Argumentation-based Fault Diagnosis for Home Networks

Changyu Dong
Department of Computing
Imperial College London
changyu.dong@imperial.ac.uk

Naranker Dulay
Department of Computing
Imperial College London
n.dulay@imperial.ac.uk

ABSTRACT

Home networks are a fast growing market but managing them is a difficult task, and diagnosing faults is even more challenging. Current fault management tools provide comprehensive information about the network and the devices but it is left to the user to interpret and reason about the data and experiment in order to find the cause of a problem. Home users may not have motivation or time to learn the required skills. Furthermore current tools adopt a closed approach which hardcodes a knowledge base, making them hard to update and extend. This paper proposes an open fault management framework for home networks, whose goal is to simplify network troubleshooting for non-expert users. The framework is based on assumption-based argumentation that is an AI technique for knowledge representation and reasoning. With the underlying argumentation theory, we can easily capture and model the diagnosis procedures of network administrators. The framework is rule-based and extensible, allowing new rules to be added into the knowledge base and diagnostic strategies to be updated on the fly.The framework can also utilise external knowledge and make distributed diagnosis.

Categories and Subject Descriptors

C2.3 [**Computer-Communication Networks**]: Network Management

General Terms

Design, Management

Keywords

Fault Diagnosis, Argumentation, Home Networks

1. INTRODUCTION

As of mid 2009, about 180 million of households worldwide had home networks [16]. By 2030, this number could reach 1 billion [12]. The use of in-home wired and wireless networking allows multiple computers to share connection to the Internet. However, many user studies have shown that managing a home network is a necessary and difficult task for most of households [13, 7, 9]. Among its subtasks, fault management is perhaps the most challenging for users.

Compared to enterprise networks, home networks are simpler: they are small (an average of six devices per network as estimated by TDG [12]) and their topologies are often not complex (one hop). However, unlike enterprise networks which are managed by skilled network administrators, home networks are managed by non-expert users who have no formal training on networking and may not have the motivation or time to learn how to diagnose and repair networking problems. For many, rebooting or contacting a third party is the only solution they understand. In addition, most fault management tools are designed for people with advanced knowledge of networking. They provide comprehensive information about the network and the devices, but leave it to the user to interpret and reason about the data to find the cause of a problem. Home users are unlikely to be able to understand the output from such tools, nor do they have the necessary background knowledge to reach the correct conclusions.

Clearly fault management in home networks must take into account the fact that the users have little technical knowledges. Therefore fault management tools for home networks must be easy to use and easy to understand: rather than just presenting networking information, tools must be able to analyse the data and form decisions based on their own knowledge. Ideally, the process should be automatic with minimum user involvement [19]. Although there are several tools targeted for home network management, their fault diagnosis capabilities are poor. As an experiment, we collected 11 home networking faults reported by our colleagues and users in internet forums. They are real faults which have been experienced by one or more users. We reproduced those faults and tested 3 tools which are available: the built-in Windows Network Diagnostics tool on Windows 7, Network Magic Pro 5.5 from Cisco Systems [3] and HomeNet Manager from SingleClick Systems [1]. The results of the tests are shown in Table 1. In the table, ✓ means the tool could identify the fault correctly and × means the tool could not identify the fault or could not detect any fault. From the table, we can see that none of the tools could correctly identify all the faults. In fact, the best tool could only identify less than half of the faults (5 out of 11). Many faults were either not detected or misdiagnosed.

HomeNets'11, August 15, 2011, Toronto, Ontario, Canada.

Fault	Diagnosis		
	Windows Diagnostics Tool	Network Magic pro	HomeNet Manager
Network cable is disconnected	✓	✓	✓
Network adapter is disabled	✓	✓	✓
Conflicting IP address	×	×	×
Default gateway address is not configured	×	Cable is not connected	There is a problem with routing table
The Internet connection to the ISP is broken	✓	✓	✓
DNS address is wrong	✓	×	A problem in the service provider's network
DNS address is not configured	×	×	A problem in the service provider's network
The address pool of the DHCP server is exhausted	Adapter doesn't have a valid IP configuration	Cable is not connected	A problem with routing table
An IPSec policy is enabled which does not allow unsecured communication	✓	A problem with the home router	A problem with the home router
Domain name resolution error due to a modified hosts file	The remote device or resource won't accept the connection	×	×
Firefox is configured to use a proxy server which is not accessible at home	×	×	×

Table 1: Faults Diagnosis Result

The problem is inherent in their design. The accuracy of the diagnoses depends largely on the quality and completeness of the tools' knowledge bases. This requires the knowledge bases to be updatable and extensible in order to accommodate new diagnostic strategies and new faults which are not considered initially. However, existing tools adopt a closed framework which hardcodes the knowledge bases thus make them hard to modify and extend.

