Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions

Xu Chen[§] Ming Zhang[†] Z. Morley Mao[§] Paramvir Bahl[†]
[†] *Microsoft Research* [§] *University of Michigan*

Abstract – Large enterprise networks consist of thousands of services and applications. The performance and reliability of any particular application may depend on multiple services, spanning many hosts and network components. While the knowledge of such dependencies is invaluable for ensuring the stability and efficiency of these applications, thus far the only proven way to discover these complex dependencies is by exploiting human expert knowledge, which does not scale with the number of applications in large enterprises.

Recently, researchers have proposed automated discovery of dependencies from network traffic [8, 18]. In this paper, we present a comprehensive study of the performance and limitations of this class of dependency discovery techniques (including our own prior work), by comparing with the ground truth of five dominant Microsoft applications. We introduce a new system, Orion, that discovers dependencies using packet headers and timing information in network traffic based on a novel insight of delay spike based analysis. Orion improves the state of the art significantly, but some shortcomings still remain. To take the next step forward, Orion incorporates external tests to reduce errors to a manageable level. Our results show Orion provides a solid foundation for combining automated discovery with simple testing to obtain accurate and validated dependencies.

1 Introduction

Modern enterprise IT infrastructures comprise of large numbers of network services and user applications. Typical applications, such as web, email, instant messaging, file sharing, and audio/video conferencing, operate on a distributed set of clients and servers. They also rely on many supporting services, such as Active Directory (AD), Domain Name System (DNS), Kerberos, and Windows Internet Name Service (WINS). The complexity quickly adds up as different applications and services must interact with each other in order to function properly. For instance, a simple webpage fetch request issued by a user can involve calls to multiple services mentioned above. Problems at any of these services may lead to failure of the request, leaving the user frustrated and IT managers perplexed.

We say one service *depends* on the other if the former requires the latter to operate properly. Knowledge of service dependencies provides a basis for serving critical network management tasks, including fault localization, reconfiguration planning, and anomaly detection. For instance, Sherlock encapsulates the services and network components that applications depend on in an *inference graph* [8]. This graph is combined with end-user observations of application performance to localize faults in an enterprise network. When IT managers need to upgrade, reorganize, or consolidate their existing applications, they can leverage the knowledge of dependencies of their applications to identify the services and hosts that may potentially be affected, and to prevent unexpected consequences [9]. When continually discovered and updated, dependencies can help draw attention to unanticipated changes that warrant investigation.

While there are network management systems that perform topology and service discovery [21, 12], IT managers currently do not have proven tools that help to discover the web of dependencies among different services and applications. They commonly rely on the knowledge from application designers and owners to specify these dependencies. These specifications can be written in languages provided by commercial products, such as Mercury MAM [4] and Microsoft MOM [5]. While straightforward, this approach requires significant human effort to keep up with the evolution of the applications and their deployment environment. This becomes a massive problem for large enterprises with thousands of applications. For example, a survey conducted by the Wall Street Journal in 2008 found that HP and Citigroup each operate over 6,000 and 10,000 line-of-business (LOB) applications [6]. Microsoft runs over 3,100 LOB applications in its corporate network, most

of which have no documentations describing their dependencies. More recently, there have been a few attempts to automate dependency discovery by observing network traffic patterns [8, 9, 18]. However, there is very little understanding about *how well these approaches work, where and how they fall short, and whether their limitations can be overcome without human intervention.*

There are a few challenges in designing a system that discovers dependencies in a complex enterprise network: First, it should require minimal human effort; Second, it should be applicable to a diverse set of applications; Third, it should be non-intrusive to applications and be easily deployable; and Fourth, it should scale with the number of services, applications, and hosts in the network. These challenges are hard to address, especially given that expert knowledge of application internals cannot be assumed for thousands of new and legacy applications. Incorporating such knowledge in a system is a formidable task.

We have built a system called Orion that overcomes all these challenges. Specifically, it discovers dependencies by passively observing application traffic. It uses readily available information contained in IP, TCP, and UDP headers without parsing higher-level application-specific protocols. Most of the computation is done locally by individual hosts, and the amount of information exchanged between hosts is small.

In this paper, we describe our experiences in designing, implementing, and deploying Orion in Microsoft's corporate network. We make the following contributions:

- We introduce a new dependency discovery technique based on traffic delay distributions. For the applications we studied, we can narrow down the set of potential dependencies by a factor of 50 to 40,000 with negligible false negatives.
- We are the first to extract the dependencies for five dominant enterprise applications by deploying Orion in a portion of Microsoft's corporate network that covers more than 2,000 hosts. These extracted dependencies can be used as input to create more realistic scenarios for the evaluation of various fault localization and impact analysis schemes
- We comprehensively study the performance and limitations of a class of dependency discovery techniques that are based on traffic patterns (including Orion). The results reveal insights into the shortcomings of such techniques when they are applied to real-world applications.
- We conduct extensive experiments to compare Orion with the state of the art (Sherlock [8] and eXpose [18]). While their false negatives are similar, the false positives of Orion are 10-95% fewer than Sherlock and 94-99% fewer than eXpose. Even though Orion cannot avoid all the false positives, we can obtain accurate dependencies using simple external tests.

In the rest of the paper, we elaborate on our techniques, implementation, and evaluation of automated dependency discovery. Additionally, we provide concrete examples about how to use extracted dependencies for fault diagnosis and reconfiguration planning.

2 Related Work

Many sophisticated commercial products, such as EMC SMARTS [1], HP OpenView [2], IBM Tivoli [3], Microsoft MOM [5], and Mercury MAM [4], are used for managing enterprise networks. Some of them provide support for application designers to specify the dependency models. However, these approaches require too much manual effort and are often restricted to a particular set of applications from the same vendor.

There is a large body of prior work on tracing execution paths among different components in distributed applications. For example, Pinpoint instruments the J2EE middleware on every host to track requests as they flow through the system [15]. It focuses on mining the collections of these paths to locate faults and understand system changes. X-Trace is a cross-layer, cross-application framework for tracing the network operations resulting from a particular task [16]. The data generated by X-Trace can also be used for fault detection and diagnosis. Both Pinpoint and X-Trace require all the distributed applications to run on a common instrumented platform. This is unlikely to happen in large enterprise networks with a plethora of applications and operating systems from different vendors.

Magpie is a toolchain that correlates events generated by operating system, middleware, and application to extract individual requests and their resource usage [10]. However, it heavily relies on expert knowledge about the systems and applications to construct schemas for event correlation.

Project5 [7] and WAP5 [22] apply two different correlation algorithms to message traces recorded at each host to identify the causality paths in distributed systems. They both focus on debugging and profiling individual applications by determining the causality between messages. The message correlation in Project5 is done by computing the cross correlation between two message streams. WAP5 developed a different message correlation algorithm based on the assumption that causal delays follow an exponential distribution for wide-area network applications. In contrast, Orion focuses on discovering the service dependencies of network applications.

Brown *et al.* propose to use active perturbation to infer dependencies between system components in distributed

applications [14]. While this methodology requires little knowledge about the implementation details of the applications, it has to use a *priori* information to learn the list of candidate services to perturb, which is inherently difficult to obtain in large enterprise networks.

