Macroscope: End-Point Approach to Networked Application Dependency Discovery

Lucian Popa
University of California
Berkeley
popa@cs.berkeley.edu

Byung-Gon Chun
Intel Labs Berkeley
byung-gon.chun@intel.com

Ion Stoica
University of California
Berkeley
istoica@cs.berkeley.edu

Jaideep Chandrashekar
Intel Labs Berkeley
jaideep.chandrashekar@intel.com

Nina Taft
Intel Labs Berkeley
nina.taft@intel.com

ABSTRACT

Enterprise and data center networks consist of a large number of complex networked applications and services that depend upon each other. For this reason, they are difficult to manage and diagnose. In this paper we propose Macroscope, a new approach to extracting the dependencies of networked applications automatically by combining application process information with network level packet traces. We evaluate Macroscope on traces collected at 52 laptops within a large enterprise and show that Macroscope is accurate in finding the dependencies of networked applications. We also show that Macroscope requires less human involvement and is significantly more accurate than state of the art approaches that use only packet traces. Using our rich profiles of the application-service dependencies, we explore and uncover some interesting characteristics about this relationship. Finally, we discuss several usage scenarios that can benefit from Macroscope.

Categories and Subject Descriptors

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

General Terms

Design, Measurement

Keywords

Application dependencies, end-point tracing

1. INTRODUCTION

Today's enterprise networks and datacenters are extremely complicated entities, running a large number of interdependent applications and services. A recent survey stated that companies such as Citigroup and HP run applications in their networks, numbering in the thousands [2]. While this sheer number is remarkable, the actual complicating issue is the manner in which they are coupled to each other. For example Microsoft Outlook, ostensibly an email client, depends on services such as AD (active directory), LDAP, DNS, network proxies, among others. When users in the network cannot "see their email", what should the network administrator look at? The key challenge facing these complicated deployments is management related. It is estimated that up to 70% of an enterprise's IT cost goes toward maintenance and configuration costs [15], *i.e.*, tasks such as fault localization, performance debugging, reconfiguration planning and dependency analysis.

Discovering (or mapping) dependencies between applications and services has been recognized previously as an important task [5, 6, 9, 12]. In much of this work, the dependency discovery task has relied solely on network-level packet trace data, *i.e.*, traces collected in the network. For example, Sherlock [5] uses the time correlation of packets between different services (in particular, the number of co-occurrences in some time interval), and Orion [9] uses the spikes of delay distribution of flow pairs. Using this data is advantageous in some ways; it can be collected quite easily, without any impact on the end-hosts. However, there are several limitations, described below, of using only this data to extract the application and service dependencies.

First, these techniques can be quite imprecise. Timing correlations between independent services can create the illusion of causality, leading to false positives. Also, when background traffic perturbs the timing of network packets, or if there is a high variation temporally in how the packets are observed, false negatives can occur. Second, approaches solely based on packet traces cannot uncover dependencies when the communication patterns are very dynamic or infrequent; this is because the dependency extraction methods require a large number of samples to converge. Third, when only packet traces are used, some human intervention is required to extract the dependencies correctly. To state this differently, existing methods require some seed ground truth to extract the application dependencies, in order to connect application level information with the network level traces. This lays the accuracy of the extracted dependencies completely at the feet of the human operator.

In this paper, we propose a new approach to extracting the dependencies of applications *automatically*. In contrast to previous methods, our approach involves a small amount of instrumentation at the end-hosts. Specifically, we collect information at the end-hosts about the particular applications that are generating network traffic and join this with network packet traces. Further, we correlate this information from multiple end-hosts to arrive at the application dependencies. We argue that this extra information, which basically

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CoNEXT'09, December 1–4, 2009, Rome, Italy.
Copyright 2009 ACM 978-1-60558-636-6/09/12 ...$10.00.

serves to replace the human knowledge required to seed the dependencies (in previous methods), can be easily collected and is more reliable. For example, contemporary enterprise deployments already feature a host of security and monitoring features installed on end-hosts and it would be easy to add (lightweight) application tracing code to these systems.

We present *Macroscope*, a system that automatically extracts networked application dependency using network-level and application level information. Macroscope consists of three components: *Tracers*, a *Collector*, and an *Analyzer*. Tracers run on individual end-hosts and log application-level process information by periodically sampling TCP/UDP connection tables and join this information with network level packet traces. This allows individual network flows to be associated with the generating application. These processed traces are then shipped to a centralized Collector. The co-located Analyzer correlates the traces across the population of end-hosts to finally extract the dependencies with various properties as will be described in later sections.

The extracted dependencies are critical for a number of management related tasks: (1) they can aid in fault diagnosis by enumerating all the key services used by an application, which allows the network operators to focus on the relevant servers and services; (2) the dependencies allow network operators to understand the impact of any outages (planned or not) and to place resources to maximize utility; (3) learning the temporal dependencies of an application can help operators determine the correct order in which to restart services (many web services today are composed of a number of services, and ordering is critical) ; (4) knowing the correct dependencies for an application can aid with detecting anomalies in the application or service (some malware is known to replace existing executables with custom versions which make connections not normally associated with the application). There are far more compelling usage scenarios, where being able to discover and enumerate the application dependencies is a crucial step, than we can enumerate here. However, we do revisit some of these in Section 5 and provide specifics of how the learned dependencies can be used.

We summarize the key contributions of this paper:

- We propose Macroscope, a novel method to identify the service-level dependencies that applications rely on. We define a notion of static dependencies and present a practical method for extracting them. Our method uses application level data in conjunction with network-level traces, to build application-service dependency graphs at multiple levels of granularity.

- We evaluate our methodology with traces collected at 53 enterprise end-hosts. We show that Macroscope is accurate in detecting application dependencies and that it outperforms by a wide margin a state-of-the-art application dependency extraction system that uses only packet traces (Orion [9]). This quantifies the gains of bringing application level information into the problem of dependency inference.

- With our rich profiles of the application-service relationship, we carry out some exploratory assessment of over 150 applications. We uncover some interesting findings about these relations, such as an elephants and mice phenomenon for application-service dependencies, a surprisingly large number of single-service applications, and others. Finally, we describe several management applications that could exploit these properties for greater effectiveness.

The rest of this paper is organized as follows. Section 2 presents Macroscope's approach. Section 3 presents Macroscope's system architecture and the algorithms it implements. We evaluate Macroscope in Section 4, discuss a set of usage scenarios in Section 5 and related work in Section 6, and finally conclude.

2. MACROSCOPE APPROACH

2.1 Dependency Definitions

We use the term *application*, denoted (app), as an application executable (and assume the name is consistent across all the end-hosts). An *application instance* refers to the application being run on a particular end-host; we represent it by the tuple (app, PID, IP), where PID is the process identifier and IP is the machine network address. We break down the PID instances by the IP to increase the number of application instance samples. For example, a process active for multiple days can be seen as a new instance each new work day (*e.g.*, user logs in the application) and we approximate this behavior by using IPs. We ignore the reuse of the same PID as the process identifier space is quite large.

An application may use one or more *services*; the latter is identified by the tuple $(protocol, port)$. Common services include DNS, LDAP, HTTP, etc. A *service instance* is identified by the tuple $(IP', protocol, port)$, where IP' represents the address of the machine on which it is running.

