Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies

Victor Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, Ming Zhang
Microsoft Research

Abstract

Localizing the sources of performance problems in large enterprise networks is extremely challenging. Dependencies are numerous, complex and inherently *multi-level*, spanning hardware and software components across the network and the computing infrastructure. To exploit these dependencies for fast, accurate problem localization, we introduce an Inference Graph model, which is well-adapted to user-perceptible problems rooted in conditions giving rise to both partial service degradation and hard faults. Further, we introduce the Sherlock system to discover Inference Graphs in the operational enterprise, infer critical attributes, and then leverage the result to automatically detect and localize problems. To illuminate strengths and limitations of the approach, we provide results from a prototype deployment in a large enterprise network, as well as from testbed emulations and simulations. In particular, we find that taking into account multi-level structure leads to a 30% improvement in fault localization, as compared to two-level approaches.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

Keywords

Network & service management, dependencies, fault localization, probabilistic inference

1. INTRODUCTION

Using a network-based service can be a frustrating experience, marked by appearances of familiar hourglass or beachball icons, with little reliable indication of where the problem lies, and even less on how it might be mitigated. Even inside the network of a single enterprise, where traffic does not need to cross the open Internet, user-perceptible service degradations are rampant. Consider Figure 1, which shows the distribution of time required for clients to fetch the home page from a major webserver in a large enterprise network including tens of thousands of network elements and over 400,000 hosts. The distribution comes from a data set of 18 thousand samples from 23 instrumented clients over a period 24 days. The second mode of the distribution represents user-perceptible lags of 3 to 10+ seconds, and 13% of the requests experience this unacceptable performance. This problem persists because current network and service monitoring tools are blind to the complex set of dependencies across systems and networks in the enterprise, needed for root cause analysis.

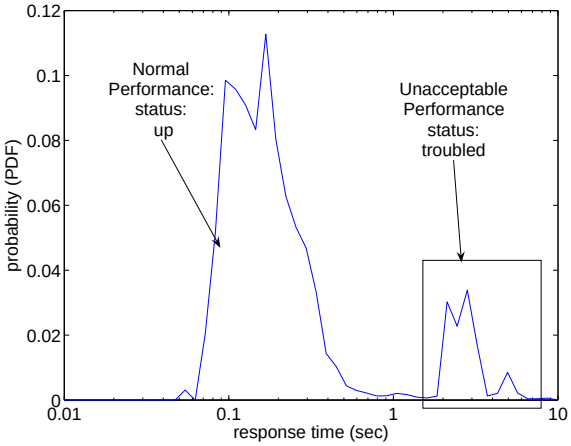


Figure 1: The response time of a major internal webserver when fetching the home page. The times are clearly bi-modal, with 13% of the requests taking 10x longer than normal and resulting in user-perceptible lags. We define the first mode in response time as indicating the service is *up* and the second mode as indicating the service is *troubled*.

Conventional management systems treat each service, which we define as an *(IPaddr, port)* pair, as being either up or down. This naive model hides the kinds of *performance failures* shown in Figure 1. In this paper, we model service availability as a 3-state value: a service is *up* when its response time is normal; it is *down* when requests result in either an error status code or no response at all; and it is *troubled* when responses fall significantly outside of normal response times. Our definition of troubled status includes the particularly challenging cases where only a subset of service requests are performing poorly.

This paper describes the Sherlock system that aims to give IT administrators the tools they need to localize performance problems and hard failures that affect an end-user. Sherlock (1) detects the existence of faults and performance problems by monitoring the response time of services; (2) determines the set of components that could be responsible; and (3) localizes the problem to the most likely component.

We faced three main challenges in creating Sherlock. First, both performance and hard faults can stem from problems anywhere in the IT infrastructure, i.e., a service, a router, or a link. Adding complexity to the problem, even simple requests like fetching a webpage involve multiple services: DNS servers, authentication servers, webservers, and the backend SQL databases that hold the webpage data. Problems at any of these can affect the success or failure of the request, but the dependencies among components in IT systems are typically not recorded anywhere, and they evolve continually as systems grow or new applications are added. As a result, Sherlock must be able to automatically discover the set of components involved in the processing of requests. Second, the

SIGCOMM'07, August 27–31, 2007, Kyoto, Japan.

failover and load-balancing techniques commonly used in enterprise networks make determining the responsible component even harder, since the set of components involved may change from request to request. Sherlock's analysis must take these mechanisms into account. Third, given the large size of enterprise networks, the challenges above must be met in manner that remains tractable even with hundreds of thousands of elements.

Sherlock meets these challenges in the following ways: First, software agents running on each host analyze the packets that the host sends and receives to determine the set of services on which the host depends. Sherlock automatically assembles an *Inference Graph* that captures the dependencies between all components of the IT infrastructure by combining together these individual views of dependency. Our algorithm uses information provided by one host to fill in any gaps in the information provided by another. Sherlock then augments the Inference Graph with information about the routers and links used to carry packets between hosts, and so encodes in a single model all the components that could affect a request. Second, our Inference Graph model includes primitives that capture the behavior of load-balancers and failover mechanisms. Operators must identify where these mechanisms are used manually or via heuristics (Section 4.2), but localization is then automatic. Third, we developed Ferret, an algorithm that efficiently localizes faults in enterprise-scale networks using the Inference Graph and measurements of service response times made by the agents.

We deliberately targeted Sherlock at localizing significant problems that affect the users of the IT infrastructure, hence our focus on performance as well as hard faults and our use of response time as an indicator for performance faults. Current systems overwhelm operators with meaningless alerts (the current management system in our organization generates 15,000 alerts a day, and they are almost universally ignored as so few prove significant). In contrast, Sherlock does not report problems that do not directly affect users. For example, Sherlock will not even detect that a server has a high CPU utilization unless requests are delayed as a result.

Sherlock aims for problem localization, which falls short in general of full problem diagnosis. For example, Sherlock can determine that a SQL server is overloaded, but not that the overload stems from a missing index. Yet, operational experience indicates that problem diagnosis and resolution often rapidly follows problem localization. Indeed, it is not uncommon that IT administrators are aware of suspicious, faulty or troubled states of a tremendous number of components along with associated methods of mitigation or repair, but, lacking localization, are unaware of which, if any, of these troubled components explain a high impact outage. If a server returns incorrect information (e.g., a DNS server returns the wrong IP address), Sherlock may help by detecting a change in the dependencies among the components but it might not directly localize the fault to the DNS server. Finally, we have not evaluated Sherlock on systems that deliberately and frequently change their dependencies and cannot predict its performance on such systems. However, measurements indicate that the vast majority of enterprise applications do not fall into this class [2].

To the best of our knowledge, Sherlock is the first system that localizes performance failures across network and services in a timely manner without requiring modifications to existing applications and network components. The contributions of this paper include our formulation of the Inference Graph and our algorithms for computing it for an entire IT infrastructure based on observations of the packets that hosts send and receive. Unlike previous work, our Inference Graph is both multi-level (in order to represent the multiple level of dependencies found in IT infrastructure) and 3-state (so we can determine whether components are up, down, or experiencing a performance fault and troubled). This paper also contributes extensions to prior work that optimize fault localization and adapt it for our three-state and multi-level Inference Graph. We extensively evaluate the effectiveness of each of Sherlock's components individually, and describe our results of deploying Sherlock in both a testbed and a large and complex enterprise network.

2. RELATED WORK

Today, enterprises use sophisticated commercial tools, such as EMC's SMARTS [21], HP Openview [13], IBM Tivoli [19], or Microsoft Operations Manager [10]. In practice, these systems have proven inadequate for finding the causes of performance problems as they treat servers and routers as independent boxes — each producing its own stream of SNMP counters, syslog messages, and alerts. Fundamentally, these box-centric measures are poor predictors of the end-to-end response time that users ultimately care about — it's not clear what CPU load on a server means users are unhappy, so it is hard to set a threshold that alerts only when users are impacted. For example, over a 10-day period our organization's well-run systems generated two thousand alerts for 160 servers that *might* be sick. Another 18 K alerts were divided among 194 different alert types coming from 877 different servers, each of which could *potentially* affect user performance (e.g., 6 alerts for a server CPU utilization over 90%; 8 for low memory causing a service to stop). Investigating all the potentially serious alerts is simply impractical, especially when many had no effect on a user. Sherlock complements existing tools by detecting and localizing the problems that affect users.