The closest prior work to Orion is Sherlock [8, 9] and eXpose [18]. The former focues on localizing faults using dependency graphs while the latter focuses on extracting and clustering significantly dependent flow groups. They both use traffic co-occurrence to identify dependencies. To determine whether one service depends on the other, they compute either the conditional probability [8] or the JMeasure [18] of the two services within a fixed time window. A key issue with both approaches is the choice of the time window size. In fact, it is fundamentally difficult to pick an appropriate window size that attains a good balance between false positives and false negatives. While they seem to extract *certain* meaningful dependencies, neither of them quantified the accuracy of their results in terms of how many true dependencies they missed or how many false dependencies they mistakenly inferred. In contrast, our technique does not rely on any co-occurrence window size. Through field deployment, we show that Orion extracts dependencies much more accurately than Sherlock [8] and eXpose [18] for a variety of real-world enterprise applications. We also validated our results with the owners of all these applications.

3 Goal & Approach

Given an enterprise network application, our goal is to discover the set of services on which it depends in order to perform its regular functions. Before describing the technical details, we first introduce a few concepts and terms that will be used in the paper. We then motivate our design decisions, outline our approach, and discuss our challenges.

3.1 Services and dependencies

Enterprise networks consist of numerous services and user applications. Applications, such as web, email, and file sharing, are directly accessed by users. Most applications depend on various network services to function properly. Typical network services include Active Directory (AD), Domain Name System (DNS), and Kerberos. These services provide basic functions, such as name lookup, authentication, and security isolation. An application or a service can run on one or more hosts.

In this paper, we do not make a formal distinction between services and applications, and we use both terms interchangeably. We use a three-tuple $(ip, port, proto)$ to denote either an application or a service. In an enterprise network, an ip normally maps to a unique host and the $port$ and $proto$ often identify a particular service running on that host. Many ports under 1024 are reserved for well-known services, such as Web (80, TCP), DNS (53, TCP/UDP), Kerberos (88, TCP/UDP), WINS (137, TCP/UDP), and LDAP (389, TCP/UDP). Another type of service is RPC-based and does not use well-known ports. Instead, these services register an RPC port between 1025 and 65535 when a host boots up. Clients who intend to use these services will learn the RPC service port through a well-known port of RPC endpoint mapper (135).

While it is a common practice to associate a service with an $(ip, port, proto)$ tuple, we may define service at either coarser or finer granularities. On the one hand, many enterprises include fail-over or load balancing clusters of hosts for particular services, which can be denoted as $(ipCluster, port, proto)$. Other services, such as audio and video streaming, could use any port within a particular range, which can be denoted as $(ip, portRange, proto)$. On the other hand, multiple services may share the same port on a host in which case we must use additional service-specific information to identify each of them.

We define service A to depend on service B, denoted as $A \rightarrow B$, if A requires B to satisfy *certain* requests from its clients. For instance, a web service depends on DNS service because web clients need to lookup the IP address of the web server to access a webpage. Similarly, a web service may also depend on database services to retrieve contents requested by its clients. Note that $A \rightarrow B$ does not mean A must depend on B to answer *all* the client requests. In the example above, clients may bypass the DNS service if they have cached the web server IP address. The web server may also bypass the database service if it already has the contents requested by the clients.

3.2 Discovering dependencies from traffic

We consider three options in designing Orion to discover dependencies of enterprise applications: i) instrumenting applications or middlewares; ii) mining application configuration files; and iii) analyzing application traffic. We bypass the first option because we want Orion to be easily deployable. Requiring changes to existing applications or middlewares will deter adoption.

Configuration files on hosts are useful sources for discovering dependencies. For instance, DNS configuration files reveal information about the IP addresses of the DNS servers, and proxy configuration files may contain the IP addresses and port numbers of HTTP and FTP proxies. However, the configuration files of different applications may be stored in different locations and have different formats. We need application-specific knowledge to parse and extract dependencies from them. Moreover, they are less useful in identifying dependencies that are dynamically constructed. A notable example is that web browsers often use automatic proxy discovery

protocols to determine their proxy settings.

In Orion, we take the third approach of discovering dependencies by using packet headers (*e.g.*, IP, UDP, and TCP) and timing information in network traffic. Such information is both easy to obtain and common to most enterprise applications. Note that it is natural to develop application-specific parsers to understand the application traffic, *e.g.*, when a message starts or ends and what the purpose of the message is. Such detailed knowledge is helpful in determining the dependency relationships between the traffic of different services, eliminating ambiguities, and hence improving the accuracy of dependency inference. Nonetheless, developing parsers for every application requires extensive human effort and domain knowledge. For this reason, we refrain from using any packet content information besides IP, UDP, and TCP headers.

Orion discovers dependencies based on the observation that *the traffic delay distribution between dependent services often exhibits "typical" spikes that reflect the underlying delay for using or providing these services.* While conceptually simple, we must overcome three key challenges. First, it is inherently difficult to infer dependencies from application traffic without understanding application-specific semantics. Packet headers and timing information are often insufficient to resolve ambiguity. This may cause us to mistakenly discover certain *service correlations* (false positives) even though there are no real dependencies between the services. Second, packet headers and timing information can be distorted by various sources of noise. Timing information is known to be susceptible to variations in server load or network congestion. Third, large enterprise networks often consist of tens of thousands of hosts and services. This imposes stringent demand on the performance and scalability of Orion. We introduce new techniques to address each of the three challenges.

Orion has three components. The *flow generator* converts raw network traffic traces into flows. The purpose is to infer the boundaries of application messages based only on packet headers and timing information. The *delay distribution calculator* identifies the potential services from the flows and computes delay distributions between flows of different services. Finally, the *dependency extractor* filters noise and discovers dependencies based on the delay distributions. We describe each of them in detail in the subsequent sections.

4 Flow Generation

In client-server applications, services and their clients communicate with each other using requests and replies. For convenience, we use a *message* to denote either a request or a reply. Orion discovers service dependencies by looking for the time correlation of messages between different services. For instance, it infers the dependency of a web service on a DNS service by observing DNS messages precede web messages. While time correlation may not always indicate a true dependency, we rely on a large number of statistical samples to reduce the likelihood of false positives.

In reality, we are only able to observe individual packets in the network instead of individual messages. Multiple packets may belong to the same message and the time correlation among themselves do not explicitly convey any dependency information. If we consider the time correlation between every possible pair of packets, we could introduce: i) too much redundancy because we count the correlation between two dependent messages multiple times; and ii) significant computation overhead because the number of packets is much larger than the number of messages.

While it is desirable to aggregate packets into messages for dependency inference, this is nontrivial because we do not parse the application payload in packets. Given that most services use UDP and TCP for communications, we aim to both reduce computation overhead and keep sufficient correlation information by aggregating packets into *flows* based on IP, TCP, and UDP headers and timing information:

TCP packets with the same five tuple (locIP, locPt, remIP, remPt, proto) are aggregated into a flow whose boundary is determined by either a timeout threshold, or TCP SYN/FIN/RST, or KEEPALIVE. Any two consecutive packets in a flow must not be interleaved by an interval longer than the timeout threshold. TCP SYN/FIN/RST flags are explicit indications of the start or the end of flows. Certain services with frequent communications may establish persistent connections to avoid the cost of repetitive TCP handshakes. They may use KEEPALIVE messages to maintain their connections during idle periods. We also use such messages to identify flow boundaries.

UDP packets with the same five tuple (locIP, locPt, remIP, remPt, proto) are aggregated into a flow solely based on timing information, since UDP is a connectionless protocol. Any two consecutive packets in a flow must not be interleaved by an interval longer than the timeout threshold.

We will evaluate the impact of flow generation on our inference results in Section 7.2.3.

5 Service Dependency Discovery

In this section, we first present an overview of our approach to discovering service dependencies. We then describe the details of our approach, including how to calculate delay distributions between different services based on flow information and how to extract dependencies from delay distributions.

