Constellation: automated discovery of service and host dependencies in networked systems

Paul Barham, Richard Black, Moises Goldszmidt, Rebecca Isaacs,
John MacCormick, Richard Mortier, Aleksandr Simma*
Microsoft Research

ABSTRACT

In a modern enterprise network of any scale, dependencies between hosts, protocols and network services are surprisingly complex, typically undocumented, and rarely static. Even though network management and troubleshooting rely on this information, automated discovery and monitoring of these dependencies remains an unsolved problem. The approach we describe in this paper attempts to close this gap by proactively inferring a network-wide map of these complex relationships using innovative machine learning techniques.

Constellation takes a black-box approach to learn explicit models of time dependencies using little more than the timings of packet transmission and reception. The parameters of these models are automatically fitted, enabling Constellation to be robust to a variety of real-world traffic behaviours. Statistical hypothesis testing on the models provides a guaranteed confidence level for the accuracy of the result. We present promising results from a prototype implementation using substantial packet traces from Microsoft's corporate network. We also discuss our experience in applying simpler statistical tests and machine learning approaches, including why they failed to perform.

1. INTRODUCTION

Shared network services enable great functional richness and flexibility for distributed applications. As a result apparently simple facilities, such as remote file sharing and email, invariably rely on multiple network services ranging from directory functions to authentication. In a large-scale deployment the appealing modularity and component reuse of shared network services leads to a highly complex system that is extremely difficult for a network operator or a user to fully understand. As well as rendering effective management problematic, the size and sophistication of networked systems often leads to security vulnerabilities, inefficient troubleshooting and anomaly detection, and ultimately user frustration. This effect was noted by Lamport in his famous quote '*a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable*' [13].

Current debugging capabilities for deployed networked systems are unsatisfactory. Individual applications may provide the ability to understand their own behaviour; a few simple general-purpose tools like *traceroute*, *tcpdump* and *ping* exist, while some systems make a variety of data available through SNMP. Analysis tools have been proposed to address this problem using operating system logging mechanisms or using intrusive instrumentation to explicitly track the causal paths of individual requests [3, 6]. However, there is still a dearth of tools that enable users, operators and developers to understand the interactions between the systems they come into direct contact with and those not within their immediate view that are part of the infrastructure.

In addition, the nature of the dependencies in these networks can vary greatly: there may be many servers for each service or many services provided by a single server. A host can assume the roles of both server and client for different instances of the same service. A client might invoke a single service variously on different servers. A computer may unwittingly have a transitive dependence upon other computers, and so on. Furthermore, these relationships are volatile, changing as applications adapt to runtime conditions such as machine load variations or link failure.

This paper presents *Constellation*, a tool that automates discovery of dependencies between hosts, protocols and network services, and proactively infers a network-wide map of these complex relationships. Figure 1 presents a small portion of such a graph, which we term a *constellation*. The machine "desktop" (doubly-circled in the figure) is the root of the constellation. The other nodes represent servers on which the root depends (either directly or indirectly), while the edges reflect the corresponding services that cause dependency. Constellation builds these dependency graphs using innovative machine learning techniques. Its lightweight, black-box approach allows deployment in a network regardless of the heterogeneity of host operating systems and applications.

Constellation has some similarities to Project5 [1], Sherlock [2], and WAP5 [17]. While those systems focus on fault localization or performance debugging of the network as a

*John is now with Dickinson College, PA. Work done while a Researcher with Microsoft Research. Aleks is with the University of California, Berkeley, CA. Work done while an intern with Microsoft Research.

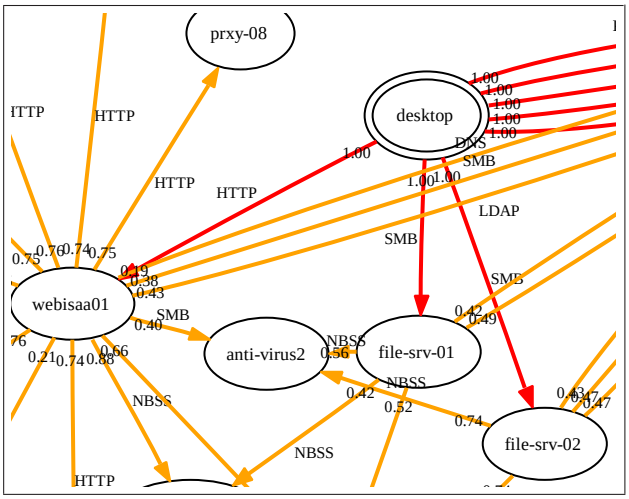


Figure 1: Small fragment of a constellation depicting the transitive dependencies of a single computer in an enterprise network.

whole, Constellation's goal is to build the map of network dependencies for an individual host.

One of the important contributions of this work is that Constellation relies on statistical hypothesis testing and false discovery rates [4, 21] to provide guarantees on the confidence of the dependencies induced from the data. This is especially critical in this domain where ground-truth is next to impossible to obtain, and indeed the provision of confidence guarantees has been posed as a challenge for the application of machine learning techniques to (distributed) systems [9].

We believe Constellation to be invaluable for a variety of higher-level activities. These include troubleshooting and diagnosis tasks, such as to search for a reported problem of unspecified provenance, as well as maintenance—providing valuable usage data for network reconfiguration planning (for example to identify those users likely to be impacted by a reconfiguration), and for detecting unexpected changes by spotting deviations from "last known good" constellations. Our current and future work is directed to building on Constellation in order to realize this vision in its entirety.

1.1 Paper outline

The Constellation system comprises two parts, a per-host model that captures the local interactions between the different services, and a recursive algorithm that uses the local models to build a network-wide constellation of dependent computers and services.

The model contains a list of the dependencies between output packet streams and input packet streams of different services, and is generated by means of a *dependency test*. In Section 2 we describe the probabilistic model that we developed for Constellation, called *Continuous Time NoisyOR*, as well as the hypothesis test that is used to decide dependency. We also present a formalization of the intuitively attractive statistical approaches that are commonly adopted, for example by the Sherlock system [2], and explain the benefits of the more complex approach used in Constellation.

To ensure that Constellation works well under real network conditions, which include packet burstiness, connection multiplexing, and variable timescales of interest for different applications, we have developed the system against a substantial packet trace from the Microsoft corporate network. In Section 3 we demonstrate good results for the Constellation test in a comparative evaluation against other techniques using "ground-truth" extracted from the network trace.

In Section 4 we describe the recursive algorithm for building constellations, as well as discussing aggregation, which is a particularly important, yet subtle, aspect of a practical system.

Constellation is a tool for learning information about network behaviour that is otherwise hard to extract. It provides the underlying functionality for higher-level tools that might use this dependency information in a myriad of ways. We explore what kind of information is revealed in a qualitative evaluation in Section 5, which includes an examination what is shown about our network by Constellation for three common tasks: browsing, email and printing.

Implementation and deployment issues are covered in Section 6, and we review related work in Section 7.

2. DEPENDENCY TESTS

Constellation is intended to be a lightweight, non-intrusive and generic tool and so must not rely on deep packet inspection or understanding of application-level semantics (indeed this is increasingly becoming a necessity with increasing proportions of encrypted traffic). Given this black-box requirement, the service associated with a channel must be identified using just standard fields in the packet headers and the correlation test must uncover evidence for interdependency between channels using only packet timestamps.

A natural approach to discovering dependent channels is to use probabilistic models to represent the inherent uncertainty and statistical tests to estimate the confidence on our inferences. The probabilistic model we developed for Constellation is known as *Continuous Time NoisyOR (CT-NOR)*, and is described more fully in Section 2.3. CT-NOR is a non-trivial extension of the Noisy-Or model [16] to deal first with continuous time, second with the fact that activity in channels is relatively rare, and third to incorporate functions representing the expected timing relations between input and output.