Significant recent research has led to methods for detailed debugging of service problems in distributed systems. Many of these systems also extract the dependencies between components, but are different in character from Sherlock. Magpie [3], FUSE [5] and Pinpoint [4], instrument middleware on every host to track requests as they flow through the system. They then diagnose faults by correlating components with failed requests. Project5 [1] and WAP5 [16] record packet traces at each host and use message correlation algorithms to resolve which incoming packet triggered which outgoing packet. These projects all target the debugging and profiling of *individual* applications, so determining exactly which message is caused by another message is critically important. In contrast, Sherlock combines measurements of the *many* applications running on an IT infrastructure to localize problems. We also show that, for fault localization, co-occurrence of packets is a reasonable indicator of dependency between accesses to two remote machines, and that valid graphs can be computed with only 1,000 samples and 20 clients (Section 6.1).

There is a large body of prior work tackling fault localization at the network layer, especially for large ISPs. In particular, BADABING [18] and Tulip [9] measure per-path characteristics, such as loss rate and latency, to identify problems that impact user-perceptible performance. These methods (and many commercial products as well) use active probing to pinpoint faulty IP links. Sherlock instead uses a passive correlation approach to localize failed network components.

Machine learning methods have been widely discussed for fault management. Pearl [15] describes a graph model for Bayesian networks. Sherlock uses similar graph models to build Inference Graphs. Rish et. al. [17] combines active probing and dependency graph modeling for fault diagnosis in a network of routers and end hosts, but they do not describe how the graph model might be automatically constructed. Unlike Sherlock, their method does not model failover servers or load balancers, which are common in en-

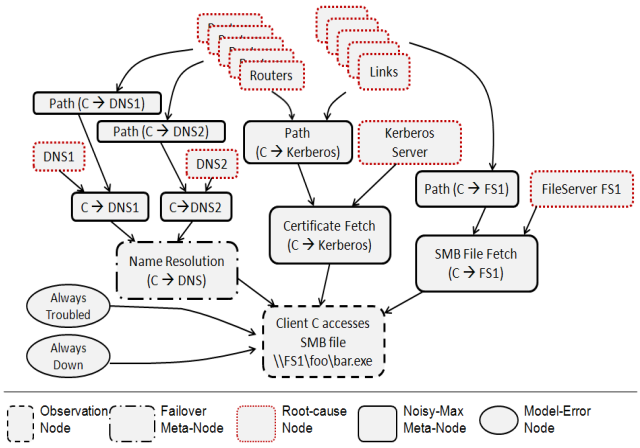


Figure 2: Snippet of a partial Inference Graph that expresses the dependencies involved in accessing a file share. Dotted boxes represent physical components and software, dashed boxes denote external observations and ovals stand-in for unmodeled or external factors.

terprise networks. Shrink [6] and SCORE [7] make seminal contributions in modeling the network as a two-level graph and using the model to find the most likely root causes of faults in wide-area networks. In SCORE, dependencies are encoded as a set and fault-localization becomes minimal set cover. Shrink introduces novel algorithmic concepts in inference of most likely root causes, taking probabilities describing strengths of dependencies into account. In Sherlock, we leverage these concepts, while extending them to deal with multi-level dependencies and with more complex operators that capture load-balancing and failover mechanisms. We compare the accuracy of our fault-localization algorithm with Shrink and SCORE in Section 6.

3. THE Inference Graph MODEL

We first describe our new model, called the Inference Graph, for representing the complex dependencies in an enterprise network. The Inference Graph forms the core of our Sherlock system. We then present our algorithm, called Ferret, that uses the model to probabilistically infer the faulty or malfunctioning components given real-world observations. We explain the details of how Sherlock constructs the Inference Graph, computes the required probabilities, and performs fault localization later in Section 4.

3.1 The Inference Graph

The Inference Graph is a labeled, directed graph that provides a unified view of the dependencies in an enterprise network, spanning services and network components. Figure 2 depicts a portion of the Inference Graph when a user accesses a network file share. The structure of dependence is inherently multi-level. The access to the file depends on contacting the Kerberos server for authentication, which in turn depends on the Kerberos server itself, as well as the routers and switches on the path from the user's machine to the Kerberos server. A problem could occur anywhere in this chain of dependencies. The challenge is to find the right level of abstraction to model these dependencies in a framework that can be feasibly automated.

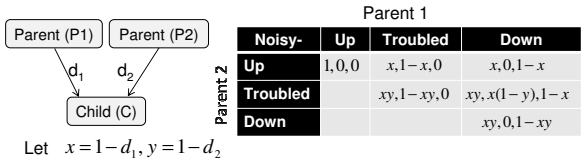
Formally, nodes in this graph are of three types. First, *root-cause* nodes correspond to physical components whose failure can cause an end-user to experience failures. The granularity of root-cause nodes in Sherlock is a computer (a machine with an IP address), a service (IP address, port), a router, or an IP link, although the model is extensible to root causes at a finer granularity. Second, *observation* nodes represent accesses to network services whose performance can be measured by Sherlock. There is a separate observation node for every client that accesses any such network service. The observation nodes model a user's experience when using services on the enterprise network. Finally, *meta-nodes* act as glue between the root-cause nodes and the observation nodes. In this paper we present three types of meta-nodes, *noisy-max*, *selector* and *failover*. These nodes model the dependencies between root causes and observations; the latter two are needed to model load-balancers and failover redundancy, respectively (described in detail in Section 3.1.1).

The state of each node in the Inference Graph is expressed by a three-tuple: $(P_{up}, P_{troubled}, P_{down})$. P_{up} denotes the probability that the node is working normally. P_{down} is the probability that the node has experienced a fail-stop failure, such as when a server is down or a link is broken. Finally, $P_{troubled}$ is the probability that a node is troubled, which corresponds to the boxed area in Figure 1, where services, physical servers or links continue to function but users perceive poor performance. The sum of $P_{up} + P_{troubled} + P_{down} = 1$. We note that the state of root-cause nodes is independent of any other nodes in the Inference Graph, while the state of observation nodes can be uniquely determined from the state of its ancestors.

An edge from node A to node B in the Inference Graph encodes the dependency that node A has to be in the *up* state for node B to be *up*. Not all dependencies are equal in strength. For example, a client cannot retrieve a file from a file server if the path to that file server is down. However, the client might still be able to retrieve the file even when the DNS server is down, if the file server's name to IP address mapping is found in the client's local DNS cache. Furthermore, the client may need to authenticate more (or less) often than resolving the server's name. To capture varying strengths in dependencies, edges in a Inference Graph are labeled with a *dependency probability*. A larger dependency probability indicates stronger dependency.

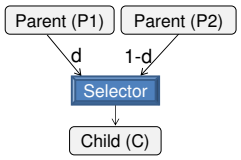
Finally, every Inference Graph has two special root-cause nodes – *always troubled (AT)* and *always down (AD)* – to model external factors not part of our model that might cause a user-perceived failure. The state of AT is set to (0, 1, 0) and that of AD is set to (0, 0, 1). We add an edge from these nodes to all the observation nodes, and describe how we assign probabilities to these edges in Section 4.

To illustrate these concepts we revisit Figure 2, which shows a portion of the Inference Graph that models a user fetching a file from a network file server. The user activity of "fetching a file" is encoded as an observation node (dashed box) in the figure because Sherlock can measure the response time for this action. Fetching a file requires the user to perform three actions: (i) authenticate itself to the system, (ii) resolve the DNS name of the file server and (iii) access the file server. These actions themselves depend on other actions to succeed. Therefore, we model them as meta-nodes, and add edges from each of them to the observation node of "fetching a file." We describe our method of computing the dependency probability for these edges in Section 4.1. Since the client is configured with both a primary and secondary DNS server (DNS1 and DNS2), we introduce a failover meta-node. Finally, note that this snippet shows a single client and a single observation. When other clients access the same servers or use the same routers/links as those shown here, their observation nodes will be connected to the same root cause nodes as those shown to create the complete Inference Graph.



Parent 2 \ Parent 1	Up	Troubled	Down
Up	1,0,0	x,1−x,0	x,0,1−x
Troubled		xy,1−xy,0	xy, $x(1-y)$,1−x
Down			xy,0,1−xy

Figure 3: Truth Table for the noisy-max meta-node when a child has two parents. The values in the lower triangle are omitted for clarity.



Let $x = 1-d$

Parent 2 \ Parent 1	Up	Troubled	Down
Up	1,0,0	x,1−x,0	x,0,1−x
Troubled		0,1,0	0, x,1−x
Down			0,0,1

Figure 4: Truth Table for the selector meta-node. A child node selects parent1 with probability d and parent2 with probability 1-d. The values in the lower triangle are omitted for clarity.

