DYSWIS: Collaborative Network Fault Diagnosis - Of End-users, By End-users, For End-users

A Framework for Automated Network Fault Detection and Diagnosis System

Kyung-Hwa Kim
Columbia University
khkim@cs.columbia.edu

Vishal Singh
Columbia University
vs2140@columbia.edu

Henning Schulzrinne
Columbia University
hgs@cs.columbia.edu

ABSTRACT

With increase in application complexity, the need for network faults diagnosis for end-users has increased. However, existing failure diagnosis techniques fail to assist the end-users in accessing the applications and services.

We present DYSWIS, an automatic network fault detection and diagnosis system for end-users. The key idea is collaboration of end-users; a node requests multiple nodes to diagnose a network fault in real time to collect diverse information from different parts of the networks and infer the cause of failure. DYSWIS leverages DHT network to search the collaborating nodes with appropriate network properties required to diagnose a failure. The framework allows dynamic updating of rules and probes into a running system. Another key aspect is contribution of expert knowledge (rules and probes) by application developers, vendors and network administrators; thereby enabling crowdsourcing of diagnosis strategy for growing set of applications.

We have implemented the framework and the software and tested them using our test bed and PlanetLab to show that several complex commonly occurring failures can be detected and diagnosed successfully using DYSWIS, while single-user probe with traditional tools fails to pinpoint the cause of such failures. We validate that our base modules and rules are sufficient to detect infrastructural failures causing majority of application failures.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network management, Network monitoring; C.2.4 [**Distributed Systems**]: Distributed Application

General Terms

Management, Design

Keywords

Network management, fault diagnosis, end-user, peer-to-peer, DHT

1. INTRODUCTION

While operating systems and computers have generally become more user-friendly and reliable, Internet usage can still be frustrating - applications fail silently, things that worked yesterday don't work today and failures are often transient.

Compared to a few years, consumer Internet usage has changed in at least four aspects: Users now expect to connect to a wide variety of networks, from home and office to WiFi hotspots and cellular networks. Applications have become more demanding, as almost every application, from calendars to games, relies on Internet connectivity and a number of applications, such as VoIP and VoD, require consistent performance. Finally, such applications often rely on the proper functioning of up to half a dozen parties, from the local wireless network to DNS servers, CDNs and various middleboxes. For all of these components, professional assistance is either unavailable or expensive, so that most users have to become unwilling network administrators (or rely on their children and technically-savvy friends for support). Thus, users have no good way to know whom to call or what to try when things go wrong.

Most applications provide, at best, minimal support to help pinpoint the potential sources of trouble. For example, if web access is slow, the cause could be high packet loss on the local wireless network due to interference, an overloaded residential Internet connection, IPv6-to-IPv4 failover, wide-area network problems, a misconfiguration in the NAT box, or a remote server problem. The appropriate action differs in each case, ranging from using a third-party DNS server to simply waiting and hoping that the server recovers.

Motivated by these real-world problems, we have developed DYSWIS ("Do You See What I See"), a system for end users and enterprises to diagnose a range of network-related problems. DYSWIS differs from other approaches in relying on the assistance of other network users, modeling the common pattern where one person asks somebody close by 'hey, is your Internet working?'. Reflecting the proliferation of services, both standardized and proprietary, DYSWIS is designed to be extensible by users and third parties, such as vendors of applications. New probes and rule sets can be added to the running system.

Thus, DYSWIS is the first complete system that automatically diagnoses common network problems for end users, using peer assistance and an extensible probing and rule framework. DYSWIS has two main roles: a framework for developers and software for end-users, which is developed based on the framework.

The main contributions of DYSWIS architecture are:

Framework design: DYSWIS is a complete framework for fault detection, user collaboration and fault diagnosis. The prototype of DYSWIS software is based on the framework and provide multiple samples which diagnose common fault scenarios.

Leveraging DHT: DYSWIS leverages DHTs to enable node collaboration and achieve Internet scale. Each node publishes their information with their location and network connection state via a DHT, so that a node can effectively discover appropriate collaborating nodes instead of contacting random nodes and receiving unwanted information from them.

Categorizing nodes: A node categorizes other nodes by their properties. It allows a node to compare probing results from different networks and reasonably infer the status of network infrastructure, which is invisible to end-users, without any help from network core devices.

Crowdsourcing expert knowledge and probing modules: We adopt rule system for flexibility, which is separated from implementation, for crowdsourcing of network experts' knowledge. DYSWIS provides a simple interface for adding new probing modules. The rule and probing module interface allows the multiple groups of developers, network administrators and application vendors to participate in writing new probe modules as well as to contributed rules which utilizes others' modules without additional effort.

Failure Coverage and Diagnosis Software: Though existing fault diagnosis probes and expert knowledge (rules) are not application specific, it covers diagnosis of infrastructural components which, we believe, plagues availability of services to users and brings user applications down. In this sense, this contribution of DYSWIS is a practical one and we believe that DYSWIS software will successfully provide assistance to end users for majority of their failure experiences.

From Section 2 to Section 5, we present the detail of framework design. In Section 6, we elaborate our implementation and in Section 7, to evaluate our approach, we present several common fault scenarios and show how DYSWIS software diagnoses these faults using our framework. In Section 8, we discuss security issues.

2. FRAMEWORK OVERVIEW

In this section, we provide an overview of DYSWIS. DYSWIS runs on end-users' personal machines such as desktops and laptop computers located at the edges of the networks. Although we believe that DYSWIS's collaborative architecture can be applied to network core devices such as routers, in this paper, we focus on how DYSWIS supports end-users and how end-users, developers, and network administrators add their own features and modify them to fulfill their specific purposes. In this context, DYSWIS is designed as a framework rather than a single application. In Section 3, 4, and 5, we describe three main modules of DYSWIS framework: fault detection, communication, and diagnosis.

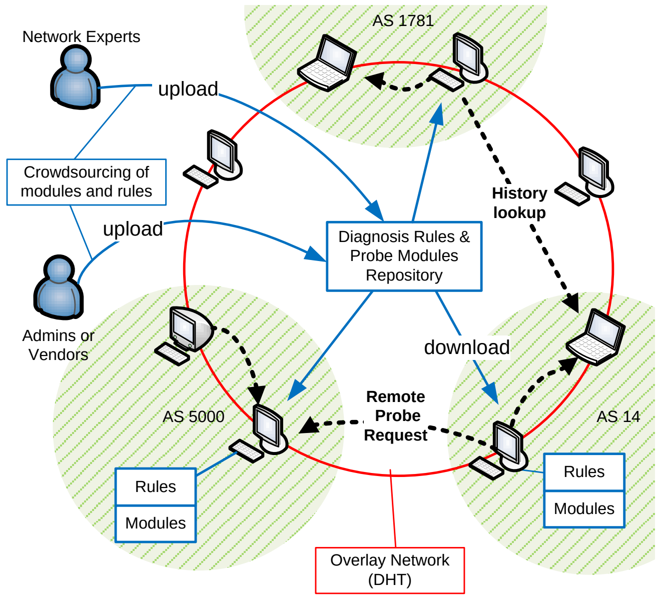


Figure 1: DYSWIS Architecture

In order to detect network faults, we monitor both incoming and outgoing network packets. Capturing those packets continuously, we monitor layer-3 protocols (TCP and UDP) as well as application protocols such as DNS, HTTP, SMTP, SIP, and RTP. We suggest three fault detection methods: finite state machine, timeout, and protocol observation. We discuss these methods in Section 3.

When DYSWIS is launched, the communication module automatically joins the DHT network and publishes the node-specific information such as subnet, IP address, whether it is using firewall or not, and whether it is behind NAT or not. A DYSWIS node periodically collects this information and keeps the list of the nodes, which are categorized into three groups: *sister node*, *near node*, and *far node*. Then, the communication module sends probe request to those nodes or responses to requests from other nodes. Section 4 discusses the process in detail.