Next we define a *dependency* (or relation), which is an association between one application and one service. A naive approach would be to define the use of *any* service as a dependency (*i.e.*, an established connection to the service from the application); however, this definition is overly broad. Ideally, we want dependencies to capture the "critical" services of an application, *i.e.*, the services that are used frequently and whose reachability predicts the success or failure of the application. Knowing such dependencies can help to locate faults and to improve application performance (*e.g.*, by replicating more of the critical and frequently used services). To approximate this set of desirable associations, we introduce the notion of a *static dependency*. Application `app` has a static dependency on service `S` if the usage of `S` can be predicted before executing `app`: (1) independently of user actions or other application instances and (2) using only the application's source code and configuration files.[1]

A static dependency is denoted as the relation $(app) \rightarrow (protocol, port)$. We name the relations of this type which do not represent static dependencies as *transient relations*. To clarify, connections to ephemeral ports, or to ports that are dynamically generated reflect transient relations. Compared to static dependencies, the services corresponding to transient relations are typically not actively managed by the enterprise and their use is limited to a few application instances. For these reasons, transient relations are less well-suited than static dependencies for detecting faults, planning enterprise resources and understanding application behavior.

By removing all connections pertaining to transient relations out of the set of all the connections established by the application, we can arrive at the set of static dependencies. In this paper we develop simple heuristic methods to extract the (static) dependencies from packet traces and application context data.

Multilevel Dependencies: Static dependencies provide a high-level view of the interactions between applications and services. However, relations at a lower granularity are often useful. For this pur-

[1] In the context of this paper, the services corresponding to static dependencies can be seen as actively managed by the enterprise (as we shall see in §4, applications cannot directly communicate outside our enterprise).

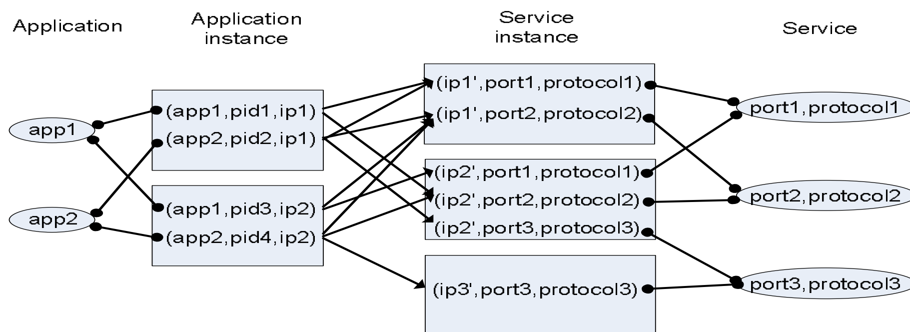


Figure 1: An example showing dependencies between applications and services. An oval represents an application or a service, and a rectangle represents a host that may run multiple applications or services.

pose, we define the following levels of granularity of the relation between applications and services:

1. $app \rightarrow (protocol, port)$; this captures the top level static dependencies.
2. $app \rightarrow (IP', protocol, port)$, specifies the individual service instances used by an application.
3. $(app, PID, IP) \rightarrow (protocol, port)$, captures the general services used by a specific application instance.
4. $(app, PID, IP) \rightarrow (IP', protocol, port)$, this finest level of granularity captures the service instance used by a particular application instance.

Figure 1 illustrates an example for two applications, each with two application instances, and three services, each with two instances. There are no transient connections in this example. Static dependencies are the connections between the applications and the services (first and last columns). The connections between the application instances in the second column and the service instances in the third column belong to the level four above.

Conversely, the relation between applications and services can also be regarded from the perspective of services, such as the applications that use one (or several) service(s), *i.e.*the mapping $(protocol, port) \rightarrow app$.

We point out that some static dependencies may actually hide numerous distinct services offered on the same port; these services can be offered by multiple hosts, which can belong to different enterprises (in this cases the destination IPs could be viewed as "ephemeral"). The most striking example of this type is the $(tcp, 80)$ HTTP service. Such static dependencies may be less effective for the administrative purposes of fault isolation and resource planning; to help administrators, Macroscope can present only the dependencies with intra-enterprise service instances. More generally, in the future, we plan to consider aggregations of destination IPs, such as in [11], in which particular destination IPs are grouped together to capture the service they represent. In an enterprise, an example would be “mail.intel.com”.

Dependency Profiling: In the dependency graph, each dependency can also have associated a *weight*, such as the usage frequency (the fraction of the application instances that use that dependency) or the traffic volume on that dependency. Depending on usage scenarios, we can use a different metric for the weights.

Finally, we can define a number of temporal relationships on the dependencies. In this paper, we only consider the *Causal Order* relation, defined as follows. Dependency `A` of application `app` is “causally ordered before” dependency `B` if `A` is often accessed before `B` by instances of `app`, and the accesses of `A` and `B` occur within a short time window W. Note that the remote-remote dependency definition used in Orion [9] is similar to the relation defined above. Unlike this definition, Orion does not consider the flow directionality; in Macroscope, dependencies are always flows initiated by the application to the service.

2.2 End-point Tracing vs Network Tracing

Macroscope’s approach of recording application context at end-hosts has two important advantages over the approaches that use only in-network monitoring for this purpose: (1) it is fully automatic and (2) it is more accurate. The price we pay is the required instrumentation of end hosts and the overhead incurred upon them. However, although CPU and memory usage statistics were not collected along with the traces used in this paper, qualitatively no user noticed any slowdown in their machines.

Fully Automatic: By capturing the application names when sampling the connection table, Macroscope can correctly associate applications with the connections being generated (and consequently, accurate determine the application-service dependencies). In contrast, with approaches that solely rely on packet traces from the network, the application information is supplied by a human who seeds a small number of application-service dependencies (the dependencies are grown transitively). Interestingly, we show in Section 4 that many applications have only a single static dependency; in these cases, automation serves no purpose for the approaches using only packet traces.

Improved Accuracy: Macroscope detects more of the actual dependencies. Previous works such as Orion [9] and Sherlock [5], rely on relations between dependencies for their detection. In general, this approach can only detect a subgraph of the real dependency graph (*i.e.*, only those dependencies that exhibit that particular relation can be detected); this detection is also quite noisy, since it relies on co-occurrences in a timed window. In contrast, the detection approach that we present in this paper, captures dependencies of an application with high accuracy, regardless of their relations to other dependencies. This detection is more accurate even for the detecting dependencies with relations between them because we can identify which are connections of the same application.

Macroscope also reduces the rate of false positives (*i.e.*, fake dependencies). For example, Orion detects application dependencies by hand picking a set of known (seed) dependencies, and finds other services that are orderly dependent on these seeds (Orion uses a sin-

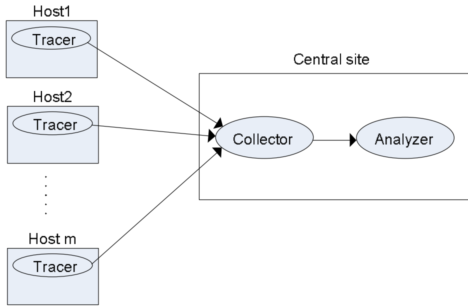


Figure 2: Macroscope system architecture.

gle seed dependency). It is easy to see how this type of detection can go wrong: assume application `app` has dependencies `A` and `B` ordered, and application `app'` has dependencies `X` and `B` ordered; hand picking service `B` as a seed dependency for `app` may lead to the wrong conclusion that `app` is also using `X`. Also, some applications have no "specific" service dependency; for example the windows executable searchprotocolhost.exe, accesses only well known services (Active Directory, EPMAP, DNS and Web). This situation is almost guaranteed to lead to false positives (and potentially false negatives) since these services are used by multiple applications with other dependencies.