3.1.1 Propagation of State with Meta-Nodes

A crucial aspect of a probabilistic model is *how* the state of parent nodes governs the state of a child node. For example, suppose a child has two parents, A and B; the state of parent A is (.8, .2, 0), which means its probability of being *up* is 0.8, *troubled* is 0.2 and *down* is 0, and the state of parent B is (.5, .2, .3). What, then, is the state of the child? While edge labels encode the strength of dependency, the nature of the dependency is encoded in the meta-node. Formally, the meta-node describes the state of the child node given the state of its parent nodes.

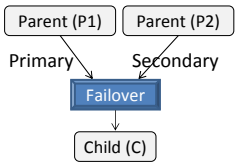
Noisy-Max Meta-Nodes are the simplest and most common meta-node. *Max* implies that if any of the parents are in the *down* state, then the child is *down*. If no parent is *down* and any parent is *troubled*, then the child is *troubled*. If all parents are *up*, then the child is *up*. *Noisy* implies that unless a parent's dependency probability is 1.0, there is some chance the child will be *up* even if the parent is *down*. Formally, if the weight of a parent's edge is d, then with probability $(1-d)$ the child is not affected by that parent.

Figure 3 presents a truth table for noisy-max when a child has two parents. Each entry in the truth table is the state of the child (i.e., its probability of being *up*, *troubled* and *down*) when $parent_1$ and $parent_2$ have states as per the column and row label respectively.[1] As an example, the second row and third column of the truth table shows the probability of the child being *troubled*, given that $parent_1$ is *down* and $parent_2$ is *troubled*: P(Child=Troubled | Parent1=Down, Parent2=Troubled) = $(1-d_1) * d_2$. To explain, the child will be *down* unless $parent_1$'s state is masked by noise (prob $1-d_1$). Further, if both parents are masked by noise, the child will be *up*. Hence the child is in *troubled* state only when $parent_1$ is drowned out by noise and $parent_2$ is not.

Selector Meta-Nodes are used to model load balancing scenarios. For example, a Network Load Balancer (NLB) in front of two servers hashes the client's requests and distributes requests evenly to the two servers. An NLB cannot be modeled using a noisy-max meta-node because the client would depend on each server with a probability of 0.5, since half the requests go to each server. Using a noisy-max meta-node will assign the client a 25% chance of being *up* even when both the servers are *down*, which is obviously incorrect. We use the selector meta-node to model NLB Servers and Equal Cost Multipath (ECMP) routing. ECMP is a commonly-used technique in enterprise networks where routers send packets to a destination along several paths. The path is selected based on a hash of the source and destination addresses in the packet. We use a selector meta-node when we can determine the set of ECMP paths available, but not which path a host's packets will use.

The truth table for the selector meta-node is shown in Figure 4, and it expresses the fact the child is making a selection. For example, while the child may choose each of the parents with probability 50%, the selector meta-node forces the child to have a zero probability of being *up* when both its parents are *down* (first number in the Down,Down entry).

[1]A (0, 1, 0) state for $parent_1$ means it is troubled.



Parent 2 \ Parent 1	Up	Troubled	Down
Up	1,0,0	0,1,0	1,0,0
Troubled	1,0,0	0,1,0	0,1,0
Down	1,0,0	0,1,0	0,0,1

Figure 5: Truth Table for the failover meta-node encodes the dependence that the child primarily contacts parent1, and fails over to parent2 when parent1 does not respond.

Failover Meta-Nodes capture the failover mechanism commonly used in enterprise servers. Failover is a redundancy technique where clients access primary production servers and failover to backup servers when the primary server is inaccessible. In our network, DNS, WINS, Authentication and DHCP servers all employ failover. Failover cannot be modeled by either the noisy-max or selector meta-nodes, since the probability of accessing the backup server depends on the failure of the primary server.

The truth table for the failover meta-node is shown in Figure 5. As long as the primary server is *up* or *troubled*, the child is not affected by the state of the secondary server. When the primary server is in the *down* state, the child is still *up* if the secondary server is *up*.

3.1.2 Time to Propagate State

A common concern with probabilistic meta-nodes is that computing the probability density for a child with n parents can take $O(3^n)$ time for a three-state model in the general case.[2] However, the majority of the nodes in our Inference Graph with more than one parent are noisy-max meta-nodes. For these nodes, we have developed the following equations that reduce the computation to $O(n)$ time.

$$\text{P(child up)} = \prod_j \left((1-d_j) * (p_j^{trouble} + p_j^{down}) + p_j^{up} \right)$$

$$1 - \text{P(child down)} = \prod_j \left(1 - p_j^{down} + (1-d_j) * p_j^{down} \right)$$

$$\text{P(child troubled)} = 1 - (\text{P(child up)} + \text{P(child down)})$$

where p_j is the j'th parent, $(p_j^{up}, p_j^{trouble}, p_j^{down})$ is its probability distribution, and d_j is its dependency probability. The first equation implies that a child is *up* only when it does not depend on any parents that are not *up*. The second equation implies that a child is *down* unless every one of its parents are either not *down* or the child does not depend on them when they are *down*.

[2]The naive way to compute the probability of the child's state requires computing all 3^n entries in the truth-table and summing the appropriate entries.

The computational cost for selector and failover meta-nodes is still exponential, $O(3^n)$, for a node with n parents. However, in our experience, these two types of meta-nodes have no more than 6 parents, and hence do not add a significant computation burden.

3.2 Fault Localization on the Inference Graph

We now present our algorithm, Ferret, that uses the Inference Graph to localize the cause of a network or service problem. We define an **assignment-vector** to be an assignment of state to every root-cause node in the Inference Graph where the root-cause node has probability 1 of being either up, troubled, or down. The vector might specify, for example, that $link_1$ is troubled, $server_2$ is down and all the other root-cause nodes are up. The problem of localizing a fault is then equivalent to finding the assignment-vector that best explains the observations measured by the clients.

Ferret takes as input the Inference Graph and the measurements (e.g., response times) associated with the observation nodes. Ferret outputs a ranked list of assignment vectors ordered by a confidence value that represents how well they explain the observations. For example, Ferret could output that $server_1$ is troubled and other root-cause nodes are up with a confidence of 90%, $link_2$ is down and other root-cause nodes are up with 5% confidence, and so on.

For any assignment-vector, Ferret can compute a score for how well that vector explains the observations. Ferret first sets the root causes to the states specified in the assignment-vector and then uses the state-propagation techniques described in the previous section to propagate probabilities downwards until they reach the observation nodes. Then, for each observation node, it computes a score based on how well the probabilities in the state of the observation node agree with the statistical evidence derived from the measurements associated with this observation node. Section 4 provides the details of how we compute this score.

How can we search through all possible assignment vectors to determine the vector with the highest score? There are 3^r vectors given r root-causes, and applying the procedure just described to evaluate the score for each assignment vector would be infeasible. Existing solutions to this problem in machine learning literature, such as loopy belief propagation [12], do not scale to the Inference Graph sizes encountered in enterprise networks. Approximate localization algorithms used in prior work, such as Shrink [6] and SCORE [7], are significantly more efficient. However, they are based on two-level, two-state graph models, and hence do not work on the Inference Graph, which is multi-level, multi-state and includes meta-nodes to model various artifacts of an enterprise network. The results in Section 6 clarify how Ferret compares with these algorithms.

Ferret uses an approximate localization algorithm that builds on an observation that was also made by Shrink [6].

OBSERVATION 3.1. *It is very likely that at any point in time only a few root-cause nodes are troubled or down.*

In large enterprises, there are problems all the time, but they are usually not ubiquitous.[3] We exploit this observation by not evaluating all 3^r assignment vectors. Instead, Ferret evaluates assignments that have no more than k root-cause nodes that are either troubled or down. Thus, Ferret first evaluates $2 * r$ vectors in which exactly one root-cause is troubled or down, next $2 * 2 * \binom{r}{2}$ vectors where exactly two root-causes are troubled or down, and so on. Given k, Ferret evaluates at most $(2 * r)^k$ assignment vectors. Further, it is easy to prove that the approximation error of Ferret, that is, the probability that Ferret does not arrive at the correct solution (the same solution attained using the brute-force, exponential approach) decreases exponentially with k and becomes vanishingly small for $k = 4$ onwards [6]. Pseudo-code for the Ferret algorithm is shown in Algorithm 1.