Previous work on the identification of correlated network traffic [2, 17] has used various approaches based on "co-occurrence" tests. The key difference between co-occurrence tests and those based on probabilistic models, is that co-occurrence conducts a *pairwise* test for dependence on each input/output channel pair, while the probabilistic approach considers *all input channels at once* to choose the best candidates for dependence with the output channel.

Direction	Service	Peer	Description
OUT	HTTP	WebServer1	HTTP requests
OUT	DNS	DNSServer	DNS requests
IN	SMB	MyLaptop	Desktop file browsing
IN	HTTP	WebServer1	HTTP responses
...	...	...	...

Table 1: Examples of channels on a host.

2.1 What is a dependency?

To establish a common vocabulary we first introduce a specific notion of a *channel* that we will use throughout the paper. A *channel* at a given host is a unidirectional flow of network packets sent or received by that host, identified by a direction (IN or OUT), the remote peer, and a service. Here, a service could be a protocol, application or network service, frequently determined in practice from the well-known TCP/UDP port number, or other information in the packet header. See Table 1 for examples.

The next definition we need is that of a *dependency* between channels or services. This definition is tied up to a period of time T of observation:

DEFINITION 1. *Let channel* A *be an input channel to some host, and channel* B *an output channel from the same host. Channel* B *is* dependent *on channel* A *in period* T *if packets in* A *caused packets in* B *during the period* T.

A few comments are in order. A rigorous definition of causality lies in the realm of philosophy and we do not attempt one here. Rather, we assert that the vast majority of experienced computer systems practitioners can almost always agree on whether a given packet "caused" another packet. For example, a packet sent in response to some request is "caused" by that request; the evaluation section gives numerous additional examples.

Also note that most systems administrators will have a different, and much stronger notion of dependency, namely that *if A depends on B then A won't work if B is broken*. Although this would be extremely desirable information to have, it is impossible to determine from passive analysis of a working system—we cannot hope to identify hot standbys and fault-tolerant services unless we observe them in use.

Finally it is worth mentioning the difference between correlation and causation. If two services are co-located on a busy server it is quite possible that scheduling behaviour will introduce timing *correlations* between their responses to clients. It is not correct to say that service one depends on service two, but an outside observer, such as the Constellation system, may not be able to tell the difference between the two scenarios.

2.2 Co-occurrence tests

An extremely simple test for whether channels are dependent can be performed by counting the number of times a packet of the input channel is observed within a small time window prior to a packet on the output channel. We can estimate the number of times such a *co-occurrence* might be expected to happen by chance on channels that are independent, and if the observed value is significantly above this threshold then the channels are assumed to be dependent

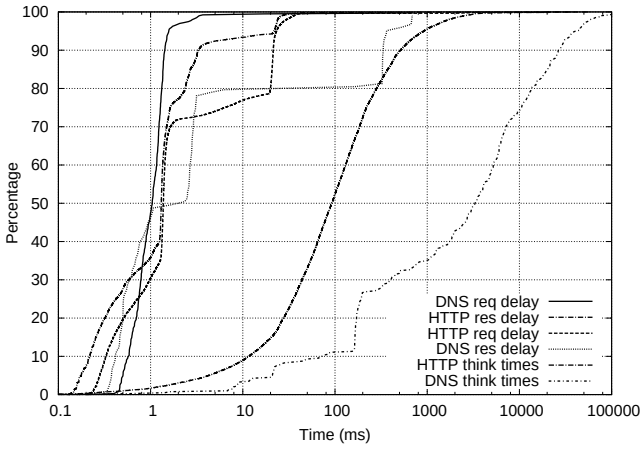


Figure 2: CDFs of forwarding delays and think times

Simple co-occurrence

A naïve way to compute the threshold is to assume that N_i input packets are uniformly distributed over interval T with mean inter-arrival time $IAT_i = T/N_i$. For a given window size of W, the probability of an accidental collision is approximately $P_{coll} = W/IAT_i$, (where W is much less than IAT_i), and hence the expected number of co-occurrences is:

$$E(C_i^o) = \frac{N_o \times W}{IAT_i} \quad (1)$$

where N_o is the number of output packets.

Sherlock[2] uses this simple pairwise co-occurrence test with a fixed window size of 10ms, and defines dependent channels as those where the fraction of co-occurrences is "*much greater than the likelihood of chance co-occurrence*".

Window sizes

It is immediately apparent that choosing the fixed window size parameter W of the above test is extremely important. This parameter is related to the service time distribution between an input and a subsequent output. In general we would expect this to be in the millisecond range, and analysis of our traces to extract service time distributions of an HTTP proxy server and a DNS server show this to be the case (Figure 2). However, there are also situations, such as submitting a job to a print spooler, where service times can be several seconds. A simple co-occurrence test with a fixed size window is unlikely to do well in both regimes. WAP5[17] uses a variant of co-occurrence without a window and attempts to fit causal delays using an exponential or gamma distribution.

The effects of burstiness

Of course, real network traffic is often bursty and in our traces the assumption of uniformity has proven to be extremely unreliable, especially on busy servers where the *mean* inter-arrival time can be much less than 10ms due to long bursts of back-to-back packets.

One heuristic which can mitigate the problems with simple co-occurrence testing is to low-pass filter the packet trace, i.e. to discard packets which occur within a short time window of the previous packet on the same channel. This will tend to discard packets from the middle of bursts and keep those at the front of bursts. This effectiveness of this heuristic is evaluated in Section 6.3.

We can explicitly take bursts into account when estimating the probability of an accidental collision using the following:

$$P_{coll} = \frac{\sum_{i=1}^{N} min(I_i - I_{i-1}, W)}{T} \quad (2)$$

Intuitively, this computes the fraction of the trace where a randomly chosen instant would lie within W of the nearest input packet, where I_i represents the time of input packet i.

Binomial confidence limit

Given an observed number of co-occurrences C_i^o we would like to compute whether the observed value is significantly above the expected number of chance collisions. If we assume that each output packet has an independent probability P_{coll} of lying within W of an input packet, and let the number of output packets be N_o, then the observed number of accidental collisions of two independent channels would be drawn from a Binomial distribution $B(P_{coll}, N_o)$ with mean $N_o \times P_{coll}$ and variance $N_o \times P_{coll} \times (1 - P_{coll})$.

The significance of the observation can be estimated by summing the tail of the Binomial distribution using the incomplete beta function:

$$P = \text{betai}(1 - P_{coll}, N_o - C_i^o, N_o + 1) \quad (3)$$

We can now use a threshold of $P > 0.95$ to accept a false positive rate of 5%. For sufficient samples, the binomial can be approximated by a Normal distribution and we can use a decision based on the number of standard deviations above the mean.

Unambiguous co-occurrence

On busy servers it is common that more than one input channel is active within a short time window of any given output packet. This makes it difficult to disambiguate the potential causes of the output packet. However, due to the bursty nature of most network traffic, there are invariably brief quiet periods where it is easier to observe genuine co-occurrences (i.e., those indicative of dependent channels).

The *unambiguous* co-occurrence test counts how often a given input channel is the only input active during the window W prior to each output packet. Although the number of such events is much smaller than with standard co-occurrence, each of these observations provides greater evidence of dependence between the channels. Indeed the unambiguous co-occurrence test can be thought of as a cheap version of the hypothesis test we describe in Section 2.3, where we compare a full model to a model without a specific input channel.[1]

In our ground-truth experiments, this technique proved to have very few false positives, but does have the drawback that is is difficult to derive an analytic confidence threshold, necessary for when we don't have ground truth, and it fails in cases such as the printer when we must take into account interactions over a 2 second window.

2.3 The Continuous Time Noisy-OR model

CT-NOR is based on the Noisy-Or model for representing causality under uncertain conditions [16] and generalizes this standard model in three ways: i) it models events in continuous time, ii) it models the fact that there are relatively long periods of inactivity, and iii) it incorporates explicit functions of time, modelling the time delays between inputs and outputs.

