two separate diagnosis tables—we need to pass the credit to the lower layer of the stack (the child) and not simply to the producer or consumer.

Principle 6: Queues signal completion In addition to inferring a module's activity state as discussed in Section 2.2, in some cases we can make two additional observations from a module's counters and use this to improve the diagnosis.

When a module is inactive (i.e., is not ACT or WAIT), we need to determine whether the module *should* be active. While the comparisons of module states is our primary way of doing this, we make an additional observation if the module supports the `buffered_msgs` metric (Table 1): either the queue is currently empty (QZ) or has something in it (QNZ); this suggests whether the module has work left to do.[1] Queue lengths can change frequently, so a one-time snapshot by itself is not a very reliable metric. Therefore we only make this observation when the module is inactive, and apply weaker certainty factors.

If the child's outbound queue is empty, it has completed its work and $DT_{OUT}[*][\text{QZ}]$ partially absolves the child unless the parent is waiting (WAIT and ANW rows). If the child has not emptied its queue ($DT_{OUT}[*][\text{QNZ}]\#C$), the blame is stronger. If the parent's outbound queue is non-empty ($DT_{OUT}[\text{QNZ}][*]\#P$), the module still has messages to transmit and is blamed accordingly.

For inbound traffic, the consumer (parent's) queue provides less insight, since the inbound queue is generally driven by higher layers. Thus we update the diagnosis only when the child is idle, as seen in the $DT_{IN}[\text{NONE},\text{IDLE}][\text{QZ}]$ cells. The child's queues, however, indicate whether the consumer (parent) has done its job, as seen in the $DT_{IN}[\text{QNZ}][\text{NONE},\text{IDLE}]$ cells.

Principle 7: Errors signal problems Finally, problems experienced by a module may be signaled by the presence of errors including timeouts, checksum errors, and dropped, retransmitted, or duplicate messages. All of these are tracked in the `errors` counter, where a nonzero counter corresponds to the observation ERR. Generally, errors provide only a weak indication of problems (σ_P in the ERR row, and σ_C in the ERR column), but they have stronger weights when they occur at lower layers of the protocol stack (children).

3.2 Algorithm

Our full algorithm is given in pseudocode below. The first phase compares parent-child pairs of nodes in the graph and diagnoses their inbound and outbound behavior using the diagnosis table given previously. The second phase considers parent-child nodes (X,Y) that are

[1] A more complete system would observe when queues are full, but we have not implemented such a check due to limitations in the metrics exported by the Linux kernel.

both blocked (i.e., that are WAIT, ANW, or NONE). For each other blocked parent of Y we absolve X. This is because multiple blocked parents with the same child (e.g., multiple TCP connections with the same IP module) suggest the child is at fault, so that blame that was applied to the parents in the first phase should be compensated for. In short, the first pass passes blame to the lower layer and partially absolves each parent module, while the second pass credits each parent for their shared disappointment.

```
// all modules with {WAIT,ANW,NONE} state
WaitingModules_IN = ... WaitingModules_OUT = ...

forall modules M in the graph: // init
  diag_OUT[M] = diag_IN[M] = 0.0

// pairwise comparisons across graph edges
forall pairs(X,Y) in NSG: // X parent, Y child
  // check outbound (X producer, Y consumer)
  forall B_X ∈ observations_OUT[X]:
    forall B_Y ∈ observations_OUT[Y]:
      diag_OUT[X] ← diag_OUT[X] ⊕ DT_OUT[B_X][B_Y]#P
      diag_OUT[Y] ← diag_OUT[Y] ⊕ DT_OUT[B_X][B_Y]#C
  // check inbound (X consumer, Y producer)
  forall B_Y ∈ observations_IN[Y]:
    forall B_X ∈ observations_IN[X]:
      diag_IN[Y] ← diag_in[Y] ⊕ DT_IN[B_Y][B_X]#P
      diag_IN[X] ← diag_in[X] ⊕ DT_IN[B_Y][B_X]#C

// absolve blocked co-parents
forall pairs(X,Y) in NSG: // X parent, Y child
  if (X ∈ WaitingModules_OUT and Y ∈ WaitingModules_OUT):
     forall (parentsOf(Y) ∩ WaitingModules_OUT) − {X}:
       diag_OUT[X] ← diag_OUT[X] ⊕ ◄
  if (X ∈ WaitingModules_IN and Y ∈ WaitingModules_IN):
     forall (parentsOf(Y) ∩ WaitingModules_IN) − {X}:
       diag_IN[X] ← diag_IN[X] ⊕ ◄
```

3.3 Diagnosis example

To demonstrate how NeST works, consider the diagnosis of inbound flows using the network stack graph shown in Figure 4. In this case, we assume that only the WebKit browser is running (with process ID 12343), with four open sockets (using file descriptors 2,5,7, and 24) to four TCP connections (respectively labeled `TCPw`,`TCPx`,`TCPy`, and `TCPz`). All of the connections are originated from local IPv4 address `10.0.1.43`, which is assigned to Ethernet interface `eth0`.