Once a fault is detected, diagnosis module determines rules to be executed and follows the pre-defined rules step by step. In most cases, the rules encode decision trees. They indicate which probing modules should be invoked and the result of previous probing module determines the next step. The probing modules can be executed locally (*local probe*) or remotely (*remote probe*). We elaborate on the diagnosis process in Section 5.

As a framework, DYSWIS has a scalable feature which everyone can contribute to develop and improve the system. We define three groups who participate to build the system: end-users, network experts, and module developers.

3. FAULT DETECTION

In this section, we describe the types of faults we focus on and show how DYSWIS detects them automatically.

3.1 Fault model

Rather than focusing on simple network connectivity problems, we are more concerned about partial network problems, which are often more complex to diagnose (e.g., network is working, but a particular application, protocol, server, or infrastructure is not), as well as network performance problems (e.g., TCP congestion, slow wireless connection).

3.2 Packet classification

DYSWIS continuously monitors users' network activity by capturing network packets. These packets are classified according to application protocols, distinguished by port numbers. Also, TCP control packets like SYN, ACK, and FIN are monitored and classified to check network faults which are related to TCP streams. After classification, each packet is delivered to corresponding *Session Trackers* which are tracking multiple active sessions in the user's system. The packets are categorized again by 4-tuples (source IP address, source port, destination IP address, and destination port) to decide to either initiate a new session or add this packet to an existing session.

For example, in TCP protocol case, if we detect a TCP SYN packet, we initiate a new TCP session tracker and it begins to record the following packets to detect any anomalies. In HTTP case, HTTP GET message will start a new session if it is not matched with any existing sessions.

3.3 Fault discovery

The second step is judging each session to determine whether network faults have occurred or not. For this, we have four mechanisms.

3.3.1 Error Response and Error Flag

The simplest method to determine network fault is to check the response code of each packet. This is dependent on protocols. For example, if *500 Internal Server Error* has been detected in a HTTP response message, we regard this as a network fault. Also, we monitor some important flag bits of the packets. For example, TCP RST flag is being monitored because it means that connection was attempted to an unavailable TCP port on a server. We regard this flag as a network fault, so we start to diagnose: (1) request other nodes to try to connect to the same port on the server; (2) compare the feedbacks from other nodes; and (3) conclude whether there was any complicated reason of RST flag or it was caused simply by user's mistake (typing a wrong port).

3.3.2 Timeout

However, if a server is unavailable, no response message would come back. In this case, we have to install a timer for each session and see whether timeout happens. For example, we can define TCP connection timeout error as "No TCP ACK packet has been detected for 15 seconds since a TCP SYN packet had been sent out." Similarly, we can define another timeout fault for SIP protocol: "No SIP 200 response has been detected for 10 seconds since a SIP Invite message had been sent out"

3.3.3 Abnormal Flow

Above approaches are not enough for some protocols. If duplicated TCP packets, which have the same sequence numbers, are detected, it means that a TCP retransmission occurred; some of the packets were dropped at some points of the network path to the receiver. Although this is a natural phenomenon rather than a serious problem, which is caused by TCP congestion control, if this happens abnormally often, there may be some problem or transient faults in the network. In this case, because it is not an explicit error, we have to follow the flow of the TCP sessions and check the sequence numbers. TCP session tracker raises a fault when there are too much retransmissions and duplicated ACKs.

3.3.4 Finite State Machine

Finite State Machine is another approach to detect network faults. It is very useful because it can integrate the three models above. Figure 3 shows a HTTP 1.1 FSM which has multiple GETs and multiple RESPONSEs in a single session. The FSM covers error messages, timeout, and unexpected flows. (Adding the function of checking sequence numbers, we can create TCP FSM in a similar manner.) The FSM for each protocol can be created manually by experts of the protocol or automatically by machine learning approach. This approach remains future work.

As we stated earlier, our main idea is to share knowledge of experts of particular protocols with general users who do not have deep understanding of those proto-

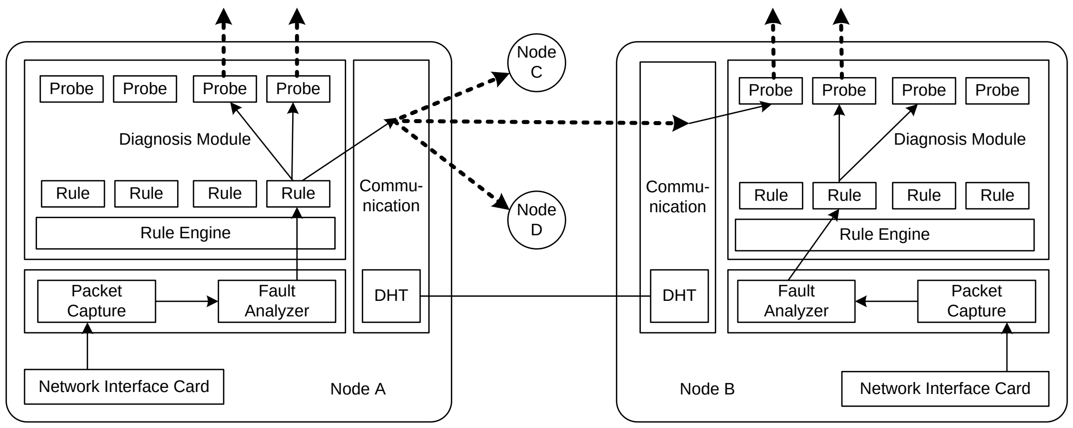


Figure 2: DYSWIS Flow from Detection to Diagnosis

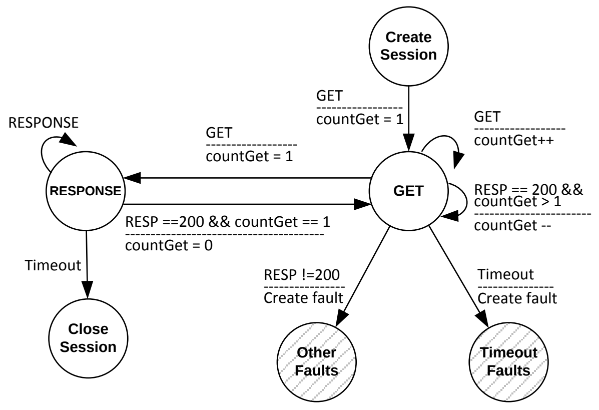


Figure 3: HTTP Session Tracker - FSM Model

cols. The experts or creator of the protocol may define the fault FSM of their protocol and distribute it with DYSWIS.

4. COMMUNICATION

The communication module discovers other available nodes and communicates with them to diagnose a fault. Instead of running a central database to maintain a list of collaborating nodes, we use a DHT because it allows us to avoid central bottleneck problem (e.g., a database failure) and reduce burden to operate central servers. Once appropriate nodes are found, we send remote probing requests to other nodes and receive the result from them (Figure 1).

As stated in Section 3.1, we are more focusing on partial network faults, so that we assume a node can connect to the DHT network or at least connected once to the DHT network and retrieved and cached the list of available collaborating nodes. However, Service Discovery Technology such as Bonjour [3] can be also adopted when we cannot join DHT network while the local area network is available.

4.1 Publishing node information

DYSWIS publishes the node's network information to the DHT network immediately after it is launched. By doing this, the node advertises to others that it is ready to help them and also will request help to them when it detects faults. DYSWIS nodes gather other nodes' information periodically from the DHT network and store the list locally. The nodes in the list are categorized into four groups:

- ***Local Node***: A node currently diagnosing the faults.
- ***Sister Node***: A node sharing the same NAT device with the *local node*.
- ***Near Node***: A node within the same subnet as the *local node*.
- ***Far Node***: A node located in any other subnets.