3. MACROSCOPE

We now describe the architecture, dependency extraction algorithms, and prototype implementation of Macroscope.

3.1 System Architecture

Figure 2 shows the overall architecture or Macroscope, which consists of Tracer, Collector, and Analyzer.

Tracer: The Tracer runs on end-hosts and has two components: one that periodically samples and logs the OS connection table and one that captures and logs packets. We have implemented the Tracer for enterprise specific laptops running Microsoft Windows XP.

The Tracer samples the TCP and UDP tables and appends this information to a file. The sample contains the time of sampling, application-level information such as application name and process id and connection tuples (protocol, ports). The Tracer samples every *five* seconds using the `GetExtendedTcpTable` and `GetExtendedUdpTable` calls. For UDP, the remote endpoint is detected from the packet traces. The Tracer stores the data in XML files.

The Tracer also captures packet level traces using WinPcap [3]. We collected the packets on the end-hosts because the data collection effort supported multiple research projects, some of which needed host mobility information. We point out that there are other ways to implement the packet tracing component of Macroscope. If the goal is to focus on dependencies that occur for applications used in the enterprise setting, then packet traces could alternatively be captured by network monitors located inside a corporate network. Moreover, Macroscope does not require a strict time synchronization between the in-network capturing device and the end hosts because the reutilization of ports opened by clients is done on a very large time scale (*e.g.*, hours) and these are actually used to synchronize the packet traces with the connection table samples.

Collector: The Collector aggregates data received from multiple hosts and joins the application-level data with the network level data. It first constructs flows (TCP/UDP sessions) from the packet traces using Bro [1]. It then combines the data in the XML connection table samples with the flow data by joining connection tuples. This data is used for analysis.

Analyzer: The Analyzer consists of a set of tools (implemented in Python) for extracting dependencies, querying dependencies and producing graphical visualizations of the dependencies. These are detailed in the next section. Macroscope can visualize the static dependencies at each of the level of granularities defined in Sec. 2.1, for a single application or for a group of applications. In addition, we can produce visualizations of the dependencies weighted by their usage frequency or traffic volumes, and depict the temporal evolution of dependencies.

Caveats of sampling: Our design that samples application-level data at end hosts has interesting tradeoffs. By increasing sampling frequencies, we can obtain more accurate snapshots of application-level data but this incurs higher resource overhead at end hosts. With the five-second sampling period, the overhead is negligible, but this sampling approach has some caveats.

First, the sampling can miss very short-lived flows. To compensate for the loss of accuracy, we use simple statistic tools as shown in the next section. Second, the connection sampling may hide information about the actual application. An application can use fork such that the same application appears as having different types of dependencies. Plug-ins (*e.g.*, browser plug-ins) may also make application dependencies vary from one instance to another. Also, some connections could be delegated by applications to local OS demons acting as proxies (*e.g.*, some applications delegate DNS queries to the lsass service).

Despite these caveats, the evaluation presented in Section 4 shows that Macroscope achieves good performance for detecting application dependencies.

3.2 Algorithm to Identify Static Dependencies

We now describe our algorithm for identifying the static dependencies of an application. The inputs to this algorithm are all the (sampled) connections outgoing from the application (the 5 tuple connection information on IPs, ports, and protocol), along with the process ID (PID) and the user identifier of the application instance that initiated the connection.

We use a two-step algorithm. In the first step we classify applications into one of two types: either it only generates connections produced from static dependencies, or it generates connections belonging to both static dependencies and transient relations. Applications that use transient connections are those that use ephemeral ports and/or peering connections such as in P2P applications. It turns out that we do have a number of applications with none, or very few, transient connections. We separate our applications into these two classes for the following reason. Those applications that generate connections from both static dependencies and transient relations will require a second processing step in which the transient and static components are further separated. However those falling into the class that has only static dependencies need no further processing in terms of identifying the static dependencies because all of the dependencies in this class are static.

The second step of our algorithm takes the data from the applications with connections arising from both static dependencies and transient relations, and tries to distinguish them. To do this, we make use of usage frequency information (defined below). A central idea is that we assume that static dependencies are common across instances of the same application, whereas transient connections are not. We believe that our traces are large enough (in

number of application instances and users) so as to have enough samples to differentiate these cases quite accurately. For example, static dependencies use the same port every time while transient connections typically use ephemeral ports. Frequently ephemeral ports are randomly assigned (although not always) and thus their usage frequency can be as low as $1/(\#ephemeral\ ports)$ (the number of ephemeral ports can be as high as 60,000) whereas the port usage frequency for static dependencies will typically be much higher.

Classifying applications: To separate our applications into the two classes described above, we use the following simple approximation method. Let N^a be the number of instances of application 'a' in the traces, that we compute by counting the number of distinct (PID, IP) tuples for a. N_s^a denotes the number of instances of N^a that used service s where s refers to a specific $(port, protocol)$ pair. Let S^a denote the total number of services, or $(port, protocol)$ pairs observed by all instances of application a in all the traces. We now define $V_s^a = N^a - N_s^a$ to capture the number of application instances that did not use service s. We define the metric M^a to be $M^a = \sqrt{\sum_s (V_s^a)^2 / S^a}$.

This metric can be interpreted as follows. When all application instances use all services, then $M^a = 0$. The larger M^a is, the farther away the application is from being one with purely static dependencies, *i.e.* it is more and more dominated by transient connections. The maximum value of $M^a = N^a - 1$ occurs when each service is used only once by one of the application instances. Note that M^a rapidly increases when transient connections are used. For example, suppose each instance of an application a uses the same 10 static dependencies and has only 1 transient connection. Then each of the transient connections will have its own variable entry V_s and the value of M^a will easily explode for traces with many application instances.

We compute M^a for each application. We classify an application as having only static connections, if M^a lies below a threshold defined as follows. Since M^a ranges from 0 to $N^a - 1$, we say that if $M^a < T(N^a - 1)$ where T is a percentage, then the application has no transient relations. For the value of T we use 85%. In Section 4 we show that our classifying heuristic is somewhat resilient to the value of the threshold T, and thus we need not worry about fine tuning its value. Clearly this heuristic does not mandate that the number of transient connections for such applications needs to be exactly zero. Our heuristic may still classify an application as having only static dependencies even if a small number of transient connections are made or if these are used by a lot of the application instances. Labeling transient relations as static dependencies will result in false positives for our method. But as we will see in Section 4, Macroscope has very few false positives and there are quite a number of applications with zero transient relations.

There are a variety of heuristics that could be defined to carry out this step (*e.g.*, set a threshold for the maximum number of static dependencies, an application with more distinct service usage is labeled as having transient connections); however, the one here is intuitive, simple and works well.

Identify Static Dependencies: Having identified those applications that contain essentially only static dependencies, we can remove them from further dependency analysis. We now focus only on the applications that generate connections due to both static dependencies and transient relations. We extract the static dependencies using the following method. First, we consider all services listening on ports below 1024 as static dependencies (regardless of their usage frequency). This is because these ports are not typically used as ephemeral ports and thus there is a high likelihood that these represent known maintained services. Second, any remaining services, *i.e.*, $(port, protocol)$ tuples, are labeled as static if they satisfy two criteria on usage frequency.