Module observations Suppose our snapshot of counters occurs when `webkit` is reading on sockets 2, 5, and 7, and not attempting to read `fd 24`. Both `fd 5` and `fd 7` are receiving messages from their TCP connections (`TCPx` and `TCPy`), so they are marked as ACT. Consequently the `webkit` application has also received messages and is marked as ACT.

Although it has been attempting to read from `fd 2`, `webkit` has not received anything on that socket, so `fd 2` is marked as WAIT. The `TCPw` connection has not received any messages, and its receive queue is empty. Since `fd 2` is waiting on `TCPw`, `TCPw`'s state is set to ANW, and since the queue is empty we make the additional observation QZ.

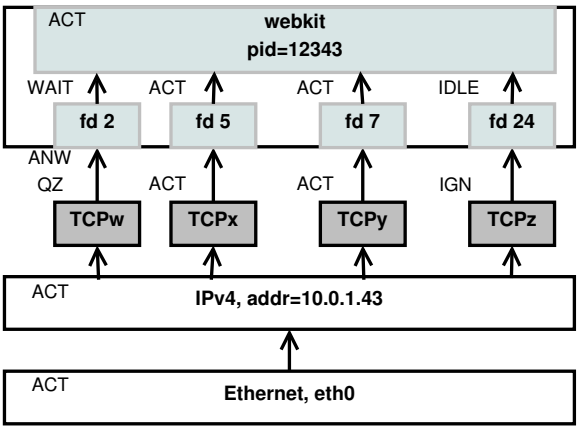


Figure 4: Network stack graph for inbound diagnosis example. Inbound observations are shown to the upper left of each module.

Since `webkit` is not reading from its final file descriptor `fd 24`, the socket is marked as IDLE and consequently its inactive `TCPz` connection is marked IGN.

Finally, both `10.0.1.43` and `eth0` have received messages (for `TCPx` and `TCPy`) so both are marked as ACT.

The diagnosis process Diagnosing these observations produces the (inbound) stack trace shown in Table 5.

Starting at the application, `webkit` is actively receiving (ACT). We cross-check `webkit`'s observations with the observations from each of its (producer) children. Comparing `webkit` with each of its active sockets `fd 5` and `fd 7`, each socket is absolved once, and `webkit` twice, with CFs -0.6 since

$$DT_{IN}[\text{ACT}_{\texttt{fd 5}}][\text{ACT}_{\texttt{webkit}}] = (\blacktriangleleft, \blacktriangleleft) = (-0.6, -0.6)$$

$$DT_{IN}[\text{ACT}_{\texttt{fd 7}}][\text{ACT}_{\texttt{webkit}}] = (\blacktriangleleft, \blacktriangleleft) = (-0.6, -0.6).$$

Its child `fd 2` is waiting, so `fd 2` is completely absolved ($\bullet = -1.0$), but `webkit`'s diagnosis is unaffected. The IDLE socket `fd 24` is completely blamed ($\circ = +1.0$), but again `webkit`'s diagnosis is left alone since we do not know the application semantics. It is possible that the socket *should* be idle from the application's point of view, so we simply blame the socket and leave it to the user to interpret the result. Hence `webkit`'s final diagnosis is $(-0.6 \oplus -0.6) = -0.84$(good).

Moving down the stack, we now compare the consumer sockets with their producer connections. Socket `fd 2` is blocked trying to read from `TCPw`. We have

$$DT_{IN}[\text{ANW}_{\texttt{TCPw}}][\text{WAIT}_{\texttt{fd 2}}] = (\triangleleft, \bullet) = (+0.6, -1.0)$$

Module	Observations	Diagnosis	From consumers (parents)	From producers (children)
webkit	ACT	−0.84 (good)	N/A	0.0 from fd 2 and fd 24, −0.6 from TCPx, −0.6 from TCPy
socket fd 2	WAIT	−1.00 (good)	−1.0 from webkit	−1.0 (ANW), 0.0 (QZ) from TCPw
socket fd 5	ACT	−0.84 (good)	−0.6 from webkit	−0.6 from TCPx
socket fd 7	ACT	−0.84 (good)	−0.6 from webkit	−0.6 from TCPy
socket fd 24	IDLE	+1.00 **(poor)**	+1.0 from webkit	0.0 from TCPz
TCPw	ANW,QZ	+0.94 **(poor)**	+0.6 (ANW), +0.6 (QZ) from fd 2	+0.6 and 0.0 from 10.0.1.43
TCPx	ACT	−0.84 (good)	−0.6 from fd 5	−0.6 from 10.0.1.43
TCPy	ACT	−0.84 (good)	−0.6 from fd 7	−0.6 from 10.0.1.43
TCPz	IGN	−1.00 (good)	−1.0 from fd 24	−1.0 from 10.0.1.43
10.0.1.43	ACT	−0.89 (good)	+0.4 (ANW) and 0.0 (QZ) from TCPw, −0.6 from TCPx, −0.6 from TCPy, 0.0 from TCPz	−0.6 from eth0
eth0	ACT	−0.60 (good)	−0.6 from 10.0.1.43	N/A