CT-NOR considers a single output channel on a given host, and simultaneously analyzes all of this host's input channels to determine which channel(s) best help to explain the output packets. It assumes there is some fixed but unknown delay function $f(t)$, specifying the probability distribution of the delay between an input and the output(s) it causes. For example, if $f(t)$ happened to be the uniform distribution $U_W(t)$ on the window $[0,W]$, then the delay between an input packet and the resulting output(s) is equally likely to be any quantity in $[0,W]$. This is exactly how the co-occurrence test models the delay between input and output. CT-NOR further assumes that the inputs from channel j produce, on average, some fixed but unknown number p^j of output packets. Readers familiar with stochastic processes will recognize the resulting model is a non-homogeneous Poisson process [10]: if the kth input packet in channel j occurs at time t_k^j, the density of output packets generated by this input is given by

$$p_k^j(t) = p^j f(t - t_k^j). \quad (4)$$

Our experiments determined two particularly useful models for the delay function f: one, f_G, is a mixture of the uniform "spike" distribution U_W and a Gaussian distribution $G_{\mu,\sigma}$ with mean μ and standard deviation σ; the other, f_E, is a mixture of the uniform and the exponential distribution E_λ with parameter λ. As shown visually in Figure 3, the Gaussian delay distribution extends the intuitions of co-occurrence within a window to also capture dependencies that can be relatively far away in time (such as with the printer in Section 5.4). The exponential distribution captures

[1] It is also possible to justify the unambiguous co-occurrence approach from a Bayesian perspective, but we do not pursue this further in this paper.

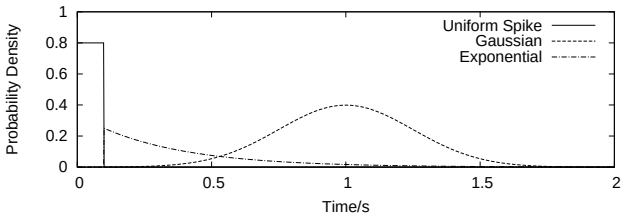


Figure 3: Example CT-NOR parametric delay distributions

the intuition that the possibility of dependence decays as the packets are further away in time. The equations for these functions are:

$$f_G(t) = \alpha U_W(t) + (1-\alpha) G_{\mu,\sigma}(t) \quad (5)$$

$$f_E(t) = \alpha U_W(t) + (1-\alpha) E_{\lambda}(t) \quad (6)$$

A key feature of the CT-NOR model is that it *simultaneously* estimates the p^j in (4) and the delay function parameters (α, W, μ, σ or α, W, λ) in (5) or (6). This is done by the standard statistical technique of "maximum likelihood estimation", and results in a likelihood value L. The likelihood is essentially the probability that a given model (complete with instantiated parameter values) would generate the specific data (input/output packets) observed. Similarly, by comparing likelihoods we are able to choose the delay function which best fits the data.

Note that CT-NOR has to overcome a chicken and egg problem: if we knew the delay function parameters, we could easily determine the probability that each input caused each output, and hence the p^j values in (4); whereas if we knew the p^j it would be easy to estimate the delay function parameters. CT-NOR solves this problem by employing an iterative algorithm similar to Expectation Maximization, an algorithm well known in statistics and machine learning [5, 19]. Full technical details of the CT-NOR model will be described in a separate publication.

Hypothesis Test

To transform the CT-NOR probabilistic model into a test, we need a decision procedure for determining whether an output channel is dependent on an input channel. In this work we take a hypothesis testing approach from classical statistics. To decide whether output channel B, is dependent on input channel A, we compare the likelihood of a CT-NOR model of B to one in which we purposefully remove the input channel A. The null hypothesis, H_0, is that the two have the same power in explaining the output packets. If we are able to reject H_0, then the channels are labeled as dependent, since the absence of the input channel A renders the second model less powerful (in explaining the output packets observed in channel B). It turns out that twice the difference in the log-likelihood in these two models behaves as a χ^2 distribution with one degree of freedom (the missing channel). From the resulting χ^2-value, as in any standard statistical test, one obtains a "p-value" which gives (roughly speaking) the probability that a more extreme result would have occurred if the null hypothesis were true. To decide whether two channels are dependent, this p-value is thresholded, and the threshold used depends on the confidence required for the task at hand. There is, however, a subtlety with applying these p-values directly in our domain because we perform *multiple* hypothesis tests (see [21, 4]). This is discussed further in Section 3.3, where an appropriate procedure is presented.

2.4 Other approaches investigated

We explored several other approaches to dependency testing with mixed results, but do not have space in this paper to describe these in detail.

One promising technique was based on a pairwise statistical test between an incoming channel A and an outgoing channel B. The intuition was that if channel B is directly dependent on channel A, then the timing relationship between the input packets in A and the output packets in B should differ significantly from the time relationship between A and a synthetic channel containing randomly distributed packets.

We explored the use of various standard hypothesis tests such as Kolmogorov-Smirnov and χ^2 to establish the difference between the distributions, and also formulated the problem as a Bayesian log-odds model selection [5]. We faced numerous problems due to inadequate representation accuracy of the distributions, but the main reason for the lack of performance is the myopic nature of the pairwise comparison which gets easily confused when ignoring the other channels.

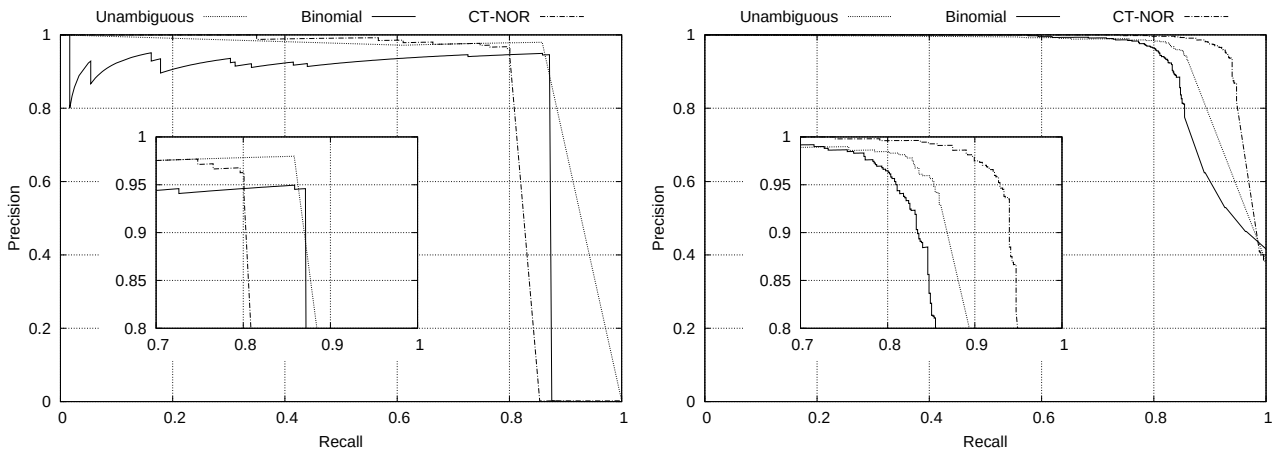
A second method rejected relatively early in the project was to divide the trace into small time slots and use Bayesian classifiers [8] to capture patterns of simultaneous input and output activity. Our experiments showed the patterns were hard to capture for a classifier since most channels are active in only a small number of slots and therefore the classifier learns that predicticing inactivity is almost always correct.

3. EXPERIMENTAL VALIDATION

All the experimental results for Constellation are obtained using a trace comprising headers and partial payloads of around 13 billion packets collected over a 3.5 week period in 2005 at Microsoft Research in Cambridge. Although the laboratory, which has approximately 500 networked machines, is physically distant from company headquarters, it is fully within Microsoft's global corporate network and so runs the same services and applications under the same global policy as the rest of the company. The trace covers every packet sent or received by the 500+ nodes on site and captures conversations with over 28000 off-site IP addresses.