Let U^{all} denote the set of all users, and $U^a \subset U^{all}$ are the users that employ application a. $U_s^a \subset U^a$ represent the users that have connected to service s through application a. We label application a as having a static dependency on service s, if both of the following hold:

$$\frac{|U_s^a|}{|U_a|} \geq U \ \text{ and } \ \frac{N_s^a}{N^a} \geq I$$

U and I are two thresholds. In our implementation, we picked 10% as the relative threshold for these two ratios. This value was derived experimentally and works quite well for our data. However, we also show in Section 4 that the method is not very sensitive to this parameter.

We use both of these criteria for the following reasons. If we considered application instances only, then a single user, that frequently used a particular application could bias the ratio N_s^a/N^a by generating a huge number of instances that use a particular service. For example, such a user could contact a particular peer often, and that peer can spuriously appear as a dependency (false positive) if it is using the same ephemeral port to provide peering service. To avoid this, we require that a minimum of users use the application and the service as well. We elect not to employ user frequency only since this detection can be inaccurate with long traces. When using long traces, the same transient connection can occur in multiple traces (since there is a finite number of ephemeral ports), which would also result in false positives.

Adjustments due to sampling: Macroscope's method of periodically sampling the connection table can miss very short connections. Thus, the usage frequency of short connections will be affected, possibly leading to an undesired bias towards mislabeling short connections as transient connections. For this reason, we use an updated value for the usage frequency of a service based on its detection probability (*i.e.*, probability of being sampled by the Tracer in the connection table). The detection probability is dictated by the length of the connection to the service. If the average connection duration is d and Macroscope's connection table sampling period is $W > d$, we define the detection probability $p_d = d/W$. For those connections shorter than W, we boost the usage frequency (ratio of instances using a service) by $1/p_d$. We only apply this adjustment if multiple samples of the service usage are available, to avoid spurious boosts. This approach may be slightly over-boosting when dependencies are regularly used multiple times by each instance, but this is acceptable since such dependencies are likely to be static ones.

4. EVALUATION

The data traces used in this paper were captured on 52 employee laptops (assigned to individual owners) in a large enterprise over a period of 11 weeks. All the instrumented systems ran a standard corporate build of Windows XP SP2. Each of these systems had a suite of enterprise applications (security and management related) installed on them. Users are free to install additional software on their systems with the exception of p2p applications, which are banned, except Skype. Data collection on the laptops proceeded even as the users moved between home and work. Correspondingly, trace files correspond to one of three modes: (i) inside the corporate network, (ii) outside, but connected to the VPN and (iii) outside and not on the VPN. Since we are chiefly interested in sce-

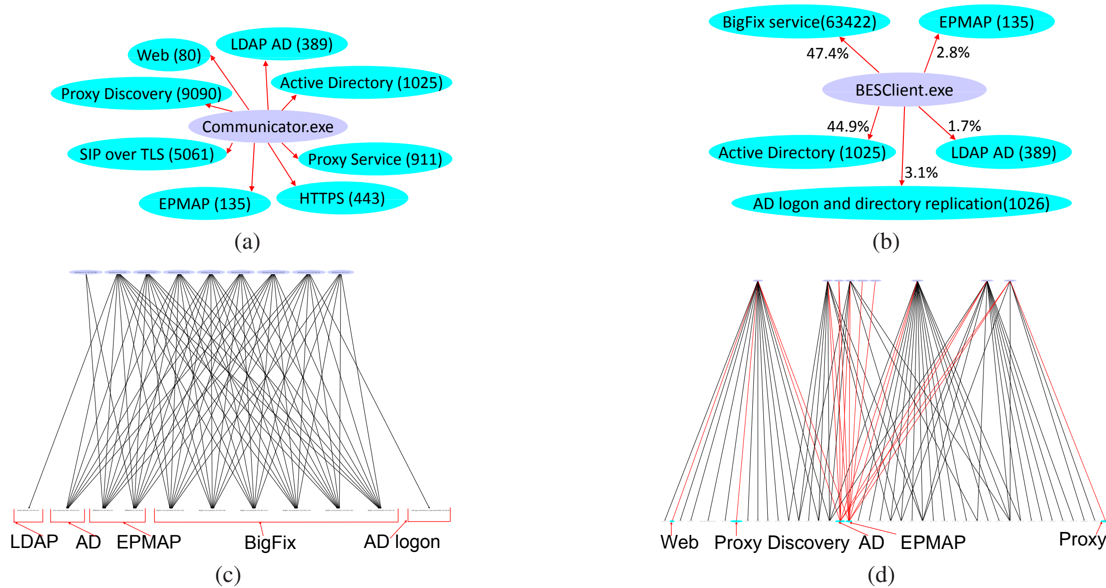


Figure 3: Macroscope Sample Charts: (a) Static dependencies for MS Office Communicator, (b) Static dependencies for BigFix BESClient (weighted by % of outgoing traffic volume), (c) Instance level dependencies for BigFix BESClient (top application instances, bottom service instances), (d) Distinct port connections for MS Outlook (top application instances, bottom services); red lines denote static dependencies.

narios that aid the IT operators inside the enterprise, we only work with the (subset of) data collected when the user is inside the corporate network. [2]

In this section, we first describe the capabilities that Macroscope enables and present some sample visualizations (§4.1). We then evaluate the accuracy of our dependency extraction methodology and compare it to an established method (§4.2). Finally, we highlight a few interesting results that are enabled by Macroscope (§4.3).

4.1 Macroscope Capabilities

Multi-level Dependencies: Macroscope can discover all the levels of granularity of the application service relations defined in Section 2.1. Filters can be applied to restrict only a subset of the services to be used in these relations. For instance, we can restrict the displayed services to only ones of static dependencies, use a subset of the data (in duration or number of users) or show dependencies based on time of day. Further, Macroscope can present these relations for a single application or for a set of applications.

As an example, Fig. 3(a) presents a visualization of the static dependencies for MS Communicator as uncovered by Macroscope. This chart was generated automatically, with the application name provided as input. Seen in the figure, $(tcp, 911)$ represents an HTTP proxy service used to communicate with hosts outside the enterprise, while the service $(tcp, 9090)$ is a proxy discovery service that hosts use to locate the nearest proxy server.

At a different level of granularity (level 4 in § 2.1), Fig. 3(c) is a visualization of all the service instances used by instances of the BigFix client application; upper vertices correspond to application instances (PID, IP) and the lower vertices represent service instances $(IP', protocol, port)$. In this figure, we see replicated services corresponding to the static dependencies (no transient connections were observed); we also see that most application instances have a regular pattern of using most static services and load balancing between them. In contrast, the access pattern in Fig. 3(d) is very different. Here, the black lines indicate transient connections, while red lines capture the static dependencies. Note that (i) there are few static dependencies relative to the transient destinations, and (ii) access patterns vary a lot across instances. There is an intuitive reason to expect a difference in access patterns between figures 3(c) and 3(d); BigFix is an enterprise application that runs without any user involvement while Outlook is almost completely user driven.

Dependency Profiling: Macroscope's procedure of joining information creates rich data that enables the creation of profiles for applications and their dependencies. In particular, Macroscope can label static dependencies by their usage frequency and traffic volume (incoming/outgoing) but also find (temporal) relationships between static dependencies such as the causal ordering introduced in Section 2.

Fig. 3(b) shows the static dependencies for the BigFix BESClient application, labeled by the percentage of outgoing traffic each dependency contributed relative to the total outgoing traffic of the application. As one can see, the outgoing traffic pattern of this application is dominated by the BigFix service (it's main purpose) and by the traffic to the Active Directory service. Macroscope can present this type of data from a selected subset of data (*e.g.*, a period in time, number of users) and also show the variation of the weights in time. Finally, Macroscope can also show how much of total incoming/outgoing traffic each application is responsible for and how many rejected flows each application generates.