Table 5: Example network stack trace, inbound results only. The last two columns are added for explanatory purposes. Socket fd 24 is blamed for not consuming (reading), and TCPw is blamed for not producing for waiting socket fd 2.

which strongly blames the producer TCPw and completely absolves the consumer socket. Cell $DT_{IN}[\text{QZ}_{\texttt{TCPw}}][\text{WAIT}_{\texttt{fd 2}}]\#P$ adds additional blame ($\lhd = +0.6$) to producer TCPw. Using the evidence already provided by its comparison with webkit, socket fd 2's final diagnosis is -1.0 (good).

Both fd 5 and fd 7 are receiving messages from their underlying TCP connections TCPx and TCPy. They have each already received ($\blacktriangleleft = -0.6$) from webkit, and receive the same from each of their active TCP connections. This results in a diagnosis of -0.84(good) for each of them.

The final socket, fd 24 is IDLE. It has already been completely blamed by its parent (webkit). Since its child is set to IGN, no additional update is made. Socket fd 24 is left with a diagnosis of $+1.0$ (idle).

Now looking at the TCP connections as consumers, TCPw has already received two pieces of blame from its waiting socket fd 2. Since the connection is potentially unresponsive (ANW) and its producer child 10.0.1.43 is active, it receives additional blame from

$$DT_{IN}[\text{ACT}_{\texttt{10.0.1.43}}][\text{ANW}_{\texttt{TCPw}}] = (\circ, \lhd) = (+0.4, +0.6)$$

which also partially blames the lower layer ($+0.4$) for the potential stall. Overall, TCPw receives a diagnosis of $(+0.6 \oplus +0.6 \oplus +0.6) = +0.94$(idle).

Finally, comparing 10.0.1.43 and eth0, both have emitted messages to their upper layers, applying ($\blacktriangleleft = -0.6$) to both modules. Combining all of the accumulated evidence results in the diagnosis shown in Table 5.

4 Implementation

To apply NeST to end-host performance monitoring, we built a GNU/Linux prototype having the architecture depicted in Figure 5. The implementation has three subsystems (shown in dark gray in the figure):

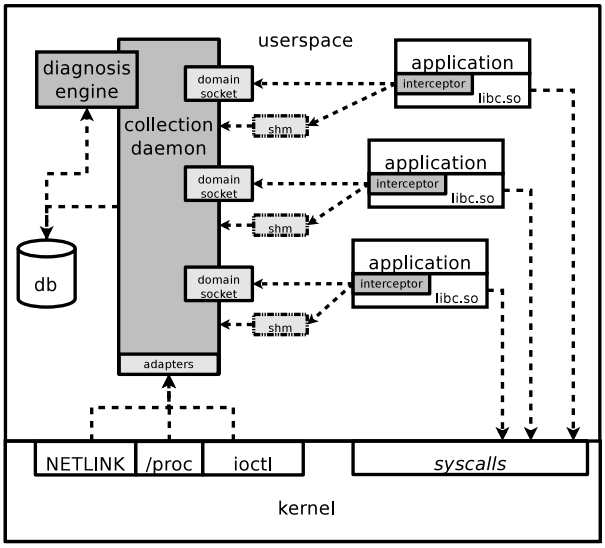


Figure 5: Network Stack Trace collection architecture

- a *collection daemon* that maintains the network stack graph and tracks the behavior of each network module by taking periodic snapshots of counters,
- an *interceptor* library that tracks application counters and provides them to the collector, and
- the *diagnosis engine* which analyzes the snapshots according to diagnosis algorithm.

We use ioctl(2), proc(5), and netlink(7) to access performance and configuration information for the network stack. Since each interface exports this information differently, we wrote simple adapters to read the

various protocol-specific counters and present them in our common format shown in Table 1. Unfortunately, the IP-level counters exported in `/proc/net/snmp[6]` count the IP packets processed by the host on *all* its interfaces. Since these counters are the only IP-level counters readily available, in our experiments we combined the IP and Ethernet interface modules together and use the counters exported by the Ethernet interface. The adapters also track the relationships between processes, sockets, and connections, so the collector can construct the stack graph and adapt it over time as entities are established and torn down.

To track application statistics, we developed an interceptor library which proxies socket-related calls to `libc` (redirected to it using LD_PRELOAD). This library tracks the number of calls made (for `total_msgs`) and the time spent in each call (for `wait_msec`) and writes the counts to a shared memory (shm) segment. As applications and sockets are created and closed, the library sends one-way messages to the collection daemon's Unix domain socket to notify the daemon of their existence and configuration. For applications this includes the process name, ID, and command line, and for sockets the file descriptor, socket type, endpoint addresses and ports, and connection status. The library also tells the daemon where to read each socket's statistics from in the shared memory area, so the daemon can monitor the socket counters without interfering with the application; application counters are simply the sum of all the application's socket counters.

