Dependency analysis via network measurements

Philip Lorenz
Betreuer: Dipl.-Inform. Lothar Braun
Hauptseminar: Innovative Internet-Technologien und Mobilkommunikation WS2010/2011
Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitektur
Fakultät für Informatik, Technische Universität München
Email: lorenzph@in.tum.de

ABSTRACT

Large scale computer networks consist of a vast number of interoperating services. Often, the interchange between those services is not documented leading to a variety of issues. Network dependency analysis aims to automate service dependency discovery. In this work several different approaches to network dependency analysis, ranging from active to passive approaches, will be introduced and evaluated.

Keywords

Network Dependency Analysis, Sherlock, Orion, Active Dependency Discovery, Traffic Dispersion Graphs

1. INTRODUCTION

Modern enterprise IT infrastructures consist of thousands of participants using a large amount of different applications. A survey performed by the Wall Street Journal in 2008 ([11]) reports that in large companies such as HP more than 6000 different applications are in use. A lot of the applications require access to one or more network services making availability of these services crucial. Often, these services are even dependent between themselves further increasing complexity. For example, a seemingly simple task such as opening a web site has at least two dependencies - the DNS server for name resolution and the web server itself which returns the page. In some networks a proxy server may also be required introducing an additional dependency.

Dependency documentation of network services is often not readily available - e.g. if the product was developed within the company without documentation guidelines. Even if the documentation is available, extracting dependency information for the multitude of services can be very time consuming and error prone. Especially, as the network evolves over time, documentation may become outdated.

Historically systems ([3, 9]) which automatically detect network topology and services were developed. However, these systems did not extract relations between the different components of a network. They rather relied on expert and business knowledge to formulate application dependencies. Other approaches rely on instrumenting the software stack in order to extract dependencies. Pinpoint ([5]) integrates into the J2EE stack, a platform for developing Java-based enterprise server applications, enabling tracing of individual requests. X-Trace ([7]) is a tracing framework supporting a number of different OSI layers (typically the network, transport and application layer). Both approaches are of limited scope as they require detailed implementation knowledge of the software stack and therefore may be troublesome to deploy.

Bahl et. al ([1]) identify several areas which would benefit from the availability of network dependency information:

Fault localisation: Consider a service that is not functioning properly. Dependency information can be used to determine the root cause of the problem as all components which may be responsible for the failure are known. Applied to the web browsing example introduced above, the proxy server may be load balanced - e.g. several physical servers are responsible for fetching the website. If one of those servers fails dependency information may be used to track down the actual machine.

Reconfiguration planning: Companies usually run a lot of servers which sometimes have been in use over the course of several years. Sometimes the tasks of a specific server are not known by the administrators. Imagine that a server which use was not documented, running a backup database, is removed during IT reorganisation. In the best case backups are available but those may be several hours old. And even if those are not too old there is a downtime which might restrict employees from performing their work. In this case dependency information aids system planners in making choices when reorganising the IT infrastructure.

Help desk optimisation: In case a component fails many different applications may be affected. For example, the failure of an Active Directory server may affect a vast amount of the users in an organisation. As dependency graphs allow the extraction of all affected components help desk employees can troubleshoot problems more quickly and efficiently as a specific problem description can be mapped to the actual root cause. This not only avoids unnecessary problem mitigation strategies (e.g. please reboot your computer) but also allows ticket prioritisation if a single failure created a lot of support requests.

Anomaly detection: Dependency graphs show the relations between network components at a given point of time. A rapid change of dependencies may be a sign of an anomaly in the system. If such a change is detected human supervisors may be alerted to further inspect the found issue.

This paper presents various systems which are used to extract dependency information from a network. In section 2 terminology used throughout the paper will be introduced and explained. Section 3 presents Active Dependency Discovery, an approach which actively influences the network in order to derive dependencies. In section 4 several non-invasive systems are introduced.

