Detailed Diagnosis in Enterprise Networks

Srikanth Kandula Ratul Mahajan Patrick Verkaik (UCSD)
Sharad Agarwal Jitendra Padhye Paramvir Bahl
Microsoft Research

ABSTRACT

By studying trouble tickets from small enterprise networks, we conclude that their operators need *detailed* fault diagnosis. That is, the diagnostic system should be able to diagnose not only generic faults (e.g., performance-related) but also application specific faults (e.g., error codes). It should also identify culprits at a fine granularity such as a process or firewall configuration. We build a system, called NetMedic, that enables detailed diagnosis by harnessing the rich information exposed by modern operating systems and applications. It formulates detailed diagnosis as an inference problem that more faithfully captures the behaviors and interactions of fine-grained network components such as processes. The primary challenge in solving this problem is inferring when a component might be impacting another. Our solution is based on an intuitive technique that uses the joint behavior of two components in the past to estimate the likelihood of them impacting one another in the present. We find that our deployed prototype is effective at diagnosing faults that we inject in a live environment. The faulty component is correctly identified as the most likely culprit in 80% of the cases and is almost always in the list of top five culprits.

Categories and Subject Descriptors
C.4 [Performance of systems] Reliability, availability, serviceability
General Terms
Algorithms, design, management, performance, reliability
Keywords
Enterprise networks, applications, fault diagnosis

1. INTRODUCTION

Diagnosing problems in computer networks is frustrating. Modern networks have many components that interact in complex ways. Configuration changes in seemingly unrelated files, resource hogs elsewhere in the network, and even software upgrades can ruin what worked perfectly yesterday. Thus, the development of tools to help operators diagnose faults has been the subject of much research and commercial activity [2, 4, 5, 6, 11, 12, 17, 21].

Because little is known about faults inside small enterprise networks, we conduct a detailed study of these environments. We reach a surprising conclusion. As we explain below, existing diagnostic systems, designed with large, complex networks in mind, fall short at helping the operators of small networks.

Our study is based on trouble tickets that describe problems reported by the operators of small enterprise networks. We observe that most problems in this environment concern application specific issues such as certain features not working or servers returning error codes. Generic problems related to performance or reachability are in a minority. The culprits underlying these faults range from bad application or firewall configuration to software and driver bugs.

We conclude that detailed diagnosis is required to help these operators. That is, the diagnostic system should be capable of observing both generic as well as application-specific faults and of identifying culprits at the granularity of processes and configuration entries. Diagnosis at the granularity of machines is not very useful. Operators often already know which machine is faulty. They want to know what is amiss in more detail.

Existing diagnostic systems fall short because they either lack detail or require extensive domain knowledge. The systems for large enterprises, such as Sherlock [2], target only performance and reachability issues and diagnose at the granularity of machines. They essentially sacrifice detail in order to scale. Other systems, such as Pinpoint for online services [5] and SCORE for ISP networks [17], use extensive knowledge of the structure of their domains. Extending them to perform detailed diagnosis in enterprise networks would require embedding detailed knowledge of each application's dependencies and failure modes. The range and complexity of applications inside modern enterprises makes this task intractable.

Can detailed diagnosis be enabled with little application specific knowledge? By developing a system called NetMedic, we show that the answer is yes. The two keys to our solution are: *i*) framing detailed diagnosis as an inference problem that is much richer than current formulations [2, 5, 17, 32]; and *ii*) a novel technique to estimate when two entities in the network are impacting each other without programmed knowledge of how they interact.

Our formulation models the network as a dependency graph of fine-grained components such as processes and firewall configuration. While dependency graphs have been used previously [2, 5, 17, 32], our formulation is different. One difference is that it captures the state of a network component using many variables rather than a single, abstract variable that denotes overall health. Different variables capture different aspects of component behavior. For instance, the variables for a process may include its resource consumption, response time for its queries, and application-specific aspects such as fraction of responses with error codes. Another difference is that our formulation allows for components impacting each other in complex ways depending on their state; existing formulations assume that faulty components hurt dependent components irrespective of the nature of the failure. These differences are necessary for observing and diagnosing a rich set of failure modes. For instance, whether or not a faulty process hurts other processes on the same machine depends on its resource consumption. For correct diagnosis, the model must capture the behavior of the process in detail as well as allow for both possibilities.

The goal of diagnosis in our model is to link affected components to components that are likely culprits, through a chain of dependency edges. The basic primitive required is inferring the likelihood that the source component of a dependency edge is impacting the destination. This inference is challenging because components interact in complex ways. And because we want to be application agnostic, we cannot rely on knowing the semantics of individual state variables.

Our insight is to use the joint behavior of the components in the past to estimate impact in the present. We search in the history of component states for time periods where the source component's state is "similar" to its current state. If during those periods the destination component is often in a state similar to its current state, the chances are that it is currently being impacted by the source component. If not, it is likely that the source component in its current state is not impacting the destination component.

Our system, NetMedic, builds on this insight to identify likely culprits behind observed abnormal behaviors in the network. The

SIGCOMM'09, August 17–21, 2009, Barcelona, Spain.

	Observed symptom	Identified cause
1	The browser saw error codes when accessing some of the pages on the Web server even though they had correct permissions.	A software update had changed the Web server's configuration. In the new configuration, it was not correctly processing some required scripts. The operator was aware of the update but not of the configuration change.
2	An application was observing intermittently high response times to its server.	An unrelated process on the server's machine was intermittently consuming a lot of memory.
3	Some of the clients were unable to access a specific feature of a Web-based application.	The firewall configuration on a router along the path was blocking https traffic that was required for that feature. The operator did not know when or how the firewall configuration had changed.
4	The mail client (Outlook) was not showing up-to-date calendar information.	A remote folder on the client machine was unmounted during a defragmentation operation. The operator did not know that defragmentation could lead to the unmounting of a remote folder.
5	None of the clients in the network could send email.	The configuration of the client was overriden with incorrect mail server type. The probable cause of the change was a bug in the client software that was triggered by an overnight update.
6	Database server refused to start.	The server was misconfigured. The operator did not know how that happened.
7	An application client was getting RPC errors when contacting the server.	A low-level service (IPSec) on the client machine was intercepting application traffic. The operator did not know how the service got turned on or that it could interfere with the application.
8	The clients were experiencing poor performance to a database server.	Another client was generating too many requests.
9	The network latency between hosts was high.	A buggy process was broadcasting UDP packets at a high rate.
10	The database server was returning errors to a subset of the clients.	A port that was being used by the problematic clients had been blocked by a change in firewall configuration on the server machine. The operator was not aware of the configuration change.

Table 1. Example problems in our logs.

rich information on component states needed for detailed diagnosis is already exported by modern operating systems and applications [20, 25]. NetMedic takes as input simple templates (e.g., a machine depends on all active processes) to automatically build the dependency graph amongst components. It implements history-based reasoning in a way that is robust to idiosyncrasies of real-world data. It uses statistical abnormality detection as a pruning step to avoid being misguided by components that have not changed appreciably. And it uses simple learning techniques to extract enough relevant information about state variables to compete favorably with a scheme that uses domain knowledge.