[2]Labeling the traces as inside/outside/VPN has been done at the collection time, by querying a host resident proxy configuration service and by considering the network interface being used. In the absence of such support, this labeling could be done using knowledge of enterprise IP address range and other information (*e.g.*, in our case, all connections outside the enterprise are done through a proxy).

Application	True Static Dependencies	End-point + Packet Traces Macroscope			Packet traces only Orion Best fn			Packet traces only Orion Best fp		
		tp	fp	fn	tp*	fp	fn	tp*	fp	fn
Communicator	8	8	0	0	8	17	0	4	3	4
PWconsole	8	7	0	1	6	19	2	1	0	7
FrameworkService	6	6	0	0	5	20	1	1	0	5
Outlook	8	8	10	0	8	11	0	8	11	0
BESClient	5	5	0	0	5	19	0	4	6	1
Winword	8	6	0	2	8	16	0	4	1	4
lsass	7	7	0	0	7	40	0	6	4	1
SearchProtocolHost	4	4	0	0	4	21	0	4	21	0
Skype	3	3	0	0	2	22	1	1	1	2
UNS	1	1	0	0	X†	X	X	X†	X	X
userinit	1	1	0	0	X†	X	X	X†	X	X
conf	2	2	1	0	1	0	1	1	0	1

*One of these true dependencies is manually added by the user.
†Since these applications have a single static dependency, this approach is not feasible.

Table 2: Accuracy of Static Dependency Detection

Executable	App. Name/Description
Communicator.exe	MS Office Communicator
PWConsole.exe	MS Office Live Meeting
Framework-Service.exe	McAfee VirusScan Framework Service: component of a antivirus product
Outlook.exe	MS Outlook
BESClient.exe	BigFix BESClient: remote administration software
WinWord.exe	MS Office Word
lsass.exe	MS Local Security Authority Subsystem Service: enforces Windows security
SearchProtocol-Host.exe	MS Windows Search Protocol Host: used for desktop and network search
Skype.exe	Skype
UNS.exe	Intel Active Management Technology notifier service client
userinit.exe	MS Windows Userinit Logon: sets up network connections and shell
conf.exe	A process belonging to MS NetMeeting

Table 1: Applications used to evaluate Macroscope's accuracy in detecting static dependencies

4.2 Accuracy of static dependency detection

To evaluate Macroscope's accuracy in determining static dependencies, we choose 12 applications for which we compare Macroscope with (1) manually verified (ground truth) static dependencies and (2) we also compared it against the results returned by Orion [9], the state of the art method for extracting dependencies by only looking at packet traces. The 12 applications are listed in Table 1. The choice for these applications was based on their popularity and our interests. The chosen applications represent a balanced mix between applications which use transient connections (7) and applications without transient connections (5).

In an ideal scenario, ground truth static dependencies would have to be derived from a careful analysis of the application source code, configuration files, and even the network itself. However, this is quite impractical to realize since we do not have access to the source code. In this paper, we instead rely on operator knowledge and common sense to extract the confirmed static dependencies in the data proceeding as follows. For every network flow, tagged with an application name, observed in the traces: (i) we check the destination server and port against a list of known servers and ports used inside the enterprise (these are all the infrastructure servers formally managed by the IT department); if we have a match, the connection corresponds to a static dependency, else (ii) we attempt to manually connect to the server and port (believing that infrastructure servers do not go away in a period of a few months) and label the dependency as static if the connection succeeds; if that does not work, as a last resort (iii) we asked the IT operators in our enterprise to verify if they believed the service to be persistent. If none of these methods classified the dependency as static, we assumed it was transient. Note that this is is a conservative approach; IT operators cannot keep track of the thousands of services that are deployed and their answers are based on a best guess.

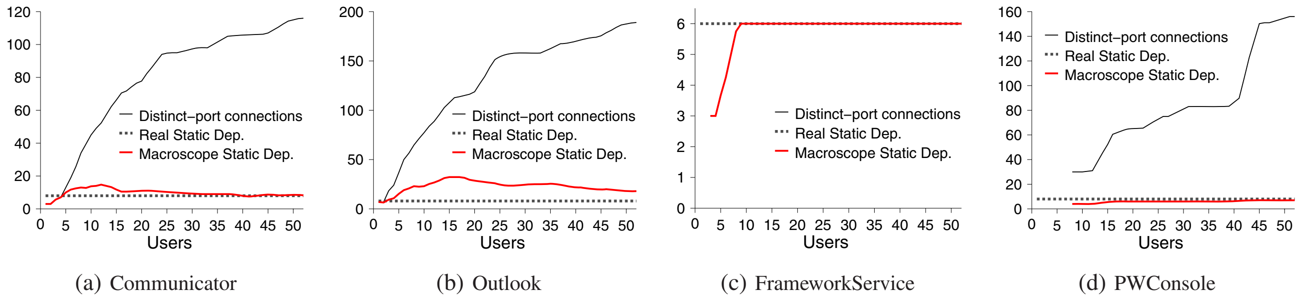
Our described approach may miss some short-lived extant static dependencies due to sampling. Given that our traces were collected over a relatively long period and that static dependencies should be observed frequently, we believe that the missing dependencies are very small in number. For example, for a 25ms dependency an expected number of approximately 200 occurrences are required for one sample; assuming even only one usage per application instance this would necessitate 200 application instances; many of the applications that we profile have seen a lot more instances.

Another artifact of the data collection process has to do with the non-transparent proxy required to contact external destinations. All connections to the outside are effected through a proxy, which has the effect of collapsing all the static dependencies to outside destinations on the proxy nodes.

Finally, while we expect the DNS service to be a static dependency for most applications, we are unable to attribute DNS connections correctly to some applications. This is because the windows kernel delegates DNS lookups to a third party (`lsass.exe`). Thus, in almost all the cases, the static dependencies of an application do not include DNS.

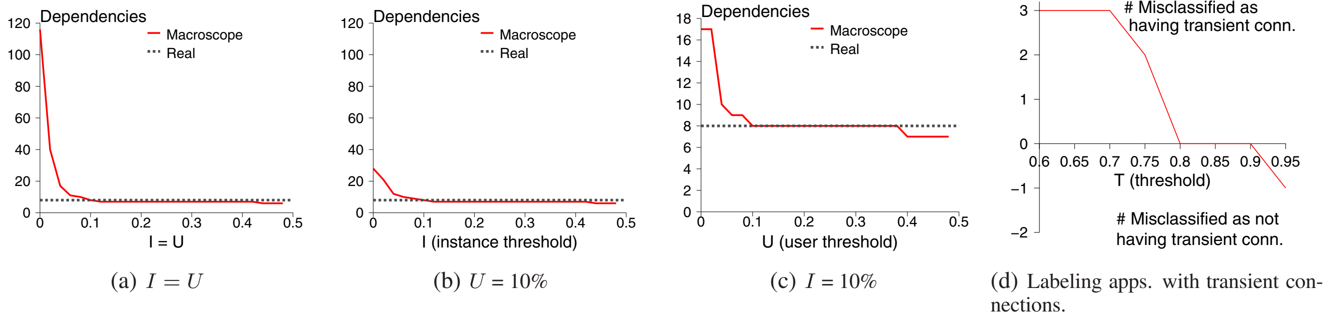
To compare Macroscope's accuracy in detecting static dependencies to that of approaches based on packet traces only, we have implemented Orion [9], the current state of the art approach in this space. This implementation represents our best effort based on the description in the paper and discussions with authors. We use Bro [1] to generate flow information from the packet traces (but we break flows using keepalive messages as in [9]).