2. BACKGROUND

Network dependency analysis attempts to recognise dependencies between members of a network. For example, a system administrator might be interested whether Host A depends on Host B or vice versa. This is a high level viewpoint as the interchange between services is not of interest. In this work a host based dependency between Host B on Host A will be expressed as $(A) \rightarrow (B)$. Note that this relation is not symmetric $((A) \rightarrow (B) \nRightarrow (B) \rightarrow (A))$ - e.g. if a call to host A depends on B it does not necessarily follow that a call to host B depends on A.

In other cases one may be interested in the actual dependencies between different services. A service can be described by its IP address and the port it provides its services on. Formally, this can be expressed as the three-tuple $(IPaddress, port, protocol)$. For example, the web server at www.in.tum.de can be expressed as $(131.159.0.35, 80, TCP)$. A dependency between two services can then be described using a similar notation as above, by replacing the host with the service part. It is important to realise that a dependency between service A and B does not necessarily mean that every access to service A also triggers an invocation of service B.

Dependencies can be split into two groups - remote-remote (RR) dependencies and local-remote (LR) dependencies. A remote-remote dependency describes that in order to access service B service A has to be invoked first. A typical example for a RR dependency is browsing the web. Before the web browser contacts the web server, the domain name of the website has to be resolved . In order to do so, the DNS service is queried and as soon as the name has been resolved the web server can be contacted on its IP address. This example also illustrates that a dependency is not always visible. Due to caching at the operating system level DNS lookups do not happen every time the web server is contacted.
On the other hand, LR dependencies are triggered by an incoming service call resulting in an outgoing service call. A web server accessing a database to provide the information for a web page is an example for this type of dependency.

Another important issue, when dealing with services, is the distinction between persistent and dynamic services. A persistent service is a long-lived service often using a well-defined port for its purpose. Examples include web or mail servers. In contrast dynamic services are usually short-lived and not meant to serve more than a couple of other clients. Peer to peer applications such as Skype typically fall into this category.

The results of a network analysis can be evaluated using three metrics:

True positives The dependencies found which are ground-truth dependencies - e.g. dependencies which have been verified to be real dependencies.

False positives Classified dependencies of the system which are not actual dependencies of the network. For example, regular background traffic may be misclassified by the system as a dependency.

False negatives Actual dependencies which were not detected by the network analysis system. There are several reasons which lead to false negatives such as a sampling rate which is too high or a lack of traffic which triggers the dependency.

In the case of a perfect analysis engine the true positives will exactly match the actual dependencies of the host or service. However, it is unlikely for a system to correctly detect all and only the correct dependencies, hence the other two metrics play an important role as well. A large number of false positives may be cumbersome if manual inspection of the results has to be performed. In contrast, even a small number of false negatives may have an impact on the operation of the system if the missing dependencies are not found by manual inspection.

3. ACTIVE SYSTEMS

Dependency analysis systems can be categorised into two groups: active and passive systems. Active systems attempt to determine dependencies by modifying the observed system. Modification includes changing parameters of components as well as injecting network traffic generated by the dependency analysis into the system. This means that the analysis process may influence the system behaviour or, in the worst case, disturb the operation of the system.

3.1 Active Dependency Discovery

Brown et. al ([4]) introduce a system called Active Dependency Discovery (ADD) which determines dependencies by actively perturbing services in the network. This approach focuses on fine grained dependency analysis hence some information about the observed system should already be available. The dependency analysis process is split into four major steps:

1. **Node/Component identification:** In this step hosts or components that are relevant to the analysed system are identified. The list of potential components may come from various data sources such as inventory management software or from coarser grained dependency models.
2. **System instrumentation:** Probes and other components are installed to measure the effect of the perturbation within the network. Potential metrics include availability or performance data (e.g. response time).
3. **System perturbation:** In order to measure the effects of the perturbation, a specific workload should