We evaluate our approach by deploying NetMedic in two environments, including a live environment with actively used desktops. In this environment, NetMedic built a dependence graph with roughly 1000 components and 3600 edges, with each component populated by roughly 35 state variables. By injecting faults drawn from our trouble tickets, which comprise both fail-stop and performance problems, we find that in almost all cases NetMedic places the faulty component in the list of top five causes. In 80% of them, the faulty component is the top identified cause. Compared to a diagnostic method based on current formulations, this ability represents a five-fold improvement. We show that NetMedic is more effective because its history-based technique correctly identifies many situations where the components are not impacting each other. Additionally, this ability requires only a modest amount of history (30-60 minutes).

2. PROBLEMS IN SMALL ENTERPRISES

To understand what plagues small enterprise networks, we analyze trouble ticket logs from an organization that provides technical support for such networks. The logs indicate that the network sizes vary from a few to a few hundred computers. To our knowledge, ours is the first study of faults in such networks.

Our logs span an entire month (Feb '08) and contain 450K cases. A case documents the information related to a problem, mainly as a free form description of oral or electronic conversation between the operator of the small enterprise network and the support personnel. Most cases span multiple conversations and describe the problem symptoms, impact, and the culprit if identified. A case also contains other information such as when the network was behaving normally and any recent changes that in the operator's knowledge may have resulted in the abnormality.

Since the logs contain only faults for which operators contacted an external support organization, they may not be representative of all problems that occur. They are likely biased towards those that operators struggle to diagnose independently and need help with.

We randomly selected 0.1% of the cases and read them manually. We decided to read the cases to get detailed insights into the nature of the problems and also because the unstructured nature of the logs defied our attempts at automatic classification. We discarded cases that were incomplete, contained internal communication between support personnel, or contained non-faults such as forgotten passwords. Our analysis is based on the remaining 148 cases. While these cases represents a small fraction of the total, we find that the resulting classification is consistent even when we use only a randomly selected half of these cases.

We first describe example cases from our logs and then provide a broader classification of all that we read.

2.1 Example problems

Table 1 shows ten problems in our logs that we find interesting. Our intent is to provide concrete descriptions of a diverse set of problems rather than being quantitatively representative. We see that the range of symptoms is large and consists of application-specific errors as well as performance and reachability issues. The range of underlying causes is large as well and consists of bugs, configuration changes, overload, and side-effects of planned activities.

While it may be straightforward to design point solutions to each of these problems, it is challenging to design a comprehensive system that covers all of them. The design and implementation of such a system is a goal of our work.

2.2 Classification results

Table 2 classifies the cases that we read along three dimensions to understand the demands on a diagnostic system—the fault symptoms that it should detect and the culprits that it should identify.

The first dimension captures whether the fault impacted an individual application or the entire machine (i.e., many applications on it). It does not relate directly to the underlying cause. For instance, the machine category includes cases where a faulty application impacted the entire machine. The data shows that most of the problem reports refer to individual applications and hence monitoring machine health alone will miss many faults. To detect these faults, a diagnostic system must monitor individual applications.

The second category is based on how the fault manifests. We see that application-specific defects account for a majority of the cases. These include conditions such as the application servers returning error codes, features not working as expected, and a high number of failed requests. The prevalence of such symptoms indicates the need to track application-specific health. Unlike the more generic symptoms, it is unclear how a diagnostic system can track application

1. *What was impacted*		
An application	125	(84.5%)
Entire machine	23	(15.5)
2. *Symptom*		
Application-specific faults	88	(59.5%)
Failed initialization	19	(12.8)
Performance	15	(10.1)
Hang or crash	15	(10.1)
Unreachability	11	(7.4)
3. *Identified cause*		
Other configuration	44	(29.7%)
Application configuration	28	(18.9)
Software bug	20	(13.5)
Driver bug	10	(6.8)
Overload	6	(4.1)
Hardware fault	3	(2.0)
Unknown	37	(25.0)

Table 2. A classification of the problems in our logs.

health without knowing application semantics or requiring help from the application. We show later how we handle this issue.

The final category shows the root causes of the faults. In 19% of the cases, the application configuration was incorrect. The biggest cause, however, was some other configuration element in the environment on which the application depends. We define other configuration quite broadly to include the lower-layer services that are running, the firewall configuration, the installed devices and device drivers etc. For 25% of the faults, the underlying cause could not be identified but recovery actions such as a reboot fixed some anyway.

Unlike other settings [10, 18, 22], it appears from the logs that in most cases incorrect configuration was not a result of mistakes on the part of the operators. Rather, configuration was overwritten by a software update or a bug without their knowledge. In many other cases, the configuration change was intentional but the operators did not realize the effects of that change.

2.3 Discussion

Statistics aside, the overall picture that emerges from the logs is that small business networks are very dynamic. They undergo frequent changes, both deliberate (e.g., installing new applications, upgrading software or hardware) as well as inadvertent (e.g., triggering of latent bugs, automatic updates). Each change impacts many components in the network, some of which may be seemingly unrelated.

Detecting individual changes is rather easy. Applications and operating systems today expose plenty of low-level information, for instance, Windows Vista exposes over 35 different aspects of a process's current behavior. However, complex interactions and unknown semantics make it hard to use this information to identify the reasons behind specific abnormalities of interest to operators.

While our study is based on small enterprise networks, we believe that the kinds of problems it reveals also plague large enterprises. Existing diagnostic systems for large enterprises such as Sherlock [2] are not capable of diagnosing such faults. In order to scale, they focus on coarser faults such as a DNS server failing completely. Our work asks whether the detailed faults that we observe in our logs can be diagnosed if scalability is not a prime concern. If our techniques can be scaled, they will benefit large enterprises as well. We discuss how to scale NetMedic in §8.

3. PROBLEM FORMULATION

We now formulate the diagnosis problem in a way that helps operators with the kinds of issues that we uncover in our logs. Our goal is to build a system that can narrow down the likely causes responsible for a wide range of faults such as poor performance, unreachability, or application specific issues. This ability is the first and perhaps the hardest aspect of troubleshooting. Once the operators have identified the true culprit using our system, they can proceed to repairing the fault. Automatic repair is not our goal in this work.

Generic variables	Application variables
% processor time	current files cached
% user time	connection attempts/sec
io data bytes/sec	files sent/sec
thread count	get requests/sec
page faults/sec	put requests/sec
page file bytes	head requests/sec
working set	not found errors/sec

Table 3. Example variables in Web server state. In all there are 28 generic and 126 application specific variables.

We want our system to have the following two properties.

1. Detail: The system should be able to diagnose both application specific and generic problems. Further, it should identify likely causes with as much specificity as possible. If a process is responsible, it should identify that process rather than the hosting machine. If application configuration is responsible, it should identify the incorrect configuration rather than simply blaming the application.

The need for detailed diagnosis clearly stands out in our logs. Most faults are application-specific. The callers often knew which machine was faulty but did not know what aspect was faulty.

2. Application agnosticism: The system should rely on minimal application specific knowledge. Enterprises run numerous applications across the board. It is intractable for a general diagnostic system to contain knowledge of every possible application.

These two properties are conflicting. How can application specific faults be detected without application knowledge? For instance, the straightforward way to detect that an application client is receiving error messages is through knowledge of the protocol. Detecting faults that are not reflected in protocol messages may require even more application knowledge. We layout the problem and explain how we reconcile these conflicting goals below.

3.1 Our inference problem

There are several approaches that one might consider to diagnose faults in a computer network. To be able to diagnose a wide range of faults, we take an inference-based approach rather than, for instance, a rule-based approach (§9). However, our goals require a richer network model than current inference models. We first describe our model and then explain how it differs from existing models.