3.1 Ground-truth datasets

In the field of machine learning, "ground-truth" is used to refer to those situations where we know the "correct" answers. The ground-truth for dependencies in a real network



(a) Precision-recall curves for dependency tests on DNS data.

(b) Precision-recall curves for dependency tests on HTTP data.

Figure 4: Ground-truth experimental results.

Dataset	Tot pkts	Proto%	Clnts	Svrs	Cache	Fwd	**Rate**
DNS	377,767	3.5%	600	148	91%	113	0.03/s
HTTP	473,291	26%	93	9	29%	4246	1.18/s

Table 2: Characteristics of the two ground-truth datasets

is, in general, quite hard to extract. Relationships are often hidden inside the configuration files or source code of individual applications, and caching and load balancing can further obscure the actual underlying dependency. Fortunately, forwarding servers for a well-understood protocol can provide an exception to this situation, because received packets can be matched with transmitted packets by simple inspection of the payload. In this section we present an evaluation of the dependency tests using two contrasting traffic types that we have been able to label: forwarded DNS traffic at the lowest level of a DNS server hierarchy, and forwarded HTTP traffic at a caching proxy server. We apply three dependency tests—CT-NOR, binomial co-occurrence and unambiguous co-occurrence—to the two ground-truth data sets.

For this experimental evaluation we use the same one hour period of the trace (10-11am on a Tuesday) throughout. DNS ground-truth was extracted by deep inspection of DNS packets forwarded by the on-site Domain Controller (DC), which is the first-hop DNS server for local machines. When the DC receives a DNS query from a client, it either forwards the request to an upstream DNS server, or else responds directly to the client if the result is cached locally. Similarly HTTP ground-truth was extracted from the on-site web proxy server, which can either respond from its own cache or forward the request to one of an array of 9 upstream proxies.

Because the web proxy performs load-balancing across all machines in the proxy bank, to avoid the uninteresting situation where every client is dependent on each of the upstream proxies (i.e. it would be impossible to make a mistake), we filtered the HTTP traffic by selectively dropping packets so each client uses no more than 4 or 5 of the proxies. The DNS ground-truth data set remained unchanged from the original trace. Table 2 contains summary statistics of the resulting ground-truth data sets, including total traffic volume at the server, percentage of traffic contributed by this protocol, the number of client and server peers, the percentage of requests that are served from the cache or forwarded upstream, and the resulting rate of forwarded requests.

It can be seen that the two datasets have significantly different characteristics—compared with the HTTP traffic, forwarded DNS traffic is infrequent and a small fraction of the total workload. This is also visible in the earlier Figure 2 which showed their processing delay, burstiness and inter-arrival time distributions. DNS queries tend to be single packets, whilst HTTP responses can frequently generate thousands of packets. It is important that a dependency test works well in both these regimes.

3.2 Precision-recall

The majority of the dependency tests described in Section 2 generate a p-value, or confidence that a particular input and output are related. Comparing this score against a fixed threshold allows the user to arbitrarily trade off the accuracy of decisions against the number of dependencies identified. To compare the effectiveness of the various tests we borrow a technique from the field of information retrieval. For each of the tests we plot their *precision* (the fraction of answers returned which are correct) against their *recall* (the fraction of all correct answers which were returned) as we vary the threshold from zero to infinity.

It is clear that DNS traffic, with low volume and large request inter-arrival-times, is well-suited to the co-occurrence tests, particularly for requests where almost 100% have a servicing delay on the DC of less than 10ms. The HTTP traffic is more challenging for any dependency test, with a higher request rate, more packet burstiness and greater vari-

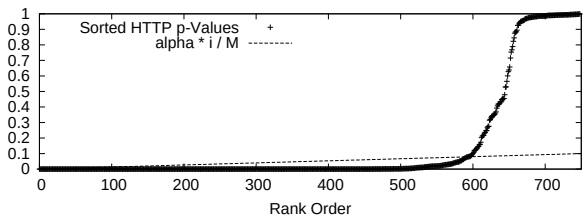


Figure 5: FDR threshold for HTTP dataset

ability in the delays due to servicing times on the proxy.

For the two co-occurrence tests we configured a window size of 10ms,[2] as used in previous systems, whilst CT-NOR was allowed to fit appropriate an delay distribution for the observed traffic. Results are shown using the precision-recall curves in Figure 4. For DNS the CT-NOR test has a lower $max(precision + recall)$ than the two co-occurrence tests, but it is notable that binomial co-occurrence suffers from a lower precision throughout. For the HTTP dataset, CT-NOR exhibits both better precision and recall, handling the higher volume of traffic and response burstiness much better.

3.3 False discovery rates

When deployed for real, Constellation does not have the luxury of data labelled with accurate ground-truth, thus we need to be able to compute the confidence on the decisions made by the algorithms proposed. In traditional statistics, it is customary to treat the decision as a hypothesis test and threshold the p-value with a significance level: this is the probability, under the conditions of H_0, that a case will be falsely rejected [21].

Unfortunately, while the p-value tells us something about the probability that a specific case will be falsely rejected, when we are performing a large number of tests this number becomes less relevant. The reason for this is best understood through a concrete example. In analyzing the HTTP dataset we must perform 1692 hypothesis tests of which we know that 646 are dependent and 1046 are non-dependent. Suppose that our hypothesis test rejects a number equal to the number of genuinely dependent cases, namely 646. With a p-value threshold of 5%, on average 52 of these (5% of 1046) will be incorrect decisions—i.e. around 10% of our decisions are incorrect. In reality, what we care about is the following: *amongst the cases that my tests say are dependent, what is the proportion of incorrect decisions?* This is known as the False Discovery Rate (FDR).

Fortunately, statisticians have been confronting this problem recently in the domain of genomics and have come up with various procedures for dealing with large numbers of hypothesis tests [4, 18]. In this paper we follow the method described in [4] which works as follows: given a desired FDR rate α and the total number of tests M, sort the array of p-values in increasing order, find the largest index i such that $p[i] \leq \alpha \times i/M$, and use $p[i]$ as the threshold. This corresponds to finding the intersection of the two lines shown in Figure 5. For the case of HTTP where we have ground-truth, and we can compute the actual values of FDR and p-values, we can see in Table 3 that the model fitted with CT-NOR is quite good. Both the actual rate of false positives and the actual rate from the pool of independent channels flagged as being dependent are below the estimated ones.

[2]This value was determined experimentally to be optimal for this data, and agrees with the value predicted by Figure 2.

Target FDR	%false +ves	p-value cutoff	actual FDR
5%	3.3%	2.3%	1.34%
10%	5.21%	4.2%	2.03%

Table 3: Estimated p-values and FDRs versus actuals for HTTP dataset.

Using the p-values from the CT-NOR test in combination with the FDR procedure, we can now compute suitable thresholds on each host for rendering constellations, with guarantees (in expectation) on the number of false dependencies that will be included.

4. BUILDING CONSTELLATIONS

Section 2 presented several techniques for discovering dependencies between the requests made of one machine and the requests it makes of others. While such per-machine information alone is often useful, Constellation also provides a recursive algorithm that connects these local models, transitively following selected dependencies from machine to machine. In this section we describe this algorithm, which has two forms: the basic version to build constellations that show what is caused by an activity, and a variant for building constellations that show what caused an activity.

Constellations can be constructed over various timescales and with different degrees of channel aggregation. The choices for these parameters are dictated by how a particular constellation is intended to be used. For example, a host's constellation for a 15 minute period will act as a record of specific dependencies that existed in that quarter hour, which might be useful information for troubleshooting. On the other hand, the appropriate timescale over which to build constellations when planning a network reconfiguration may well be a full day or even a week. Similarly, channel aggregation has a significant effect on the information captured by a constellation, as well as implications for runtime performance of the algorithm. We discuss aggregation in some detail in Section 4.3.