DYSWIS stores and queries the node information using (key, value) pairs. The key is a single string, which is used when a node wants to find a particular group of nodes stated above (See Section 4.2). It indicates whether a node has a public IP address or not and which subnet the node is located at. A single node constructs multiple keys with its information because common DHTs only support exact matching query. The value corresponding to the key contains the node's IP address, port number, and properties such as name of operating system, network connection (e.g., wireless), or wireless information. Followings are samples of (key, value) pairs.

Note that a single node publishes multiple (key, value) pairs. For example, in Figure 4, a node that has a public IP address publishes (A1, B1) and (A2, B1). Key A1

Key A1: "*public*"
Key A2: "*public@128.59.16.1/24*"
Value B1: "*IP=128.59.21.16; Port=1234; ASN=14; subnet=128.59.16.0/24; Firewall=No; Wireless=No; OS=WindowsXP;*"

Figure 4: A public node

Key C1: "*NAT*"
Key C2: "*NAT@128.59.21.16*"
Value D1: "*IP=192.168.0.1; Port=1234; ASN=14; subnet=128.59.16.0/24; Firewall=No; Wireless=No;* "

Figure 5: A node with a private network address

means that the node has a public IP address and key A2 has additional information, the subnet of the node. Also, in Figure 5, a node behind a NAT publishes (C1, D1) and (C2, D1). Key C1 means that the node is behind a NAT and key C2 implies that the node's external network address is 128.59.21.16. This (key, value) pairs allow other nodes to easily find appropriate nodes. For example, key C2 is used when a node tries to discover a *sister node*. We discuss the search process in the next section.

4.2 Searching nodes

To discover available collaborating nodes, DYSWIS queries the DHT network with keys which depend on what kind of collaborative nodes are required to assist the diagnosis process. For example, if a diagnosis rule needs a *near node*, DYSWIS queries to DHT with a key formatted like A2 in Figure 4. The *subnet-address* field should be filled with the subnet address of the *local node*. If the local node is behind a NAT, we often need to discover *sister nodes* to obtain the view from the same environment. To discover them, we create a key with a format of C2 in Figure 5. Note that, in this case, we use the public the IP address instead of subnet address. It means, 'we are seeking a node that is behind a NAT which has a particular public IP address, 143.248.1.123', which implies a node sharing the very same NAT device with the *local node*. To seek a *far node*, we simply query with a key, "public". It returns a list of public nodes (near and far nodes). In this case, in order to create a list of *far node*, we check the subnet address field in values and filters *near node* out. Also, with the same format of the key, we can discover a node located at a specific subnet. This is used when we need a probing result from the nodes in a particular subnet where our target server to be diagnosed is located at. The examples of keys are summarized in Table 1. The next step is sending probe request to discovered nodes and receive result from them. The IP addresses and port numbers contained in the returned value tell us where we can send the probing requests. We present this step in the next section.

4.3 Request and Response

DYSWIS sends probing requests to remote nodes using the IP address and port number obtained from the response of DHT query. The request contains a name of probing module to be invoked and fault information to be used as parameters. The response from the remote probing can be either a return value of probing module (Section 5.1.2) or '*no response*'. Sometimes, '*no response*' provides an important clue to diagnose the fault. For example, if a node can contact some *near nodes* while it fails to contact every *far node*, we can guess there is a network connection issue from the local subnet to outside network.

5. DIAGNOSIS

DYSWIS automatically records detected faults in the fault history repository and begins to diagnose the fault immediately if the user configured to do it. Fault diagnosis is processed in a particular sequence which is predefined by diagnosis rules. The diagnosis rules indicate which probing modules are to be invoked to diagnose the network fault.

5.1 Probing module

In this section, we present probing modules in detail.

5.1.1 Probing model

We advocate active probing which means the nodes probe the faults dynamically in real time. This is different from passive probing which merely shares pre-obtained knowledge such as configuration data or past successful history of a node. We more focus on active probing in our system because the pre-obtained knowledge can be stale.

Also, in the remainder of this paper, we use *local probing* to denote probing executed by the node itself which detected the fault and *remote probing* to denote a probing executed by other nodes to help diagnose the fault.

5.1.2 Interface

In order to reduce redundancy and increase reusability, each probing module must have minimum functions. In our module design and interface, we have two main features: (1) Each probing module has only one probing function; (2) Identical format of input parameters and return values - The unified format enables scalable system.

- ***Input parameters***: (1) Type of probing (remote or local); (2) Fault description (e.g., protocol, fault type); and (3) Parameters of probing request (e.g., target server).

Table 1: Searching nodes

Key Format	Example	Return Values
public@[subnet X]	"public@128.59.16.0/24"	Near nodes
public@[subnet Y]	"public@143.248.1.0/21"	Far nodes in a specified subnet Y
public	"public"	Random public nodes. (near and far nodes) - Check subnet field in the values and exclude near nodes to get only far nodes
NAT[IP address Z]	"NAT@128.59.21.16"	Sister nodes behind the same NAT device

- ***Return value***: The return value is either *success* or *failure* - it indicates the result of requested probing. For example, in the case of TCP connection test, *success* is returned when connection to the target server was successful.

Note that probing module can be invoked either locally or remotely. If remote probe is needed, the communication module invoke remote nodes' probing modules via network as described Figure 2. Since all nodes have the same probing module interfaces, they can be remotely invoked without conflict. In our implementation, we provide a Java `interface` for probing modules, and thus module developers create a class and `implement` the `interface` to add a new module.

5.2 Rule system

Using a rule system, DYSWIS separates diagnosis strategies from implementation of probing modules. The rule system determines which probing modules need to be invoked in which order and where - local or remote - the probing modules should be. A rule also analyzes the feedbacks from other nodes and provides diagnostic advice to users. In DYSWIS framework, there are two groups who participate to build new diagnosis strategy: Probing module programmer and network experts (e.g., network administrators, application vendors). Programmers create new probing modules which fit to new protocol which they want to add or modify basic probing modules which we provide while the network experts write rules to determine the sequence of executing probing modules. When they build new diagnosis strategy, they simply list up necessary probing modules and construct new rule with them.

5.2.1 Decision Tree

Decision tree is a straightforward way to make a diagnosis rule. When a diagnosis process begins, it invokes the first probing module and decides the next step depending on the probing result. This is repeated until we reach a leaf of tree, which is either a conclusion or executing another rule. As described earlier, in the case of remote probing, each decision entry usually has three branches: *success*, *failure*, and *no response*. The decision tree need to be encoded to a rule language. Figure 6 is a sample of DYSWIS's lisp-style rule syntax.

```
...
(deffunction TcpConnection1-no (args)
(bind other-port (RemoteProbe "ListenOtherPort" "far-node"
args))
(if (eq "success" (LocalProbe "TcpConnectionProbe" other-port))
then
    (RemoteProbe "TcpConnectionProbe" "sister-node" args)
    (Analysis "the port is blocked.")
else
    (Analysis "inside node test is needed")
    (TcpConnection2-no args)))
...
```

Figure 6: Sample rule

6. IMPLEMENTATION

We wrote the first version of DYSWIS. Since the framework provides a number of APIs which hide the detail of underneath operations such as capturing packets, searching nodes, and executing diagnosis rules, it allows programmers who do not have experience with distributed system and remote procedure call to easily add their own diagnosis modules and rules without concerning themselves with finding collaborative nodes and sending them probing requests. Also, on top of the framework, we provide various diagnostic packages in order to prove our approach as well as provide sample diagnosis modules for the real-world network problems. Integrating those packages and DYSWIS framework, we provide complete standalone software with user-friendly GUI (on Linux, Mac OS, and Windows XP). Figure 11 shows screen dumps of DYSWIS software running on Windows XP and Mac OS.