We model the network as a dependency graph between components such as application processes, host machines, and configuration elements. There is a directed edge between two components if the source directly impacts the destination. The dependency graph may contain cycles–in particular, two components may be connected by edges in both directions. Our system automatically constructs the dependency graph.

The state of a component at any given time consists of visible and invisible parts, of which only the former is available to us. For instance, the visible state of an application process includes generic aspects such as its processor usage and some application-specific aspects. The invisible state may include values of program variables and some other application-specific aspects. Table 3 shows a subset of the variables that form the Web server process's visible state in our prototype. We represent visible state using multiple variables, each corresponding to a certain aspect of the component's behavior. The set of variables differs across components. The diagnostic system is unaware of the semantics of the variables.

Given a component whose visible state has changed relative to some period in the past, our goal is to identify the components likely responsible for the change. In other words, we want to identify the causes underlying an observed effect. Each identified causes has the properties that: *i*) its visible state changes can explain the observed effect; *ii*) its visible state changes cannot be explained by visible state changes of other components.

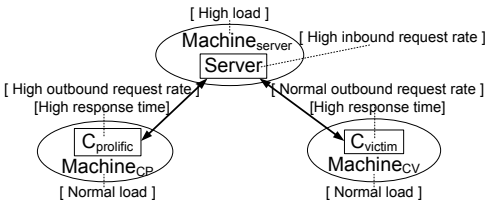


Figure 1. Illustration of Problem 8 in Table 1. The rectangles are processes and the ellipses are host machines. The relevant state of the components is shown in brackets.

For instance, consider Figure 1, which illustrates Problem 8 of Table 1. Both clients experience high response times because $C_{prolific}$ is overwhelming the server. Suppose we want to diagnose why the response time is high for C_{victim}. Although the load on Server leads to high response times, we want to identify $C_{prolific}$ as the culprit, since $C_{prolific}$ is responsible for both Server's high load and C_{victim}'s high response times, and its behavior cannot be explained by other visible factors. It may have been externally impacted, but lacking further visibility, the diagnosis will identify it as the culprit.

We do not assume that the effect being diagnosed represents a deterioration. Thus, our system can be used to explain any change, including improvements. This agnosticism towards the nature of change and the lack of knowledge of the meaning of state variables lets us diagnose application-specific behaviors without application knowledge. If applications export their current experience, e.g., number of successful and failed transactions, the system treats these experiences as part of the state of the application process and diagnoses any changes in them. We assume that the state variables are well-behaved—small changes in component behaviors lead to relatively small changes in variable values and significant behavioral changes are detectable using statistical methods. We find that this assumption holds for the state variables exported by the components in our prototype.

3.2 Limitations of existing models

Existing models [2, 5, 17] differ from our formulation in three important ways that makes them unsuitable for detailed diagnosis. First, they use a single variable to represent component health. However, if exposing and diagnosing a rich set of failure modes is desired, component state must be captured in more detail. One might be tempted to abstract away the detail and just present a faulty-or-healthy status for each component, but some types of component failures impact other components while others do not. For instance, an application process has the ability to hurt other processes on the same machine, but typically, it hurts them only when it consumes a lot of resources and not otherwise. To correctly determine if a process is impacting others, its state must be captured in more detail.

In principle, a component with multiple variables is logically equivalent to multiple components with a variable each. In practice, however, the difference is significant. Dividing a component into constituent variables forces us to consider interactions within those variables. Given the internal complexities of components and that there can be hundreds of variables, this division significantly increases the complexity of the inference problem. Further, as we will show, keeping a multi-variate component intact lets us extract useful information from the collective behavior of those variables.

Second, existing models assume a simple dependency model in which a faulty component hurts each dependent component with some probability. Turning again to the faulty process example above, we can see that whether a component impacts another depends in a more complex way on its current state.

Finally, existing models do not allow circular dependencies by which two components have a direct or indirect mutual dependence. When viewed in detail, circular dependencies are commonplace. For instance, processes that run on the same machine are mutually dependent, and so are processes that communicate.

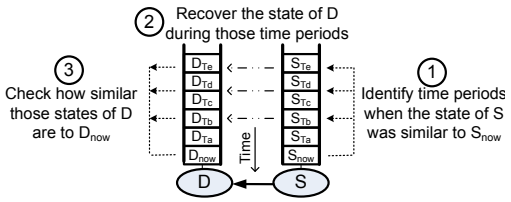


Figure 2. Computing the weight of the edge from S to D.

4. USING HISTORY TO GAUGE IMPACT

Solving our inference problem requires us to estimate when a component might be impacting another. The primary difficulty in this estimation is that we do not know *a priori* how components interact. Our lack of knowledge stems from application agnosticism. Even if we had not chosen an application-agnostic approach, it appears unrealistic to embed detailed knowledge of component interaction into the design of the diagnostic system. For instance, there is no general way to specify how network path congestion impacts application processes because the impact varies across applications.

One could use time to rule out the possibility of impact along certain dependency edges. A component that is currently behaving normally is likely not impacting one that is currently abnormal. For instance, in Figure 1, because the host machine of C_{victim} is behaving normally, we can rule it out as a possible culprit. However, time-based elimination is limited because it cannot deduce what is impacting what. Returning to the example, we see that both clients as well as Server and $Machine_{server}$ are abnormal. Time-based elimination alone cannot tell which of these might be the culprit. Instead, we must use a more precise analysis based on the states of various components.

Our level of detail makes the challenge more daunting. Component states include many variables (e.g., some applications expose over fifty variables in our implementation); it is not uncommon for at least some variables to be in an abnormal state at any time. Amidst this constant churn, we need to link observed effects to their likely causes, while ignoring unrelated contemporaneous changes and without knowing *a priori* either the meanings of various state variables or the impact relationship between components.

We address this challenge using a novel, history-based primitive. This primitive extracts information from the joint historical behavior of components to estimate the likelihood that a component is currently impacting a neighbor. We use this estimated likelihood to set edge weights in the dependency graph. The weights are then used to identify the likely causes as those that have a path of high impact edges in the dependency graph leading to the affected component.

We provide in this section the intuition underlying our history-based primitive; we explain in §5.3 how exactly it is implemented in a way that is robust to the real-world complexities of component states. In Figure 2, assume that the current state of the source component S is S_{now} and of the destination D is D_{now}. We want a rough estimate of the probability that S being in S_{now} has driven D into D_{now}. We compute this by searching through the history for periods when the state of S was "similar" to S_{now}. Informally, similarity of state is a measure of how close the values are for each variable. We quantify it in a way that does not require the knowledge of the semantics of the state variables and appropriately emphasizes the relevant aspects of the component's behavior. The edge weight is then a measure of how similar to D_{now} is the state of D in those time periods. If we do not find states similar to S_{now} in the history, a default high weight is assigned to the edge.

Intuitively, if D's state was often similar to D_{now} when S's state was similar to S_{now}, the likelihood of S being in S_{now} having driven D into D_{now} is high. Alternately, if D was often in dissimilar states, then the chances are that S_{now} does not lead to D_{now}.