Algorithm 1 Ferret{Observations O, Inference Graph G, Int X}

```
Candidates ← (up|trouble|down) assignments to root causes
    with at most k abnormal at any time
List_X ← {}                          ▷ List of top X Assignment-Vectors
for R_a ∈ Candidates do              ▷ For each Assignment-Vector
    Assign States to all Root-Causes in G as per R_a.
    Score(R_a) ← 1                   ▷ Initialize Score
    for Node n ∈ G do                ▷ Breadth-first traversal of G
        Compute P(n) given P(parents of n)   ▷ Propagate
    end for
    for Node n ∈ G_O do              ▷ Scoring Observation Nodes
        s ← P( Evidence at n | prob. density of n)   ▷ How well
            does R_a explain observation at n?
        Score(R_a) ← Score(R_a) * s  ▷ Total Score
    end for
    Include R_a in List_X if Score(R_a) is in top X assignment
        vectors
end for
return List_X
```

Ferret uses another practical observation to speed up its computation.

OBSERVATION 3.2. *Since a root-cause is assigned to be* **up** *in most assignment vectors, the evaluation of an assignment vector only requires re-evaluation of states at the descendants of root-cause nodes that are not* **up**.

Therefore, Ferret preprocesses the Inference Graph by assigning all root-causes to be up and propagating this state through to the observation nodes. To evaluate an assignment vector, Ferret needs to re-compute only the nodes that are descendants of root-cause nodes marked troubled or down in the assignment vector. After computing the score for an assignment vector, Ferret simply rolls back to the pre-processed state with all root-causes in the *up* state. As there are never more than k root-cause nodes that change state out of the hundreds of root-cause nodes in our Inference Graphs, this reduces Ferret's time to localize by roughly two orders of magnitude without sacrificing accuracy.

In the studies presented in this paper, we use the Ferret algorithm exactly as described above. However, the inference algorithm can be easily extended to leverage whatever domain knowledge is available. For example, if prior probabilities on the failure rates of components are known (e.g., links in enterprise networks may have a much higher chance of being congested than down [14]), then Ferret can sort the assignment vectors by their prior probability and evaluate in order of decreasing likelihood to speed up inference.

4. THE SHERLOCK SYSTEM

Now that we have explained the Inference Graph model and Ferret fault localization algorithm, we describe the Sherlock system that actually constructs the Inference Graph for an enterprise network and uses it to localize faults. Sherlock consists of a centralized *Inference Engine* and distributed *Sherlock Agents*. Sherlock requires no changes to routers, applications, or middleware used in the enterprise. It uses a three-step process to localize faults in the enterprise network, illustrated in Figure 6.

[3]There are important cases where this observation might not hold, such as rapid malware infection and propagation.

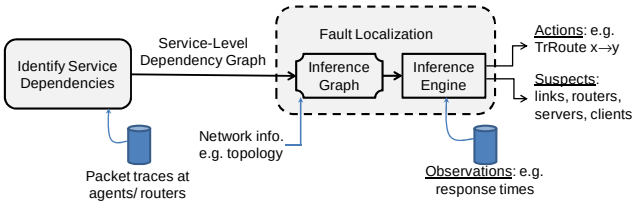


Figure 6: Sherlock Solution Overview

First, Sherlock computes a *service-level dependency graph (SLDG)* that describes the services on which each client and service depends. Each Sherlock agent is responsible for monitoring the packets sent and received by one or more hosts. The agent may run on the host itself, or it may obtain packet traces via sniffing a nearby link or router. From these traces, the agent computes the dependencies between the services with which its host(s) communicates and the response time distributions for each service (Section 4.1). This information is then relayed to the inference engine as described in Section 5, where the engine aggregates the dependencies between services computed by each agent to form the SLDG. The SLDG is relatively stable, changing when new hosts or applications are added to the network, and we expect it will be recomputed daily or weekly.

Second, the inference engine combines the SLDG with the network topology to compute a unified Inference Graph over all services in which the operator is interested and across all Sherlock agents (Section 4.2). This step can be repeated as often as needed to capture changes in the network.

Third, the inference engine runs Ferret over the response time observations reported by the agents and the Inference Graph to identify the root-cause node(s) responsible for any problem observed. This step is executed whenever agents observe large response times.

4.1 Discovering Service-Level Dependencies

Each Sherlock agent is responsible for computing the dependency between the services its host accesses. We define the *dependency probability* of a host on service A when accessing service B as the probability the host needs to communicate with service A before it can successfully communicate with service B. A value of 1 indicates a strong dependency, where the host machine always contacts service A before contacting B. For example, a client will visit a web server soon after receiving a response from DNS server providing the web server's IP address, so the dependency probability of using DNS when visiting a web server will be greater than 0. Due to caching, however, the probability may be less than 1.

Because we define services in terms of IP addresses and ports, Sherlock does not rely on parsing application-specific headers. It could be easily extended to use a finer-grain notion of a service if such parsers were available.

4.1.1 Computing the Dependency Probability

Sherlock computes the dependency between services by leveraging the observation that if accessing service B depends on service A, then packets exchanged with A and B are likely to co-occur.

Using this observation, we approximate the dependency probability of a host on service A when accessing service B as the conditional probability of accessing service A within a short interval, called the *dependency interval*, prior to accessing service B. We compute the conditional probability as the number of times in the packet trace that an access to service A precedes an access to service B within the dependency interval.

There is a tension in choosing the value of the dependency interval which is well known in machine learning [8]. Too large an interval will introduce false dependencies on services that are accessed with a high frequency, while too small an interval will miss some true dependencies.

The Sherlock agents use a simple approach that works well in practice. The dependency interval is fixed at 10 ms, which in our experience discovers most of the dependencies. The agents then apply a simple heuristic to eliminate false positives due to chance co-occurrence. They first calculate the average interval, I, between accesses to the same service and estimate the likelihood of "chance co-occurrence" as $(10ms)/I$. They then retain only the dependencies where the dependency probability is much greater than the likelihood of chance co-occurrence.

Our heuristic for computing dependency works best when a response from service A precedes a request to service B. But without deep packet inspection, it is not possible to explicitly identify the requests and responses in streams of packets going back and forth between the host and A and the host and B. In practice, we have found it is sufficient to group together a contiguous sequence of packets to a service as a single access to the service. In Section 6.1, we show that this simple approximation produces reasonably accurate service-level dependency graphs.

4.1.2 Aggregating Probabilities Across Clients

All agents periodically submit the dependency probabilities they measure to the inference engine. However, because some services are accessed infrequently, a single host may not have enough samples to compute an accurate probability. Fortunately, many clients in an enterprise network have similar host, software and network configurations (e.g. clients in the same subnet) and are likely to have similar dependencies. Therefore, the inference engine aggregates the probabilities of similar clients to obtain more accurate estimates of the dependencies between services.

Aggregation also provides another mechanism to eliminate false dependencies – for example, a client making a large number of requests to the proxy server will appear to be dependent on the proxy server for all the services it accesses. To eliminate these false dependencies, the inference engine calculates the mean and standard deviation of each dependency probability. It then excludes clients with a probability more than five standard deviations from the mean. Section 6.1 evaluates the effectiveness of this aggregation.

4.2 Constructing the Inference Graph

Here we describe how the Inference Engine combines dependencies between services reported by the Sherlock agents with network topology information to construct a unified Inference Graph.

For each service S, the inference engine first creates a noisy-max meta-node to represent the service. It then creates an observation node for each client reporting response time observations of that service and makes the service meta-node a parent of the observation node. The engine then examines the service dependency information of these clients to identify the set of services $\mathcal{D}_S$ that the clients are dependent on when accessing S. The engine then recurses, expanding each service in $\mathcal{D}_S$. Once all service meta-nodes have been created, for each of these nodes the inference engine creates a root-cause node to represent the host on which the service runs and makes this root-cause a parent of the meta-node.

The inference engine then adds network topology information to the Inference Graph by using traceroute results reported by the agents. For each path between hosts in the Inference Graph, it adds

a noisy-max meta node to represent the path and root-cause nodes to represent every router and link on the path. It then adds each of these root-causes as parents of the path meta-node.

Optionally, the operators can tell the inference engine where load balancing or redundancy techniques are used in their network, and the engine will update the Inference Graphs, drawing on the appropriate specialized meta-node. Adapting the local environment to the configuration language of the inference engine can also be done with scripting. For example, in our network the load-balanced web servers for a site follow a naming convention and are called $sitename$* (e.g., msw01, msw02). Our script looks for this pattern and replaces the default meta-nodes with selector meta-nodes. Similarly, the agent examines its host's DNS configuration using ipconfig to identify where to place a failover meta-node to model the primary/secondary relationship between its name resolvers.