We currently snapshot counters every 100 ms, which approximates Internet round trip times [9] and matches the default interval for WiFi beacons. Collecting at this rate makes it possible to detect short stalls without excessive overhead. While the diagnosis engine can be run in real-time (e.g., each time a snapshot is made) our implementation does not yet do this. Instead, we store the collected snapshots and post-process the counters to create the diagnosis and evaluate the results. This also allows us to experiment with different rule sets as discussed in the next section.

5 Evaluation

We used NeST to diagnose injected faults in a controlled setting. We found that NeST correctly diagnosed the faulty module more than 99% of the time, while incorrectly blaming modules only 3% of the time.

5.1 Experimental setup

We ran our experiments on Emulab [19] using the topology shown in Figure 6, varying the types of applications running (download-only, upload-only, or simultaneous upload/download) and number of simultaneous connections. Each series of tests included a control group running normally, plus experiments with randomly injected faults targeting the network and applications. These faults include:

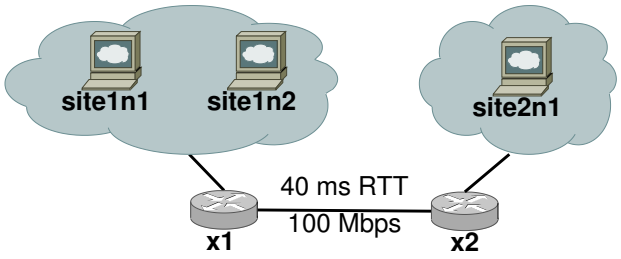


Figure 6: Emulab experiment topology. Two sites are connected by a single 100 Mbps bottleneck link between routers `x1` and `x2`. We inject faults by pausing applications, and dropping all packets on specific TCP connections and IP flows on router `x1`.

1. Pausing the application to force it to stop reading and writing from network sockets
2. Dropping packets on certain TCP connections
3. Dropping packets on certain IP-to-IP flows

In our unloaded test network, a module should be stalled (and receive a positive diagnosis) only when it is targeted for a fault; this is a *gold standard* positive result. Otherwise the gold standard diagnosis should be 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 an unresponsive (positive) outbound diagnosis; conversely upload sockets should have an unresponsive (positive) inbound diagnosis.[2] The same holds for the applications themselves.

5.2 Diagnosis Results

Figure 7 shows portions of the diagnosis timelines for sample applications with specific injected faults. Each line in the plot shows the inbound or outbound diagnosis value for a single module; the x-axis is the experiment time, and the y-axis the level of certainty that the module is unresponsive (positive values) or active/waiting (negative values). Values in the grey area below zero show good performance, while values in the white area above zero indicate modules having trouble. We tabulate these results in the next subsection.

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

In Figure 7(a), a network fault is repeatedly injected that drops all IP packets between the local host and the remote server; this causes all of the TCP connections from the remote host to stop sending, so the monitored host receives no packets. Each of the twelve fault-injection periods are marked with light blue vertical shading. When more than one TCP connection is blocked, their observations combine to blame the `ipv4/eth` module, as seen when the dotted green line goes above zero (annotated with "ipv4/eth blamed"). Their shared co-waiting status (many parents waiting on one child) causes each TCP connection to be absolved by the second pass of the diagnosis algorithm (Section 3.2).

Note that the problem periods may continue long after the injected fault is removed, as seen in time period t571.4–t572.9: the remote sending TCP modules have hit retransmission timeouts and are delaying transmissions far longer than the actual problem exists, which is not uncommon. As each TCP connection is able to get packets through the network, the TCP modules recover (and are not blamed since they are now active) and `ipv4/eth` is blamed less (t572.9), and the TCP modules that do not recover are blamed instead (TCP#344 @ t572.9, TCP#343 @ t573.0, and TCP#341 @ t574.7).

Figure 7(b) demonstrates NeST's ability to detect problems on particular TCP connections. In this case, a `wget` [18] download is running, with three additional incoming connections in the background. A fault is injected to drop all packets on the `wget` download (TCP connection #471), and the connection is blamed for not providing data to the socket (#470) until it recovers. Since there are additional active connections during this time, NeST is able to accurately pinpoint the source of the stall—congestion on a particular TCP connection.

An example of a malfunctioning application is shown in Figure 7(c). A modified instance of `iperf` [13] is running two upload connections to the remote server. Just before t368.5, we signal the `iperf` process to stop reading and writing from its network sockets. As it stops writing to socket#175, that socket is blamed for being IDLE, but the application is considered good overall since it is still writing to socket#175. At t368.8, it stops writing to its second socket, and both sockets and the application are blamed for being IDLE. Once the application resumes writing to its sockets again at t369.2, the traffic returns to normal and no modules are blamed.