This reasoning is reminiscent of probabilistic or causal inference [13, 23]. But because component states are multi-dimensional, real-valued vectors, we are not aware of a method from these fields

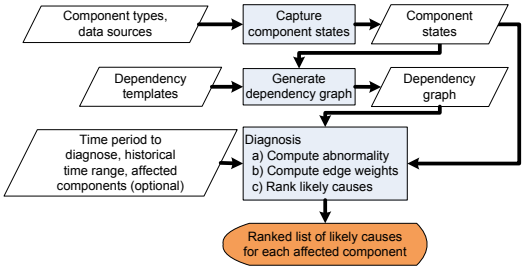


Figure 3. The work-flow of NetMedic.

that we can directly apply. Crudely, what we are computing is the conditional probability $Prob(D = D_{now}|S = S_{now})$ and assuming that it reflects causality. Conditional probability in general does not measure causality, but we find that the assumption holds frequently enough in practice to facilitate effective diagnosis. Further, we do not infer complex probabilistic models to predict the conditional probability for each pair of S-D states; such models typically require a lot of training data. Instead, we estimate the required probability on demand based on whatever historical information is available.

Consider how our use of history helps in Figure 1. The estimated impact from Server to C_{victim} will be high if in the past time periods when Server had high inbound request rate, C_{victim} had high response time along with normal outbound request rate. The estimated impact from Server to $C_{prolific}$ will be low if during those time periods, $C_{prolific}$ had normal outbound request rate. On the other hand, the estimated impact from $C_{prolific}$ to Server will be high if $C_{prolific}$ never had high outbound request rate in the past or if Server had high inbound request rate whenever it did. This way, we obtain a high impact path through Server from $C_{prolific}$ to C_{victim}, without the need for interpreting client and server state variables.

Whether the weight is correctly determined for an edge depends of course on the contents of the history. We find that estimating the correct weight for every edge is not critical. What is important for accurate diagnosis is an ability to correctly assign a low weight to enough edges such that the path from the real cause to its effects shines through. We show later that our method can accomplish this using only a modest amount of history.

5. DESIGN

The workflow of NetMedic is depicted in Figure 3. Its three main functional pieces capture the state of network components, generate the dependency graph, and diagnose based on component states and the dependency graph. We describe each piece below.

5.1 Capturing component state

There are many ways to partition a network into constituent components. Our partitioning is guided by the kinds of faults that appear in our logs—components in our current design include application processes, machine, and network paths, as well as configuration of applications, machine, and firewalls. The machine component bundles the hardware and the OS.

In addition, we also include a virtual component, called NbrSet (short for Neighbor set). A NbrSet represents the collective behavior of communication peers of a process. Its state variables represent information such as traffic exchanged and response time aggregated based on the server-side port. In the presence of redundant servers (e.g., for DNS), it helps model their collective impact on the client process. Similarly, it models the collective impact of all the clients for a server process. Using a NbrSet instead of individual dependencies allows us to model the dependencies more accurately [2].

The granularity of diagnosis is determined by the granularity of the modeled components. For instance, using the full network path as a component implies that culprits will not be identified at the level of individual switches. Our framework, however, can be extended to include finer-grained components than those in our current design.

Machine	CPU utilization, memory usage, disk usage, amount of network and other IO
Process	*Generic variables:* CPU utilization, memory usage, amount of network and other IO, response time to servers, traffic from clients *Application specific variables:* Whatever is available
NbrSet	State relevant to communication peers, e.g., inbound and outbound traffic, response time
Path	Loss rate and delay
Config	All relevant key-value pairs

Table 4. Example state variables that NetMedic captures.

NetMedic periodically captures the state of each component as a multi-variable vector. State is stored in one-minute bins. The bin size represents a trade-off—bigger bins have lower overhead of capturing component state but limit our ability to diagnose short-lived faults. The value of a variable represents some aspect of the component behavior during that time bin. The number of variables and their meanings vary across components. Table 4 shows a subset of aspects that are currently included for each component type.

A process is identified by its complete command line, rather than the process ID. Such identification ensures that across machine reboots and process restarts, process instances with the same command line (e.g., $c:\backslash mssql\backslash bin\backslash sqlservr.exe - ssqlexpress$) are considered to be the same functional component [30].

Process state is a union of two parts. The first part captures generic, application-independent aspects such as resources consumed and traffic exchanged. We maintain traffic information per port and also log which other processes this process communicates with, which is used for dependency graph generation. The second part of process state consists of application specific variables and reflects different aspects of current application experience such as fraction of failed requests, number of requests of a certain type, etc. Including it in the process state lets us diagnose application specific abnormalities without application knowledge.

We describe in §6 how various component state variables are captured, including how application-specific variables are captured without application knowledge.

5.2 Generating the dependency graph

We model the network as a dependency graph among components in which there is an edge from a component to each of its directly dependent components. We automatically generate this graph using a set of templates, one template per component type. Figure 4 shows the set of templates we have currently defined. A template has a component type in the center, surrounded by other component types that impact it directly. Edges in the real dependency graph correspond to edges in the templates. For instance, if the template for a machine shows that it depends on its processes, we introduce an edge from each of its processes to it.

The templates in Figure 4 can be easily interpreted. They show that a machine depends on its processes and its configuration. An application process depends on its configuration, its NbrSet, its host machine, and the configuration of the machine. While a process relies on other processes on the machine because of resource sharing, we do not include that dependency directly in the templates. For non-communicating processes, that dependency is indirect, mediated by the machine. We currently ignore inter-process interaction that does not involve exchanging IP packets (e.g., through shared memory). IP communication is captured using NbrSet. The NbrSet of a process depends on local and remote firewall configurations, the processes it is communicating with and the network paths. Finally, a network path between two machines depends on all machines that inject traffic into it and the amount of other traffic, that is, traffic from hosts outside the monitored network.

In our current templates, configuration components do not depend on anything else. If configuration changes explain the effect being diagnosed, we identify the configuration component as the culprit, without attempting to identify what changed the configuration. Extending NetMedic to remember what modified the configuration can enable such identification if needed [30].

We can see from the templates that the resulting dependency graphs can be quite complex with a diverse set of dependencies and many cycles, e.g., Process1 → NbrSet of Process2 → Process2 → NbrSet of Process1 → Process1. The next section describes how we perform an accurate diagnosis over this graph.

5.3 Diagnosis

Diagnosis takes as input the (one-minute) time bin to analyze and the time range to use as historical reference. This time range does not need to be contiguous or adjacent to the time bin of interest. We only assume that it is not dominated by the fault being diagnosed. For instance, if a configuration fault occurs at night but its effect is observed the next morning, NetMedic needs historical reference before the fault (e.g., the previous morning) to diagnose the effect. Optionally, the operator can also specify one or more affected components whose abnormal behavior is of interest. If left unspecified, we identify such components automatically as all that are behaving abnormally. The output of the system is a ranked list of components that are impacting each affected component of interest. There is a separate list for each affected component.

Diagnosis proceeds in three steps (Figure 3). First, we determine the extent to which various components and variables are statistically abnormal. Second, we compute weights for edges in the dependency graph. Third, we use edge weights to compute path weights and produce a ranked list of likely culprits.

5.3.1 Computing abnormality

Given historical values of a variable, we want to detect how abnormal its value is at the time of diagnosis. For purposes that will become clear later, we need a fine-grained measure of abnormality in addition to a simple binary decision as to whether a variable is abnormal. While the semantics of some variables may be known, most have application-specific, undocumented semantics. Our goal is not to craft a perfect detector but to design a simple one that works well in practice without knowing semantics before hand.