Orion (along with Sherlock [5] and eXpose [12]) infer relationships between services based on temporal co-occurrence. Application context is not recorded; instead, applications are manually associated (seeded) with a set of dependent services. For example, suppose the services $(tcp, 53)$, $(tcp, 80)$ and $(tcp, 443)$ are related based on the co-occurrence. One can supply the knowledge that



(a) Communicator (b) Outlook (c) FrameworkService (d) PWConsole

Figure 4: Convergence of dependency finding vs. Number of traces



(a) $I = U$ (b) U = 10% (c) I = 10% (d) Labeling apps. with transient connections.

Figure 5: Sensitivity of static dependency detection. First three charts: Number of static dependencies identified by Macroscope for different thresholds of I (instances) and U (users) for Microsoft Communicator. Last chart: accuracy of detecting the applications with transient connections.

$(tcp, 443)$ a dependency of the web browser traffic. With this seed, the other two services (related to the seed) also become associated with the web browser application.

Main Results: Table 2 synthesizes the evaluation of Macroscope's accuracy in detecting static dependencies by comparison with the ground truth static dependencies and Orion's results. In Table 2, true positives (tp) are the actual static dependencies detected by the algorithms. False positives (fp) are dependencies identified as static (by Macroscope or Orion) but which are not static dependencies in reality. False negatives (fn) are static dependencies that are not detected as such (are "missed") by the applied algorithms.

For each application, we use two different seeds for Orion; one which optimizes the number of false positives and the other optimizes for false negatives. Table 2 summarizes the results. The number of true positives shown in Table 2 for Orion includes the manually specified dependency. To detect the best seeds, we have inspected the results of all the different static dependencies when picked as seeds. For clarity, in Table 2 we use a single entry for the services that are offered on both UDP and TCP (DNS, LDAP). In this case, we count one false positive if the service is spuriously detected as a dependency on at least one of the two transport protocols and count one false negative if none of the two are detected. We aggregate the Orion delay distribution bins to reflect the dependencies used in this paper and improve the Orion data set [9].

Table 2 shows that Macroscope has very few false positives or false negatives: it detects over 95% of the static dependencies and has a low number of false positives, amounting to about 18% of the number of static dependencies. Also, Macroscope offers a significant improvement compared to Orion: in the case of the best seed pick for false negatives, Orion detects about 91% of dependencies (the manual seeds were counted as actually detected), but has a high false positive ratio, over 315% compared to the number of actual static dependencies. In the case of the best seed pick for false positives, Orion has fewer false positives, amounting to about 80% of the number of static dependencies, but detects only about 57% of the static dependencies. Any other seed results in worse performance in at least one of the two metrics.

It is expected that Macroscope would be superior to the approaches that only use packet traces, because it has more information at its disposal. Here we quantify that gain. The results show that Macroscope is able to maintain both low false positives and low false negatives, whereas Orion in this case can attain at most one of these goals.

Macroscope's false positives occur due to mislabeling, resulted from applying its algorithm. As shown, for most applications Macroscope has at most one false positive. `Outlook` stands out in the table because of the large number of false positives. We discovered that the connections to the mail server (all mail in the enterprise is delivered over an MS Exchange connection) tended to reuse ephemeral ports and the connections were long lived; some of these transient ports are used by up to 18 users (34% of all users) and up to 15% of the instances, quite above the usage frequency of many static services in general. Macroscope's false negatives can occur for two reasons: (1) lack of sampling of any connection to that service, or (2) misclassification of Macroscope's algorithm due to infrequent usage, the cause for the false negatives in Table 2.

Sample Period		5s	10s	20s	30s
I=U=10%	tp	95%	85%	84%	82%
	fp	18%	16%	16%	16%
I=U=5%	tp	95%	92%	87%	85%
	fp	52%	46%	83%	82%

Table 3: Effect of sampling rate and filtering thresholds.

Sensitivity Analysis: We now evaluate the effect on detecting static dependencies of the number of traces, of Macroscope's parameters (§ 3.2) and of the sampling rate.

Figure 4 presents the number of static dependencies of four applications as extracted from an increasing subset of user traces (a user is represented by an individual laptop). The figure shows that Macroscope converges towards the actual number of static dependencies as more data is available.

Fig. 5 shows the number of identified static dependencies for MS Office Communicator, given different values of the the instance (I) and user (U) relative thresholds (parameters described in Section 3.2). Both I and U are varied in the range 0%–50%. Fig. 5(a) shows the number of identified static dependencies when $I = U$, in Fig. 5(b) U is set to 10% and only I is varied, and in Fig. 5(c), I is set to 10% and U is varied. As one can see, there is a knee of the selection threshold above which the number of identified static dependencies decreases very slowly (almost remains constant), but may result in a few false negatives; in this context, our goal is to set the threshold at the knee of this curve.

Fig. 5(d) evaluates the sensitivity of the algorithm that classifies applications into using transient connections or not to its parameter T (§3.2). Fig. 5(d) shows how many applications were misclassified out of the 12 applications used in Table 2; a positive value means misclassified as having transient connections while a negative value means misclassified as not having transient connections (there are no cases when applications are misclassified both ways). In the range of 0.8-0.9 for T the classification is flawless. A value too high for T can lead to misclassifying applications as not having transient connections which results in false positive dependencies, while a value too low leaves the dependency detection up to the second part of the algorithm, where infrequently used static dependencies may turn into false negatives.

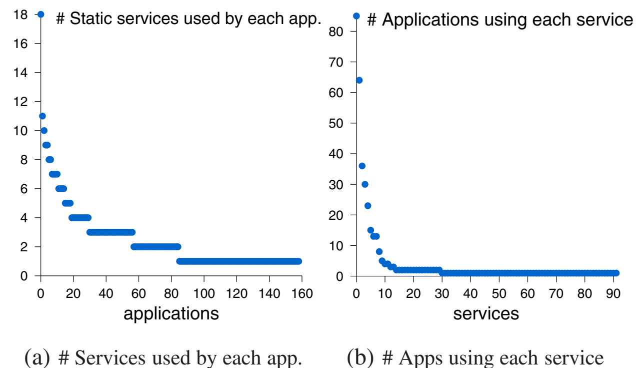
Table 3 shows the effects of the connection table sampling rate together with the effects of Macroscope's thresholds on the static dependency extraction. The table presents the total number of true positives and the total number of false positives detected by Macroscope relative to the total number of actual static dependencies (ground truth); the results are computed by adding up (aggregating) all the dependencies for all the applications in Table 1. We can see that Macroscope's accuracy gradually degrades along with the sampling frequency, and that using a lower threshold to select static dependencies can increase the number of true positives at the expense of an increased number of false positives.

4.3 Dependency Characteristics

Characterizing static dependencies: The distribution of the relations between applications and services is very skewed. A few applications depend on a lot of services, while most applications depend on few services. Also, a few of the services are used by a lot of applications, while most services are used by one application. This type of analysis is unavailable to approaches using only packet traces and we are not aware of previous works profiling the application-service relation for all the applications used in a large enterprise.