4.1 Algorithm

Building a constellation involves a breadth-first transitive-closure across the local models of each machine to construct the dependency graph. The core step considers every output request of interest from the current machine M to a target machine T. This output request from M is transformed to the corresponding input request at T, and the local model at T

used to add the list of its correlated output requests to the work-set of requests to evaluate. Additionally, the output request from M is also inverted at M to determine any correlated input responses, and M's local model then consulted to discover any additional dependencies arising from considering these input responses.

The work-set is seeded with a set of output requests from M, and the algorithm is applied breadth-first to the work-set until it is empty, or some pre-defined maximum depth or size is reached. The output of the algorithm is the set of input-to-output dependencies traversed, representing the constellation of the original seed requests. Such a constellation can then be presented in graphical form, for example as seen in Figure 6.

4.2 Causal constellations

The algorithm above builds a constellation that describes *what was caused by* a set of requests originating from a particular host. It is also useful to be able to run the core step of the algorithm backward on a single machine model to determine *what caused* a request, and thus those services on which a particular request depended.

To do so, the core step takes an output channel at a machine and examines the local model for input channels correlated to that output. These input channels are plausible causes of the output: input requests indicate that activity on some other machine caused the output at this machine, while input responses indicate that the output is actually the result of earlier activity at this machine. Using the machine's local dependency map, input responses can be mapped to the local output request that caused them, resulting in a constellation which represents all possible transitive causes for the initial seed output.

Causal constellating is most useful for seeding the work-set of the standard constellation algorithm. For example, suppose we wish to find the constellation for HTTP requests from some client computer C. We first start by computing the causal constellation locally at C to find the other output channels which are causally required by C before it can send HTTP requests (such as DNS). Then we use these as the seed work-set to determine the transitive dependencies at other machines. If we simply looked at the transitive dependencies arising from the HTTP output request alone, we would not discover the many other services and servers actually required to browse the web.

4.3 Aggregation

Aggregation is an important and subtle part of the system, which is best introduced by comparing two scenarios, the first in which aggregation is desirable and the second in which it is not.

First, suppose a large set of clients send HTTP requests to a caching web proxy which in turn spreads cache misses over an array of upstream proxies. It is possible to track dependencies for individual clients through the proxy, showing, for example, that client C1 was dependent on servers S2 and S4, whereas C2 was dependent on S1 and S7. However in many likely uses of a constellation the important information would be that clients were dependent on all the upstream servers, since a request from any client might be sent to any server. That is, we wish to see that, in aggregate, clients dependent on the proxy have a dependency on all the upstream servers.

In contrast, imagine a mail hub that sends IDENT[12] callbacks to clients submitting mail.[3] Aggregating as suggested above would show the mail hub's SMTP service to be dependent on the IDENT services of very many client computers. However, the only information relevant for the constellation of a particular client computer is the dependence of SMTP on *its own* IDENT service: the aggregation of clients here is misleading.

Constellation supports three different types of aggregation that can be carried out on packet events at a single machine. *Port aggregation* aggregates into single channels those packets with identical IP addresses and service port numbers but distinct client port numbers. This unifies, for example, the several HTTP connections opened by a browser to a single server when downloading the multiple components of a single page. We apply port aggregation throughout our results as we have always found it useful to do so (very little useful information is incorporated into the constellation by distinguishing client port numbers).

Client aggregation aggregates into a single channel those packets representing input requests and output responses with identical service ports across multiple clients. This is helpful for addressing questions such as "on what does a typical client depend for web browsing?". *Service aggregation* aggregates *all* packets with the same service port, and this addresses questions such as "what are the dependencies among services in my large datacenter?", where each service may be provided by hundreds of servers, some of which may provide more than a single service.

Note that the implementation of the constellation building algorithm handles aggregated channels slightly differently. For example, it must deal with cases such as the output request channel of one client transforming to the input request channel from an aggregated set of clients at the server.

One interesting line of future work is to develop automated aggregation strategies by clustering channels using characteristics such as the expected number of packets that each input channel causes on the output channel. This would avoid the situation of inadvertently aggregating channels incorrectly, as described above, and enable different views of constellations that are customisable depending on the monitoring or diagnostic task being pursued. Another is that aggregation may also be carried out on the scores that are output from dependency testing at a single machine, and also

[3]The precise protocols described here are not present in our Windows environment but they provide a simple and realistic example of the issue we observed.

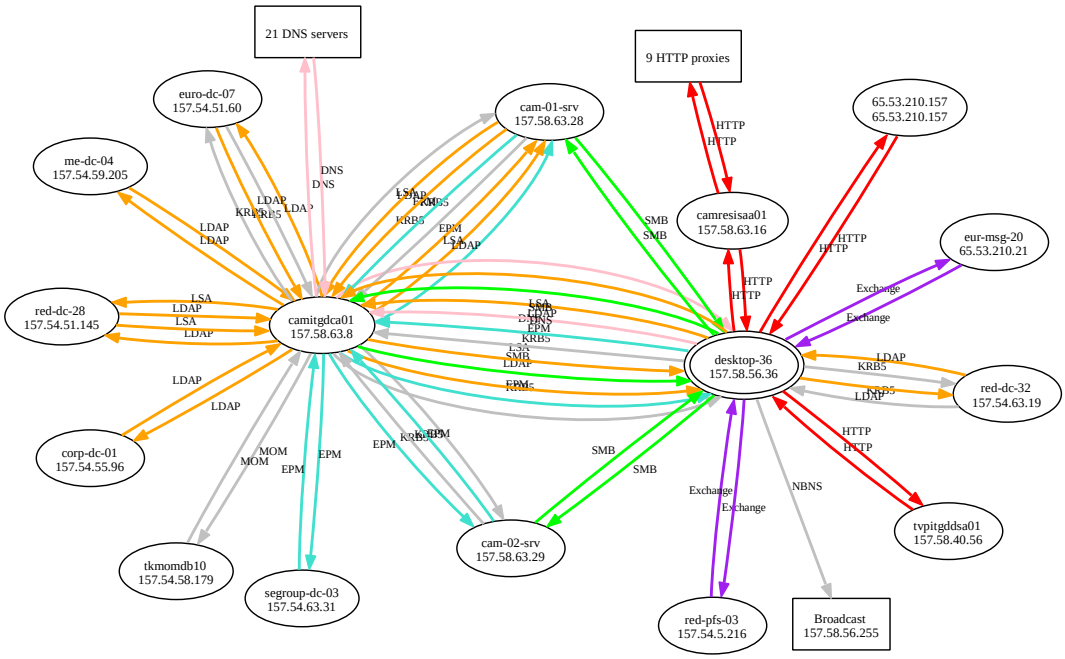


Figure 6: A constellation rooted at the host desktop-36 highlights the large number of services invoked on the Domain Controller (camitgdca01) by the client, both by direct connection and also indirectly with transitive dependencies via the two on-site file servers (cam-01-srv and cam-02-srv). Note that all the leaf nodes represent uninstrumented, off-site servers, and therefore do not show any further onward transitive dependencies.

across the scores computed by multiple machines. Doing so requires a metric-specific way to aggregate scores, and we are currently investigating how to do this and the impact it has.

5. QUALITATIVE EVALUATION

In this section we describe a qualitative investigation into how a higher-level tool might make use of constellations. In Section 5.1 we show the entire constellation for a desktop computer and discuss the information that it reveals. Next, drawing on our domain knowledge about likely relationships between hosts and services in the network, we look whether Constellation exposes any such relationships in three scenarios:

1. Web browsing (HTTP), for which we would expect to find a dependency on name resolution (DNS and WINS).

2. Running an email client (Outlook). Here there are various ill-understood network dependencies on services such as authentication and Active Directory.

3. Printing. We know that there is a transitive dependency from a client to a printer via the print server, however the situation otherwise is not clear-cut because of the way printer status changes are communicated to multiple clients. In addition, we believe that arbitrary spooling delays make detection of this dependency very challenging for any statistical or probabilistic test.

We examine each of these in Sections 5.2, 5.3 and 5.4 respectively. In Section 5.5 we explore the degree to which Constellation would reduce the search space of computers in our network for a hypothetical network management tool that identifies the source of a failure.