For abnormality computation, we assume that the values of the variable approximate the normal distribution. Per the central limit theorem, this is a reasonable assumption because the values of our variables tend to be sums or averages (e.g., memory usage) over the sampling time bin. If μ and σ are the variable's mean and standard deviation over the historical time range, the abnormality of value v at the time of diagnosis is $|\mathrm{erf}(\frac{v-\mu}{\sigma\sqrt{2}})|$, where $\mathrm{erf}(.)$ is the error function. The formula is double the probability of seeing values between μ and v in a normal distribution with parameters μ and σ. It ranges from 0 to 1, and the higher end of the range corresponds to values that are far from the mean, i.e., towards the tails of the normal distribution.

Given the abnormality for each variable, the abnormality of a component is the maximum abnormality across its variables.

The abnormality values computed above are used in two ways. They can be used directly, for instance, as multiplicative factors. This usage is robust to the exact method for computing abnormality as long as the first order trend of the variable values are captured such that less likely values have higher abnormality.

The abnormality values are also used to make a binary decision as to whether a variable or component is abnormal. For this decision, we use a threshold of 0.8. Like all binary decisions of abnormality, we face a trade-off between flagging a non-existent abnormality and missing a real one. We prefer the former because our edge weight computation assumes that normally behaving components do not impact others. Thus, declaring potentially abnormal components as normal is less desirable than the other way around. Our chosen threshold reflects this preference.

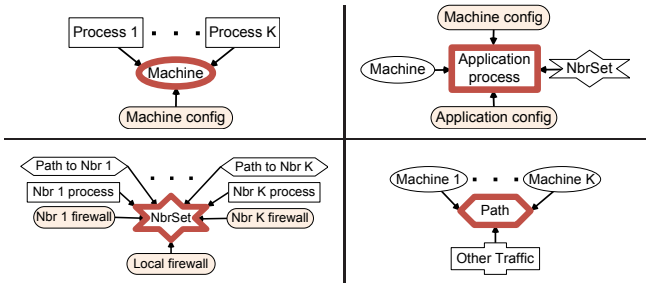


Figure 4. The templates used by NetMedic to automatically generate the dependency graph.

5.3.2 Computing edge weights

Let S and D be the source and destination of a dependency edge. If either S or D is behaving normally, it is unlikely that S is impacting D and we assign a low weight to the edge. The exact value of the edge weight is not critical in this case. However, since computing path weights involves multiplying edge weights, edge weights of zero are brittle in the face of errors. Hence, we use an edge weight of 0.1 in our experiments.

If both S and D are abnormal, we use their joint historical behavior to determine the edge weight. Let S_{now} and D_{now} be their respective states during the time bin of diagnosis. We first divide the history where both components co-exist into K equal-sized chunks, each consisting of one or more time bins. Within each chunk we identify the time bin in which S was in a state most similar to S_{now}. We then compute how similar on average D was to D_{now} during those times. More precisely:

$$E(S \rightarrow D) = \frac{\sum_{k=1}^{K}(1 - |D_{t_k} - D_{now}|) \times w_k}{\sum_{k=1}^{K} w_k}, \quad (1)$$

where t_k is the time bin in chunk k where the state of S was most similar, and $|D_{t_k} - D_{now}|$ is the difference between the two state vectors. The differencing of two states (explained below) produces a number between 0 and 1.

The term w_k is a relative weighting factor for different chunks. We specify $w_k = 1 - |S_{t_k} - S_{now}|$ if $|S_{t_k} - S_{now}| \leq \delta$; it is 0 otherwise. This specification places a higher weight on historical states that are more similar. And it excludes chunks of time where the most similar source state differs by more than δ. Because historical states that differ more already have a lower weight, the main reason for this cutoff is to avoid computing the probability based on dissimilar states alone. Our experiments use a relaxed δ of 1/3.

Dividing the history into K disjoint chunks and looking for the most similar state in each helps base the weight computation on a diverse set of time windows. Alternately, we could pick K time bins where the source state was most similar. But this method could bias results to temporally close bins that may be dependent, leading to a less effective factoring out of other aspects that impact the destination state. We find that even small values of K suffice for accurate diagnosis. We use $K = \min(10, \text{number of time bins in history})$ for experiments in this paper.

When no usable historical information exists, e.g., because of insufficient history or because similar source states do not exist, we assign a high weight of 0.8 to the edge. This assignment assumes that a fault is more likely to stem from a component that was not seen in a similar state previously. It has the desired behavior of assuming impact rather than exonerating possibly responsible components.

The basic procedure for differencing states: When computing state differences, our intent is to get a robust measure of how differently a component is behaving at different points in time. State

differences are based on differences in the values of individual variables. The difference between two state vectors with L variables is $\sum_{i=1}^{L} |d^i|/L$, where d^i is the difference of the i-th variable normalized by the observed range. That is, $d^i = (v^i_{t_k} - v^i_{now})/(v^i_{max} - v^i_{min})$, where $v^i_{t_k}$ and v^i_{now} are the values of the variable at the two time bins, and v^i_{max} and v^i_{min} are the maximum and minimum values observed across all time. Normalization means that the difference for each variable is between 0 and 1. It ensures that a variable does not dominate because its values are drawn from a bigger range.

Configuration components are handled differently for computing state differences. The difference is zero if the values of all variables are identical. It is one otherwise. For configuration components, any change in the value of even a single variable could represent a significant functional shift. We thus err on the side of deeming every such change as significant.

Robust weight assignment with unknown variable semantics: The procedure above is a starting point; while it works well in some cases, it is not robust to the presence of a large and diverse set of variables in component states. The underlying problem is that it equally emphasizes all variables, irrespective of the fault being diagnosed, the uniqueness of the information represented by that variable, or whether the variable is relevant for interaction with the neighbor under consideration. Equal emphasis on all variables dilutes state differences, which hinders diagnosis. For instance, even when a runaway process is consuming 100% of the CPU, its state may appear similar to other times if the vast majority of its state variables are unrelated to CPU usage.

If we knew variable semantics, we could pick and choose those that matter to the fault being diagnosed. We now describe extensions to the basic procedure that create a similar effect without requiring knowledge of variable semantics. The simplest of our extensions leverages the abnormality of variables and the others are based on automatically inferring the relevant properties of state variables.

a) Weigh variables by abnormality: Instead of treating the variables equally, we use abnormality of a variable as the relative weight in the state difference. This weighting biases the state difference towards variables related to the effect currently being diagnosed. For instance, while diagnosing an effect related to CPU usage, the abnormality of aspects related to CPU usage will be higher.

b) Ignore redundant variables: We ignore variables that represent redundant information with respect to other variables of the component. This extension helps prevent an over-representation of certain aspects of the component's behavior. For instance, our machines export used as well as available memory, each in units of bytes, kilobytes, and megabytes. If we include all six variables, the state differences will be biased towards memory-related aspects, making it harder to diagnose other aspects.

To discover variables that are not redundant, we want to look for independent components [14]. Instead of running a full-blown independent component analysis, we approximate via a simple heuristic that works well in our setting. We compute linear correlation between pairs of variables in the component and then identify cliques of variables such that the Pearson correlation coefficient between every pair of variables is above a threshold (0.8). We select one variable to represent each clique and deem others to be redundant.

c) Focus on variables relevant to interaction with neighbor: Among the remaining variables, we ignore those that are irrelevant to interaction with the neighbor under consideration. For instance, while considering the impact of a machine on an application process, we exclude variables for error codes that the process receives from a peer process. By reducing the noise from irrelevant variables, this exclusion makes weight assignment more robust.