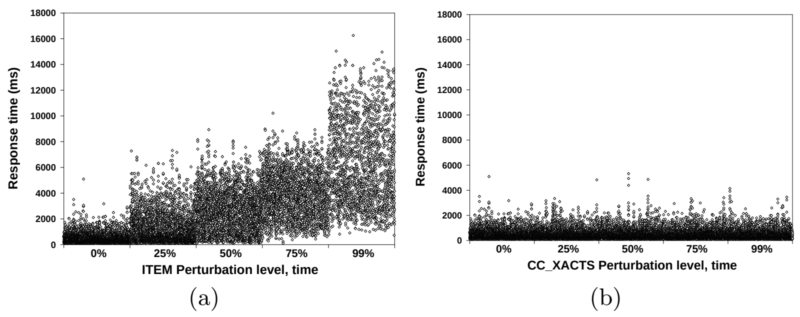

Figure 1: Perturbation intensity vs response time of an intranet portal ([4])

be chosen and then be continuously applied to the system. In case of an Internet portal, one possible workload could be a list of frequently visited subpages. As soon as the workload is applied, components should be perturbed at varying intensity. For example, one could simulate network loss ranging from 0%, meaning no packet loss, to 100%, resulting in complete loss of connectivity. During this step the instrumentation systems set up in step 2 are used to log the system response to the perturbation. It is also possible to perturb multiple components at the same time which enables the discovery of complex dependencies such as load balancers or replicated components.

In their work, Brown et. al use a e-commerce system to perform fine-grained dependency analysis. This system had a database backend which the authors perturbed by locking individual tables making data retrieval impossible until the lock expired. The authors then recorded the effect on the response time while requesting different parts of the site (e.g. viewing a list of products). Figure 1 shows the effect on the website response time after applying the perturbation at varying intensities to two tables of this e-commerce system (ITEM containing the items available for sale and CC_XACTS which contains credit card transactions).

4. **Dependency extraction:** After the perturbation step is completed, models of the logged data can be created. In this step, the various metrics recorded through instrumentation are related to the perturbation settings at the given point in time. The goal is to identify dependencies by determining the statistical significant correlations. This does not only allow the extraction of dependencies but also the strength of the dependency for the given workload. When looking at the example in Figure 1, it can be clearly seen that a longer locking time on table ITEM leads to a higher response time of the web server. This indicates that a dependency between the sample workload and the given database table exists. On the other hand, no unusual increase in response times for the CC_XACTS table can be seen. This suggests that the workload is independent of the given table. While in their work, perturbation is not performed on the network layer, other perturbation methods such as simulating packet loss, can be used to find the dependency on the database server. In order to lower the cost of dependency extraction, the raw results can be aggregated.

Active approaches such as ADD have several disadvantages when applying them to real world networks. Due to their invasive nature the performance of the network may be negatively affected, simply due to adding additional load to the network. In the case of ADD, perturbation is bound to negatively affect the network services if it is applied to the production environment. Hence, ADD is best used in development environments which simulate the actual network. However, this may cause additional problems if the development environment does not exactly behave like the production environment. Another problem of ADD is its dependency on domain knowledge. In the best case only the workload has to be created but generating an exhaustive workload may prove to be difficult, potentially leading to false positives. Additionally, ADD requires the installation of probes throughout the network hence a priori knowledge about the network topology is required. The acquisition of network topology is not within the scope of ADD but will likely require manual intervention which filters candidate instrumentation targets.

4. PASSIVE SYSTEMS

In contrast to active systems, passive dependency analysis does not interfere with normal system behaviour. In order to derive dependencies only information produced by the network itself is used - e.g. no traffic is generated in order to determine the dependencies within a network.

4.1 Sherlock

Bahl et. al ([2]) introduce a system which aims to aid IT administrators in troubleshooting problems. In order to achieve this goal, a dependency analysis component was developed which passively monitors the network and attempts to automatically create a graph describing the network components and the services provided within the network.