As seen in Figure 7(c), there may be a delay between the initial fault injection and its impact on the targeted module. Additionally, once the fault is removed, the impacted module may require an indeterminate amount of time to recover (Figure 7(b)). This is often due to retransmission backoff caused by repeated timeouts.

To determine when a module is actually impacted by the injected fault, once a fault is injected we watch for the module's packet and wait period counters to both become zero (corresponding to observations IDLE and NONE). An *impact period* ends whenever the counters increase again, usually due to local retransmissions. An injected fault may cause more than one impact period if the counters return to zero while the fault is still active.

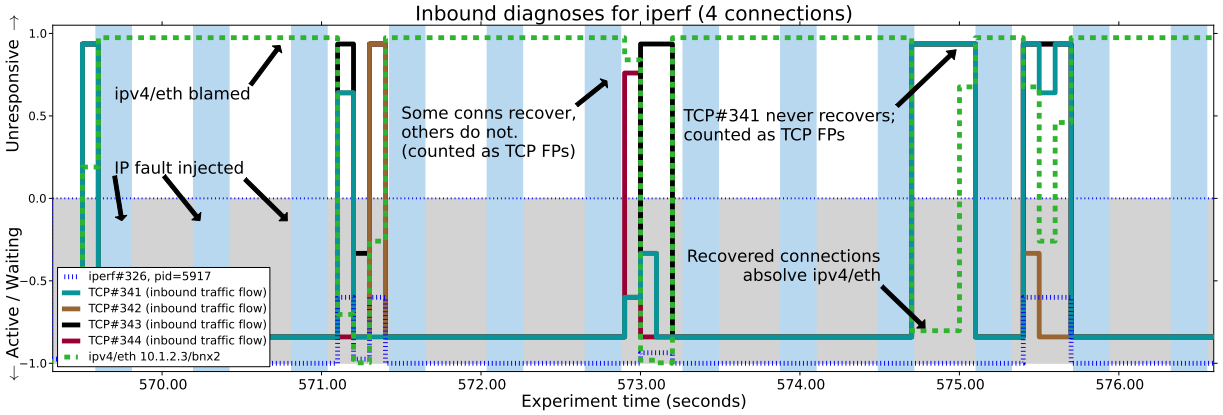
	Layer	Module	Positive Accuracy TPR	FPR	PPV	Negative Accuracy TNR	FNR	NPV
Inbound Diagnoses	6	iperf	99.4	0.5	99.6	99.5	0.6	99.3
	6	wget	100.0	0.0	99.6	100.0	0.0	100.0
	5	socket	99.9	0.1	99.9	99.9	0.1	99.9
		s:iperf	99.9	0.2	99.9	99.8	0.1	99.8
		s:wget	100.0	0.0	99.6	100.0	0.0	100.0
	4	TCP	89.6	3.0	48.6	97.0	10.4	99.7
	2+3	ipv4/eth	99.2	8.3	55.8	91.7	0.8	99.9
	all modules		**99.1**	**2.4**	**90.8**	**97.6**	**0.9**	**99.8**
Outbound Diagnoses	6	iperf	100.0	0.0	100.0	100.0	0.0	100.0
	6	wget	99.8		100.0		0.2	0.0
	5	socket	99.9	0.0	100.0	100.0	0.1	99.8
		s:iperf	100.0	0.0	100.0	100.0	0.0	100.0
		s:wget	99.8		100.0		0.2	0.0
	4	TCP	100.0	2.7	40.3	97.3	0.0	100.0
	2+3	ipv4/eth	98.1	3.6	62.5	96.4	1.9	99.9
	all modules		**99.8**	**2.2**	**95.9**	**97.8**	**0.2**	**99.9**
All Diagnoses	6	iperf	99.4	0.2	99.6	99.8	0.6	99.8
	6	wget	99.8	0.0	100.0	100.0	0.2	99.8
	5	socket	99.9	0.1	100.0	99.9	0.1	99.9
		s:iperf	99.9	0.1	99.9	99.9	0.1	99.9
		s:wget	99.8	0.0	100.0	100.0	0.2	99.8
	4	TCP	93.4	2.9	44.7	97.1	6.6	99.8
	2+3	ipv4/eth	98.8	5.9	58.2	94.1	1.2	99.9
	all modules		**99.6**	**2.3**	**94.0**	**97.7**	**0.4**	**99.8**

Table 6: Diagnosis accuracy from microbenchmarks; columns are defined in Table 7. Top section is for inbound flows, middle section for outbound flows, and combined results at the bottom.