We designed DYSWIS to attain scalability, modularity, and reusability. First, DYSWIS uses the AzDHT library [12], an implementation of Kademlia [19] to build our own overlay network. Once appropriate nodes have been discovered via DHT, XML-RPC is used to interact with those nodes. It enables the nodes to communicate with XML-encoded messages and allows a user to remotely invoke probing methods on the machines of the collaborating nodes with desired parameters and receive corresponding results.

A diagnosis package is a collection of multiple diagnosis modules. Usually, it includes several modules which probe the same protocols such as HTTP, DNS, and TCP. Otherwise, some modules which probe a particular environment such as wireless or NAT can be aggregated into an independent diagnosis package. We utilize OSGi [21] to handle these diagnosis packages. OSGi is a java-based framework which protects each Java class from another class's accessing its variables and methods. Using OSGi, DYSWIS protects each diagnosis packages from other packages as we expect programmers would participate to write different diagnosis packages. We also leverage this technology to update diagnosis packages dynamically and automatically. Once DYSWIS is installed on users' machines, it will check the central update server periodically and download updated or newly added resources (diagnosis modules and rules). OSGi framework dynamically replaces running modules with the updated modules without re-launching the entire software.

The rule system enables users add or modify existing rules without re-compiling the source code. Also, rule developers can easily create new rules without analyzing the source code. We expect this feature encourages not only programmers but also administrators without knowledge of programming to participate to write rules. The Jess rule engine and library [24] are used in DYSWIS.

7. CASE STUDIES

In this section, we elaborate several network fault diagnosis cases to evaluate DYSWIS framework and its collaborative approach. In order to provide convincing samples, we have focused on common faults on widely-used network protocols such as DNS, TCP, and HTTP. Also, we introduce a couple of strategies to diagnose network faults which occur under particular environments such as NAT and wireless. In each case, we explain how to develop a new diagnosis strategy using DYSWIS and show how it infers the cause of the faults.

To develop a new strategy, three steps are needed: (1) defining a network fault; (2) writing a diagnosis rule; and (3) writing probing modules. For each fault case, we have implemented probing modules and wrote sample rules. Note that although new rules and probing modules are required for each fault, existing modules which are implemented by other developers can be also reused. For example, probing modules used in TCP protocol diagnosis can be reused in HTTP protocol diagnosis. In fact, in many cases, building a rule for a new diagnosis strategy simply means drawing a new decision tree and filling out the tree with existing probing modules.

Throughout following sections, we describe the detail of the probing modules and rules to diagnose the selected network faults. We believe that these diagnosis cases successfully verify our claim that collaboration of users is actually helpful to diagnose complex network faults.

To evaluate DYSWIS framework, we deployed the linux version of DYSWIS on multiple PlanetLab nodes and used them as *far nodes*. Also, we tested multiple injected faults using our internal testbed consisting of three laptops and a NAT device. The internal nodes discover other DYSWIS nodes on PlanetLab via the DHT network using our search method. The failure scenarios which we generated for evaluating DYSWIS software are explained using the following case studies:

7.1 Network connectivity

The end-user's local machine itself often causes network faults. For example, a wireless device of a laptop may be turned off by mistake or the LAN cable may be unplugged. Thus, the first step of the diagnosis is usually checking network connectivity. We define our network connectivity test rule using various test modules such as checking network cables, IP address, DNS server address, TCP stack, and network interfaces. If one of the tests fails, we conclude that the user has a network connectivity problem and invoke more specific rule for the fault.

The network connectivity test is executed locally and does not require the collaboration of other nodes. However, the result of the test determines whether the diagnosis is done or a further diagnosis step, which requires help from other nodes, is needed. For example, if an unplugged LAN cable is detected during the test, no further steps are needed. In this case, we simply display what we observed to end-users and terminate the diagnosis. However, in some network connectivity problems, it is hard to figure out the cause only with the symptom. For example, when a user has an incorrect IP address such as 0.0.0.0, we can reasonably guess that there are problems related to the DHCP server but it is difficult to determine whether the server has a problem or DHCP client behaves incorrectly. Consequently, to point out more detailed cause, we proceed to diagnose the problem invoking the DHCP diagnosis rule.

Usually, those following steps require the collaboration of other nodes. Network connectivity test provides important information about which categories of nodes are not available. In above example of incorrect IP address, we know it is impossible to contact to *far nodes* because we cannot send proper IP packets to other nodes. Instead, we suggest to discover *sister* or *near nodes* using service discovery technology discussed in Section 4.

7.2 DNS

The Domain Name System (DNS) is one of the major causes of end-user network problems. In this section, we look into the possible network faults related to DNS and present a diagnostic strategy.

Table 2: DNS Error list

Fault Name	Error Code
No error	0
Format error	1
Server failure	2
Name not found	3
Not implemented	4
Refused	5
No response	9

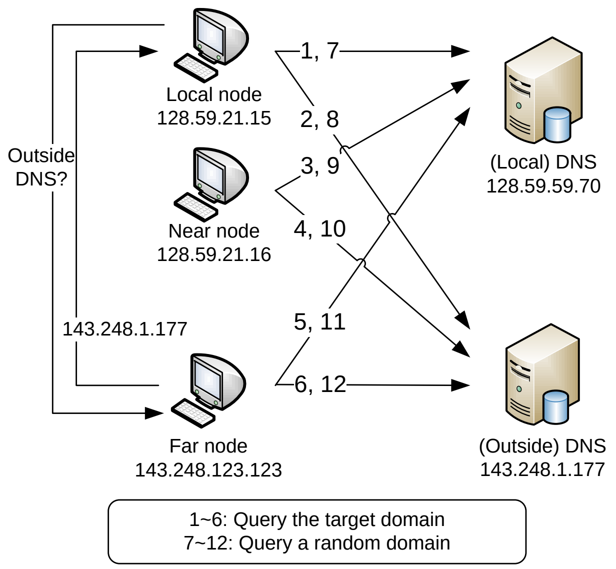


Figure 7: DNS Query

We detect DNS problems with two methods. First, they can be detected by packet monitoring as stated in Section 3. If DNS query packets are detected but there is no answer from DNS server within a certain amount of time, DYSWIS concludes that a DNS timeout error has occurred. Otherwise, if a DNS answer packet has arrived but it contains an error flag, it will also be considered as a DNS error. We define our own DNS error codes based on DNS error flag (Table 2). DNS faults also can be detected by network connectivity test as stated in Section 7.1.

DNS fault is a convincing example that clearly explains why end-user collaboration is indispensable. Due to the hierarchical structure of DNS, a number of DNS servers, which are responsible to different level of domains, are running in the Internet and they interact with each other to update domain resolution information. Although this hierarchical structure has significant benefits such as scalability and resilience to central server failure, it also makes the system complicated and causes confusion in diagnosing DNS fault because the states of servers and domain data stored on those servers cannot be completely synchronized. This means they produce different views between the end-users. For example, a local DNS server for a particular subnet may be down due to its administrative problem while other servers located in other subnets are working correctly. It is also possible that two DNS servers have different information about a domain because one of them has not been updated yet. However, there is a limited number of ways to diagnose those faults from the side of end-users because they are only aware of one or two local DNS servers. The naive possible diagnosis steps are as follows: (1) check whether at least one DNS server is configured; (2) check whether the DNS server is alive; (3) try to send domain resolving requests to the server; and (4) if it responds, analyze the answers. While several new facts can be learned through these steps, some of them may simply reproduce the faults which are already observed or lead to a conclusion that is not significantly meaningful. To overcome this shortage, we request multiple collaborating nodes to repeat step (3) and (4) to collect more helpful feedback, so that we can draw a more detailed conclusion.