Unless stated otherwise, all the constellations in this section were generated using CT-NOR with an exponential delay function and FDR of 5% at each host.

5.1 The constellation of a desktop computer

A constellation covering a one hour period rooted at an arbitrarily selected desktop computer is shown in Figure 6. It was generated with client aggregation enabled, and therefore shows the expected dependencies for *any* client responsible for traffic on the same channels within that hour. For example, the client's outgoing HTTP request to the web proxy (camresisaa01) correlates with forwarded HTTP requests to all 9 upstream HTTP proxies, even though in reality that client's requests may actually only have been forwarded to a subset of the proxies (of course, in the ground-truth validation of Section 3 client aggregation was not used).

For comparison, we also generated this constellation over a period of 1 day with CT-NOR using an FDR of 5%, and with binomial co-occurrence using a threshold of 0.95 (i.e. a p-value rate of 5%). The CT-NOR full day constellation

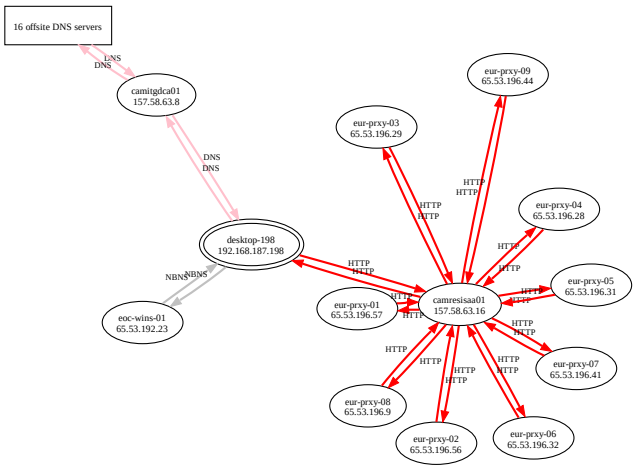


Figure 7: HTTP dependency constellation for one hour.

contains 171 additional servers over the one hour constellation, of which the vast majority—150—are upstream DNS servers. For binomial co-occurrence, the single hour constellation is almost identical to that generated by CT-NOR, but the full day graph includes 105 new nodes plus 108 additional upstream DNS servers. Out of these 105, two thirds turn out to be client (desktop) computers, which strongly implies that almost all of these dependencies are unlikely to be correct. Clearly over a longer time period the likelihood of chance co-occurrence between channels is problematic for the pairwise test.

5.2 Name resolution and web browsing

The relationship between HTTP and DNS is well-known and does not need further elaboration here. We expect to find evidence of this in our traces, as well as a less universally common dependency between Netbios Name Service (NBNS) and HTTP. This is because the Microsoft enterprise intranet contains a large number of different disjoint DNS domains, all of which have development and test DNS subdomains present on the intranet, in addition to the obvious subdomains belonging to `corp.microsoft.com` itself. The use of multiple federated Active Directory forests, each with many AD, domains also leads to non-trivial use of non-local NBNS naming within the network.

To investigate the relationship between HTTP and name resolution we first examine the CT-NOR dependency models of all hosts in the network. Over a time period of one hour we find DNS response-to-HTTP request correlations for 47% of clients sending HTTP requests, while over 24 hours this proportion rises to 83%. The increase is explained by the effects of client-side caching of DNS name lookups (typically 15 minutes). The relationship between the Netbios name resolution protocol and HTTP requests is also significant: we find that 11% of clients issuing HTTP requests have the NBNS response-to-HTTP request correlation in their single hour constellation and 33% in their constellation for one day.

Secondly, we use the causal mode of the constellation building algorithm, described in Section 4, to see how Constellation would answer the question "what hosts and services are required in order to browse the web?" We generate causal constellations from a seed edge that is the outgoing HTTP request channel to the proxy server, and find that the resulting graphs fall into 3 main groups, depending on whether just HTTP is involved, or if name resolution using DNS or WINS is also used.

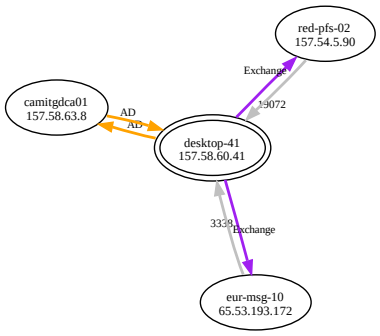
It is apparent from this that the essential infrastructure for web browsing from our site comprises the 10 proxy servers and the local domain controller. A large number of other DNS servers (168 over the course of the day) also appear in over 60% of these constellations, while 25 constellations contain the WINS servers used for Netbios Name Resolution. Figure 7 shows a typical "HTTP dependency" constellation containing DNS and WINS servers as well as the HTTP proxies.

These findings reassure us that Constellation observes the dependencies we expect to see in this network, as well as shedding light on interesting aspects such as the volume of NBNS name lookup that took place for web browsing.

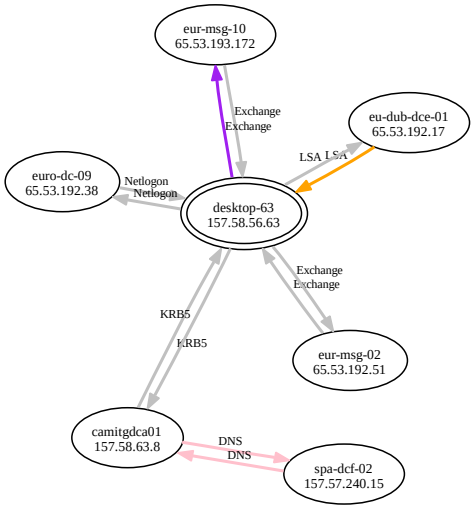
5.3 Email

The dependencies for email are complex and highly variable over time. The email client used on all desktops in the network is Outlook, which maintains numerous, long-lived connections to a number of Exchange servers situated in an offsite data centre (and therefore uninstrumented, unfortunately). The nature of the computers and services used by an Outlook client at any given time depends upon many things. For example, services such as the RPC endpoint mapper and DNS will be required when a new email session starts up. The global Active Directory catalog stores mailbox configuration and the global address book—the frequency of AD updates will impact communication by clients with domain controllers. Contact with a domain controller also occurs to invoke authentication services and protocols such as Netlogon and Kerberos. In our network, Exchange servers play dedicated roles, either supporting personal mailboxes or else shared email folders (public folders), and so the constellation for many clients includes not just their primary exchange server, but possibly one or more public folder servers also.

In contrast to the DNS/HTTP scenario we explored above, email TCP connections tend to be long-lived (on the order of hours or even days). Therefore we constructed causal constellations for email over a period of one day rather than one hour. We found the vast majority of constellations to be very small, with an average of 4 nodes and 6 edges. However, as expected, the structure of these dependency graphs is a lot less consistent across the set of hosts than for HTTP. Nevertheless, it is notable that 90% of the constellations contain the DC, usually via Kerberos and DNS, in addition to their Exchange server, and one third also contain a Public Folder



(a) Typical 24-hour email constellation.



(b) Complex email dependency constellation.

Figure 8: 24-hour Email dependency constellations

server. Other hosts found in the email constellations include file servers, web servers, non-local domain controllers and various security nodes. One of the common configurations for email dependency is depicted in Figure 8(a), and one of the more complex constellations that we found is shown in Figure 8(b).