Application	Description	# of static dependencies
outlook.exe	MS Outlook	18
iexplore.exe	MS Internet Explorer	11
x1service.exe	Enterprise search engine	10
svchost.exe	MS Windows component dynamically loading DLLs	9
winproj.exe	Component of MS Project	9
communicator.exe	MS Office Communicator	8
firefox.exe	Mozilla Firefox	8
spoolsv.exe	MS Windows service managing spooled print jobs	7
mmc.exe	MS Windows Management Console	7
pwconsole.exe	MS Office Live Meeting	7

Table 4: Top 10 applications with most static dep.



(a) # Services used by each app. (b) # Apps using each service

Figure 6: Application-Service relation distribution.

Fig. 6(a) shows a chart of the distribution of the number of static services used by each of the 159 applications present in the traces as detected by Macroscope; as we have shown, detection is not perfect but is quite accurate. The average number of static dependencies is 2.5, and the median is 1. There are 73 applications (out of 159) having only one static dependency as detected by Macroscope. Among these, most depend on: the proxy used to communicate with machines outside the enterprise (28 applications), Web (10), SSH (10), HTTPS (5) and LDAP (3).

Table 4 presents the top ten applications with the most static dependencies.

Conversely, Fig. 6(b) shows the number of applications using each of the services of the static dependencies. The number of distinct services that applications statically depend on is 92, the average number of applications using one of these static dependencies is 4.6, the maximum is 85 and the median is 1. 62 of these services are specific to a single application. Examples of services used by a single application include POP3 $(tcp, 995)$, Telnet $(tcp, 23)$, NTP $(tcp, 123)$, IMAP $(tcp, 993)$, McAfee Update $(tcp, 8440, 8442)$, HTTP alternate $(tcp, 8080)$, Intel UNS $(tcp, 16993)$, etc.

Table 5 presents the top ten services associated with static dependencies based on the number of different applications that use them.

Besides characterizing the dependencies of the enterprise applications, these results also offer an extra motivation for our work. The results show that: (1) end-point detection is necessary due to the large number of applications using only a few services ; (2) it is difficult to manually create a list of all the services to be verified for faults (in fact only by profiling 12 applications we have identified a

Service	Service Name	# Apps. use it
('tcp', 911)	Enterprise Proxy	85
('tcp', 80)	Web	64
('tcp', 443)	HTTPS	36
('tcp', 9090)	Proxy Discovery	30
('tcp', 135)	EPMAP	23
('tcp', 389)	LDAP AD	15
('tcp', 1025)	Active Dir. (AD)	13
('tcp', 22)	SSH	13
('tcp', 3180)	MS SQL Svr.	8
('tcp', 21)	FTP	5

Table 5: Top 10 services used by most applications

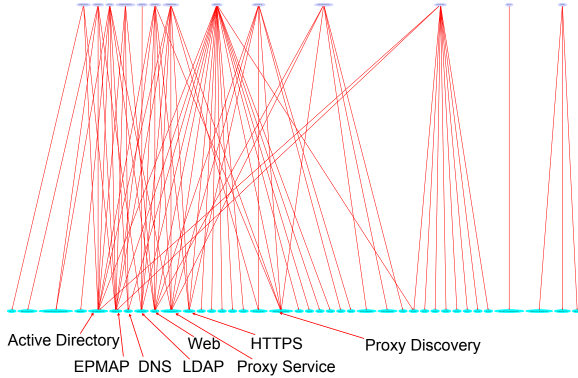


Figure 7: Static dependencies of 12 applications.

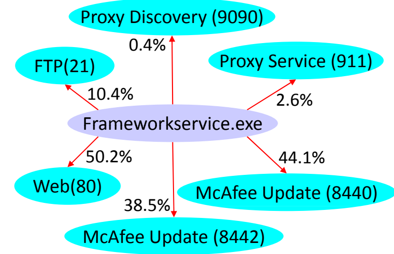


Figure 8: FrameworkService - weighted static dependencies by usage frequency (#uses / #instances).

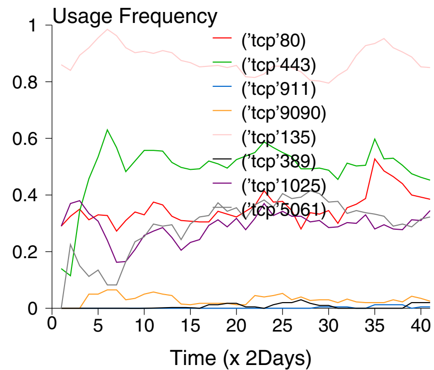


Figure 9: Communicator - dependency usage frequency in time (2 day slices, curvefit).

lot of services not on the engineering list of well known services)[3]; and (3) isolating faults for most applications should be fast (due to the small number of static dependencies).

Transient connections vs static dependencies: Out of the 159 applications encountered, our algorithm identifies 136 as not having transient connections and 23 of these as having transient connections. For applications without transient connections, Macroscope achieves its best accuracy in detecting static dependencies since it detects no false positives and, for large traces, false negatives should be very few. As a surprising side note, several transient services (destinations of application outgoing connections using ephemeral ports) are used by multiple applications, for example three of these services are used by 6 different applications.

Inside vs Outside Services: Out of the 159 applications, 73 communicate only with internal services, 51 communicate only with external services while 35 use both internal and external services. This quantifies the level of service an enterprise IT management department can provide its employees when it conducts fault isolation, server replication, and so on (assuming it cannot fix problems on external servers).

Multi-application dependencies: Fig. 7 shows the static dependencies of all the 12 applications used in Table 2; the upper vertices represent the applications while the lower vertices represent services). There are quite a few common services among these applications, some of these are highlighted in the figure.

Profiled Dependencies: Fig. 8 shows the static dependencies for the McAfee Virusscan Framework Service application, and each dependency is labeled with its usage frequency (*i.e.*, the ratio between the number of instances of the application for which Macroscope detected the use of the dependency and the total number of instances of the application). Note that even if the usage frequency of some dependencies is lower than the 10% threshold used to select static dependencies, they are considered static dependencies because this application is classified as not using transient connections. From Fig. 8 one can see that the two update services are used with comparable frequencies and that half of the instances used the web.

An interesting feature of Macroscope is that it can trace the variation of dependencies in time. Fig. 9 shows the usage frequency for the static dependencies of the Office Communicator for multiple 2 day periods; the usage frequency was extracted from each of the 2 day periods independently. As the chart shows, these weights are fairly stable in time, which indicates that the application code based and its usage pattern were constant throughout the traces. This fine granularity detection is difficult to achieve when using only packet traces due to the larger number of samples required for an accurate measurement.

5. DISCUSSION AND FUTURE WORK

5.1 Macroscope Usage

As we showed previously, Macroscope can generate visualizations for the application and service dependencies at various granularities and scopes. Here, we walk through some concrete use cases

[3] If this list were to be complete then Macroscope can classify connections using this oracle and accuracy would be almost perfect (with at most a few, not sampled, false negatives).

where network operators can use the visualizations effectively to solve a (network) management related activity. We intend the following discussion not as a particular prescription of how Macroscope should be used (the discussion is high level), but more as a way to illustrate the different ways in which Macroscope can aid network operators in carrying out their tasks more effectively.

Fault Localization: One of the common complaints registered with IT helpdesks in enterprise networks relates to applications (email, intranet apps, etc.) not functioning. In such a case, the debugging process normally involves rebooting as a first step (which almost always leads to a diatribe from the user about IT operators and their limited knowledge). The situation could be very different with Macroscope in place; the operator could generate a graph such as Fig. 3(a) and compare the last snapshot of activity (where the fault occurred) to a snapshot derived from a different time. Missing edges in the visualization graph are easily identified.