First of all, as Figure 7 shows, we choose two collaborating nodes: a *near node* and a *far node*. Also, in order to compare the results, we choose two DNS servers to diagnose: the *local node*'s default DNS server and a random outside DNS server that is obtained from the *far node*. Finally, we query two domain names at each DNS server. One is the target domain name that originally caused the DNS error and another one is a random well-known domain name such as *google.com*. Consequently, including the *local node*, total three nodes participate in the diagnosis and they independently execute four distinct probing tests respectively. Therefore, we have total 12 feedback from the probing tests. Figure 7 shows these 12 queries. The arrows from 1 to 6 indicates queries of target domain and 7 to 12 means queries of well-known domain name. The result of each probing is one of the errors on the list of Table 2. Theoretically, the number of result combinations is 9^{12} because there are 9 possible errors. However, fortunately, the actual number of the result in the real-world is much less.

In this diagnosis example, we use a matching rule rather than drawing a decision tree because it is more straightforward to understand. In other words, we match a fault conclusion to each set of results. For example, if the error codes from 12 probing tests are all zero, which means every test was successful, the final diagnosis result is 'The problem was temporary and DNS is working correctly now'. Similarly, if the error codes from first 6 probing tests are 3 (name not found) while last 6 are zero (no error), we conclude that the queried domain name does not exist in any DNS server; the local DNS server is fine but the domain name is wrong. Table 3 shows the observed results of probing tests and

Table 3: DNS diagnosis rule

no	Diagnosis	Observed Probing Result	Local	Remote
1	Temporary problem; Resolved now	0*** **** ****	O	O
2	The default DNS server is down	9090 9090 9090	X	O
3	The default DNS server does not have an entry of queried domain while others have	3000 3000 3000	X	O
4	The default DNS server refuses to answer your query while other DNS servers do not	2020 0000 0000 5050 0000 0000 9090 0000 0000	X	O
5	The default DNS server is fine. But your network has a problem; DNS packets are blocked	9999 0000 0000	X	O
6	The queried domain name does not exist in any DNS server	3300 3300 3300	X	O
7	The domain has been created recently. The DNS server has not been updated yet	Identical to #2 or #5, but authoritative DNS has an entry.	X	O
8	Unknown problem	Others	X	X

matched diagnosis for each result. Also, the last two columns show that only one conclusion can be diagnosed by *local probing* and others cannot be diagnosed without *remote probing*.

Another advantage we take from collaboration is that we can obtain alternative solutions. For example, in the cases of problem #2, #3, and #4 in Table 3, it is obvious that the local DNS server has some problems while others are working correctly. Thus, we can temporally configure those other servers as default DNS servers until the original DNS server has recovered. Otherwise, if those outside DNS servers restrict the queries from other subnet, we can simply request the collaborating nodes to query the domain to their DNS servers and resolve the IP address on behalf of the *local node*. Finally, DYSWIS suggests those alternative DNS server or resolved IP address to end-users. However, there is a security issue that malicious nodes might provide compromised information. To mitigate this risk, we suggest asking multiple nodes to collect several alternative solutions and providing the most frequently answered solutions to the users. It is very rare that those random nodes provide the same compromised information.

It is true that the DYSWIS's collaborating approach neither guarantee to solve the problem nor always figure out the exact cause. However, note that our goal is 'reasonably pointing out the error spots' rather than solving all the network problems, which is impossible for end-users especially when the problems are located in the network core. Considering this point, multi-user collaboration apparently helps towards our goal compare to single-user probing. Also, as stated above, in many cases, alternative solution that temporarily resolves the problem can be obtained from other users - in this case, it is an IP address of the requested domain name.

7.3 TCP Connection

In this section, we describe how to diagnose TCP connection failures. The potential cause can be located at any device or link on the path from the *local node* to the target server. Our diagnosis goal in this example is pointing out the correct location of the cause.

Approximately 60% of web access problems are due to connection failure of TCP [22]. However, traditional diagnostic tools such as *ping* and *traceroute* are not always able to diagnose such failures because they use ICMP packets which are based on IP packets. It is possible that a web-server or the routers on the path to the server respond to or forward the ICMP packets while they block or drop TCP packets for a number of different reasons. On the contrary, it is also possible that routers or server do not respond to ICMP packets while they accept TCP packets. In both cases, *ping* and *traceroute* will produce a misleading the diagnosis. To determine the reasons correctly, we implemented a *TCP connection probing module* that tries to send TCP SYN packet to the target server. We execute this module locally and remotely. In the previous section, we assumed that we can communicate with any collaborating nodes. However, if there is any network connection problem, it is possible that we even cannot reach some of those nodes.

A possible combination of the collaborating nodes is showed in Figure 8. We choose three nodes as collaborating nodes for TCP connection test: a *local node*, a *sister node*, a *near node*, and a *far node*. When the *local node* fails to connect to the target server, the TCP connection diagnosis begins and the three nodes are randomly selected from each category via DHT network, and requested to try to connect to the target server. The result of each node, either 'Success', 'Failure', or 'No response', is sent back to the *local node* and it starts

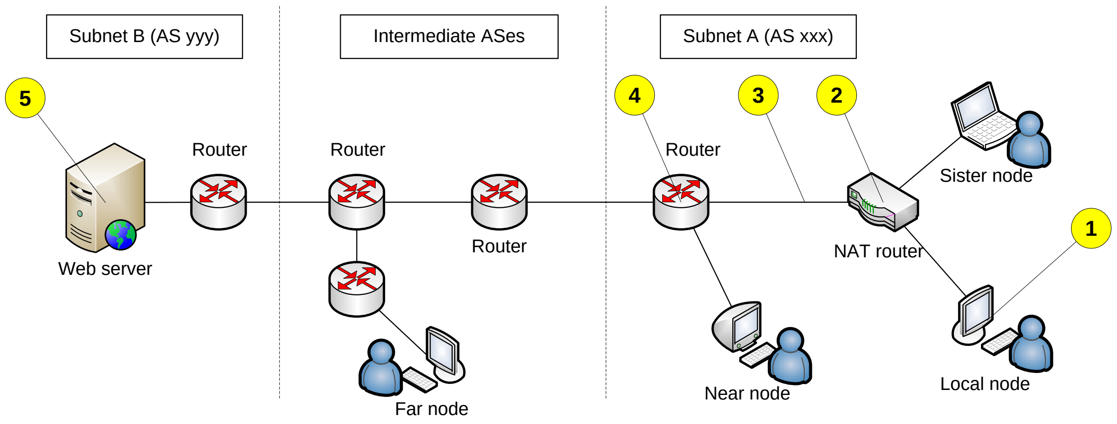


Figure 8: TCP Connection Diagnosis

to analyze the results.

It is notable that the *TCP Connection probing module* can be implemented in a couple of lines. It simply tries to send TCP SYN packet to the target server and check if TCP ACK returns within a particular amount of time. In our Java code, including error handling, only about 20 lines are enough to implement this module. However, executed at multiple points of different networks, this light-weighted probing module can collect important clues which indicate where the cause of the network fault is located.

For example, in Figure 8, we pinpoint the location of the cause among the five candidates (from 1 to 5) using the observations from each node. We compare all results from each node and conclude that the cause is located at (or related to) #1 if only the local node failed, #2 if only the local and sister node failed, #3 ifi the local and sister node cannot connect to any outside networks, #4 if the local, sister, and near node failed, or #5 if every node failed.