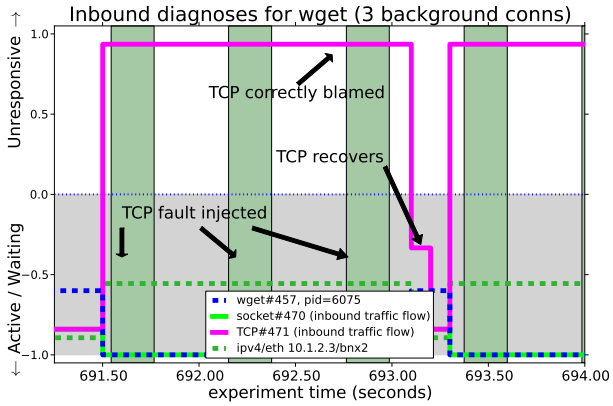
5.3 Diagnosis Accuracy

We now consider the ability of our diagnosis system to detect these injected faults. As mentioned previously, the gold standard result is positive 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 *impacted* by an injected fault. In all other cases, the gold standard result is negative; in general, each module should receive a clean bill of health. Table 6 shows the accuracy of our diagnosis algorithm using these criteria.

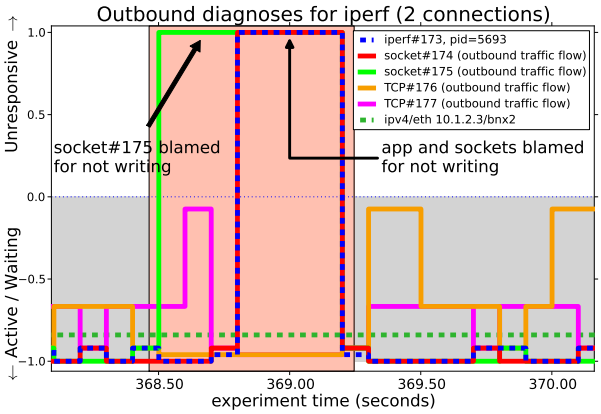
NeST is able to accurately detect faults in applications or individual sockets in well over 99% of the cases, with few False Positives. This is expected, since the `wait_msec` counter allows us to observe their state directly. At first glance, the IP/Eth module and TCP modules appear to have much worse results: While we ac-



(a) Inbound diagnosis with faults injected (blue areas) on the IP-to-IP flow; some modules are omitted. The IPv4/Ethernet module is correctly blamed for problems while multiple TCP connections are affected. Once some of the connections recover, IPv4/Ethernet is absolved (diagnosis < 0). Sometimes individual TCPs connections do not recover, and they are blamed individually.



(b) Inbound diagnosis with fault injected (light green areas) to drop all packets on TCP connection #471. Initially everything is performing well, and no modules are blamed. The fault is injected and the TCP module is blamed. Three additional connections are running in the background (not shown).



(c) Outbound diagnosis with fault injected (red area) to pause application from reading/writing. Socket#174 stops writing first, and is blamed; when both sockets stop writing the application is idle, and the application and sockets are blamed.

Figure 7: Diagnosis timelines for specific injected faults.

curately detect almost all of the gold standard positives, we have a significant False Positive Rate (FPR) for inbound IP/Eth traffic (8.3%), and appear to misdiagnose 10.4% of the inbound TCP faults. Likewise, we detect 100% of outbound TCP connections' problems, but we have a significant number of False Positives, leading to a low Positive Predictive Value (PPV)—according to our gold standard evaluation, when our diagnosis blames an outbound TCP connection, it is correct only 40.3% of the time.

The main reason for this apparent lack of precision is that our our evaluation criteria are overly conservative. There are two main cases which appear in our experiments: when there is a single active flow, and when multiple flows experience congestion.

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 in Section 2.3, in such single-flow situations our diagnosis blames both the TCP and IP/Eth modules. These are counted as IP/Eth False Positives when the TCP module is targeted, and TCP False Positives when the IP/Eth module is targeted.

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 as each TCP connection blames the lower layer. When this occurs during a TCP fault injection period, it is counted as a TCP False Negative since TCP is absolved by its peers.

Abbr.	Name	Explanation
TPR	True-Positive Rate (Sensitivity)	What % of the actual positives were diagnosed correctly?
FPR	False-Positive Rate	What % of the actual negatives were False Positives?
PPV	Positive Predictive Value (Precision)	When the diagnosis is positive, what % of the time is it correct?
TNR	True-Negative Rate (Specificity)	What % of the actual negatives were diagnosed correctly?
FNR	False-Negative Rate	What % of the actual positives were False Negatives?
NPV	Negative Predictive Value	When the diagnosis is negative, what % of the time is it correct?

Table 7: Abbreviations for evaluation tables.

5.4 Significance of various features

To test the importance of the various diagnosis features described in Section 3, we selectively disabled some of the features and compared the total number of True Positive and False Positive diagnoses against the full-feature results presented in Table 6.