4.1.1 Architecture

Sherlock consists of a centralised *Inference engine* and several distributed *Sherlock agents*. The agents sniff network packet data and compute the dependencies for their attached network segments and the corresponding response time distributions. This data is then relayed to the inference engine which uses the information to perform fault localisation. An agent may be installed as a system service on single hosts

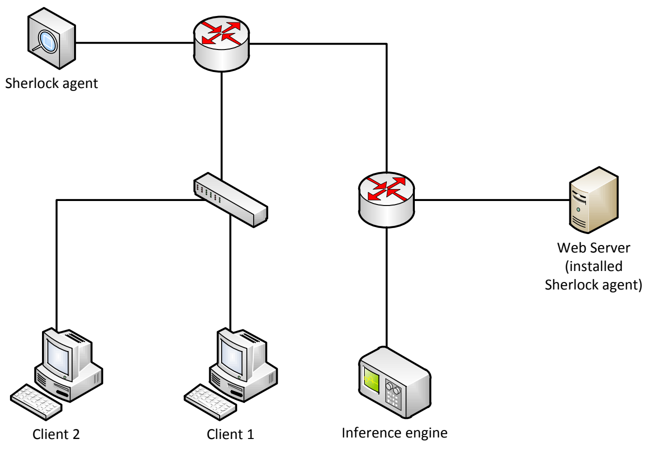

Figure 2: Sample of a Sherlock deployment

but can also be used to process data received from a monitoring port at a router or other network hardware. Figure 2 illustrates a sample deployment of Sherlock within a network. It includes an agent connected to a router and one agent directly installed on a web server. Data is collected at an independent host.

4.1.2 Dependency extraction

Sherlock analyses the packets captured, trying to find correlations between single packets directed towards a service. Rather than interrelating all packages, which would result in a severe performance loss, a time-window based approach is used: Let t_0 be the time at which an outgoing service request to service B is observed. Sherlock will choose all other outgoing service requests within the time window Δt before t_0 as dependency candidates. The remote-remote dependency probability that a host accessing service B is dependent on service A can then be expressed as the conditional dependency $Pr[A|B]$ - e.g. the number of times within the trace that A was accessed within the time window before seeing an invocation of B divided by the total number of invocations of B. Figure 3 illustrates such a packet time line. In this case Output 1 is the packet which is analysed. Let the time window Δt be set to 5 seconds. Both, Output 2 and Output 3, have sent packets indicating potential dependencies.

In order to deal with chance co-occurrence, which may be falsely assumed if another service is called often during the trace, Sherlock applies a simple heuristic to filter the results. Let I be the average invocation time interval of the noisy service. Only if the conditional probability is a lot larger than $\frac{\Delta t}{I}$, the dependency is assumed to be valid. Applying this technique to the example introduced above will exclude Output 3 as a dependency (the average interval I for this output is 2) as the resulting chance co-occurrence factor is larger than 1.

The dependency extraction process can solely be controlled by the selection of the time window length Δt. Choosing this value too high may introduce false positives as services called with a relatively high frequency will be falsely classified as dependencies. On the other hand, picking a value which is too low may result in false negatives. According to the authors, a time window of 10ms has proven to be a good choice detecting the majority of the service dependencies.

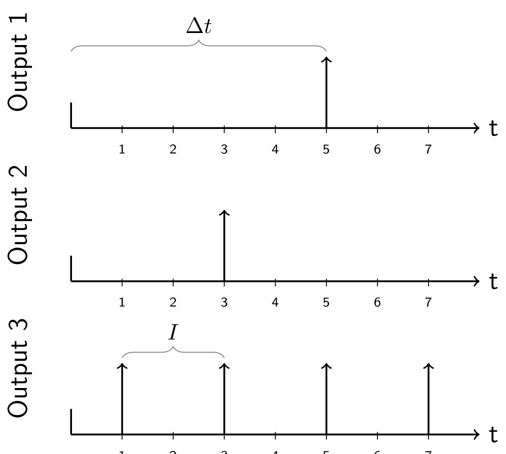

Figure 3: Exemplary packet flow

Data generated by the agents is then transmitted to the central inference engine which further aggregates the data eliminating potential false positives. For example, a client which always relies on a proxy server to perform its networking tasks may introduce false positives. Additionally, this aggregation enables the discovery of seldom accessed service dependencies as the combination of multiple data sources may provide enough data points to mark the dependency as statistically significant.