Finally, the inference engine assigns probabilities to the edges in the Inference Graph. The service-level dependency probabilities are directly copied onto corresponding edges in the Inference Graph. The special nodes *always troubled* and *always down* are connected to observation nodes with a probability of 0.001, which implies that 1 in 1000 failures are caused by a component not in our model. Edges between a router and a path meta-node use a probability of 0.9999, which implies that there is a 1-in-10,000 chance that our network topology or traceroutes are incorrect and the router is not actually on the path. In our experience, Sherlock's results are not sensitive to the precise setting of these parameters (Section 6.2).

4.3 Fault Localization Using Ferret

As described in Section 3.2, Ferret uses a scoring function to compute how well an assignment vector being evaluated matches external evidence. A scoring function takes as input the probability distribution of the observation node and the external evidence for this node and returns a value between zero and one. A higher value indicates a better match. The score for an assignment vector is the product of scores for individual observations.

The scoring function for the case when an observation node returns an error or receives no response is simple – the score is equal to the probability of the observation node being down. For example, if the assignment vector correctly predicts that the observation node has a high probability of being down, its score will be high.

The scoring function for the case when an observation node returns a response time is computed as follows. The Sherlock agent tracks the history of response times and fits two Gaussian distributions to the historical data, namely $Gaussian_{up}$ and $Gaussian_{troubled}$. For example, the distribution in Figure 1 would be modeled by $Gaussian_{up}$ with a mean response time of 200 ms and $Gaussian_{troubled}$ with a mean response time of 2 s. If the observation node returns a response time t, the score of an assignment vector that predicts the observation node state to be $(p_{up}, p_{troubled}, p_{down})$ is computed as $p_{up} * Prob(t|Gaussian_{up}) + p_{troubled} * Prob(t|Gaussian_{troubled})$. In other words, if the response time t is well explained by the $Gaussian_{up}$ and the assignment vector correctly predicts that the observation node has a high probability of being up, the assignment vector will have a high score.

When Ferret produces a ranked list of assignment vectors for a set of observations, it uses a statistical test to determine if the prediction is sufficiently meaningful to deserve attention. For a set of observations, Ferret computes the score that these observations would arise even if all root causes were up – this is the score of the null hypothesis. Over time, the inference engine obtains the distribution of $Score$(best prediction) $-$ $Score$(null hypothesis). If the score difference between the prediction and the null hypothesis exceeds the median of the above distribution by more than one standard deviation, the prediction is considered significant.

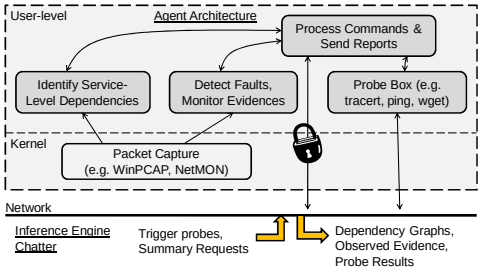


Figure 7: The components of the Sherlock Agent, with arrows showing the flow of information. Block arrows show the interactions with the inference engine, which are described in the text.

5. IMPLEMENTATION

We have implemented the Sherlock Agent, shown in Figure 7, as a user-level service (daemon) in Windows XP. The agent observes ongoing traffic from its host machine, watches for faults, and continuously updates a local version of the service-level dependency graph. The agent uses a WinPcap [20]-based sniffer to capture packets. We augmented the sniffer in several ways to efficiently sniff high volumes of data–even at an offered load of 800 Mbps, the sniffer misses less than 1% of packets. Agents learn the network topology by periodically running traceroutes to the hosts that appear in the local version of the service-level dependency graph. Sherlock would easily accommodate layer-2 topology as well, if it were available. The Agent uses an RPC-style mechanism to communicate with the inference engine. Both agent and inference engine use role-based authentication to validate incoming messages.

The choice of a centralized inference engine makes it easier to aggregate information, but raises scalability concerns about CPU and bandwidth limitations. Back-of-the-envelope calculations show that both requirements are feasible even for large enterprise networks. An Sherlock Agent sends 100B observation reports once every 300s. The inference engine polls each agent for its service-level dependency graph once every 3600s, and for most hosts in the network this graph is less than 40 KB. Even for an extremely large enterprise network with 10^5 Sherlock Agents, this results in an aggregate bandwidth of about 10 Mbps.

The computational complexity of fault localization scales linearly with graph size, so we believe it is feasible even in large networks. Specifically, computational complexity is proportional to the number of root causes in the inference graph $\times$ the graph depth. Graph depth depends on the complexity of network applications, but is less than 10 for all the applications we have studied.

6. EVALUATION

We evaluated our techniques by deploying the Sherlock system in a portion of our organization's enterprise network shown in Figure 8. We monitored 40 servers, 34 routers, 54 IP links and 2 LANs for 3 weeks. Out of approximately 1,500 clients connected to the 2 LANs, we deployed Sherlock agents on 23 of them. In addition to observing ongoing traffic, these agents periodically send requests to the web- and file-servers, mimicking user behavior by browsing webpages, launching searches, and fetching files. We also installed packet sniffers at R1 and 5 routers in the datacenter, enabling us to conduct experiments as if Agents were present on all clients and

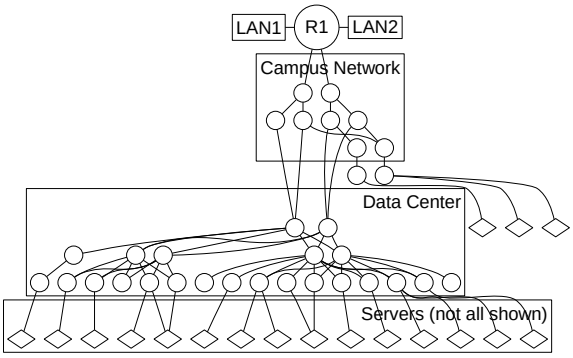


Figure 8: Topology of the production network on which Sherlock was evaluated. Circles indicate routers; diamonds indicate servers; clients are connected to the two LANs shown at the top. Multiple paths exist between hosts and ECMP is used.

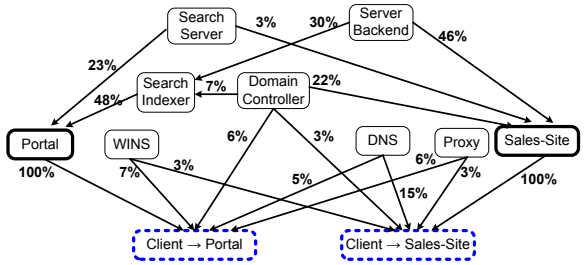


Figure 9: Inferred service dependency graphs for clients accessing the main web portal and the sales website. There is significant overlap in their dependencies.

servers connected to these routers. These servers include our organization's internal web portal, sales website, a major file server, and servers that provide name-resolution and authentication services. Traffic from the clients to the data center was spread across four disjoint paths using Equal Cost Multi-Path routing (ECMP).

In addition to the field deployment, we use both a testbed and simulations to evaluate our techniques in controlled environments (Section 6.2). The testbed and simulations enable us to study Ferret's sensitivity to errors in the Inference Graphs and compare its effectiveness with prior fault localization techniques, including Shrink [6] and SCORE [7].

6.1 Discovering Service Dependencies

We now evaluate Sherlock's algorithm for discovering service-level dependencies and quantify the amount of data and time required for stable results. We carefully examined the service-level dependency graphs computed by Sherlock for fifteen production web and file servers in our organization, and we corroborated the correctness and completeness of these dependencies with our system administrators. Below, we show the dependency graphs for two typical web servers and one file server, and we highlight the lessons we learned.

Figure 9 shows the service-level dependency graphs for visiting our organization's main web portal and sales website. Arrows point from servers that provide essential services to servers or activities that depend on these services. Edges are annotated with weights which represent the strength of the dependencies. Two things are worth noting. First, clients depend on name lookup servers (DNS, WINS), authentication servers (Domain Controller), and proxy servers to access either of these websites. Clients must communicate with the authentication servers to validate certificates

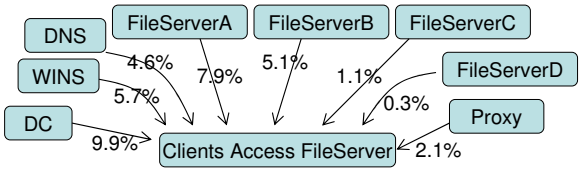


Figure 10: Inferred service dependency graph for clients accessing a file server.