Features we modified include:

- Replacing ANW state with NONE
- Replacing IGN state with NONE
- Replacing both ANW and IGN with NONE
- The QZ and QNZ observations
- The ERR observation

Results of this analysis show that our current feature set is the best we have considered, with the possible exception of the use of errors to indicate unresponsive modules.

The results are presented in Table 8. If removing the feature improved the results, more True Positives and/or fewer False Positives occur, and the percentage improvement is highlighted in **bold font**. If removing the feature reduced accuracy, there are fewer True Positives and/or more False Positives, and the percentage decline is shown in *italic font*. Since application and socket activity states are observed directly, the change in features made no difference in their overall diagnosis results, so their rows are omitted.

Removing the ANW and IGN states selectively tests the effectiveness of the NONE rules, specifically the DT_*[NONE][NONE] cells which blame both the producer and consumer a small amount (○).

Removing the ANW state results in fewer positive diagnoses overall for `ipv4/eth`, which hurts its ability to properly blame `ipv4/eth`, detecting 67.4% fewer of the faults in inbound flows, but with 40.5% fewer False Positives than the full feature set. On the other hand, TCP is blamed significantly more often. This follows the premise that we use ANW to propagate blame to the lower layers of the stack (Principle 4), and removing it is a significant setback.

Removing the IGN state has an odd effect—we miss more of the positive diagnoses (fewer TP), but also have many more False Positives. The lack of IGN blames quiet modules even when no applications are active, but the weak CF ○applied by the DT_*[NONE][NONE] cells fail to blame them strongly enough at other times.

Removing both ANW and IGN improves the true positive rate overall, at the cost of many more false positives. The increase in True Positives is largely due to blaming TCP even when its lower layer is also inactive (NONE), and again the NONE rules apply blame imprecisely. Overall, these experiments indicate that some rewriting the NONE state with ANW and/or IGN is needed to reliably diagnose faults at lower layer of the protocol stack. The NONE rules are not accurate enough at accounting for the overall system state (specifically, whether applications and sockets are waiting or idle), which misdiagnoses lower layers of the stack.

If we were to discard the observations of a module's queue, we decrease the overall number of positive diagnoses. This is not unexpected, since the majority of CFs assigned by the QZ and QNZ observations are positive. It appears that we have a discrepancy in our outbound rules which blames TCP too often (35.6% FP reduction for TCP OUT), but even so, when this rule is missing too much blame is applied to `ipv4/eth`, which is blamed 20.7% more often. So, we find that tracking queues is necessary to place blame correctly, even if our rules have not implemented it perfectly.

For the limited evaluation we performed, it appears that tracking the presence of errors (ERR) independently of the activity state does not improve the diagnosis. We expect that accounting for errors will be much more significant when diagnosing connections over wireless networks, which we leave to future work.

6 Related work

Many systems have been proposed for diagnosing some aspect of network or software performance [1, 2, 8, 10, 15–17], some geared toward software analysis [10,17] or specific network protocols [6, 8, 15, 16], 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 [6]. 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.

	Feature removed:		ANW		IGN		ANW,IGN		QZ,QNZ		ERR	
	Layer /	Module	ΔTP	ΔFP	ΔTP	ΔFP	ΔTP	ΔFP	ΔTP	ΔFP	ΔTP	ΔFP
IN	4	TCP	**11.6**	*135.9*	*-11.0*	*296.2*	**11.6**	*330.6*	*-43.8*	**-28.8**	·	**-3.5**
	2+3	ipv4/eth	*-67.4*	**-40.5**	*-40.8*	*188.2*	**0.8**	*200.8*	·	·	·	**-0.9**
	all modules		*-2.5*	*69.3*	*-2.6*	*252.4*	**0.7**	*278.3*	*-2.6*	**-17.8**	·	**-2.5**
OUT	4	TCP	·	*160.7*	·	*425.6*	·	*462.0*	·	**-35.6**	·	**-3.6**
	2+3	ipv4/eth	*-76.9*	**-51.9**	*-41.2*	*601.5*	**2.0**	*632.0*	·	*20.7*	·	·
	all modules		*-1.2*	*111.7*	*-0.7*	*466.2*	**0.0**	*501.2*	·	**-22.7**	·	**-2.8**
ALL	4	TCP	**7.1**	*147.7*	*-6.7*	*357.6*	**7.1**	*393.0*	*-26.6*	**-32.1**	·	**-3.5**
	2+3	ipv4/eth	*-70.9*	**-44.1**	*-40.9*	*317.8*	**1.3**	*336.0*	·	*6.5*	·	**-0.6**
	all modules		*-1.7*	*87.1*	*-1.4*	*342.4*	**0.3**	*372.1*	*-0.9*	**-19.9**	·	**-2.6**

Table 8: Which features are necessary? Each pair of columns shows the percentage change in total numbers of True Positives (TP) and False Positives (FP) when removing a given feature vs. using the full feature set. All **good changes** are shown in **bold font**, and *bad changes* in *italics*. A dot (·) indicates "no change."