7.4 Port Blocking

In the problems presented in Netprints [2], 5 out of 25 recent home networks problems were related to port forwarding or port blocking. They are very common when a node uses a NAT box, firewall, or proxy server. In this section, we describe how DYSWIS diagnoses those problems and shows the steps to build the diagnosis rule and modules. For our experiment, we set up a *local node* and a *sister node* which are sharing a NAT device as described in Figure 9. First, we assume a fault scenario, which often occurs in real world, that a particular outbound port is blocked. For example, port 3389 for Windows Remote Desktop Protocol (RDP) is blocked, and thus even though the RDP server is running correctly, the connection to the server from client computer fails. This fault can be detected by detection module of TCP connection diagnosis described in Section 7.3. After detection, by the collaboration of other nodes, the diagnosis process narrows down the cause of the problem. However, in many cases, it is still not enough to point out specific cause. For instance, suppose that TCP connection diagnosis concluded that the cause is located at the private network because the *local node* and *sister node* failed to connect to the target server while the *far nodes* succeeded. In this case, we can reasonably guess that NAT device is blocking our connection but cannot assure it. Also, we have to consider carefully whether there are other possibilities. Now, we describe the steps to leverage DYSWIS framework to build a new diagnosis strategy to investigate those possible causes.

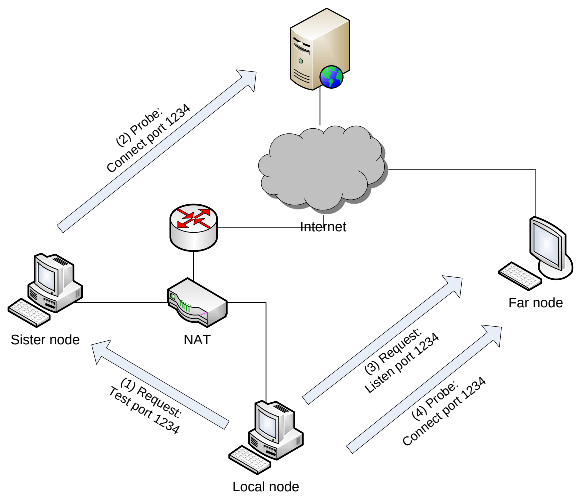

(2) Probe: Connect port 1234
Internet
Sister node
NAT
Far node
(1) Request: Test port 1234
(3) Request: Listen port 1234
(4) Probe: Connect port 1234
Local node

Figure 9: Port Blocking

1. ***Investigate the fault scenario***: Any experts such as network administrators or product ven-

dors write a rule for port blocking diagnosis. In this particular case, let us assume that they empirically imagine three possible causes: (1) NAT device is blocking a particular outbound port. (Note that, in this example, the *near* and *far nodes* already succeeded to the target server, thus we do not consider the case of inbound port blocking on the target server.); (2) Personal local firewall is running and restricting the connection; and (3) The target server is blocking the connection from the local node's IP address. (The *local node* and *sister node* are using the same public IP address)

2. ***Select probing modules***: Now, the experts choose probing modules to be executed to diagnose the fault. Since we already have basic probing modules, the experts simply choose desired modules among them. In this case, *TCP connection module* and *Request-listen module* can be used.

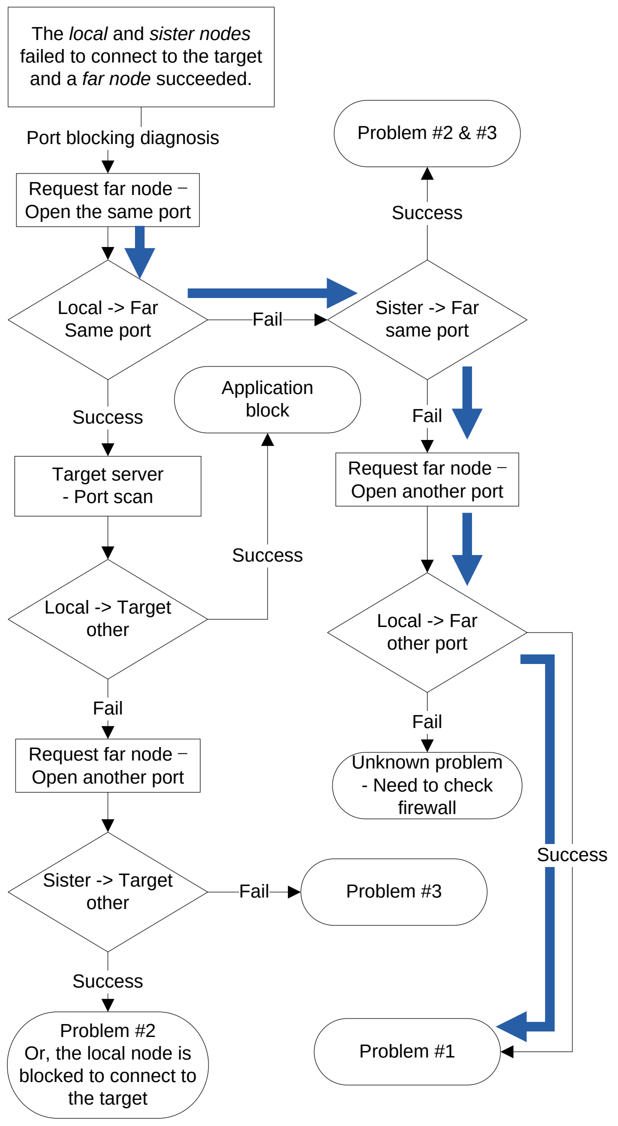
3. ***Create a decision tree***: Draw a decision tree and encode it to a rule. The sample decision tree for this particular case is described in Figure 10. The decision tree contains *local probing* as well as *remote probing*.

4. ***Deploy and update***: The experts deploy the rules using their own website or DYSWIS repository. If any new modules are created, they can also be deployed at the same time. End-users' system automatically check the DYSWIS repository and download new rules and modules. Otherwise, users can also download them from the website of the experts.

Now, we set up a fault scenario - outbound port blocked - with real network devices. Figure 9 shows our experimental environment. To create the fault artificially, we configure firewall function in the NAT device to block the port. In this environment, *ping* and *traceroute* do not provide any clues to diagnose the fault. They return the same result as usual. Even if we test a TCP connection to the target server, the only thing we can observe is that the connection continues to fail. We integrate above *Tcp connection diagnosis* and *Port blocking diagnosis* and run those diagnosis in the testbed. We can finally observe that the diagnosis process follows the thick arrows in Figure 10: (1) request a far node to open the same port as the target server; (2) try to connect to the far node - fail; (3) request a sister node to repeat the same action - fail; (4) try to connect to another port - success; and (5) reaches the problem #1 - the NAT device blocks a particular port number.

8. SECURITY ISSUES

Authentication: DYSWIS nodes will require an authentication mechanism as they allow other nodes to

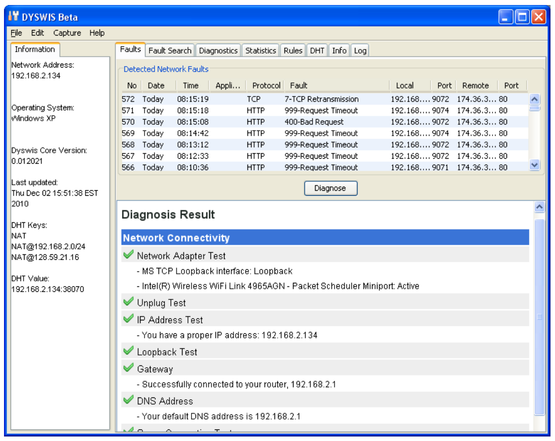


The *local* and *sister nodes* failed to connect to the target and a *far node* succeeded.
Port blocking diagnosis
Request far node – Open the same port
Local -> Far Same port
Fail
Sister -> Far same port
Success
Problem #2 & #3
Fail
Request far node – Open another port
Local -> Far other port
Success
Fail
Unknown problem - Need to check firewall
Problem #1
Success
Target server - Port scan
Local -> Target other
Success
Application block
Fail
Request far node – Open another port
Sister -> Target other
Fail
Problem #3
Success
Problem #2 Or, the local node is blocked to connect to the target

Figure 10: Port blocking decision tree - problem #1: outbound port blocked; problem #2: local firewall; problem #3: target sever blocks the local node

request probing. So that only those nodes which are authenticated as a part of DYSWIS network will be allowed to request probes or historical results. We believe that DYSWIS nodes can utilize authentication mechanism similar to other P2P based applications such as file sharing applications or communication applications (e.g., Skype). We plan to provide this in future release of DYSWIS.