5.1 Overview

Orion discovers service dependencies by observing the time correlation of messages between different services. Our key assumption is if service A depends on service B, the delay distribution between their messages should not be random. In fact, it should reflect the underlying processing and network delays that are determined by factors like computation complexity, execution speed, amount of communication information, and network available bandwidth and latency. For instance, a web client may need to go through DNS lookup and authentication before accessing a web service. The *message delay* between the DNS and web services is the sum of: 1) the time it takes for the client to send an authentication request after the DNS reply is received; 2) the transmission time of the authentication request to the authentication service; 3) the processing time of the authentication request by the authentication service; 4) the transmission time of the authentication reply to the client; and 5) the time it takes for the client to send a web request after the authentication reply is received. Assuming the host and network load are relatively stable and relatively uniform service processing overhead, this message delay should be close to a "typical" value that exhibits as a "typical" spike in its delay distribution.

There could be multiple typical values for the message delay between two dependent services, each of which corresponds to a distinct execution path in the services. In the above example, the client may bypass the authentication if it has a valid authentication ticket cached. As a result, the message delay between the DNS and web services will simply be the time it takes for the client to send a web request after the DNS reply is received. This will lead to two typical spikes in the delay distribution.

While there could be thousands of hosts in the network, Orion focuses on discovering service dependencies from an individual host's perspective. Given a host, it aims to identify dependencies only between services that are either used or provided by that host. This implies the dependency discovery algorithm can run independently on each host. This is critical for Orion to scale with the network size. By combining the dependencies extracted from multiple hosts, Orion can construct the dependency graphs of multi-tier applications. The dependency graphs of a few three-tier applications are illustrated in Section 7.1.

In the remainder of the section, we will describe a few important techniques in realizing Orion. This includes how to scale with the number of services, reduce the impact of noise, and deal with insufficient number of delay samples.

5.2 Delay distribution calculation

Orion uses the delay distribution of service pairs to determine their dependency relationship. Given a host, we use $(IP_{loc}, Port_{loc}, proto)$ and $(IP_{rem}, Port_{rem}, proto)$ to represent the local and remote services with respect to that host. We are interested in two types of dependency: i) Remote-Remote (RR) dependency indicates the host depends on one remote service to use another remote service. This type of dependency is commonly seen on clients, *e.g.*, a client depends on a DNS service $(DNS_{rem}, 53_{rem}, UDP)$ to use a web service $(Web_{rem}, 80_{rem}, TCP)$; ii) Local-Remote (LR) dependency indicates the host depends on a remote service to provide a local service. This type of dependency is commonly seen on servers, *e.g.*, the web service on a server $(Web_{loc}, 80_{loc}, TCP)$ depends on an SQL database service $(SQL_{rem}, 1433_{rem}, TCP)$ to satisfy the web requests from its clients.

Orion calculates the delay distribution based on the flow information generated in the previous stage (Section 4). Since a host may observe many flows over a long time, Orion uses two heuristics to reduce the CPU and memory usage. First, it calculates the delays only between flows that are interleaved by less than a predefined time window. Ideally, the time window should be larger than the end-to-end response time of any service S in the network (from the time a client sends the first request to a service that S depends on till the time the client receives the first reply from S) to capture all the possible dependencies of S. In single-site enterprise networks, a time window of a few seconds should be large enough to capture most of the dependencies that we look for, given the end-to-end response time of services in such networks is typically small. In multi-site enterprise networks which are interconnected via wide-area networks, we may need a time window of a few tens of seconds. We currently use a three-second time window for our deployment inside Microsoft's corporate network.

The second heuristic to reduce overhead is based on the observation that a host may communicate over a large number of services, many of which may not be persistent enough to attract our interest. For instance, clients often use many ephemeral ports to communicate with servers and the "services" corresponding to these ephemeral ports are never used by other services. Orion keeps track of the number of flows of each service in the recent past and uses a flow count threshold to distinguish between ephemeral and persistent services. It calculates and maintains delay distributions only for persistent services pairs. The flow count threshold is determined by the minimum number of statistical samples that are required to reliably extract dependencies. We use a default threshold of 50 in the current system. Note that the window size and the flow count threshold only affect the the computation and storage overhead but not the accuracy of Orion.

Since Orion does not parse packet payload to under-

stand the actual relationship between flows, it simply calculates the delay between every pair of flows, *e.g.*, $(LocIP_1, LocPt_1, RemIP_1, RemPt_1, proto_1)$ and $(LocIP_2, LocPt_2, RemIP_2, RemPt_2, proto_2)$, that are interleaved by less than the time window. We treat each delay sample as a possible indication of both a RR dependency, *e.g.*, $(RemIP_2, RemPt_2, proto_2) \rightarrow (RemIP_1, RemPt_1, proto_1)$, and an LR dependency, *e.g.*, $(LocIP_1, LocPt_1, proto_1) \rightarrow (RemIP_2, RemPt_2, proto_2)$, and add it to the delay distributions of both service pairs. This implies that there could be "irrelevant" samples in the delay distribution that do not reflect a true dependency between the service pair. This could be problematic if a delay distribution is dominated by such irrelevant samples. Nonetheless, in our current deployment, we identified only one false negative that is possibly caused by this problem (Section 7.2).

Suppose a host uses m remote services and provides n local services, Orion needs to maintain delay distributions for $(m \times m)$ RR service pairs and $(m \times n)$ LR service pairs for that host in the worse case. Because Orion discovers dependencies for each host independently, m and n are determined by the services observed at that host rather than all the services in the network. This allows Orion to scale in large enterprises with many services. We evaluate the scalability of Orion in Section 7.4.1.

5.3 Service dependency extraction

We now describe three important issues related to extracting dependencies from delay distributions: mitigating the impact of random noise, detecting typical spikes, and dealing with insufficient samples.

5.3.1 Noise filtering & spike detection

The delay distribution of service pairs calculated from flow information is stored as a histogram with a default bin width of 10ms. There are 300 bins if we use a three-second time window. We denote *bin-height* as the number of delay samples that fall into each bin.

Raw delay distributions may contain much random noise due to host and network load variations. The noise will introduce numerous random spikes in the delay distribution, which could potentially interfere with the detection of typical spikes. Realizing this problem, we treat each delay distribution as a signal and use signal processing techniques to reduce random noise. Intuitively, the number of typical spikes corresponds to the number of commonly-executed paths in the services, which is at most a few for all the services we study. In contrast, random noise tends to introduce numerous random spikes in the signal, which is more evident in the high frequency spectrum.

This prompts us to use Fast Fourier Transform (FFT) to decompose the signal across the frequency spectrum

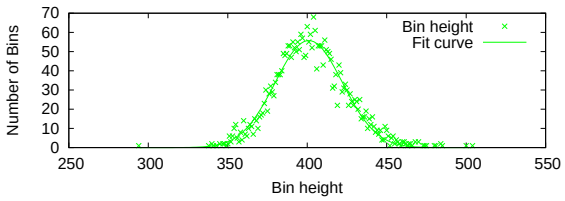

Figure 1: *Bin-heights fit well with normal distribution*

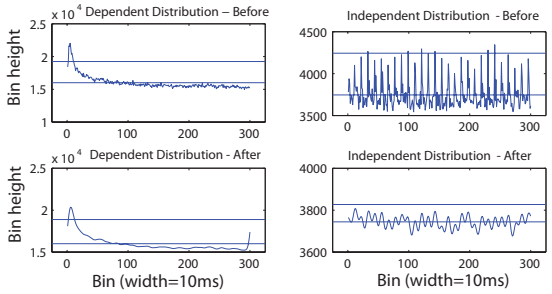

Figure 2: *Delay distributions before and after filtering*

and apply a low-pass filter to mitigate the impact of random noise [13]. The choice of low-pass filter reflects the trade-off between tolerance to noise and sensitivity to typical spikes. We have tried a few commonly-used filters and find that Kaiser window ($50 \leq \beta \leq 200$) [17] achieves a reasonable balance between the two goals. The effect of filtering is not particularly sensitive to the choice of β within the above range and we use $\beta = 100$ in the current system.