CONMan [3, 4] 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 network management, and it relies on a centralized *Configuration Manager* to track modules throughout the network. While the Configuration Manager is able to diagnose some network *faults* by tracking a subset of the NeST counters across all the modules in the network [4], NeST is able to reliably diagnose temporary performance *degradation*, a much more difficult goal. In contrast to the centralized, cooperative scheme described in CONMan, NeST is entirely 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.

Another system that shares our goal of automated network diagnostics is NetPrints [1], which can 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, one-way traffic without responses, and lack of any inbound/outbound traffic—that may indicate certain types of network faults. These network features and configurations are combined to detect and potentially repair misconfigurations on the end host, firewall, or router. Like CONMan, NetPrints relies on a separate diagnostic server to collect and analyze information provided by end hosts, which is problematic during periods of extremely poor network performance.

The use of dependency graphs is common in network and systems performance analysis [1–3, 7, 14]. As mentioned previously, CONMan creates a dependency graph much like the one used in NeST, with individual network protocols making up the nodes of the graph.

Sherlock [2] 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 [7] 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 NeST—discovery of inter-system dependencies across an enterprise network, and in the case of Sherlock diagnosis of service and network delay spikes and failures. Because of its use of the interceptor library, NeST 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.

NetMedic [14] 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 NeST 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 the all of 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.

NetLogger [10] and Pip [17] 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, but are limited to those calls that have been instrumented. They also provide only a coarse view of network delays, based upon matching the request/response pairs, and don't provide insight into the network behavior itself. The expense of adding this instrumentation to application source code is also a high barrier to entry.

NeST is perhaps most inspired by the Web100 project [16], 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. The Pathdiag tool [15] uses Web100 instrumentation to detect common network performance problems along a path, including host misconfigurations, large router queue sizes, and excessive loss. Both are limited to diagnosing behavior from the TCP sender side.

7 Conclusion

This paper has presented NeST, the Network Stack Trace. By observing a handful performance counters, considering the shape of the network stack as a whole, and following a few principles of producer/consumer diagnosis, any host can independently determine which connections, applications, or interfaces are causing problems, and take steps to troubleshoot further or work around it. Experimental results show that NeST is over 99% accurate in detecting network stalls, with a false positive rate of less than 3%. We believe NeST represents an important step forward for automated network performance diagnosis; it supports independent end-host diagnosis, which is useful by itself, and can form the basis of richer distributed system performance diagnosis, a topic we plan to consider in future work.

References

[1] AGGARWAL, B., ET AL. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI* (Apr 2009).

[2] BAHL, P., ET AL. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM* (2007), ACM, pp. 13–24.

[3] BALLANI, H., AND FRANCIS, P. CONMan: a step towards network manageability. In *SIGCOMM* (2007).

[4] BALLANI, H., AND FRANCIS, P. Fault management using the CONMan abstraction. In *INFOCOM* (2009).

[5] BUCHANAN, B. G., AND SHORTLIFFE, E. H. *Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Boston, MA, 1984.

[6] CHANDRA, R., PADMANABHAN, V. N., AND ZHANG, M. WiFiProfiler: cooperative diagnosis in wireless LANs. In *MobiSys* (2006), ACM, pp. 205–219.

[7] CHEN, X., ET AL. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI* (2008), USENIX Association.

[8] CHENG, Y.-C., ET AL. Automating cross-layer diagnosis of enterprise wireless networks. In *SIGCOMM* (2007).

[9] DABEK, F., ET AL. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM* (2004).

[10] GUNTER, D., ET AL. Netlogger: A toolkit for distributed system performance analysis. In *IEEE MASCOTS* (2000).

[11] HECKERMAN, D. E., AND HORVITZ, E. J. On the expressiveness of rule-based systems for reasoning with uncertainty. AAAI'87, AAAI Press, pp. 121–126.

[12] HECKERMAN, D. E., AND SHORTLIFFE, E. H. From certainty factors to belief networks. *Artificial Intelligence in Medicine 4*, 1 (1992), 35–52.

[13] Iperf 2.0.5. `http://iperf.sourceforge.net/`.

[14] KANDULA, S., ET AL. Detailed diagnosis in enterprise networks. In *SIGCOMM* (2009), pp. 243–254.

[15] MATHIS, M., ET AL. Pathdiag: Automated TCP diagnosis. In *PAM* (Apr 2008).

[16] MATHIS, M., HEFFNER, J., AND REDDY, R. Web100: Extended TCP instrumentation for research, education and diagnosis. In *SIGCOMM* (2003), pp. 69–79.

[17] REYNOLDS, P., ET AL. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006), pp. 115–128.

[18] GNU wget 1.12. `http://www.gnu.org/s/wget/`.

[19] WHITE, B., ET AL. An integrated experimental environment for distributed systems and networks. In *OSDI* (Dec. 2002), USENIX ATC, pp. 255–270.