Use of DYSWIS nodes for DOS: One concern with use of DYSWIS for network fault diagnosis is security. It is possible that an adversary can use the DYSWIS nodes to launch a Distributed Denial of Service (DDoS) attack by initiating probes against a ser-


DYSWIS Beta
File Edit Capture Help
Information
Faults Fault Search Diagnostics Statistics Rules DHT Info Log
Diagnose

Network Address:
192.168.2.134

Operating System:
Windows XP

Dyswis Core Version:
0.012021

Last updated:
Thu Dec 02 15:51:38 EST 2010

DHT Keys:
NAT
NAT@192.168.2.0/24
NAT@128.59.21.16

DHT Value:
192.168.2.134:38070

Detected Network Faults

No	Date	Time	Appli...	Protocol	Fault	Local	Port	Remote	Port
572	Today	08:15:19		TCP	7-TCP Retransmission	192.168....	9072	174.36.3...	80
571	Today	08:15:18		HTTP	999-Request Timeout	192.168....	9074	174.36.3...	80
570	Today	08:15:08		HTTP	400-Bad Request	192.168....	9072	174.36.3...	80
569	Today	08:14:42		HTTP	999-Request Timeout	192.168....	9074	174.36.3...	80
568	Today	08:13:12		HTTP	999-Request Timeout	192.168....	9072	174.36.3...	80
567	Today	08:12:33		HTTP	999-Request Timeout	192.168....	9072	174.36.3...	80
566	Today	08:10:36		HTTP	999-Request Timeout	192.168....	9071	174.36.3...	80

Diagnosis Result

Network Connectivity

✓ Network Adapter Test
- MS TCP Loopback interface: Loopback
- Intel(R) Wireless WiFi Link 4965AGN - Packet Scheduler Miniport: Active

✓ Unplug Test

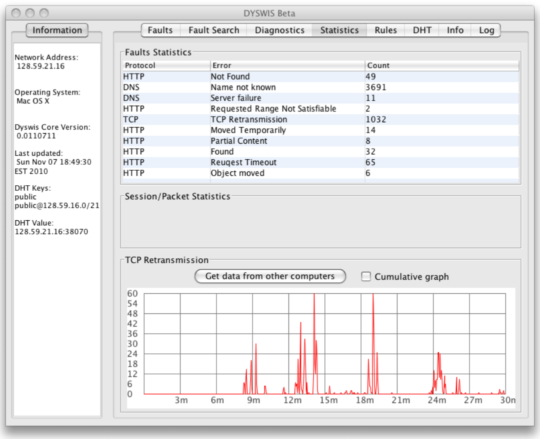
✓ IP Address Test
- You have a proper IP address: 192.168.2.134

✓ Loopback Test

✓ Gateway
- Successfully connected to your router, 192.168.2.1

✓ DNS Address
- Your default DNS address is 192.168.2.1

(a) DYSWIS on Windows - Fault Diagnosis


DYSWIS Beta
Information
Faults Fault Search Diagnostics Statistics Rules DHT Info Log
Session/Packet Statistics
TCP Retransmission
Get data from other computers
Cumulative graph
60
54
48
42
36
30
24
18
12
6
0
3m
6m
9m
12m
15m
18m
21m
24m
27m
30m

Network Address:
128.59.21.16

Operating System:
Mac OS X

Dyswis Core Version:
0.0110711

Last updated:
Sun Nov 07 18:49:30 EST 2010

DHT Keys:
public
public@128.59.16.0/21

DHT Value:
128.59.21.16:38070

Faults Statistics

Protocol	Error	Count
HTTP	Not Found	49
DNS	Name not known	3691
DNS	Server failure	11
HTTP	Requested Range Not Satisfiable	2
TCP	TCP Retransmission	1032
HTTP	Moved Temporarily	14
HTTP	Partial Content	8
HTTP	Found	32
HTTP	Reuqest Timeout	65
HTTP	Object moved	6

(b) DYSWIS on Mac OS - TCP congestion history

Figure 11: DYSWIS screen dumps

vice. However, DYSWIS nodes before executing a probe perform a look up in probe history to check if the service (or a specific server) or network was probed recently and a usable result exists. This will prevent redundant probes to be executed.

Validation of new probes and diagnostic rules: The authenticity and correctness of probes and diagnosis rules provided by application developer or network/system administrator community is another matter of concern. Since we will open DYSWIS nodes open for contributions for probe modules and diagnosis rules, we need a way to ensure DYSWIS nodes are not used to achieve some malicious behavior on the end host or the network. One approach can be verifying the contributed probing modules and diagnosis rules in a sandbox environment before allowing it to be disseminated to all DYSWIS nodes. We recognize this issue but believe that current deployment and usage is not hindered by this and we will provide a solution for this in the future. Additionally, using diagnosis rules, we can leverage DYSWIS' s mechanism itself to self monitor DYSWIS network against malicious nodes.

9. RELATED WORK

Network fault detection and diagnosis have been an area of interest for a number of years. Many solutions have been proposed and implemented by industry and academia; centralized and distributed, active probing and passive detection, rule based systems and machine learning based approaches. Existing solutions are either applicable to a specific application or specific kind of failures or applicable to enterprise or Internet service provider scenarios. They are neither general nor scalable to cover the diverse failures seen by end users.

We discuss prior work in networked system diagnosis for end users and how DYSWIS relates to it.

End user based diagnosis: We argue that user perceived experience helps in classifying a behavior as failure. This in turn may depend on network events (at the ISP) or status of infrastructural service. DYSWIS proposes end user based detection and diagnosis. There are several proposals in this space, specifically, Glasnost [13] which discovers service differentiation by ISP based on traffic tests between an end point and another controlled end point in the network. CrowdSourcing [10] proposes methodology to detect network events based on users experiences. They aim to detect events impacting user perceived application performance, by running on the end systems themselves. Effective diagnosis of routing disruptions from end systems [28] proposes end user based collaborative active probing. Tulip [18] probes routers to localize anomalies such as packet reordering and loss.

Cooperative diagnosis: Webprofiler [1] leverages end host cooperation to pool observation on the success and failure of web accesses. Although, it is similar to DYSWIS in approach, it is only applicable to web failures and is based on passive observations alone. WifiProfiler [8] relies on cooperation among wireless clients to diagnose problems and resolve them. None of the existing cooperative diagnosis techniques provides mechanism for sharing (or crowdsourcing) of diagnosis rules and probes and are very specialized. DYSWIS provides ability to write general purpose as well as specialized diagnosis rules using layered failure handlers.

P2P based diagnosis: AutoMON [6] uses a P2P-based solution to test network performance and reliability. The distributed testing and monitoring nodes are coordinated by using a DHT (Distributed Hash Table) algorithm, which helps in locating resources or agents. Unlike DYSWIS, nodes do not cooperate nor use historical information about failures. In Connected Home fault management is performed in a distributed fashion, following an agent-based approach, but only focusing on the Connected Home scenario.