For each delay distribution, we plot the corresponding bin-height distribution. Each point in the bin-height distribution represents the number of bins with a particular bin-height. Interestingly, we find these bin-height distributions closely follow normal distribution, as illustrated by an example in Figure 1. Based on this observation, we detect typical spikes whose bin-heights are among the top $x\%$ in the bin-height distribution. The parameter x determines the degree of tolerance to noise and sensitivity to typical spikes. In practice, we find x between 0.1% and 1% works pretty well. We use a bin-height threshold of $(mean + k \times stdev)$ to detect typical spikes, where $mean$ and $stdev$ are the the mean and standard deviation of the bin-heights. With $k = 3$, we will detect typical spikes whose bin-heights are among the top 0.3% in the bin-height distribution.

Figure 2 shows two examples of noise filtering and spike detection. The two horizontal lines in each graph represent the $mean$ and the $(mean + k \times stdev)$ of the bin-heights. The two graphs on the left are the delay distributions of a true dependency before and after filtering. Clearly, filtering does not eliminate the typical spike. The two graphs on the right are the delay distributions of a non-dependency. In this case, filtering significantly reduces the random spikes that could have led to false positives. Note that noise filtering is effective only

against random spikes in delay distributions. It has little effect on other non-typical spikes introduced by certain unexpected service pair interaction.

5.3.2 Client & service aggregation

Orion requires a reasonably large number of samples in a delay distribution to reliably detect typical spikes. To avoid inaccuracy due to a lack of samples, it ignores delay distributions in which the number of samples is fewer than the number of bins in the histogram. This could be problematic for individual clients who use many remote services infrequently. Fortunately, clients in an enterprise network often have similar host, software, and network configurations. They also have a similar set of dependencies when using a particular remote service. Orion aggregates the delay distributions of the same service pairs from multiple clients to improve the accuracy of dependency extraction. Note that the service dependencies of clients may have slight difference, *e.g.,* due to different software versions. By doing client aggregation, Orion will discover the aggregated dependencies of all the clients, which could be a superset of the dependencies of each individual client.

To facilitate client aggregation, we may have to perform service aggregation as well. Many enterprise networks use a failover or load balancing cluster of servers to provide a particular service. Clients may communicate with any of the servers in the cluster. By treating such a cluster of servers as a whole and representing the service with a $(ipCluster, port, proto)$, it provides us much more opportunities in performing client aggregation. Similarly, a server may provide the same service (*e.g.,* audio and video streaming) on any port in a particular range. We may represent such a service with $(ip, portRange, proto)$ to help client aggregation.

While client and service aggregations help to improve accuracy, they require extra information beyond that embedded in the packet headers. In Microsoft's corporate network, most servers are named based on a well-defined convention, *e.g.,* xxx-prxy-xx is a proxy cluster and xxx-dns-xx is a DNS cluster. We develop a simple set of naming rules to identify the clusters. We also examine the configuration files to obtain the port range for a few services that do not use a fixed port. In enterprises where such naming convention does not exist, we may have to rely on IT managers to populate the host-to-cluster mapping information. Normally, this type of information already exists in large enterprises to facilitate host management. We can also leverage existing work on service discovery to obtain this information [11].

5.4 Discussion

We focus on discovering the service dependencies for client-server applications, which are dominant in many enterprise networks. Their dependencies change only when they are reconfigured or upgraded. As a result, the dependencies that we aim to discover are usually stable over several weeks or even months. As we will see in Section 7.3, this is critical because Orion may need a few days of statistical samples to reliably infer dependencies. Recently, peer-to-peer (p2p) applications have gained popularity in enterprise networks. In contrast to traditional client-server applications, they are designed to be highly resilient by dynamically changing the set of hosts with which a client communicates. The dependencies of these applications could change even during short periods of time. As future work, we plan to investigate how to discover the dependencies of p2p applications.

Orion discovers service dependencies by looking for typical spikes in the delay distributions of service pairs. While conceptually simple, false positives and false negatives may arise due to various types of noise (*e.g.,* different hardware, software, configuration, and workload on the hosts and load variation in the network) or unexpected service pair interaction (*e.g.,* although service $A \rightarrow B$, the messages of A and B could be triggered by other services). While the impact of random noise can be mitigated by taking a large number of statistical samples, unexpected service pair interaction is more problematic. In Section 7.2, we will illustrate examples of false positives where non-dependent service pairs show strong time correlations.

We emphasize that the issues above are not just specific to Orion but to the class of dependency discovery techniques based on traffic patterns. We will comprehensively evaluate and compare their performance using five real-world enterprise applications in Section 7. In spite of these issues, Orion is surprisingly effective in discovering service dependencies. In fact, it not only discovers the majority of the true dependencies but also successfully eliminates most of the false positives. While some false positives are unavoidable, their numbers are sufficiently small to be removed with some simple testing.

Orion requires a large number of statistical samples to reliably extract service dependencies. This makes it less applicable to services which are newly deployed or infrequently used. It may also miss dependencies that rarely occur, such as DHCP. One possible solution is to proactively inject workloads to these services to help accumulate sufficient number of samples.

6 Implementation

We now describe the implementation of Orion as shown in Figure 3. Orion has three major components that run on a distributed set of hosts in an enterprise network. The *flow generators* convert raw traffic traces into flow records in real time. The *delay distribution calculators* run on the same set of hosts as the flow generators. They continually update the delay distributions for all the service pairs relevant to the services that administrators are

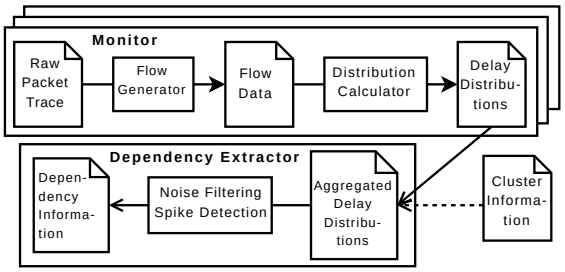

Figure 3: *System architecture*

interested in. A centralized *dependency extractor* collects and analyzes the delay distributions from multiple hosts to extract dependencies.

In a fully distributed deployment, each host runs a flow generator and a delay distribution calculator to build its own delay distributions (host-based deployment). Such organization scales well given the localized nature of computation. Traffic traces can be captured by WinPcap or TDI drivers (a Windows API). The latter allows us to get the port and protocol information even when traffic is encrypted by IPSec. When such a fully distributed deployment is not possible, Orion can operate on packet sniffers connected to span ports on switches and routers that are close to hosts (network-based deployment). It will build the delay distributions for each host on the same subnet.

6.1 Flow generator

A flow generator reads the $(ip, port, proto)$ and timing information from the raw traffic traces and outputs flow records. It maintains a hash table in memory, which keeps track of all the active flow records using the five-tuple $(locIP, locPt, remIP, remPt, proto)$ as keys. $locIP$ corresponds to a monitored host. Each flow record contains a small amount of information, *e.g.*, the timestamps of the first and the last packets, the direction and TCP flag of the last packet, and the current TCP state of the flow. Based on this information, we can determine whether to merge a new packet into an existing flow record, flush an existing flow record, or create a new one. To keep the hash table from growing excessively, we expire old flow records periodically. The current version is implemented in C using the libpcap library with roughly 2K lines of code.