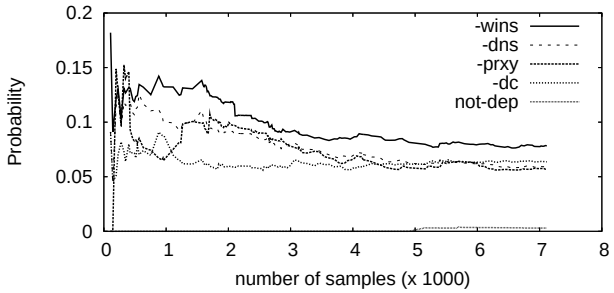


Figure 11: Dependency probabilities for accessing the web portal converge to stable values as the inference engine receives more samples from clients.

that control access and use the proxy servers to retrieve external pages that are embedded in the websites' pages. Second, both websites also share substantial portions of their back-end dependencies. The same search server crawls both websites and generates indexes that are used by the websites to answer client queries. The presence of such overlap in production environments bodes well for our techniques, as it means Sherlock can construct succinct Inference Graphs and Ferret can localize faults with fewer observations.

Figure 10 shows the dependency graph for visiting a major file server. As before, clients depend on DNS, WINS, Domain Controller (DC), and proxy servers to access the file server. Interestingly, clients actually depend on four different file servers – FileServerA-FileServerD to access the main file server. It turns out that the name of the main file server is just the root name of a distributed file system. The actual files are stored on several file servers, each of which is responsible for a portion of the name space. The client requests are sent to the file servers based on the location of the clients and the requested files.

To summarize, our observations are three-fold. First, there is significant variety in service-level dependencies – some servers redirect a majority of their requests while others exclusively serve the requests locally. Second, even when two services appear to have similar dependencies, there are differences in the strength of the dependencies. For instance, clients may heavily depend on domain controllers to access certain web servers which contain lots of sensitive information, but this does not apply to accessing the web portal. Finally, dependencies change over the time – we have seen content move across machines from one building to another. Hence, we conclude that an automated algorithm for inferring dependencies is necessary and useful.

Impact of number of samples: Section 4.1 describes how Sherlock computes service-level dependency graphs by aggregating the results from multiple clients. In this section we examine how many samples are required to produce stable probability estimates. Figure 11 shows how dependency probabilities for clients accessing the web portal converge as the algorithm uses more samples. We show the probabilities for a set of true dependencies and the one

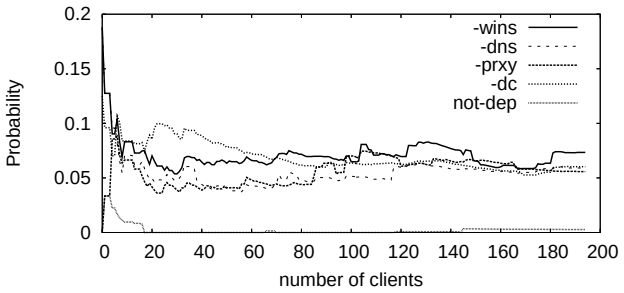


Figure 12: Dependency probabilities for accessing the web portal converge as the inference engine aggregates samples from more clients.

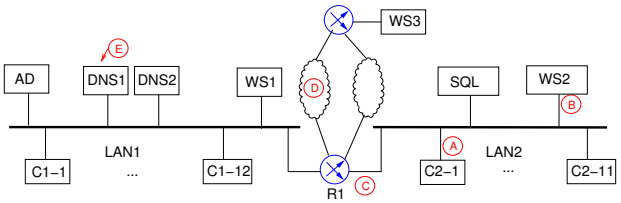


Figure 13: Physical topology of the testbed. Hosts are squares, routers are circles. Example failure points indicated with circled letters. Hosts labeled C* are clients and WS* are webservers.

false dependency with the largest probabilities among false dependencies. Note that the probabilities of the four true dependencies (DC, DNS, WINS, and proxy) quickly exceed those of the false dependency, even with only 200 samples. At about 4,000 samples, the probabilities of all the true dependencies converge to their final values. The Inference Engine normally receives this number of samples in a few hours during a regular day. Once converged, we find the service-level dependencies are stable over several days to a couple of weeks.

Impact of number of clients: Figure 12 shows how dependency probabilities for clients accessing the web portal converge as Sherlock aggregates samples from more clients. We show probabilities for the same set of dependencies as before. Not surprisingly, when we aggregate the results from very few clients, the false dependency has a higher probability than some of the true dependencies. Aggregating over even 20 clients reduces the false dependency probability to a trivial value, showing the importance of aggregation in eliminating false positives.

6.2 Localizing Faults in Enterprise Network

We now turn our attention to Ferret, the fault localization algorithm. We evaluate Ferret's ability to localize faults in an enterprise network and its sensitivity to errors in the inference graph. We also compare it with prior work.

We begin with a simple but illustrative example where we inject faults in our testbed (Figure 13). The testbed has three web servers, one in each of the two LANs and one in the data center. It also has an SQL backend server and supporting DNS and authentication servers (AD). *WebServer*$_1$ only serves local content and *WebServer*$_2$ serves content stored in the SQL database. Note that the testbed shares routers and links with the production enterprise network, so there is substantial real background traffic. We use packet droppers and rate shapers along with CPU and disk load generators to create scenarios where any desired subset of clients,

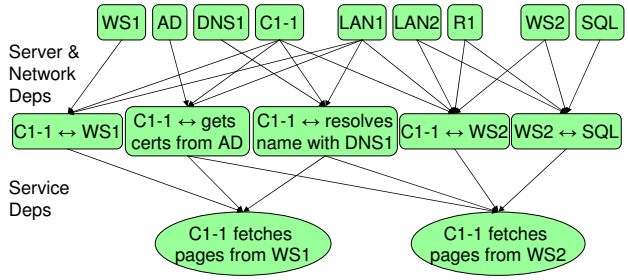


Figure 14: Inference graph for client C_{1-1} accessing *WebServer*$_1$ (*WS*$_1$) and *WebServer*$_2$ (*WS*$_2$). For clarity, we elide the probability on edges, the specialized (failover) metanode for *DNS*$_1$ and *DNS*$_2$, and the activities of other clients.

servers, routers, and links in the testbed appear as failed or overloaded. Specifically, an *overloaded link* drops 5% of packets at random and an *overloaded server* has high CPU and disk utilization.

Figure 14 shows the inference graph constructed by Sherlock, with some details omitted for clarity. The arrows at the bottom-level are the service-level dependencies inferred by our dependency discovery algorithm. For example, to fetch a web page from *WebServer*$_2$, client C_{1-1} has to communicate with *DNS*$_1$ for name resolution and *AD* for certificates. *WebServer*$_2$, in turn, retrieves the content from the SQL database. Sherlock builds the complete inference graph from the service-level dependencies as described in Section 4.2.

Unlike traditional threshold-based fault detection algorithms, Ferret localizes faults by correlating observations from multiple vantage points. To give a concrete example, if *WebServer*$_1$ is overloaded, traditional approaches would rely on instrumentation at the server to raise an alert once the CPU or disk utilization passes a certain threshold. In contrast, Ferret relies on the clients' observations of *WebServer*$_1$'s performance. Since clients do not experience problems accessing *WebServer*$_2$, Ferret can exclude *LAN*$_1$, *LAN*$_2$ and router R_1 from the potentially faulty candidates, which leaves *WebServer*$_1$ as the only candidate to blame. Ferret formalizes this reasoning process into a probabilistic correlation algorithm (described in Section 3.2) and produces a list of suspects ranked by their likelihood of being the root cause. In the above case, the top two root cause suspects are *WebServer*$_1$ with a likelihood of 99.9% and Router R_1 with a likelihood of 9.0*10^{-9}%. Ferret successfully identifies the right root cause while the likelihood of the second best candidate is negligibly small.

Ferret can also deal with multiple simultaneous failures. To illustrate this, we created a scenario where both *WebServer*$_1$ and one of the clients C_{1-3} were overloaded at the same time. In this case, the top two candidates identified by Ferret are *WebServer*$_1 \cap C_{1-3}$ with a likelihood of 97.8% and *WebServer*$_1$ with a likelihood of 1.6%. *WebServer*$_1$ appears by itself as the second best candidate since failure of that one component explains most of the poor performance seen by clients, and the problems C_{1-3} reports with other services might be noise.