While this does not describe the entire troubleshooting process, the fact is that a tool such as Macroscope allows the operators to reason about the problem aided by the visualization, rather than recite from a canned playbook ("step 1: reboot computer and try again"). Macroscope can also generate application specific visualizations across a group of malfunctioning applications as shown in Fig. 7 or across a group of users (as shown in Fig. 3(c)). If the IT operator(s) receive a number of trouble tickets, such a visualization can tell them what is working, and what else needs to be looked at. Given this information, the operators can easily generate a plausible set of servers/services to check. The visualizations generated by Macroscope can also improve efficiency; nodes in the graph can be ordered by degree or weight and operators can search for faults "greedily".

Load Balancing, Replication and Capacity Planning: In most modern enterprise networks, important infrastructure services are replicated at each geographic site (to reduce latency and cut down bandwidth costs). However, the utilizations across these servers varies dramatically. A routine network maintenance task is that of placing additional resources (servers, storage, etc) to improve performance. Macroscope can issue queries that can inform how users are "pinned" to servers across locations and to identify the locations that can benefit the most from an upgrade. By the same coin, the visualizations can also inform network operators how many applications and users will be impacted by a single service outage.

Consider Figures 3(c) and 7. Clearly, not all the services are equally significant. The network that wants to upgrade gets the best "bang for the buck" by upgrading hardware (or managing the utilization in some other way) for the services/servers that have a lot of incident edges.

Detect Malicious or Misbehaving Applications: Macroscope can enable operators to identify and pinpoint misbehaving or malicious applications. In many malware instances, a popular application is replaced by a trojaned version whose behavior is different from the original application. Macroscope enables the building of application "profiles" that can help detect these changes. To describe it at a high level, the static dependencies and their associated properties (usage frequency, traffic weight, destinations) essentially describe the profile of an application. This profile can be built over time (and for individual users). The historical computed profiles can be compared to profiles generated from more recent data. Fig. 9 shows the usage frequency of different services over a window of time (one of the elements in the profile). It is very likely that a trojan application that replaces it will perturb the usage frequency, which can be detected. Moreover, Macroscope can detect applications doing port scanning by looking at the number of rejected connections.

5.2 Future Work

Causal order of dependencies: Macroscope can evaluate the causality relation between dependencies. We say two dependencies A and B are causally ordered (and write A->B) if A usually finishes just before B is started (and the vice-versa is not true). In this context, "before" means within a time window. These constructions allow network operators to understand how services are composed, to set an order among the services to be verified for faults and to improve overall application responsiveness (the latency of the application is related to the latency of the longest path in such a graph). We have implemented this algorithm, but due to the unavailability of ground truth for the causal order (to validate the results), we do not include any such results in this paper.

Service side dependencies: If server side traces are available in addition to client side traces (used in this paper), our work can be extended to create complex dependency graphs, of depth higher than one; *e.g.*, not only capture that application A depends on service S, but also that S itself (the application implementing it) depends on S_1, and that S_1 depends on S_2, and so on.

Kernel-level tracing: Macroscope's tracing method can be further improved. The connection table sampling can be replaced with fine granularity monitoring of system calls in the kernel. In this way, UDP dependencies can be detected without the use of packet level traces; however, packet level traces would still be useful for profiling dependencies (*e.g.*, associate them traffic volumes) and for identifying unsuccessful connections. Also, by tracing interprocess communications, this method may more easily detect the dependencies accessed indirectly through the use of an OS service such as lsass. The disadvantage of such a method may be the tracing overhead, which is proportional to the network traffic (*i.e.*, the overhead is higher when the computer is busier), unlike the almost constant overhead of the connection table sampling.

Identifying applications: In the future, the Tracer could identify applications by using a hash of the executable file instead of the executable name, to avoid name collisions. However, this may thoroughly reduce the available samples, *e.g.*, different compilers can generate different executable files.

6. RELATED WORK

In recent years, extracting networked application dependencies has been recognized as a critical task in the larger problem of managing enterprise networks and data-centers. Previous efforts to address this problem, notably Sherlock [5], eXpose [12], and Orion [9], are based on network-level tracing. That is, they operate solely on network traffic traces to extract the dependencies. Sherlock uses time correlation of packets between different services and computes conditional probability to decide dependencies [5]. eXpose partitions the packet trace into time windows and looks for service dependencies that occur within the same time window by computing a modified JMeasure from an activity matrix [12]. Orion computes spikes of delay distribution of flow pairs and uses thresholds to decide if two services depend each other [9]. Constellation [6] uses machine learning algorithms to extract dependencies from the timings of packets sent and received. These methods require no modification or instrumenting of individual end-hosts and hence can be deployed quite easily. However, as we show in this paper, they require manual steps and can have high false negatives and positives. In Macroscope, we take a different approach, choosing to instrument end-hosts to collect application level data. Thus, Macroscope is harder to deploy but the benefit is that we overcome the challenges faced by methods relying only on network data.

To address the challenge of debugging distributed systems, researchers have studied application [10] or middleware execution paths [8] tracing. These systems focus on a particular distributed system and require tracing in a particular tracing framework and modification of applications; thus, they are hard to do in today's complex heterogeneous service environment. Project 5 [4] and WAP5 [17] aim to debug distributed systems by taking a black-box debugging approach. They passively monitor messages and infer causal execution paths from these message traces. Magpie [7] and Pinpoint [8] take a gray-box debugging approach that combines prior knowledge, observations, and inference. Pip [16] detects deviation of distributed systems by comparing actual behavior with expected behavior. In contrast, Macroscope focuses on automatic dependency extraction of networked applications for a large number of applications and services, and does this without application modification.

Network monitoring and application traffic classification has been studied extensively in the past. BLINC [14] classifies traffic by using the behavioral patterns at the social, functional and application level. Kannan et al. [13] proposed algorithms that semi-automatically extract application session structures from network connection-level traces. The term session denotes a set of connections related to a single application task (*e.g.*, FTP session).

7. CONCLUSIONS

We have presented Macroscope, a system that applies end-point tracing to detect application dependencies on services. Macroscope is completely automatic, simple enough to be practical, accurate in detecting static dependencies and enables application profiling. In the future, we plan to develop network management algorithms for fault isolation and service replication that incorporate Macroscope's profiles of the dependency relationship between applications and services.

8. REFERENCES

[1] Bro. `http://www.bro-ids.org/`.

[2] Taming technology sprawl. `http://online.wsj.com/article/SB120156419453723637.html?mod=techno%logy_main_promo_left`.

[3] Winpcap. `http://www.winpcap.org/`.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

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

[6] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma. Constellation: automated discovery of service and host dependencies in networked systems. *MSR-TR-2008-67*, 2008.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[8] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[9] X. Chen, M. Zhang, Z. M. Mao, and V. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, 2008.

[10] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[11] F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, and K. Papagiannaki. Exploiting temporal persistence to detect covert botnet channels. In *RAID'09*.

[12] S. Kandula, R. Chandra, and D. Katabi. What's going on? learning communication rules in edge networks. In *ACM SIGCOMM*, 2008.

[13] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-automated discovery of application session structure. In *IMC*, 2006.

[14] T. Karagiannis, K. Papaginnaki, and M. Faloutsos. BLINC: Multilevel traffic classification in the dark. In *ACM SIGCOMM*, 2005.

[15] Z. Kerravala. Configuration management delivers business resiliency. In *The Yankee Group*, 2002.

[16] P. Reynold, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[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*, 2006.