6.2 Delay distribution calculator

The delay distribution calculator keeps a buffer that holds the recent flow records of each monitored host. The flow records in the buffer are sorted based on their starting time and ending time. When a new flow record arrives, we use its starting time and ending time minus the three-second time window to expire old records in the buffer. For each monitored host, we also maintain a set of delay distributions for the service pairs related to that host. We go through all the existing flow records in the buffer, compute the delay between the new flow record and each

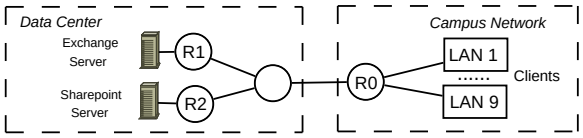

Figure 4: *Deployment in Microsoft corporate network*

of the existing ones, and insert the delay samples into the delay distributions of the corresponding service pairs. Each delay distribution is maintained as a histogram with 300 bins and 10ms bin width. We implement this component using Perl with roughly 500 lines of code.

6.3 Dependency extractor

The centralized dependency extractor waits for dependency extraction requests for a particular service from administrators. When a request arrives, it will retrieve the delay distributions of relevant service pairs from the servers where the service is hosted and the clients. Depending on whether there are enough samples to reliably extract dependencies, the dependency extractor may perform client and service aggregation when clustering information is available. Aggregation is done by adding the bin-heights of the same bins in the delay distributions of the same service pair. After retrieving and possibly aggregating the delay distributions, we ignore those delay distributions with fewer samples than the number of bins. For each remaining delay distribution, we use Matlab to perform Fast Fourier Transform, filter noise with Kaiser window, and then detect typical spikes whose bin-heights exceed $(mean + k \times stdev)$. If any typical spike exists, we consider the corresponding service pair a dependency and output the list of all the dependencies in the end. We use a combination of Perl and Matlab codes for aggregation, noise filtering, spike detection, and report generation, with a total of 1K lines of code.

7 Experimental Results

We deployed Orion in a portion of Microsoft's corporate network illustrated in Figure 4. Because we cannot directly access the clients and the production servers, we choose an network-based deployment by connecting packet sniffers to span ports on routers. We monitored the traffic of 2,048 clients in 9 LANs at router R_0 and the traffic of 2 servers in the data center at routers R_1 and R_2. The client traffic must traverse R_0 to reach the data center, where most of the services are hosted. From the traffic at R_0, we extract the RR dependencies for five representative applications from the client's perspective. By examining the traffic at R_1 and R_2, we extract the LR dependencies for two of the five applications from the server's perspective. The results in this section were obtained during a two-week period in January 2008. We thoroughly evaluate Orion in its accuracy of dependency extraction, its convergence properties, and its scalability and performance.

Type	Sharepoint	DFS	OC	SD	Exchg
# of Instances	1693	1125	3	34	34
# of Clients	786	746	228	196	349

Table 1: Popularity of five enterprise applications

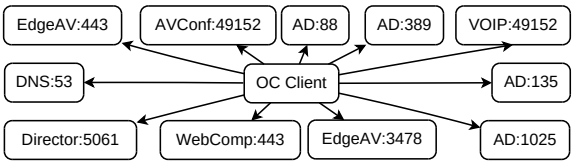

Figure 5: *OC client dependencies*

7.1 Dependencies of five applications

Microsoft's corporate network has thousands of applications. We select five distinct applications based on their popularity. These five applications include Office Communications (integrated instant messaging, VoIP, and audio and video conferencing), Exchange (email), Sharepoint (web), Distributed File System (file serving), and Source Depot (version control system).

For each application, Table 1 lists the number of clients and the number of application instances of the same type based on the traffic at R_0. Clearly, each application attracts a reasonably large fraction of the monitored clients. There are also many application instances of the same types in the network. Since the same type of applications have similar dependencies, our results may be easily extended to many other application instances. For each application, we obtain its true dependencies from the deployment documents written by the application owners. This is one key distinction from previous work which does not have access to such ground truths to perform comprehensive validations. Note that due to the large amount of time and effort involved, application owners can only create these documents for a small subset of important applications.

There are four infrastructural services that most applications depend on. Among them, active directory (AD) and proxy services are provided by load balancing clusters and DNS and WINS services are provided by failover clusters. We aggregate all the servers in the same cluster and represent each service as an $(ipCluster, port, proto)$. Since most services support both UDP and TCP, we omit the $proto$ field for simplicity in the remaining of this section. We next describe the service dependencies of the five applications studied based on the ground truths from deployment documents also confirmed by their application owners.

7.1.1 Office communications (OC)

Office Communications (OC) is an enterprise application that combines instant messaging, VoIP, and audio and video (AV) conferencing. It is one of the most popular applications and is being used by 50K+ users in Microsoft's corporate network. Figure 5 illustrates the dependencies of OC clients. They depend on eleven services to exploit the full functionality of OC: 1) DNS:53 for server name lookup during login; 2) Director:5061 for load-balancing login requests; 3) AD:88 for user authorization; 4) AD:389 for querying relevant domain objects; 5) AD:1025 (an RPC port) for looking up user profile; 6) AD:135 for learning the port number of the AD:1025 service; 7) EdgeAV:3478 for AV conferencing with external users via UDP; 8) EdgeAV:443 for AV conferencing with external users via TCP; 9) AVConf:49152 for AV conferencing with internal users; 10) VoIP:49152 for voice-over-IP; 11) WebComp:443 for retrieving web contents via HTTPS.

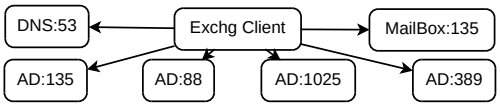

Figure 6: *Exchange client dependencies*

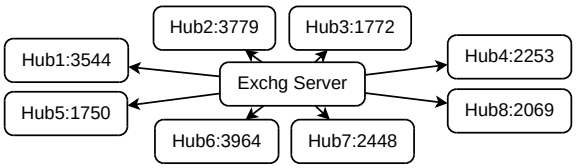

Figure 7: *Exchange server dependencies*

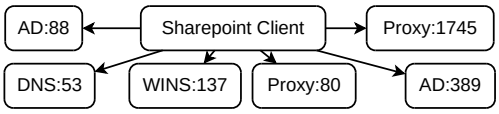

Figure 8: *Sharepoint client dependencies*

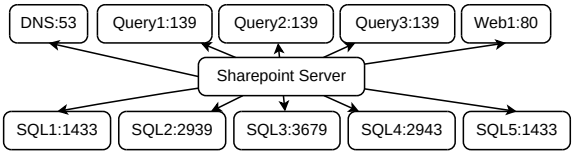

Figure 9: *Sharepoint server dependencies*

7.1.2 Exchange

Exchange is an enterprise email application. It is being used by all the users in Microsoft. Figure 6 and 7 illustrate its client and mailbox server dependencies. Exchange clients depend on six services to use the email service, five of which have been explained before. Because clients use RPC to communicate with the email service, it also depends on the endpoint mapper service on the mailbox server (Mailbox:135) to learn the RPC port of the email service. The email service on the mailbox server depends on eight services to answer the requests from Exchange clients, each of which is an email submission service running on a hub transport server. Note that we can obtain a three-tier dependency graph of Exchange by combining the client-side dependencies with the server-side dependencies.