We infer whether a variable is relevant to interaction with the neighbor by checking if it is correlated to any of the neighbor's variables. Specifically, we compute the linear correlation between this variable and each variable of the neighbor. We consider the variable relevant if the Pearson correlation coefficient is greater than a threshold (0.8) for any neighbor variable. Linear correlation does not capture all kinds of relationships but is easy to compute and works well for the kinds of variables that we see in practice.

The state difference for non-configuration components after applying these three extensions is $(\sum_1^L |d^i| \cdot a^i \cdot r^i)/(\sum_1^L a^i \cdot r^i)$, where L and d^i are as before and a^i is abnormality of the variable. The term r^i is a binary indicator that denotes if the i-th variable is included in the computation. It is 1 if the variable is relevant to interaction with the neighbor and represents non-redundant information.

d) Account for aggregate relationships: Some variables in machine state (e.g., CPU usage) are sums of values of process variables. Similarly, some variables in server process state (e.g., incoming traffic) are sums of values across its client processes. We discover and account for such relationships when computing state differences. The following discussion is in the context of a machine and its processes. The same procedure is used for server and its client processes.

If the variable values of different components were synchronized in time, discovering aggregate relationships would be easy. The sum of the values of appropriate process variables would be exactly the value of a machine variable. But because variables values may be sampled at different times, the sum relationship does not hold precisely. We thus use an indirect way to infer which machine variables are aggregates. We instantiate virtual variables whose values represent the sum of identically named process variables; one virtual variable is instantiated per name that is common to all processes. Even though we do not know their semantics, variables have names (e.g., "CPU usage"), and a name refers to the same behavioral aspect across processes. We then check if any machine state variable is highly correlated (with coefficient > 0.9) with a virtual variable. If so, we conclude that the machine variable is an aggregate of the corresponding process variables.

We use aggregate relationships in several ways. First, we replace the variable value in the machine with that of the virtual variable, i.e., sum of values of the corresponding process variable. Second, when computing the edge weight from a machine to its process, we subtract the contribution of the process itself. Specifically, as a pre-processing step before searching for similar machine states, the value of each aggregate variable in the machine state at each time bin is reduced by the value of its corresponding process variable. The remaining process is as before.

Such pre-processing lets us compute the state of the process's environment without its own influence. Without it, we may not find a similar machine state in history and hence falsely assign a high weight for the machine-to-process edge. Consider a case where a runaway process starts consuming 100% CPU. If such an event has not happened before, we will not find similar machine states in the history with 100% CPU usage. Instead, by discounting the impact of the process, we will likely find similar machine states and find that it is only the process that is behaving differently. These findings will correctly lead to a low weight on the machine-to-process edge.

Finally, when estimating the impact of a process on the machine, if similar process states are not found, we assign weight based on the contribution of the process. That is, we do not use the default high weight. For each aggregate variable, we compute the fraction that the process's value represents in the aggregate value. The maximum such fraction is used as the weight on the edge. This modification helps by not blaming small processes just because they are new. Arrival of new processes is common, and we do not wish to impugn such processes unless they also consume a lot of resources.

5.3.3 *Ranking likely causes*

We now describe how we use the edge weights to order likely causes. The edge weights help connect likely causes to their observed effects through a sequence of high weight edges. However, unlikely

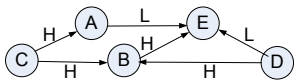


Figure 5. An example dependency graph. The labels on edges denote whether the computed weight was high (H) or low (L).

causes may also have high weight edges leading to the effects of interest.These include those that lie along paths from responsible causes but may also include others if weights on those edges overestimate the impact.

As an example, consider the dependency graph in Figure 5. For simplicity, we show whether the edge weight is high (H) or low (L) instead of numeric values. Assume that we set out to diagnose the abnormal behavior of the component labeled E and that the real culprit C is impacting it through B. Accordingly, C is connected to E through a path of high weight edges, but so are B and D (via the path D-B-E). Let us further assume that C is also hurting A and that the high weight from D to B is erroneous.

Our goal is to rank causes such that more likely culprits have lower ranks. A compact representation of our ranking function is shown in Figure 6. The rank of a component c with respect to an affected component of interest e is based on the product of two measures, and components with larger products are ranked lower. The first measure $I(c \rightarrow e)$ is the impact from c to e. The second measure $S(c)$ is a score of the global impact of c.

Together, the two measures help achieve our goal. The impact $I(c \rightarrow e)$ from one component to another is the maximum weight across all acyclic paths between them, where path weight is the geometric mean of edge weights. Per this measure, in Figure 5, B, C, and D have high impact on E but A has a low impact. The score $S(c)$ of a component is the weighted sum of its impact on each other component in the network, where the abnormality of the component is used as the weight. Components that are highly impacting more abnormal components will have a higher score. Per this measure, in Figure 5, C will have a lower rank than B and D, despite the inaccurate weight on the $D - B$ edge because it has high impact to many abnormal nodes. Of course, in any given situation whether the real culprit gets a low rank depends on the exact values of edges weights and component abnormalities. We find in our evaluation that real culprits have low ranks the vast majority of the time.

6. IMPLEMENTATION

We have implemented NetMedic on the Windows platform. Our implementation has two parts—data collection and analysis. The first part captures and stores the state of various components. The second part uses the stored data to generate the dependency graph and conduct diagnosis.

The main source of data is the Windows Performance Counter framework [20]. Using this framework, the operating system (OS) and applications export named counters and update their values. Each counter represents a different aspect of the exporter's behavior. "Performance" is a misnomer for this framework because it exposes non-performance aspects as well. The OS exports many machine-wide counters such as processor and memory usage. It also exports generic process-level aspects such as resource consumption levels. In addition, many processes export application-specific counters. See Table 3 for some counters exported by the Web server.

NetMedic reads the values of all exported counters periodically. We do not interpret what a counter represents but simply make each counter a state variable of the component to which it belongs. While most counters represent values since the last time they were read, some represent cumulative values such as the number of exceptions since the process started. We identify such counters and recover their current behavior by subtracting the values at successive readings.

The Performance Counter interface does not tell us which processes in the network are communicating with each other. We use

$$\text{Rank}(c \rightarrow e) \propto (I(c \rightarrow e) \cdot S(c))^{-1}$$

$I(c \rightarrow e) = \max(\text{weight } W(p) \text{ of acyclic paths } p \text{ from } c \text{ to } e)$

$$W(p) = \left(\prod_{j=1}^{n} E(e_j)\right)^{\frac{1}{n}} \text{ where } e_1 \cdots e_n \text{ are edges of the path,}$$

$E(\cdot)$ is edge weight

$S(c) = \sum_{e \in C} I(c \rightarrow e) \cdot A_e$ where C is set of all components,

A_e is the abnormality of e

Figure 6. Our methodology for ranking causes.

a custom utility that snoops on all socket-level read and write calls. This snooping yields the identity of the calling processes along with the IP addresses and ports being used on both ends. It lets us connect communicating processes and measure how much traffic they exchange. We also estimate response times from these socket-level events as the time difference between read and write calls. Including these response times as a variable in the process state lets us diagnose faults that delay responses even if the application does not expose this information as a counter.

We measure path loss rate and delay by sending periodic probes to machines with which a monitored machine communicates. For paths that go outside the monitored network, we measure the part up to the gateway.