5.4 Printing

Printing proves to be an interesting application to consider. A simple operation of the Constellation system sometimes results in a selection of client computers appearing in the constellation, as seen in Figure 9, and not always the actual printer. This comes about because of the way spooling is handled. Print jobs are spooled no less than three times: first on the client computer, second on a central print server, and finally on the printer itself. In addition, the print server makes callbacks to any client with an open job or status window on the printer when the printer changes status, and this callback is in the form of an RPC delivered to a named pipe over the remote filesystem protocol (SMB). Thus it appears that the print server acts as a filesystem client to a selection of desktop computers before and after submitting every job

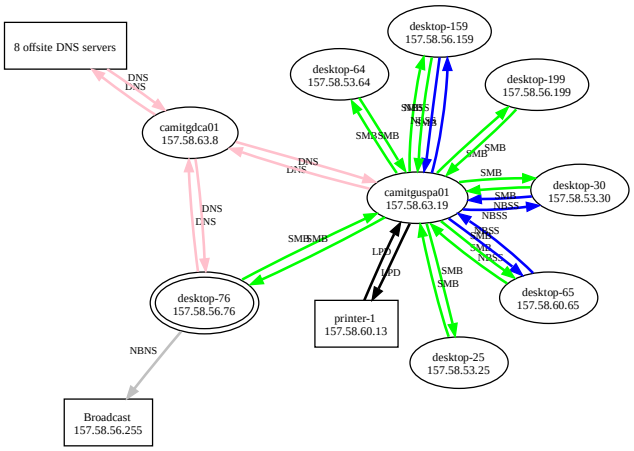


Figure 9: A printing dependency constellation containing 6 clients whose interactions with the print server camitguspa01 occur during the same time period as those of the root node, desktop-76.

to a printer.

It is hard to even determine what is a correct or useful constellation here; in some ways these clients, which might have jobs ahead in the queue, might reasonably be considered real dependencies since the print job of the constellation of interest may be dependent on the other jobs completing successfully. From another point of view these additional clients should not be entangled in a general constellation; we may not find useful a constellation in which every client that prints is dependent on every other client that prints.

One of the most important issues here is the length of time between the initial print job being spooled to the server and from the server to the physical printer which can often be several seconds. This is an example of a situation in which a co-occurrence based test will never generate useful results, but the CT-NOR method manages to correctly identify dependencies on each of the printers, learning delay distributions of $Gauss(\mu \approx 2.0, \sigma \approx 1.2)$ or $Exp(\lambda \approx 0.6)$ with the Gaussian model showing a marginally higher log likelihood.

5.5 Pruning the search space

Consider a naïve diagnostic tool that, when prompted by a user, attempts to discover the reason for the failure of an activity. The tool first pings all the hosts to which the user's host has made a request in the preceding hour, and requests all those which are available to do the same. In this way the tool builds a picture of all hosts on which the user's activity *could* have depended. We simulate the behaviour of such a tool by running Constellation with a zero confidence threshold.

Figure 10 shows the number of new nodes searched at each depth level, together with the same numbers for Constellation using a confidence threshold of 95% and Constellation with a zero confidence threshold. It can readily be

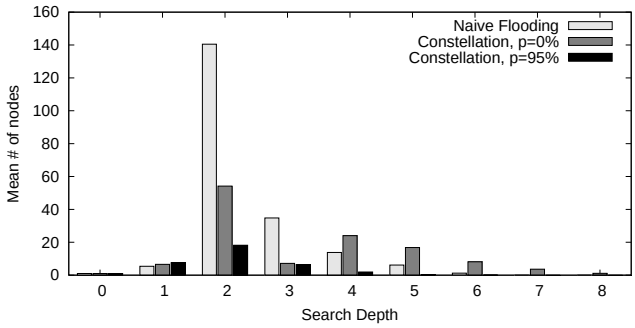


Figure 10: Pruning the search space using Constellation

seen that using Constellation to constrain the search to relevant hosts dramatically prunes the search space. Note that, due to the nature of our trace collection, a large fraction of the nodes found early in the search process will be offsite, preventing us from continuing the search. With full network visibility we would expect the difference to be much more marked.

6. DEPLOYMENT ISSUES

Constellation was designed to be deployed as a distributed system with each host building its own local traffic model, and a distributed algorithm for contruction of constellations. However, as our experimental evaluation demonstrates, it is equally possible to centralise traffic collection, processing and analysis. In this section we discuss the design tradeoffs of various deployment alternatives.

6.1 System architecture

The processing pipeline used by Constellation is shown graphically in Figure 11. This modular structure has the advantage of much flexibility. Packet timestamps on channels can be captured locally on the machines or centrally with a router tap. Filtering and pre-aggregation can be performed at the capture site, or the data can be moved remotely for processing, and so on. A deployment could choose to centralise for a (possibly complete) subset of machines at any stage of the pipeline shown above, or to operate the steps fully distributed, in any combination. We conceive of several likely deployment configurations from the flexibility available, depending on the domain of interest:

In a large hosting environment Constellation would be deployed on individual machines, either as a daemon running tcpdump or as a kernel-mode driver. In this mode each machine learns its own network dependencies and may be able to take advantage of operating-system specific knowledge (e.g. from `ps` or `netstat`) to identify services and channels.

A network operator using Constellation for management or auditing purposes may choose to centralise traffic monitoring at a router tap, giving a Constellation deployment that does not rely on the participation of individual computers.

In many situations it would make sense to monitor traffic and perform the dependency tests at the level of virtual machine hypervisor thus obtaining the efficiencies of distributed deployment without changing the running machine images. The results could be useful for datacenter management, e.g., to optimise the placement of VMs and shared services.

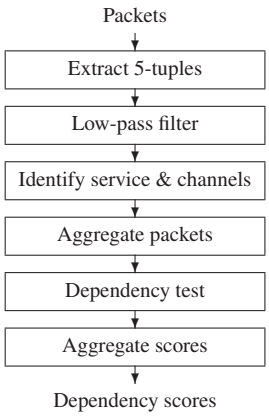


Figure 11: Constellation processing pipeline.

Likewise the algorithm to construct constellations from dependency scores could be centralised or distributed. A distributed implementation would remotely invoke a machine to query its local model returning the dependencies. In a manner similar to DNS queries, the recursion could be carried out either by that machine or the initiator.

Centralised constellation construction (at one or a small number of machines) is likely if centralised traffic monitoring has already been chosen. Indeed this is exactly what we do for the experimental evaluation in this paper that studies previously captured packet traces. Our experiments showed that a single machine is quite capable of performing this computation in real time for a site possessing hundreds of machines. A centralised (or logically centralised) system makes it much simpler to apply consistent access control to potentially sensitive information.

An intermediate design is to monitor traffic and learn pairwise correlations locally, which are then combined at a single location to generate constellations as required. This is the design employed by the Sherlock system [2]. Variations on this scheme include replication of correlation data amongst nodes in a peer-to-peer system. A peer-to-peer design would make the Constellation service more resilient to machine failures whilst retaining scalability: when some machines have failed it would still be possible to recover their most recent dependency information.

Security requirements, privacy concerns, and trust levels within the network will also influence these design choices. These same issues mean that Constellation, like `netstat` and other tools revealing potentially sensitive information, is unlikely to be deployed over the public Internet.

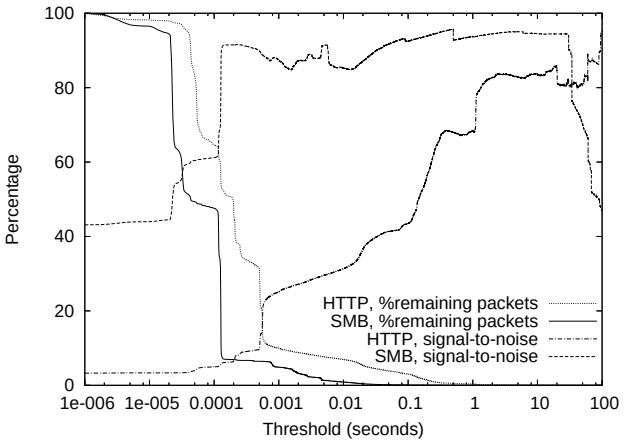


Figure 12: Impact of low-pass filter on extracted signal-to-noise and percentage of total packets processed for HTTP and SMB datasets. As LPF threshold increases so does the SNR while the percentage of packets processed decreases.