4.2 Orion

Orion by Chen et. al ([6]) is a dependency analysis engine sharing many basic concepts with the Sherlock system. However, several changes were made to improve the quality of the dependency detection.

To identify a single service invocation, Sherlock groups all contiguous packets with the same source and destination address and port without considering other transport layer attributes. In contrast, Orion aggregates individual packets depending on the protocol headers into flows. In the case of UDP, a stateless protocol, a timeout mechanism is used to determine the flow boundaries. For TCP packets, header flags are used. Example flags include the SYN, FIN, RST but in case of long-living connections the KEEPALIVE messages can be used as well. The reason for including KEEPALIVE messages is simple: If they were not used, the length of flows may include too many packets negatively influencing the dependency extraction performance. The utilisation of flows offers several advantages over a raw packet based approach: (i) the computational overhead is kept low as the number of samples decreases (consider 1 flow vs at least 3 packets for a TCP handshake) (ii) avoid redundancy and therefore skewed results which may occur if multiple packets are transmitted for a single service invocation.

Orion supports both remote-remote and local-remote dependency detection. For each potential dependency, a delay distribution is built. Hence, a system offering n local services and accessing m remote services will have $n \times m$ LR delay distributions and $m \times m$ RR delay distributions. Similar to Sherlock, Orion uses a time window in order to further reduce processing overhead. However, in the case of Orion the time window is significantly larger (3 seconds) but flows are grouped into smaller intervals, named bins.

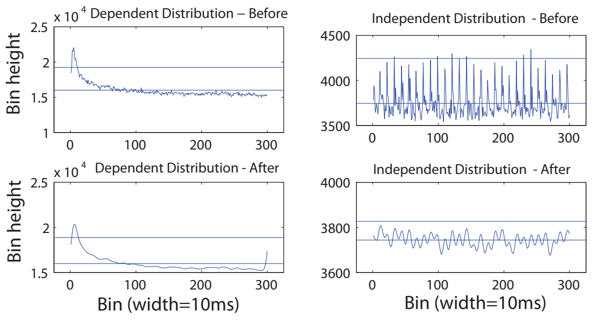

Figure 4: Delay histograms before and after the application of a low pass [6]

Similarly to Sherlock, Orion has to deal with chance co-occurrence which may introduce false positives in the dependency extraction results. As these independent packets do not follow a specific pattern they introduce random spikes in the delay distribution histogram (Figure 4). Orion treats the delay histogram as a signal and uses a common signal processing technique to eliminate the random noise. First the signal is transformed into the frequency domain (e.g. by using the Fast Fourier Transform). Afterwards, a low pass filter is applied, removing the high frequencies from the signal. This results in a smoothened signal as shown in the graphs at the bottom of Figure 4. Orion decides the validity of a dependency based on a specific bin-height threshold (indicated as a horizontal line in the graphs) - e.g. if there is at least one bin with a height above that threshold the dependency is regarded as valid. The impact of filtering can be seen on the right hand side graph where it is applied to a true negative. While without filtering, several peaks were above the threshold, these are eliminated after the application of the low pass, preventing false positives.

Similarly to Sherlock, Orion performs aggregation of the client data sets. However, not only service invocations are aggregated but also services themselves. In corporate networks, frequently used services such as DNS servers and proxy servers are load balanced in order to improve performance. Orion allows those clusters to be represented as a single server through manual input. Aggregation of these clusters may be semi-automated if a logical pattern is available to group the hosts providing these services. An example of such a pattern are reverse DNS names such as $dns-x.network.com$, where x is a number for a specific host part of the cluster.