7.1.3 Sharepoint

Sharepoint is a web-based enterprise collaboration application. We studied one of the most popular internal

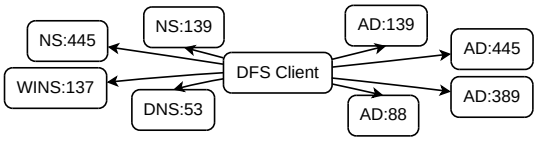

Figure 10: *DFS client dependencies*

Figure 11: *SD client dependencies*

Sharepoint websites. Figures 8 and 9 illustrate its client and front-end server dependencies. Sharepoint clients depend on six services to use the web service, three of which have been explained before. The remaining three services are: 1) WINS:137 is required because the web service uses a NetBios name which can only be looked up via WINS; 2) Proxy:80 is for notifying clients with proxy settings that Sharepoint is an internal website; 3) Proxy:1745 is for notifying clients without proxy settings that Sharepoint is an internal website. The web service on the front-end server depends on ten services to answer requests from clients: 1) five SQL services that store most of the web contents; 2) one web service that stores the remaining web contents; and 3) three query services that handle search requests. We can obtain a three-tier dependency graph of Sharepoint by combining the client-side dependencies with the server-side dependencies.

7.1.4 Distributed file system (DFS)

DFS is an enterprise service that can organize many SMB file servers into a single distributed file system. We study one of the DFS services where internal users can download most of the installation packages of Microsoft softwares. Figure 10 illustrates the dependencies of DFS clients. They depend on eight services to access the files in DFS, four of which are unique to DFS. AD:445 and AD:139 help clients find the DFS namespace servers (NS). NS:445 and NS:139 redirect clients to the appropriate file servers.

7.1.5 Source depot (SD)

Source depot (SD) is a CVS-like version control system. We study one of the SD services that is frequently used by our monitored clients. Figure 11 illustrates the service dependencies of SD clients. There are only four dependencies, all of which have been explained before.

7.2 Accuracy of dependency discovery

We first examine the accuracy of the dependencies discovered by Orion for each of the five applications depicted above. We then further remove false positives with additional testing, compare our results with prior work based on co-occurrence probability, and study the effects of noise filtering and flow generation.

For each application, we first create a *candidate set* of services that the application could possibly depend on if we do not make any inference. For a client-side or a server-side application, it is simply the full set of the remote services that the client or the server ever communicates with. We classify the services in the candidate set into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) by comparing the inferred dependencies with the true dependencies presented in the previous section.

The rows starting with Orion in Table 2 and 3 present the breakdown of the client-side (with aggregation) and server-side dependencies respectively. The number of services in the candidate sets varies from several hundreds to hundreds of thousands for different applications, reflecting the difficulty in extracting the dependencies manually. Orion can narrow down the set of potential dependencies ($TP + FP$) to fewer than a hundred, making it much easier to identify the true dependencies with some additional testing. This represents a factor of 50 to 44K reduction from the original candidate sets. Furthermore, we only miss two true dependencies on the server side (Table 3), one for each application. There is no false negative for any of the applications on the client side (Table 2).

For the server-side Sharepoint web service, we miss one dependency on a database service. Further investigation indicates that there is no typical spike in the delay distribution between the two services, likely due to the noise induced by the background indexing traffic between the two services which are unrelated to the web service. For the server-side Exchange email service, we miss the dependency on one of the eight email submission services. While visual inspection does reveal a typical spike in the corresponding delay distribution, it is not significant enough to be caught by our spike detection.

The number of FP's varies from 3 for the client-side Sharepoint service to 77 for the client-side OC service. They fall into two categories depending on whether they contain any significant, non-typical spikes in their delay distributions. The FP's in the first category are unlikely to be caused by random noise. Manual inspection indicates most of these non-typical spikes can be explained by the existence of certain *correlation* between the service pairs. As one example, the OC client has false dependency on the exchange email service, apparently due to many clients running both applications simultaneously. In another example, the Exchange client has false dependency on the proxy service. This can happen when Exchange clients open emails with embedded external web contents, causing these clients to access external websites via proxy. The FP's in the second category are apparently due to the limitation of our spike detection algorithm to fully distinguish noise from spikes.

	Exchange client				DFS client				Sharepoint client				OC client				SD client			
	tp	fp	fn	tn	tp	fp	fn	tn	tp	fp	fn	tn	tp	fp	fn	tn	tp	fp	fn	tn
Orion	6	26	0	14K	8	13	0	1497	6	3	0	703	11	77	0	25K	4	4	0	369
Sher_{10}	6	178	0	14K	8	102	0	1408	6	65	0	641	9	125	2	25K	4	52	0	321
Sher_{100}	6	57	0	14K	8	93	0	1417	6	168	0	538	10	85	1	25K	4	29	0	344
eXpose	5	443	1	14K	8	570	0	940	6	565	0	141	10	1416	1	24K	4	323	0	50
noFilter	6	49	0	14K	8	25	0	1485	6	6	0	700	11	159	0	25K	3	19	1	354
noFlow	6	2488	0	12K	8	988	0	522	6	534	0	172	11	3594	0	21K	4	198	0	175

Table 2: Client side dependencies after aggregation

	Exchange server				Sharepoint server			
	tp	fp	fn	tn	tp	fp	fn	tn
Orion	7	34	1	230K	9	6	1	660K
Sher_{10}	8	68	0	230K	8	7	2	660K
Sher_{100}	7	61	1	230K	9	19	1	660K
eXpose	7	557	1	230K	7	396	3	660K
noFilter	4	44	4	230K	6	3	4	660K

Table 3: Server side dependencies

7.2.1 Removing false positives

In the process of extracting dependencies from the candidate set, we have to trade off between FP's and FN's. Our primary goal is to avoid FN's even at the expense of increasing FP's. This is because we have almost no way to recover a true dependency once it is removed from the candidate set. In cases where dependencies are used for fault localization or reconfiguration planning, missing dependencies may lead to unanticipated consequences that are expensive to diagnose and repair. (see Section 8 for details).

To further reduce the FP's in Table 2, we perform controlled experiments on the client side. For each of the five applications, we use a firewall to block the services in the FP and TP sets one-by-one. Blocking the services in the FP set will not have any impact on the application while blocking the services in the TP set will disrupt its service function. To eliminate caching effect, we must start with a clean state for each test. Because this is a manual process, it took us roughly one working day to successfully identify all the 35 true dependencies from the 158 potential ones. We did not conduct such experiments on the server side because we have no control over the servers. Administrators can do such testing during maintenance hours to minimize the disruption to users. Note that developing test cases requires human knowledge of only how to drive applications but not of application internals. The former is relatively widely available while the latter is usually arduous to extract.

7.2.2 Comparison with Sherlock & eXpose

Sherlock [8] and eXpose[18] attempt to extract dependencies from network traffic. They are both based on the idea that the traffic of dependent services are likely to co-occur in time. They use a fixed time window W to compute co-occurrences and then use a threshold T either on the conditional probability (in Sherlock) or on the JMeasure (in eXpose) to identify dependencies. While they both seem to extract certain meaningful dependencies, neither of them quantified the accuracy of their results in terms of false negatives or false positives.