Ferret's fault localization capability is also affected by the number of vantage points. For example, in the testbed where *WebServer*$_2$ only serves content in the SQL database, Ferret cannot distinguish between congestion in *WebServer*$_2$ and congestion in the database. Observations from other clients whose activities depend on the database but not *WebServer*$_2$, would resolve the ambiguity.

Ferret's ability to correctly localize failures depends on having observations from roughly the same time period that exercise all paths in the Inference Graph. To estimate the number of observa-

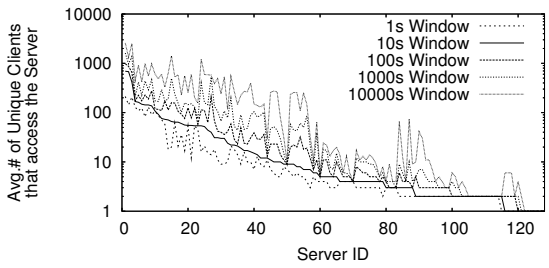


Figure 15: Average number of unique clients accessing the 128 most popular servers in a 10-second time window. The top 20 servers have more than 70 unique clients in every 10 s window.

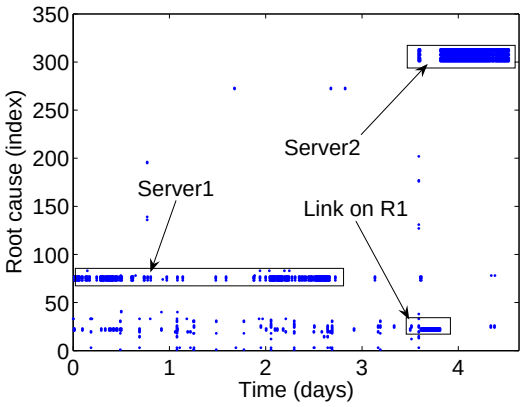


Figure 16: Root causes of performance problems identified by Ferret over a 5-day period. Each Y-axis value represents a separate component in the inference graph and a dot indicates the component is troubled or down at that time.

tions available, we measured the average number of unique clients that access a server during time windows of various sizes. We do this for the 128 most popular servers in our organization using time window lengths varying from 1 second to 10^5 seconds (roughly 3 hours). The data for Figure 15 were collected over a 24-hour period during a normal business day. It shows that there are many unique clients that access the same server in the same time window. For instance, in a time window of 10 seconds, at least 70 unique clients access every one of the top 20 servers. Given that there are only 4 unique paths to the data center and 4-6 DNS/WINS servers, we believe that accesses to the top 20 servers alone provide enough observations to localize faults occurring at most locations in the network. Accesses to less popular services leverage this information, and need only provide enough observations to localize faults in unshared components.

6.2.1 *Evaluation of Field Deployment*

We now report results from deploying Sherlock in our organization's production network. We construct the Inference Graph using the algorithm described in Section 4.2. The resulting graph contains 2,565 nodes and 358 components that can fail independently.

Figure 16 shows the results of running the Sherlock system over a 5-day period. Each Y-axis value represents one component, e.g. a server, a client, a link, or a router, in the inference graph and the X-axis is time. A dot indicates a component is in the troubled or down state at a given time. During the 5 days, Ferret found 1,029 instances of performance problems. In each instance, Ferret returned a list of components ranked by their likelihood of being the root cause. This figure illustrates how Sherlock helps network managers

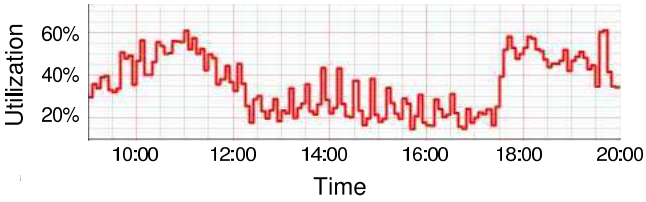


Figure 17: 5-minute averages of link utilization reported by SNMP. Oscillations around 14:00 correspond to observed performance issue.

by highlighting the components that cause user-perceived faults.

By Ferret's computations, 87% of the problems were caused by only 16 components (out of the 358 components that can fail independently). We were able to corroborate the 3 most notable problems marked in the figure with external evidence. The *Server*$_1$ incident was caused by a front-end web server with intermittent but recurring performance issues. In the *Server*$_2$ incident, another web server was having problems accessing its SQL backend. The third incident was due to recurring congestion on a link between R_1 and the rest of the enterprise network. In Figure 16, when Ferret is unable to determine a single root cause due to lack of information, it will provide a list of most likely suspects. For example in the *Server*$_1$ incident, there are 4 dots which represent the web server, the last links to and from the web server, and the router to which the web server is directly connected.

Sherlock can also discover problems that might be overlooked by using traditional threshold-based techniques. For instance, in the *Server*$_2$ incident, both the web server and SQL backend were functioning normally and traditional threshold-based techniques would not raise any alerts. Only requests requiring interaction between the web server and the SQL backend experience poor performance, but this is caught is by Sherlock.

In a fourth incident, some clients were experiencing intermittent poor performance when accessing a web server in the data center while other clients did not report any problem. Ferret identified a suspect link on the path to the data center that was shared by only those clients that experienced poor performance. Figure 17 shows the MRTG [11] data describing the bandwidth utilization of the congested link. Ferret's conclusion on when the link was troubled matches the spikes in link utilization between 12:15 and 17:30. However, an SNMP-based solution would have trouble detecting this performance incident. First, the spikes in the link utilization are always less than 40% of the link capacity. This is common with SNMP counters, since those values are 5-minute averages of the actual utilization and may not reflect instantaneous high link utilization. Second, the 60% utilization at 11:00 and 18:00 did not lead to any user-perceived problems, so there is no threshold setting that catches the problem while avoiding false alarms. Finally, due to scalability issues, administrators are unable to collect relevant SNMP information from all the links that might run into congestion.

6.2.2 *Comparing Sherlock with Prior Approaches*

Sherlock differs from prior fault localization approaches in its use of multi-level inference graph instead of two-level bipartite graph, and its use of probabilistic dependencies. Comparing Sherlock with prior approaches allows us to evaluate the impact of these design decisions.

To perform the comparison, we need a large set of observations for which the actual root causes of the problems are known. Because it is infeasible to create such a set of observations using a

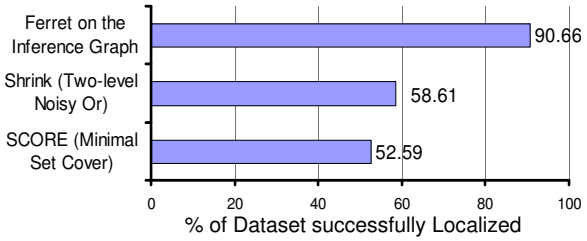


Figure 18: Multi-level probabilistic model allows Ferret to correctly identify 30% more faults than approaches based on two-level probabilistic models (Shrink) or deterministic models (SCORE).

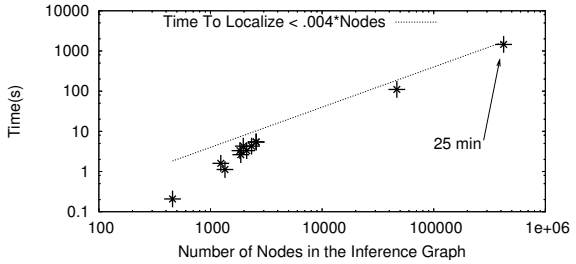


Figure 19: The time taken by Ferret to localize a fault grows linearly with the number of nodes in the Inference Graph.

testbed, we conduct experiments with simulations. We first created a topology and its corresponding inference graph that exactly matches that of the production network. Then we randomly set the state of each root cause to be troubled or down and perform a probabilistic walk through the inference graph to determine the state of all the observation nodes. Repeating this process 1,000 times produced 1,000 sets of observations for which we know the actual root causes. We then compare different techniques on their ability to identify the correct root cause given the 1,000 observation sets.

Figure 18 shows that by using multi-level inference graphs, Ferret is able to correctly identify up to 32% more faults than Shrink, which uses two-level bipartite graphs. Figure 9 and Figure 14 show that multi-level dependencies do exist in real systems, and representing this type of dependency using bipartite graphs does lose important information. SCORE [7] uses a deterministic dependency model in which a dependency either exists or not. For example, the caching of names makes DNS a weak dependency. If such weak dependencies are included, the SCORE model causes many false-positives, yet excluding these dependencies results in false-negatives.