NetMedic monitors machine, firewall, and application configuration stored in the Windows registry as well as files. We read all relevant information once upon start and register callbacks for future changes. Machine configuration includes information about running services, device drivers, and mounted drives. Application configuration may be spread over multiple locations. Currently, the list of locations for an application is an input to NetMedic, but we plan to automatically infer where application configuration resides using software package managers and by tracking application read calls [30].

Our data collectors are light-weight. In our deployment, the average processor usage due to data collection is under 1%. The exact usage at a given time depends on the level of activity on the machine. The amount of data transmitted for analysis is under 250 bytes per second per machine. From these overheads and our experience with data analysis, we believe that the current version of NetMedic can scale to 100-machine networks, which suffices for small enterprises. See §8 for a discussion on scaling NetMedic further.

While the data collection part of our system knows the meanings of some variables (e.g., traffic exchanged), we do not use that information in the analysis. Treating variables with known and unknown meanings identically greatly simplifies analysis. It also makes analysis platform-independent and applicable to a range of environments with different sets of known variables. All that is required to port NetMedic to a different environment is to implement data collection on non-Windows machines. Much of the needed information is already there, e.g., in syslog or the proc file system [25] in Linux. Developing a Linux prototype is part of our future work.

7. EVALUATION

We now evaluate NetMedic to understand how well it does at linking effects to their likely causes. We find that NetMedic is highly effective. Across a diverse set of faults it identifies the correct component as the most likely culprit (§7.2) in over 80% of the cases. This ability only slights degrades in the face of simultaneously occurring faults (§7.5). In contrast, a coarse diagnosis method performs rather poorly—only for 15% of the faults, is it able to identify the correct component as the most likely culprit. We show that the effectiveness of NetMedic is due to its ability to cut down by a factor of three the number the edges in the dependency graph for which the source is deemed as likely impacting the destination (§7.3). We also find that the extensions to the basic procedure for edge weight assignment significantly enhance the effectiveness of diagnosis (§7.4) and a modest amount of history seems to be sufficient (§7.6).

Evaluation Platforms: We have deployed our prototype in two environments. The primary one is a live environment. The deployment spans ten client machines and a server machine inside an organization. The clients are actively used desktops that belong to volunteers and have all the noise and churn of regularly used machines.

Because we are not allowed to instrument the real servers in this environment, we deploy our own. As is common in small enterprises, our server machine hosts multiple application servers, including Exchange (email), IIS (web) and MS-SQL (database). Co-hosted application servers are challenging for diagnostic systems as application interactions are more intertwined. The server processes already export several application specific counters.

We implemented custom client processes to communicate with our application servers. The existing client processes on the desktops communicate with the real servers of our organization, and we could not experiment with them without disrupting our volunteers. Our clients export application specific counters similar to those exported by real clients, such as number of successful and failed requests, requests of various types, etc.

Our second environment consists of three clients machines and a server. Because this environment is completely dedicated to our experiments, it is a lot more controlled. We do not consider it to be a realistic setting and unless otherwise stated, the results below are based on the first environment. We present some results from the controlled setting to compare how NetMedic behaves in two disparate environments with different workloads, applications etc.

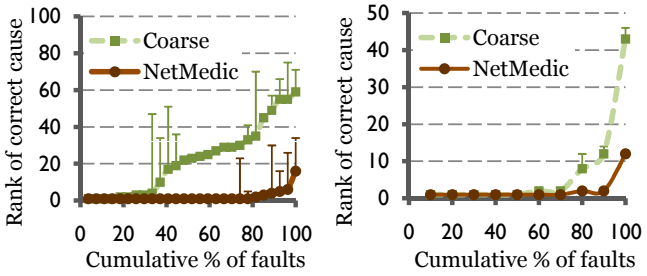
Methodology: Ideally, we would like to diagnose real faults in our deployment but are hindered by the inability to monitor real servers. We are also hindered by ground truth, which is required to understand the effectiveness of diagnosis, being often unavailable for real faults. Hence, most of the results below are based on faults that we inject. We do, however, present evidence that NetMedic can help with faults that occur *in situ* (§7.7).

We inject the diverse set of ten faults shown in Table 1. We stay as close to the reported fault as possible, including the kind of application impacted. For instance, for Problem 1, we misconfigure the IIS server such that it stops serving ASPX pages but continues serving HTML pages. Similarly, to mimic Problem 4, we made an email client depend on information on a mounted drive.

Except for the experiments in §7.5, where we inject multiple faults simultaneously, each fault is injected by itself. We inject each fault at least 5 times, at different times of the day (e.g., day versus night), to verify that we can diagnose it in different operating conditions. Cumulatively, our experiments span a month, with data collection and fault injection occurring almost non-stop.

For diagnosis, we specify as input to NetMedic a one minute window that contains a fault. We did not specify the exact effect to diagnose; rather NetMedic diagnoses all the abnormal aspects in the network. Unless otherwise specified, for each fault we use an hour-long history. The historical period is not necessarily fault-free. In fact, it often contains other injected faults as well as any naturally occurring ones. We do this for realism. In a live environment, it is almost impossible to identify or obtain a fault-free log of behavior.

A coarse diagnosis method: We know of no detailed diagnosis techniques to compare NetMedic against. To understand the value of detailed history-based analysis of NetMedic, we compare it against a *Coarse* diagnosis method that is based loosely on prior formulations that use dependency graphs such as Sherlock and Score [2, 17]. This method uses the same dependency graph as NetMedic. But unlike NetMedic, it captures the behavior of a component with one variable that represents whether the component is behaving normally. The determination regarding normal behavior is made in the same way as in NetMedic. Also unlike NetMedic, *Coarse* has simple component dependencies. A component impacts a neighboring component with a high probability (of 0.9) when both of them are abnormal. Otherwise, the impact probability is low (0.1). The exact values of these probabilities are not significant, as long as one is high and the other is low. Once these edge weights are assigned, the causes are ranked in a manner that is similar to NetMedic. Keeping the ranking method the same for *Coarse* lets us focus the evaluation in this paper on our method for inferring impact among neighbors. We omit results that show that our ranking method outperforms several other alternatives.



(a) Live environment (b) Controlled environment

Figure 7. Effectiveness of *Coarse* and NetMedic for each fault.

Metric: Our metric to evaluate diagnosis is the rank assigned to the real cause for each anticipated effect of a fault. For each fault, we report the median and the maximum rank assigned across its multiple effects. For instance, for Problem 1, all Web clients that browse ASPX pages are expected to be affected. We study the rank assigned to the configuration of Web server for each such client. The median rank represents average case behavior, i.e., what an operator who is diagnosing a randomly chosen effect of the fault would experience. The maximum rank represents the worst case.

What should the rank be for the diagnosis to be useful to an operator? Clearly, lower ranks are better, with a rank of one being perfect. However, even the ability to place the real cause within the top few ranks helps administrators avoid many potential causes that they would otherwise have to consider (close to 1000 in our deployment).

7.1 Dependency graph properties

We briefly describe the dependency graph constructed across the eleven machines in our live environment. The exact numbers vary with time but the graph has close to a 1000 components and 3600 edges. With roughly 70 processes per machine, most of the nodes in the graph correspond to processes. Correspondingly, the vast majority of the edges are between components on the same machine, such as edges between machines and processes. Edges that connect components on different machines (e.g., due to communicating processes) are a much smaller fraction. Hence, the dependency graph is highly clustered, with clusters corresponding to machines and the graph size grows roughly linearly with the number of machines. This linear growth in graph complexity makes it easier to scale NetMedic to larger networks.