6.2 Performance

The CT-NOR dependency test has a computational complexity bounded above by $O(c_o b_o c_i (c_i + \log b_i))$ where c_i and c_o are the number of input and output channels respectively, and b_i and b_o are the number of packets on the busiest input and output respectively. Fortunately realistic data avoids this bound in the common case. For a sample hour of data 92% of hosts required less than one second of CPU with our prototype, and only four hosts required more than one minute. In a fully-distributed deployment, this corresponds to a typical CPU overhead of less than 1%. For those machines with high traffic volumes, or with a large number of input and output channels, it is easy to periodically sample the traffic to reduce the processing overhead. Another useful way to speed up performance is to low-pass-filter the traffic to reduce the number of samples that the test must consider. In the next section we discuss this technique.

6.3 Low pass filter

Although Constellation uses *packet* timing relationships as inputs, we actually expect causation to be exhibited between *message* timing relationships, i.e., between communications at layers above the network stack. Transport protocols like TCP often impose a burst structure on traffic, where each burst consists of several packets with very small inter-arrival times. Compounding this, service protocols like SMB and HTTP have massive variance in message sizes, e.g., an SMB message might be a simple *read* request, or it might be a response comprised of the entire file. Correlating timings between individual packets on IN and OUT channels thus becomes influenced by other features such as the size of higher-layer messages. It therefore seems natural to apply a low-pass filter (LPF) to remove packets on a channel which are "too close" to their predecessors, on the basis that such packets are likely part of a burst and actually correspond to a single prior service activity.

To examine the impact of applying an LPF we analysed the signal-to-noise ratio (SNR) for two protocols, where the the *signal* is the number of service-layer messages over all channels present, and the *noise* is the number of packets remaining. Figure 12 presents the SNR for HTTP and SMB as the LPF threshold increases from 1 µsec. For SMB we see that an LPF threshold of 1 ms significantly increases the SNR, so that around 90% of remaining packets represent real service-layer messages; increasing the threshold above 1 ms has little further impact. For HTTP the progression is smoother for threshold values between 1 ms and about 1 minute.

Applying an LPF has a second effect: dropping packets from channels substantially decreases the processing costs for the dependency metrics, but with the potential disadvantage that the presence of fewer signal packets might increase the real time over which adequate data must be observed to form a constellation. The pair of decreasing lines in Figure 12 show the proportion of packets remaining as the LPF threshold increases for the SMB and HTTP data. As expected, at the threshold value where the SNR dramatically increases (around 0.1–1 ms for both HTTP and SMB), the percentage of remaining packets on the channels dramatically decreases. This supports the claim that the packets dropped by the LPF are largely message continuation packets, serving no useful purpose for service correlation.

We repeated the ground-truth experiments of Section 3 applying an LPF of 200ms, and found insignificant differences in the precision-recall curves for all 3 dependency tests. Applying a 200ms LPF to the sample hour we found that 98% of hosts complete processing within one second of CPU, and the worst case host is reduced to under 8 seconds.

7. RELATED WORK

Constellation is a general-purpose, black-box tool for inferring dependencies, which leads necessarily to a statistical approach and hence some degree of uncertainty in the results. This contrasts with approaches that monitor system and network activity to explicitly track causal paths, for example Pinpoint [6] and Magpie [3]. The latter techniques are highly accurate, but require supporting mechanisms such as request identifier propagation or event logging on all participating systems.

Project5 [1] and WAP5 [17] make use of packet timestamps for correlation in a similar fashion to Constellation. Those systems aim to expose delays and bottlenecks in local and wide area networks respectively, by recording the application-level messages sent and received by a process (which may comprise multiple network packets). Various linking algorithms are then applied to determine the most likely "causal path", with estimated delays at each hop. In contrast, Constellation makes no attempt to infer a single causal path. Rather, the dependency test of Section 2 identifies all pairs of correlated channels at each host, enabling a

complete picture of the inter-dependence of all services in a network.

The Sherlock project [2] can be viewed as complementary to Constellation. While Sherlock adopts similar definitions for services and channels, it uses a different algorithm for detecting dependencies: perhaps the CT-NOR test and other statistical ideas of Section 2 could be plugged into Sherlock's infrastructure. Comparing the outputs of different dependency detection algorithms is a challenging problem in its own right. Once detailed results from Sherlock become available, this will become a fruitful area for ongoing work.

Given the local nature of channel correlations, it is straightforward to run the constellation building algorithm in a distributed fashion, initiated on-demand by the root host and recursively querying other nodes. This has the benefit of limited traffic overhead across the participating nodes, and because correlations are maintained by each host locally, Constellation does not require sophisticated information management schemes such as Astrolabe [20] or SDIMS [23]. A drawback of this approach is that all machines in the Constellation system must be functional at the time when the constellation is constructed. This could be problematic when using the system for reactive network troubleshooting.

The need for efficient network diagnosis tools and architectures that enable network management has been stressed before [7, 11, 22]. These papers propose augmenting the network with "a knowledge plane", separate and alongside the existing network, reporting on its current status. In contrast, Constellation provides a basic service for inferring network dependencies over which more sophisticated network management utilities can be built, and it has the advantage of being very easy to deploy over existing infrastructure.

Constellation aims at providing the user or network operator with the capability of comprehending and troubleshooting complex network behaviours. While end-user diagnosis tools have been proposed in the past [14, 15], these solutions identify problems of specific applications or protocols (e.g., TCP reordering, queuing and loss events) and cannot identify network-wide dependencies of host or services.

8. CONCLUSION AND FUTURE WORK

Constellation is a new approach for inferring service dependencies in a computer network, deployable on existing infrastructure and using only lightweight monitoring. It automatically discovers the network-wide map of dependencies from a particular computer, removing the drudgery and guesswork of combing through tcpdump and Ethereal outputs to glean an understanding of how network services are related. In this paper we have presented results against a real network trace that are both accurate with respect to a ground-truth dataset, and that we have shown to be useful for end-users and network administrators alike.

The sound probabilistic models and statistical test that we have developed provide a solid basis on which to build, and open new possibilities that we are planning to explore in the near future. For example, we can use the local models to compute probability scores and confidence measures on whole paths (in addition to local dependencies). This, coupled with strategies for maintaining and comparing constellations over time, will enable administrators to quickly identify change, localise problems, and discard alarms that are consequences of root causes along these paths. It is towards this vision that we will continue this research path.

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'03*, pages 74–89, Oct. 2003.

[2] V. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM'07*, Aug. 2007.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI'04*, Dec. 2004.

[4] Y. Benhamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society*, 57(2-3):125–133, 1995.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] 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*, pages 309–322, Mar. 2004.

[7] D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the Internet. In *SIGCOMM'03*, Aug. 2003.

[8] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian Network Classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[9] M. Goldszmidt, I. Cohen, A. Fox, and S. Zhang. Three research challenges at the intersection of machine learning, statistical induction, and system. In *HotOS'05*, 2005.

[10] G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes*. Oxford University Press, 2001.

[11] J. M. Hellerstein, V. Paxson, L. Peterson, T. Roscoe, S. Shenker, and D. Wetherall. The Network Oracle. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2005.

[12] M. S. Johns. Identification Protocol. RFC 1413, IETF, Feb. 1993.

[13] L. Lamport. Quarterly quote. *ACM SIGACT News*, 34, Mar. 2003.

[14] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *SOSP'03*, pages 106–119, Oct. 2003.

[15] V. N. Padmanabhan, S. Ramabhadran, and J. Padhye. NetProfiler: Profiling wide-area networks using peer cooperation. In *IPTPS'05*, Feb. 2005.

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

[17] 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'06*, May 2006.

[18] J. Storey. A direct approach to false discovery rates. *Journal of the Royal Statistical Society*, 64:479–498, 2002.

[19] M. Tanner. *Tools for statistical inference*. Springer, 1993.

[20] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[21] L. Wasserman. *All of Statistics*. Springer, 2004.

[22] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *HotNets-II*, 2003.

[23] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM'04*, Aug. 2004.