6.2.3 Time to Localize Faults

We now study how long it takes Ferret to localize faults in large enterprise networks. In the following simulations, we use a topology which is the same as the one in our field deployment. We then add more clients and servers to the topology and use the measurement results in Figure 15 to determine the number of unique clients that would access a server in a given time window. The experiments were run on an AMD Athlon 1.8GHz machine with 1.5GB of RAM. Figure 19 shows that the time it takes to localize injected faults grows almost linearly with the number of nodes in the Inference Graph. The running time of Ferret is always less than 4 ms times the number of nodes in the Inference Graph. With an Inference Graph of 500,000 nodes that contains 2,300 clients and 70 servers, it takes Ferret about 24 minutes to localize an injected fault. Note that Ferret is easily parallelizable (see pseudo-code in Algorithm 1) and implementing it on a cluster would significantly reduce the running time.

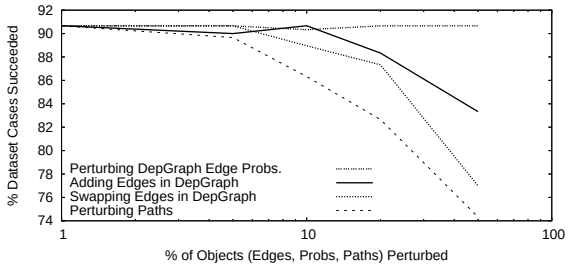


Figure 20: Impact of errors in inference graph on Ferret's ability to localizing faults.

6.2.4 Impact of Errors in Inference Graph

Sometimes, errors are unavoidable when constructing inference graphs. For example, service-level dependency graphs might contain false positives or false negatives. Traceroutes might also report the wrong intermediate routers. To understand how sensitive Ferret is to errors in inference graphs, we compare the results of Ferret on correct inference graphs with those on perturbed inference graphs.

We deliberately introduce four types of perturbation into inference graphs: First, for each observation node in the inference graph, we randomly add a new parent. Second, for each observation node, we randomly swap one of its parents with a different node. Third, for each edge in the inference graph, we randomly change its weight. Fourth, for each network-level path, we randomly add an extra hop or permute its intermediate hops. The first three types of perturbation correspond to errors in service-level dependency graphs and the last type corresponds to errors in traceroutes.

We use the same inference graph as the one in the field deployment and perturb it in the ways that are described above. Figure 20 shows how Ferret behaves in the presence of each type of perturbation. Each point in the figure represents the average of 1,000 experiments. Note that Ferret is reasonably robust to all four types of errors. Even when half the paths/nodes/weights are perturbed, Ferret correctly localizes faults in 74.3% of the cases. Perturbing the edge weights seems to have the least impact while permuting the paths seems to be most harmful.

6.2.5 Modeling Redundancy Techniques

Specialized meta-nodes have important roles modeling load-balancing and redundancy, such as ECMP, NLB, and failover. Without these nodes, the fault localization algorithm may come up with unreasonable explanations for observations reported by clients. To evaluate the impact specialized meta-nodes, we again used the same inference graph as the one in the field deployment. We created 24 failure scenarios where the root cause of each of the failures is a component connected to a specialized meta-node (e.g. a primary DNS server or an ECMP path). We then used Ferret to localize these failures both on inference graphs using specialized meta-nodes and on inference graphs using noisy-max meta-nodes instead of specialized meta-nodes.

In 14 cases where the root cause was a secondary server or a backup path, there is no difference between the two approaches. In the remaining 10 cases where a primary server or path failed, Ferret correctly identified the root cause in all 10 of the cases when using specialized meta-nodes. In contrast, when not using specialized meta-nodes Ferret identified the wrong root cause in 4 cases.

6.3 Summary of Results

The key points of our evaluations are:

- First, we corroborated the inferred service-level dependency graphs of fifteen servers with our administrators and found them to be mostly correct except for a few false-positives. Our algorithm is able to discover service dependencies in a few hours during a normal business day.
- Second, service dependencies vary widely from one server to another and the inference graph of an enterprise network may contain hundreds to thousands of nodes, justifying the need for an automatic approach.
- Third, in a field deployment we show that the Sherlock system is effective at identifying performance problems and narrowing down the root-cause to a small number of suspects. Over a five day period, Sherlock identified over 1,029 performance problems in the network, and narrowed down more than 87% of the blames to just 16 root causes out of the 350 potential ones. We also validated the three most significant outages with external evidence. Further, Sherlock can help localize faults that may be overlooked by using existing approaches.
- Finally, our simulations show that Sherlock is robust to noise in the Inference Graph and its multi-level probabilistic model helps localize faults more accurately than prior approaches that use a two-level probabilistic model.

7. DISCUSSION

To save money and datacenter space, many enterprises are consolidating multiple servers onto a single piece of hardware via virtual machines (VMs). We expect Sherlock techniques to be unaffected by this trend, as most VM technologies (e.g. Xen, VMware, VSS) assign each virtual server its own IP address, with the host machine implementing a virtual Ethernet switch or IP router that multiplexes the VMs to the single physical network interface. To all the algorithms described in this paper, each VM appears as a separate host, with the hosts joined together by a network element.

One source of failures that we have not modeled is the software running on hosts. For example, if a buggy patch were installed on the hosts in a network, it could cause correlated failures among the hosts. Unless the inference graph models this shared dependency on the patch, then blame for the failures will be incorrectly placed on some component that *is* widely shared (e.g., the DNS service). Extending our inference graph to these common failure modes will be an important next step.

Using Sherlock as a research tool, we are now conducting a longitudinal study of the distributed applications used by our organization to determine how many different types of applications exist, whose dependencies we can automatically extract, and whose we cannot. We expect to find convoluted systems and protocols for which Sherlock will not be able to extract the correct dependency graph. However, we are hopeful as this paper has shown Sherlock's success on variety of common application types.

8. CONCLUSIONS

In this paper we describe Sherlock, a system that helps IT administrators localize performance problems across network and services in a timely manner without requiring modifications to existing applications and network components.

In realizing Sherlock, we make three important technical contributions: (1) We introduce a multi-level probabilistic inference model that captures the large sets of relationships between heterogeneous network components in enterprise networks. (2) We devise techniques to automate the construction of the inference graph by using packet traces, traceroute measurements, and network configuration files. (3) We describe an algorithm that uses an Inference Graph to localize the root cause of the network or service problem.

We evaluate our algorithms and mechanisms via testbeds, simulations and field deployment in a large enterprise network. Our key findings are: (1) service dependencies are complicated and continuously evolving over time thus justifying a need for automatic approaches to discovering them. (2) Our service dependency inference algorithm is able to successfully discover dependencies for a wide variety of unmodified services in a timely manner, (3) our fault localization algorithm shows great promise in that it narrows down the root cause of the performance problem to a small number of suspects helping IT administrators in their constant quest to track down frequent user complaints, and finally, (4) comparisons to other state-of-art techniques show that our fault localization algorithm is robust to noise and it localizes performance problems more quickly and accurately.

9. REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP*, Oct. 2003.

[2] W. Aiello, C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. V. der Merwe. Analysis of Communities of Interest in Data Networks. In *PAM*, Mar. 2005.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, Dec. 2004.

[4] M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI'04*, Mar. 2004.

[5] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostic, M. Theimer, and A. Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*, 2004.

[6] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. In *Proc. MineNet Workshop at SIGCOMM*, 2005.

[7] R. R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. IP Fault Localization Via Risk Modeling. In *Proc. of NSDI*, May 2005.

[8] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[9] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet Path Diagnosis. In *SOSP*, Oct. 2003.

[10] Microsoft Operations Manager. http://www.microsoft.com/mom/.

[11] Multi Router Traffic Grapher. http://www.mrtg.com/.

[12] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Uncertainity in Artificial Intelligence*, 1999.

[13] HP Openview. http://www.openview.hp.com/.

[14] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A First Look at Modern Enterprise Traffic. In *IMC*, Oct. 2005.

[15] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[16] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-box Performance Debugging for Wide-area Systems. In *WWW*, May 2006.

[17] I. Rish, M. Brodie, and S. Ma. Efficient Fault Diagnosis Using Probing. In *AAAI Spring Symposium on Information Refinement and Revision for Decision Making*, March 2002.

[18] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving Accuracy in End-to-end Packet Loss Measurement. In *SIGCOMM*, 2005.

[19] IBM Tivoli. http://www.ibm.com/software/tivoli/.

[20] http://www.winpcap.org/.

[21] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High Speed and Robust Event Correlation. In *IEEE Communications Magazine*, 1996.