Due to their operating system independent design Orion and Sherlock offer great flexibility. There are several deployment possibilities which ease the integration within the network. Additionally, this approach enables the detection of exotic dependencies as the amount of logging data generally exceeds those of other solutions. However, this comes at the cost of accuracy. The time-window based approach leads to a trade-off between false and true positives. In addition, as any statistical approach, these systems are highly dependent on the amount of sampling data. This means that the more samples are available the better the detection will become. Another limitation of the approach stems from the layer 4 and below restriction. Both systems do not attempt to parse application payload and will therefore always be restricted in the dependencies they can find.

4.3 Macroscope

Popa et. al introduce Macroscope ([10]) which levitates some of the problems solely packet based dependency analysis systems have due to statistical uncertainties. Macroscope follows a similar architecture as Sherlock and Orion. Network traces and application data is collected at multiple *tracers* deployed on end-systems. The *tracers* relay the data to a central *collector* which aggregates and preprocesses the data and passes it on to the *analyzer* for dependency extractions.

Macroscope uses operating system knowledge about active connections in order to identify RR dependencies of single applications. Most operating systems allow querying active connections using either system calls (e.g. on Windows `GetExtendedTCPTable` or `GetExtendedUDPTable`) or through the filesystem (e.g. on Linux in `/proc/net`). These lists contain the source IP and port, as well as the target IP and port, and the unique process identifier of the process owning the connection. Rather than constantly polling for connection information, Macroscope samples this data periodically in order to minimise resource usage. However, choosing a sampling interval which is too large may result in missed dependencies, especially if the connection duration is always lower than the interval.

Macroscope distinguishes between transient relations and static dependencies. These dependencies are similar to the concept of persistent and dynamic services introduced in section 2 - e.g. a transient relation corresponds to a dynamic service call while a static dependency involves a persistent service call. In order to generate the dependency output the system first classifies all applications into two groups (i) applications with only static dependencies (ii) applications with static dependencies as well as transient connections. Mathematically, the classification into the two groups can be expressed as follows: Let N^a be the number of application instances a within the trace, N_s^a the number of instances of application a using service s, $V_s^a = N^a - N_s^a$ (i.e. the number of application instances which did not use service s) and S^a the number of services contacted by all application instances of type a. The transient dependency metric is then calculated as follows:

$$M^a = \sqrt{\sum_s \frac{{V_s^a}^2}{S^a}} \qquad (1)$$

When M^a is 0 all applications instances use all services (as V_s^a is 0). However, if an application only uses transient connections the value of M^a will be maximal at $N^a - 1$ as for each service V_s^a will be $N_a - 1$ (i.e. each instance is the only one using service s). This means that the closer the value of M^a is to 0, the more likely it is that the application only has static dependencies. Using this metric, the authors classify all applications which have a value less than a certain percentage of the maximum value (e.g. $M^a \leq T \times (N^a - 1)$) into group (i). If an application belongs to this group all of the services it invokes are regarded as dependencies. If the metric is above the threshold an application falls into group (ii) and requires further processing before the static dependencies can be extracted: First of all, all invocations targeting a port below 1024 are considered to be static dependencies[1]. For all other static dependencies the following two conditions must hold:

$$\frac{U_s^a}{U^a} \geq U \wedge \frac{N_s^a}{N^a} \geq I \qquad (2)$$

where U^a are the number of users using application a, U_s^a is the number of users that have connected to service s through application a. Note that N^a is the number of application sessions (e.g. the number of unique *(Process Identifier, Application, Source IP)* tuples) while U^a is the number of active application installations a (e.g. the number of unique *(Application, Source IP)* tuples). Essentially, U is the relative number of users using application a which access service s. This metric prevents biased results if a single application installation, making up a large part of the sample set, uses service s frequently. On the other hand, I captures the relative amount of application sessions accessing service s. Dependencies are regarded as static if both values exceed 10% (based on experimental results of the authors) - i.e. at least 10% of the application installations, as well as application sessions, accessed service s.