Each component provides a rich view of its state in our deployment. Processes have 35 state variables on average, roughly half of which are generic variables representing resource usage while the rest are application specific and vary with the application. IIS server, for instance, exports 128 application-specific variables. Machines have over a hundred variables in their state. Thus, there are plenty of variables that are already exported by real applications and operating systems for detailed diagnosis to be possible. But the sheer scale of this observable state makes understanding variable semantics daunting. NetMedic's ability to be application agnostic allows diagnosis to work even as new applications emerge or variable semantics change.

7.2 Effectiveness of diagnosis

Figure 7(a) shows the effectiveness of NetMedic and *Coarse* across all faults injected in the live environment. The lines connect the median ranks and the error bars denote the maximum ranks. The two curves are independently sorted based on the median rank.

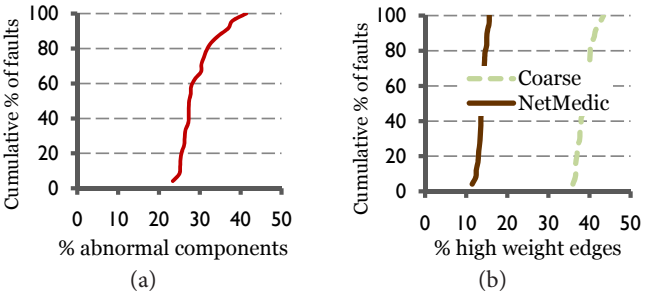


Figure 8. (a) CDF of the percentage of components that are abnormal during a fault. (b) CDF of the percentage of edges that are assigned a high weight in the dependency graph.

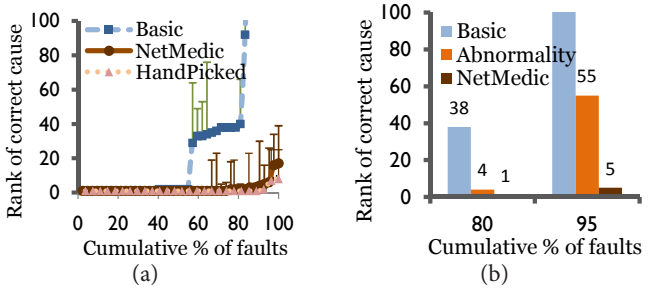


Figure 9. Value of NetMedic's extensions to the basic procedure.

We see that for 80% of the faults the median rank of the correct cause is one with NetMedic. That is, NetMedic frequently places the real culprit at the top of the list of likely causes. For all cases except one, the median rank of the correct cause is five or lower. The maximum ranks are often close to the median ranks, representing good worst-case behavior as well. These results suggest that NetMedic can help operators diagnose such faults in their networks.

In contrast, diagnosing these faults with *Coarse* would likely be a frustrating exercise. The correct cause is assigned a rank of one in fewer than 15% of the cases. For over 60% of the cases, the correct cause has a median rank of more than ten.

We examined cases where NetMedic assigned a median rank greater than three to the correct cause. We find that these often correspond to performance faults, which include Problems 2, 8 and 9 in Table 1. The side-effects of these faults lead to abnormality in many components in the network. For instance, a process that hogs the CPU disturbs many other processes on its machine, each of which can appear abnormal. A few of the victim components can get ranked lower than the correct cause if there is insufficient history to correctly determine the direction of impact. Diagnosis of non-performance faults, which tend to be more prevalent (§2.3) turns out to be easier as they disturb fewer components in the network.

Let us consider now the results from the controlled environment shown in Figure 7(b). We reduce the *y*-axis range for this graph because the environment has fewer components. We see that NetMedic effectively diagnoses faults in this setting as well.

Interestingly, *Coarse* performs much better in this setting. In the live environment, for the worst 20% of the cases, its median rank is 35 or higher. Here, the median rank is 8 or higher, a sharp improvement even after accounting for the difference in the numbers of components. Thus, in going from the controlled to the more dynamic and realistic setting, the ability of *Coarse* degrades sharply. This degradation stems from the fact that the live environment has more abnormal components. Because of its simplistic component states and dependency models, *Coarse* cannot effectively infer which components are impacting each other, and many components get ranked lower than the real culprit. NetMedic, on the other hand, shows no such degradation in our experiments and appears better equipped towards handling the noise in real environments. The next section investigates in more detail why the methods differ.

7.3 Why NetMedic outperforms *Coarse*?

NetMedic outperforms *Coarse* primarily because at the level of detail that we observe at, components are often abnormal. As a result, *Coarse* assigns a high weight to many edges and ends up erroneously connecting many non-responsible components to the observed effects. By looking at component states in detail and allowing for complex dependencies, NetMedic assigns a low weight to many edges even when both end points are abnormal simultaneously.

Figure 8(a) shows the CDF of the percentage of components that are abnormal during the periods covering various faults. We see that this percentage is quite high (20-40%).

Figure 8(b) shows the CDF of the percentage of edges in the dependency graph that are assigned a high weight (> 0.75) by each scheme. We see that this percentage is 35-45% for *Coarse* and 10-15% for NetMedic, which represents reduction by a factor of 3. This reduction in likely spurious high-weight edges leads to fewer possible causes being strongly connected to the affected component, resulting in fewer false positives and lower ranks for real causes.

Simply changing the requirement for deeming a component as abnormal (e.g., using a higher abnormality threshold or requiring more state variables to be abnormal) may reduce false positives. But we find that doing so can hurt. It runs the risk of excluding the real culprit from the list altogether; the culprit or a component on the path from it to the effect of interest may appear normal.

7.4 Benefit of extensions

We now study the value of the extensions to edge weight assignment by comparing them to two other methods. The first is the basic procedure, without the extensions. For the second method, instead of automatically inferring relationships between variables, we hand code them, based on our knowledge of what each variable represents. Given that the number of variables is quite large, we hard code knowledge of only those that are relevant for diagnosing the faults that we inject. Beyond programming these relationships, the rest of the procedure stays the same. Comparison with this "HandPicked" method quantifies any reduction in diagnostic effectiveness due to our desire to be application agnostic and treating these variables as opaque.

Figure 9(a) shows the diagnostic effectiveness of all three methods. Comparing the basic procedure with *Coarse* in Figure 7(a) reveals that it more frequently assigns a rank of one to the correct cause. This frequency is 44% versus the 14% of *Coarse*. But overall, the basic procedure is quite fragile. In fact in the worst 20% of the cases, it assigns a higher rank to the correct cause than *Coarse*.

The extensions help make the basic idea practical—an 80% frequency of assigning a rank of one to the correct cause and a significant reduction in the ranks of the correct cause for half the faults. Closer examination reveals that such faults often correspond to performance issues. As mentioned previously, performance faults have more side effects than configuration faults. The extensions are better able to sift through this noise.

Figure 9(a) also shows that the performance of NetMedic is close to *HandPicked*. Thus, the extensions extract enough semantic information for our task to not require embedding knowledge of variable semantics into the system.

To investigate in more detail, we separately consider the extension that weighs variables based on their abnormality values and the other three extensions that infer variable relationships. Figure 9(b) shows the median rank for 80th and 95th percentile of the faults with the basic procedure, with only the abnormality extension, and NetMedic, which includes all extensions. We see that both factoring in abnormality and variable relationships are useful.

7.5 Multiple simultaneous faults

We now study the ability of NetMedic to diagnose multiple, simultaneously occurring faults. In a dynamic network, simultane-