Shared knowledge based: Netprints [2] is similar to DYSWIS in its use of shared knowledge; it aims to diagnose and resolve problems in home router configurations. DYSWIS is fault detection and diagnosis framework, Netprints resolves problems by using other user's experiences. In that sense, Netprints is comparable to Autobash [25] which helps to recover from system configuration errors by recording the user actions to fix a problem and then replaying the same commands on another computer experiencing same problem and then testing it and rolling back in case test fails. DYSWIS relates to AutoBash in the sense that DYSWIS also needs tests to examine if a service is correctly running; which depends on network responses (protocol error codes) unlike Autobash which uses terminal output and specific keywords on command line after command execution. Netprints applies decision tree based learning for working and non working configuration for different home routers, for different applications. Since, the scope of problem is only home router configuration, decision tree learning is sufficient but it cannot be generalized to a lot of end user problems which occur under a wide variety of conditions. DYSWIS seeks cooperation from other nodes, hence, validates if others under similar external conditions are experiencing the problem are not.

Peer state comparison based: Strider [20] is a black-box approach for diagnosing Windows registry problems by performing temporal and spatial comparisons with respect to known healthy states. It is limited because of lack of ability to track configuration changes for each application without fine grained instrumentation. PeerPressure [27] improves Strider as it does not need to obtain state from a healthy machine. Instead, it relies on registry settings from a large population of machines assuming that most of these are correct. It is prone to false positives and need not identify combinations of configuration settings that are problematic.

Rule based systems also called expert systems diagnose based on a set pre-encoded rules [7, 17, 14] which represent the dependency between components or network protocols and failures. They are limited as they need to be updated for a change in network. However, DYSWIS tries to achieve crowdsourcing of rules for different failures and subsystems. Also, DYSWIS approach is based on fact that a large fraction of commonly occurring failures are because of infrastructural failures [22] and can be diagnosed by diagnosing these infrastructural components from multiple point of views.

Inference-based systems like [4, 15] diagnoses faults based on a model of dependency graph. These models typically learn dependency graph based on traffic flows or instrumentation and apply scalable inference. Cohen et al. [11] solves the problem of automated diagnosis on a per server basis. They identify combinations of low-level system metrics (e.g., CPU usage) that correlate well with high-level service metrics (e.g., average response time) and use Tree-Augmented Bayesian Networks (TANs) for this. In contrast, DYSWIS does not rely on any particular correlation or classification technique instead leverages domain knowledge of experts and captures this knowledge into rules which use probes and queries when fired. Additionally, it is not limited to solving at single host level.

Network traffic based dependency discovery: This class of work aims to infer application relationship from traffic flows [9, 23, 4, 16] based on co-occurrence and delay distribution properties. The aim of such works is to determine related applications based on correlated flows observed on each host. Orion [9] exploits the time correlation of different services by calculating the delay distributions of any two-flow pairs. Another class of work relies on tracing the request execution paths [26] to infer the causal dependency between different application components. Magpie [5] is a instrumentation based tool that extracts a request's processing path via analyzing the event logs generated by different system components.

10. CONCLUSION

DYSWIS diagnoses complex end-user's network problems using end-user collaboration. We provide a new framework for collaborative approach and diagnosis strategies for various fault scenarios. We provide a detailed design to discover and communicate with collaborating nodes. Also, we provide a framework for administrators and developers to participate to contribute to expand the diagnosis system.

We have implemented the prototype of DYSWIS framework and present how easily the participants add new rules and modules on top of the framework in order to diagnose several common network faults. We set up these scenarios with real network devices and diagnosed them using those rules and modules we have created. While local probing with traditional diagnosis tools fail to point out the cause of these fault scenarios, our evaluation presents that DYSWIS can effectively narrow down the problematic regions and pinpoint the detailed causes.

11. REFERENCES

[1] S. Agarwal, N. Liogkas, P. Mohan, and V. N. Padmanabhan. Webprofiler: cooperative diagnosis of web failures. In *Proceedings of COMSNETS*, Bangalore, India, January 2010.

[2] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: diagnosing home network

misconfigurations using shared knowledge. In *Proceedings of NSDI*, Berkeley, CA, USA, 2009.

[3] Apple inc. Bonjour. http://www.apple.com/support/bonjour/.

[4] P. Bahl, R. Ch, A. Greenberg, S. K, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of SIGCOMM*, Kyoto, Japan, August 2007.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of OSDI*, San Francisco, CA, USA, December 2004.

[6] A. Binzenhöfer, K. Tutschku, B. auf dem Graben, M. Fiedler, and P. Arlos. A p2p-based framework for distributed network management. In *Proceedings of EuroNGI Workshop*, Villa Vigoni, Italy, July 2005.

[7] S. Brugnoni, G. Bruno, R. Manione, E. Montariolo, E. Paschetta, and L. Sisto. An expert system for real time fault diagnosis of the italian telecommunications network. In *Proceedings of IFIP/IEEE IM*, Amsterdam, The Netherlands, The Netherlands, 1993.

[8] R. Ch, V. N. Padmanabhan, and M. Zhang. Wifiprofiler: Cooperative diagnosis in wireless lans. In *Proceedings of MobiSys*, June 2006.

[9] X. Chen, M. Zhang, Z. Morley, and M. P. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of OSDI*, San Diego, CA, USA, December 2008.

[10] D. R. Choffnes, F. E. Bustamante, and Z. Ge. Crowdsourcing service-level network event monitoring. In *Proceedings of ACM SIGCOMM*, New Delhi, India, September 2010.

[11] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of OSDI*, Berkeley, CA, USA, 2004.

[12] David Brodsky. Azdht. http://trekie.sinister.cz/tairent/README.azdht.html.

[13] M. Dischinger, M. Marcon, S. Guha, P. K. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling end users to detect traffic differentiation. In *Proceedings of NSDI*, San Jose, CA, USA, April 2010.

[14] HP technologies inc. Openview. http://www.openview.hp.com.

[15] D. John, P. Prakash, R. R. Kompella, and R. Chandra. Shedding light on enterprise network failures using spotlight. *Reliable Distributed Systems, IEEE Symposium on*, pages 167–176, 2010.

[16] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of SIGCOMM*, Barcelona, Spain, August 2009.

[17] G. Khanna, M. Yu Cheng, P. Varadharajan, S. Bagchi, M. P. Correia, and P. J. Veríssimo. Automated rule-based diagnosis through a distributed monitor system. *IEEE Trans. Dependable Secur. Comput.*, pages 266–279, October 2007.

[18] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level internet path diagnosis. In *Proceedings of ACM SOSP*, New York, NY, USA, October 2003.

[19] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS*.

[20] Y. min Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, and C. Yuan. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of USENIX LISA*, San Diego, CA, USA, October 2003.

[21] OSGi Alliance. Osgi. http://www.osgi.org/Main/HomePage.

[22] V. N. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye. A study of end-to-end web access failures. In *Proceedings of CoNEXT*, New York, NY, USA, 2006.

[23] L. Popa, B.-G. Chun, I. Stoica, J. Chandrashekar, and N. Taft. Macroscope: End-point approach to networked application dependency discovery. In *Proceedings of CoNEXT*, Rome, Italy, December 2009.

[24] Sandia National Laboratories. Jess. http://www.jessrules.com/.

[25] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *Proceedings of SOSP*.

[26] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *Proceedings of USENIX*, San Diego, CA, USA, October 2009.

[27] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of OSDI*, San Francisco, CA, USA, December 2004.

[28] Y. Zhang, Z. M. Mao, and M. Zhang. Effective diagnosis of routing disruptions from end systems. In *Proceedings of NSDI*, San Francisco, CA, USA, April 2008.