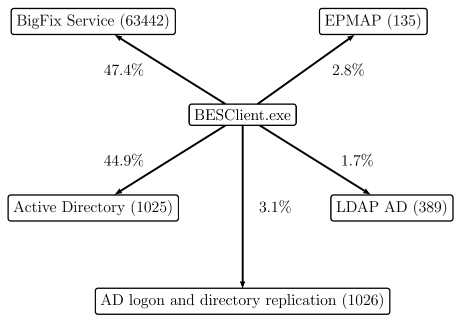

Figure 5: Macroscope sample output (based on [10])

Next to dependency extraction, Macroscope also offers dependency profiling mechanism. This enables detailed dependency analysis, for example by inspecting the amount of traffic generated by a dependency. Also, causal relations between dependencies can be derived. Figure 5 shows a graphical output by the Macroscope system for the BigFix Enterprise Suite, an application for remote system administration. The edges are labelled with the relative traffic usage of each dependency. It can be clearly seen that the majority of traffic is directed towards the BigFix service itself followed by the Active Directory server.

Due to the use of operating system knowledge, Macroscope has a better dependency detection ratio than other solutions such as Orion or Sherlock. Other than the identification of transient relations, there are no further statistical based steps in this approach. This leads to a low number of false positives as completely independent connections are not even considered to be a dependency. However, this comes at the price of flexibility. Macroscope requires the installation at each endpoint in order to determine application dependencies. While this may be feasible in some environments, such as in homogeneous setups, it may become more difficult if a multitude of platforms has to be supported. Additionally, the overhead of deploying Macroscope on every system may be problematic. Another issue of the operating system based approach is, that some dependencies will be simply missed as messages invoking them are not dispatched by the application itself. One popular example are DNS name queries which are usually handled directly by the operating system and therefore are not directly associated to the querying application in the connection table.

[1]These are well known ports specified by IANA at `http://www.iana.org/assignments/port-numbers`

4.4 Traffic Dispersion Graphs

Iliofotou et. al ([8]) introduce Traffic Dispersion Graphs (TDG) in order to extract dependency information from a network. While their work concentrates on identifying peer to peer applications many of the heuristics are also applicable to generic dependency analysis.

A TDG is a graph $G = (V, E)$, where the vertices, V, represent the nodes within a network and the edges, E, connect two nodes only if a flow between the two nodes exists. Edges are directed so that the initiator of the connection is represented. In the case of TCP connection, SYN or SYN/ACK packets are used to derive the direction while in the case of UDP the first packet of the data sample is used.

The resulting graphs can be filtered using *edge filters*. For example, the destination port of a connection could be used to filter for a specific service such as HTTP on port 80. These filters are named *TDG port filters*. Other potential filters include filtering by traffic rate or by traffic count.

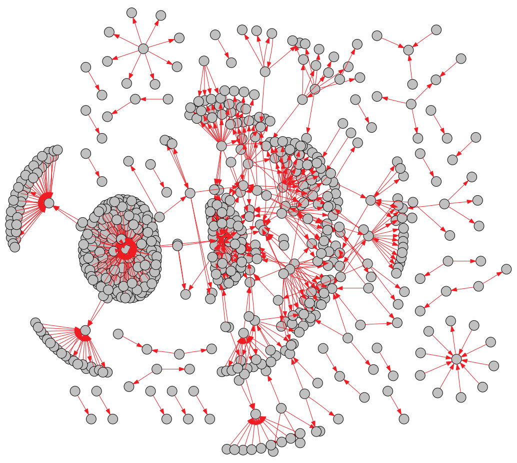

Figure 6: Visualisation of a TDG with a DNS port filter [8]

Visualising TDGs is a very expressive way of showing dependencies of a network. By using a graph layouting algorithm which places connected nodes within the same area, single dependencies can be spotted quickly. Distinguishing servers from clients can be done by looking at the degree of a node and the direction of the edges. For example, Figure 6 shows a TDG with a DNS port filter set. The DNS servers can be easily recognised due to the amount of incoming edges.