While conceptually simple, a key problem with both approaches is the choice of W. As we explained earlier in Section 7.2.3, the delay between two dependent services reflects the underlying processing and network delay. This delay could vary from a few milliseconds to hundreds of milliseconds. If W is small (as in Sherlock), we may miss the dependencies between the service pairs whose typical delays exceed W. If W is large (as in eXpose), we are likely to capture many co-occurrences of non-dependent service pairs. In contrast, Orion identifies dependencies by looking for typical spikes in the delay distributions. It does not make any assumption about the location of the typical spikes in the distribution.

We implemented both Sherlock and eXpose (without cluster pruning) for comparison. We use $W = 10ms$ and $100ms$ for Sherlock and the $W = 1s$ for eXpose. (In their papers, Sherlock uses $W = 10ms$ and eXpose uses $W = 1s$.) We tune their threshold T so that their FN's roughly match ours, and then compare the FP's. The results are in the rows starting with Sher_{10}, Sher_{100}, and eXpose in Tables 2 and 3. Clearly, Orion has far fewer FP's than Sherlock or eXpose in all the cases. For the client side of Exchange, DFS, Sharepoint, and SD, the FP's inferred by Orion are only 5% - 50% of those inferred by Sherlock or eXpose. Assuming that the testing time to remove FP's grows linearly with the number of potential dependencies, Orion will save approximately two to twenty days of human testing time compared with Sherlock and eXpose.

7.2.3 Effect of noise filtering & flow generation

Orion relies on noise filtering to mitigate the impact of random noise on dependency discovery. It is important to understand to what extent noise filtering helps to reduce FP's and FN's. In Table 2 and 3, the results in the rows starting with "noFilter" are computed by applying spike detection directly to the delay distributions without filtering noise. Judging from the Orion results, noise filtering is effective in reducing FP's and/or FN's in all but one case. Even in the Sharepoint server case where Orion has 3 more FP's, we consider it worthwhile given the decrease of 3 FN's.

Orion aggregates packets into flows to reduce redun-

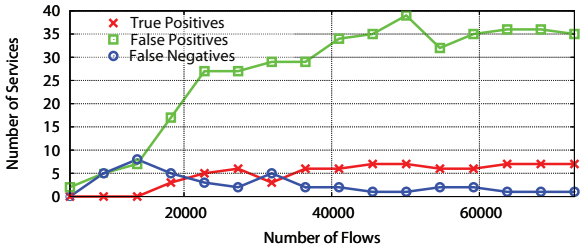

Figure 12: *Impact of flows on Exchange server dependencies*

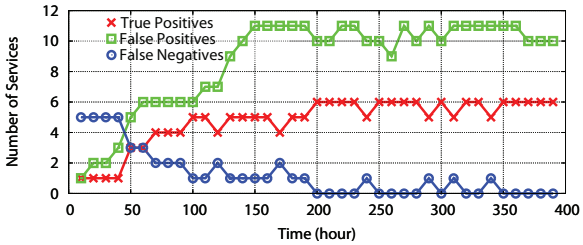

Figure 13: *Impact of time on Exchange client dependencies*

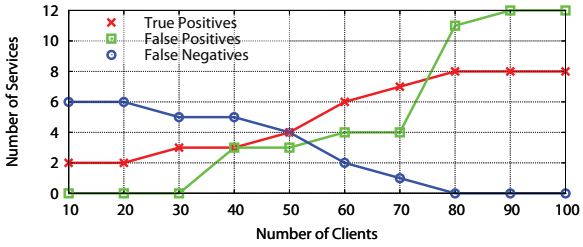

Figure 14: *Impact of aggregation on DFS client dependencies*

dant correlation and computation overhead. Without flow generation, big flows are treated more favorably than small flows since they will contribute more samples in the delay distributions. Such systematic bias may lead to undesirable spikes in the delay distribution. In Table 2, the row starting with "noFlow" contains the results without flow generation. Compared with the Orion results, FN's stay the same but FP's are much larger, most likely due to the over-counting of delay samples related to big flows. In terms of performance, calculating delay distribution directly from packets is roughly ten times slower than from flows. This is because there are significantly more packet pairs than flow pairs.

7.3 Convergence of dependency discovery

We now study the convergence properties of Orion along three dimensions: time, flows, and clients. This is important because the dependencies of an application may change from time to time due to reconfiguration. We evaluate whether Orion can produce stable results before the next change happens. Furthermore, Orion discovers dependencies based on the delay samples computed from flows. Understanding its requirement on the number of flows is essential for us to judge the reliability of the results. Finally, Orion sometimes needs client aggregation to overcome the problem of a lack of sufficient samples. Measuring the impact of clients helps to avoid unnecessary overhead due to excessive client aggregation.

Figure 12 illustrates how the inferred dependencies of Exchange server change as more flows to the Exchange server are used for dependency discovery. The X-axis is the number of flows. The number of samples in all the delay distributions related to the Exchange service grows with the number of flows. Clearly, Orion can discover more TP's when more flows are used. When the number of flows reaches 65K, Orion discovers all the TP's and the number of FP's also stabilizes. This suggests that the variation of inferred dependencies ($TP + FP$) is a good indication of whether Orion needs more flows. The convergence behavior of other applications exhibits similar trend. Depending on the application, Orion needs 10K to 300K flows to obtain stable results.

Figure 13 shows how the inferred dependencies of Exchange client evolve over time. Not surprisingly, the accuracy of the inferred dependencies gradually improves as Orion uses longer duration of traces. After 200 hours, it has discovered all the TP's and the inferred dependencies fluctuate only slightly thereafter. This confirms that we can stop the inference when the inferred dependencies converge. For all the applications, the convergence time varies from two to nine days. We consider this acceptable since the dependencies of production client-server applications are usually stable for at least several weeks to several months in enterprise networks. Nonetheless, this convergence time could be a bit long for newly deployed applications. We may expedite the discovery process by applying dependency templates derived from other networks or provided by application designers to pre-filter the set of possible dependencies.

Figure 14 illustrates how the inferred dependencies of DFS client vary as we aggregate more clients. It is evident that client aggregation is important for improving the inference accuracy, especially when no individual clients have a sufficient number of delay samples. The FN's drop from 5 to 0 as the aggregated clients increase from 10 to 90. After that, the inferred dependencies become stable even when more clients are aggregated. This suggests excessive client aggregation will only lead to more overhead instead of benefit. For the remaining applications, we need to aggregate 7 (SD) to 230 (Sharepoint) clients to identify all the TP's.

7.4 Performance & Scalability

In this section, we focus on the performance and scalability of Orion. We are interested in answering the following questions: i) does it scale with the number of services in the network? ii) what is the CPU and memory usage in a network-based or a host-based deployment? iii) how quickly can dependencies be extracted when administrators need such information?

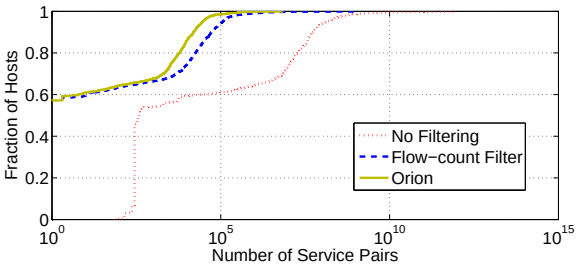

Figure 15: *Flow-count and time-window filters improve scalability*

7.4.1 Scalability of delay distribution calculator

As described in Section 5.2, Orion uses a time window of three seconds and a flow count threshold of 50 to filter unnecessary service pairs. To illustrate the effectiveness of these two heuristics, Figure 15 plots the CDF of the number of service pairs without any filter, only with the flow count filter, and with both filters. The X-axis is the number of service pairs and the Y-axis is the cumulative fraction of hosts. The flow count filter reduces the service pairs by almost three orders of magnitude. After applying the time-window filter, 99% of the hosts have fewer than 10^5 service pairs. As we show next, the actual memory usage is reasonably small for both the network-based and the host-based deployment.

7.4.2 Performance of flow generator & delay distribution calculator

As shown in Figure 4, we currently use the network-based deployment by running Orion on dedicated sniffing boxes attached to three routers in the network. In this deployment, each sniffing box may capture large volumes of traffic from multiple hosts in the same subnet. We want to understand whether the flow generator and delay distribution calculator can keep up with such high traffic rate. We use the traffic at R_0 for our evaluation because it contains the aggregate traffic of all the clients and is bigger than the traffic at the other two routers. We run the flow generator and delay distribution calculator on a Windows Server 2003 machine with 2.4G four-core Xeon processor and 3GB memory. We measured their performance during the peak hour (2 - 3 PM local time) on a Thursday. The aggregate traffic rate is 202 Mbps during that period. The flow generator processed one hour of traffic in only 5 minutes with an 8MB memory footprint. The delay distribution calculator finished in 83 seconds and used 6.7MB memory.

We also measure the performance of the host-based deployment. Given that Orion has to share resources with other services on the same host, we focus on its CPU and memory usage. We perform the evaluation on a regular client machine with 3GHz Pentium4 processor and 1GB memory. The packet sniffer, flow generator, and delay distribution calculator normally use 1% of CPU and 11MB memory in total. Such overhead is reasonably small for long-term monitoring on individual hosts.

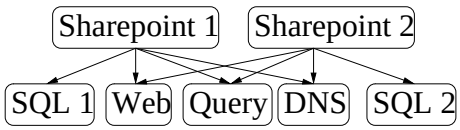

Figure 16: *Using dependency graph for fault localization*

7.4.3 Performance of dependency extractor

We now evaluate the execution time for extracting dependencies from the delay distributions. The results are measured on a Windows Server 2003 machine with 2G dual-core Opteron processor and 4GB memory. For all the applications, the dependency extraction was finished within two minutes. This is short enough for administrators to run on-demand. We also measured the network usage of client aggregation. Aggregation is required for merging the delay distributions from multiple clients when there are insufficient delay samples. During the two-week evaluation period, the total size of the delay distributions from all the 2,048 clients is under 1MB after compression. This suggests it is feasible to use a centralized dependency extractor since it is unlikely to become the bottleneck.

8 Operational Use of Dependencies

We now provide examples of how dependencies can facilitate fault localization and reconfiguration planning. These examples are by no means exhaustive. Administrators may find dependencies useful for other network management tasks, *e.g.*, impact analysis and anomaly detection.

The existence of complex dependencies between different services makes it extremely challenging to localize sources of performance faults in large enterprise networks. For instance, when a Sharepoint service fails, it could be caused by problems at the DNS servers, SQL servers, web servers, or query servers. Manually investigating all these relevant servers for each performance fault is time-consuming and often infeasible.

A dependency graph summarizes all the components that are involved in particular services. Combined with observations from multiple services, it enables fast and accurate fault localization. Figure 16 illustrates an example of a dependency graph with two Sharepoint services. For simplicity, we ignore problems in the network and the port numbers in the graph. $Sharepoint_1$ and $Sharepoint_2$ use the same DNS, query, and web servers. However, they use different SQL servers for storing contents. Suppose $Sharepoint_1$ is experiencing problems while $Sharepoint_2$ is not. From the dependency graph, we deduce that the source of the problem is unlikely at

the DNS, query, or web servers since $Sharepoint_2$ has no problems using them. This leaves SQL_1 as the most plausible candidate for the source of the problem.

While the above example is fairly simple, a dependency graph of a large enterprise network will be substantially more complex. In fact, it is almost impossible to inspect manually. Fortunately, there have been known techniques that automate the fault localization process by applying Bayesian inference algorithms to dependency graphs [8, 20, 19]. We omit the details here since they are not the focus of this paper.

Another use of dependencies is in reconfiguration planning. Large enterprises have many services and applications, which are continually being reorganized, consolidated, and upgraded. Such reconfigurations may lead to unanticipated consequences which are difficult to diagnose and repair. A classic example involves a machine configured as a backup database. Since there is no explicit documentation about this dependency, the machine is recycled by administrators. Later on, when the primary database fails, applications that depend on the database becomes completely unavailable.

To avoid such unanticipated consequences, administrators must identify the services and applications that depend on a particular service before any changes can be made to that service. This often is a slow and expensive process. Given the dependencies extracted from all the service pairs, we can easily search for all the services that directly or indirectly depend on a particular service. This will significantly save the time administrators spend in assessing and planning for the changes.

Besides our own research prototype, a production implementation of Orion based on the TDI driver is currently being deployed in the Microsoft IT department (MSIT). The administrators will initially use the dependencies extracted by Orion for reconfiguration planning. Orion has been set up on two web services and one Windows Messenger service. Preliminary results indicate Orion has successfully discovered the set of expected dependencies on database servers, AD servers, and presence servers. The plan is to roll out Orion to over 1,000 services managed by MSIT in the next six months.

9 Conclusion

In this paper, we present a comprehensive study of the performance and limitations of dependency discovery techniques based on traffic patterns. We introduce the Orion system that discovers dependencies for enterprise applications by using packet headers and timing information. Our key observation is the delay distribution between dependent services often exhibits "typical" spikes that reflect the underlying delay for using or providing such services. By deploying Orion in Microsoft's corporate network that covers over 2,000 hosts, we extract the dependencies for five dominant applications. Our results from extensive experiments show Orion improves the state of the art significantly. Orion provides a solid foundation for combining automated discovery with simple testing to obtain accurate dependencies.

References

[1] EMC SMARTS. `http://www.emc.com/products/family/smarts-family.htm`.

[2] HP OpenView. `http://www.openview.hp.com`.

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

[4] Mercury MAM. `http://www.mercury.com/us/products/business-availability-center/application-mapping`.

[5] Microsoft MOM. `http://technet.microsoft.com/en-us/opsmgr/bb498230.aspx`.

[6] Taming Technology Sprawl. `http://online.wsj.com/article/SB120156419453723637.html.html?mod=technology_main_promo_left`.

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

[8] 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 *Proc. ACM SIGCOMM*, 2007.

[9] P. V. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering Dependencies for Network Management. In *HotNets*, 2006.

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

[11] G. Bartlett, J. Heidemann, and C. Papadopoulos. Understanding passive and active service discovery. In *IMC*, 2007.

[12] R. Black, A. Donnelly, and C. Fournet. Ethernet Topology Discovery without Network Assistance. In *ICNP*, 2004.

[13] O. E. Brigham. The fast fourier transform and its application. In *Prentice-Hall*, 1988.

[14] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Intergrated Network Management*, 2001.

[15] M. Chen, A. Accardi, E. Kcman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *NSDI*, 2004.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenkar, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[17] J. F. Kaiser and R. W. Schafer. On the Use of the Io-Sinh Window for Spectrum Analysis. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, 1980.

[18] S. Kandula, R. Chandra, and D. Katabi. What's Going On? Extracting Communication Rules In Edge Networks. In *Proc. ACM SIGCOMM*, 2008.

[19] S. Kandula, D. Katabi, and J. P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. In *MineNet*, 2005.

[20] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. In *NSDI*, 2005.

[21] B. Lowekamp, D. R. O'Hallaron, and T. R. Gross. Topology Discovery for Large Enternet Networks. In *SIGCOMM*, 2001.

[22] 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.