Mathematically several graph related metrics can be used to inspect TDGs. For example, a TDG can be analysed using the average degree of nodes. The average degree is the number of incoming and outgoing edges of a node. Graphs with a large number of high average degree nodes are typically tightly connected. Another metric available is the In-and-Out degree (InO). The InO is the percentage of nodes which have a non-zero in-degree as well as a non-zero out-degree. Typically, servers will have a low InO as they generally do not use many outgoing connections.

TDGs show their strength when used for detection of single services. Using Traffic Dispersion Graphs, network phenomena can be either inspected visually or identified using graph metrics. While the authors already introduce various metrics which can be used to categorise filtered TDGs, automatic dependency discovery based on these graphs still has to be part of further research. However, TDGs can already be used to detect certain applications or for anomaly detection within networks.

5. SUMMARY

This work introduces several different dependency analysis systems ranging from active to passive approaches. In the following section the different approaches will be compared, highlighting their strength and weaknesses.

Active Dependency Discovery enables fine grained dependency detection at the cost of generality. Because of the active approach some knowledge about the service tested, such as its external communication protocol, must be known. Additionally, a typical workload has to be created before dependency extraction can begin. Depending on the type of service, this can be automated. In other cases, manual creation of the workload may be needed. However, the results of the dependency extraction are more fine grained than those of the other systems. For example, certain parts of the workload can be related to specific dependencies.

In contrast Orion and Sherlock do not need detailed knowledge of the system itself. As they work on raw packets, data can be collected anywhere in the network. This enables large-scale deployment and dependency detection for any application in the network without manual user intervention. However, as with any statistical approach results do not always match ground truth dependencies. While both systems can be configured for specific workloads there is a trade-off between the number of false positives and true negatives potentially requiring analysis by human operators after dependency extraction.

Macroscope tries to levitate this problem by utilising operating system knowledge to extract dependencies. As a result, both the number of false positives and especially the number of true negatives are significantly lower compared to Orion and Sherlock. However, due to the endpoint installation dependency, deployment within the network, especially in heterogeneous networks, is more difficult.

Traffic Dispersion Graphs define a methodology for capturing network communication in a graph data structure. This data structure makes it possible to visualise relations between nodes in a network but also enables the application of graph metrics for computational feature extraction. TDGs aim to detect the presence of specific applications rather than automatically inferring all dependencies between services. They provide a high level viewpoint of the network communication structure enabling the detection of network anomalies.

6. REFERENCES

[1] P. Bahl, P. Barham, R. Black, R. Ch, M. Goldszmidt, R. Isaacs, S. K, L. Li, J. Maccormick, D. A. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering dependencies for network management. In *In Proc. V HotNets Workshop*, pages 97–102, 2006.

[2] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2007.

[3] R. Black, A. Donnelly, and C. Fournet. Ethernet Topology Discovery without Network Assistance. In *ICNP*, pages 328–339, 2004.

[4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 377–390. IEEE, 2002.

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. *Dependable Systems and Networks, International Conference on*, 0:595, 2002.

[6] X. Chen, M. Zhang, Z. Mao, and P. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 117–130. USENIX Association, 2008.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Networked Systems Design and Implementation*, number April, 2007.

[8] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network traffic analysis using traffic dispersion graphs (TDGs): techniques and hardware implementation. 2007.

[9] B. Lowekamp, D. O'Hallaron, and T. Gross. Topology discovery for large ethernet networks. In *SIGCOMM*, SIGCOMM '01, New York, NY, USA, 2001. ACM.

[10] L. Popa, B. Chun, I. Stoica, J. Chandrashekar, and N. Taft. Macroscope: end-point approach to networked application dependency discovery. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 229–240. ACM, 2009.

[11] J. Scheck. Taming Technology Sprawl, 2008.