In this paper, we describe an open fault management framework for home networks. The goal of the framework is to simplify network troubleshooting for non-expert users. The framework is based on assumption-based argumentation which is a technique for knowledge representation and reasoning. The framework is rule-based and extensible, allowing new rules to be added into the knowledge base and diagnostic strategies to be modified on the fly. The framework can also utilise external knowledge and support distributed diagnosis.

2. RELATED WORK

A comprehensive survey of network fault diagnosis is presented in [17]. However, most existing systems target for enterprise networks and require users to have in-depth knowledge of networking technologies. As home networks are usually managed in-house by non-expert users, current systems are unwieldy to be used.

A recent user study [19] suggests that network diagnosis tools for home network have to be easy-to-understand and easy-to-use. To this end, several research activities have been carried out. HomeMaestro [14] was designed to diagnose performance related faults. The system monitors flows in a home network and detects performance issues by using time-series and cross-correlation analysis. It can also identify whether the problems are caused by contention for network resources. However, the system cannot be adopted or applied for diagnosing non-performance related faults. NetPrints [4] allows automated diagnosis of problems caused by misconfigurations. The system records and aggregates configurations of network devices from different households. The configurations are classified as good or bad and are correlated to network faults. If a client experiences a recognised fault in the future, the system can give suggestions on the client's configuration. HNDR [8] can log various data in a home network and later the data can be used in supporting fault diagnosis.

3. ARGUMENTATION-BASED FAULT DIAGNOSIS

3.1 Assumption-based Argumentation

Argumentation [10, 6, 5] is the theory about arguments. In our work, we use assumption-based argumentation [11] to formalise the fault diagnosis process and to represent background information regarding faults. In assumption-based argumentation, *arguments* are essentially backward deductions based on logic rules and supported by sets of *assumptions* or *facts*. An argument *attacks* another if the first supports the contrary of an assumption in the second. Computationally, the reasoning process in argumentation systems can be represented as building a dispute tree. It mimics the behaviour of human debate: one party (*proponent*) proposes and tries to defend his theory while another party (*opponent*) tries to attack the theory. More formally, in assumption-based argumentation we have the following definitions:

DEFINITION 1 ([11]). *Given a deductive system $(\mathcal{L}, \mathcal{R})$, with a language $\mathcal{L}$ and a set of inference rules $\mathcal{R}$, and a set of assumptions $\mathcal{A} \subseteq \mathcal{L}$, an* ***argument*** *for $c \in \mathcal{L}$, i.e. the conclusion or claim, supported by $\mathcal{S} \subseteq \mathcal{A}$ is a tree with nodes labelled by sentences in $\mathcal{L}$ or by the symbol τ which stands for an empty set of premises, such that:*

- *the root is labelled by c*

- *for every node N*
  - *if N is a leaf then N is labelled either by an assumption or by τ;*
  - *if N is not a leaf and l_N is the label of N, then there is an inference rule $l_N \leftarrow b_1, ...b_m$ ($m \geq 0$) and either $m = 0$ and the child of N is τ or $m > 0$ and N has m children, labelled by $b_1, ...b_m$ respectively*
- *S is the set of all assumptions labelling the leaves.*
- *an argument for claim c **supported** by a set of assumptions S is denoted by $S \vdash c$*
- *an argument $S_1 \vdash c_1$ **attacks** an argument $S_2 \vdash c_2$ if and only if the claim c_1 of the first argument is the contrary of one of the assumptions in S_2*
- *a set of arguments Arg_1 attacks a set of arguments Arg_2 if an argument in Arg_1 attacks an argument in Arg_2*
- *a set of arguments Arg **defends** an argument arg if Arg attacks all arguments that attack arg*

The benefit of using assumption-based argumentation as our theoretical foundation for the fault diagnosis framework is that it naturally models the fault diagnosis procedure of network administrators. To diagnose a fault, a network administrator first needs to propose what are the potential causes given the symptom. Then the potential causes are eliminated through a systematic checking process. For causes that are not eliminated, further investigation needs to be conducted to identify the root of the problem. This is exactly how the assumption-based argumentation system reasons about faults. The symptoms are captured as goals and the background knowledge is captured as inference rules. The possible causes are the assumptions. Arguments are formed regarding the symptom and the assumptions. Arguments may be attacked by other arguments and therefore eliminated. For arguments which successfully defend, further tests will be done to confirm the assumptions.

Let us illustrate it with a simple example. A possible symptom might be "cannot access the Internet" and two possible inference rules regarding this symptom might be "if the network media (cable and wireless) is disconnected, then the computer cannot access the Internet" and "if the connection to the ISP is broken, then the computer cannot access the Internet". In the fault diagnosis process, the system may first assume "the network media is disconnected" as the cause. The argument "the computer cannot access the Internet because the network media is disconnected" is a valid argument supported by the assumption. However, if there is another inference rule "if we can ping another local devices in the network, then the network media is not disconnected" and a ping test is successful, then another argument "the network media is not disconnected because can ping another neighbour in the network" can be formed which attacks the first one. Since the second argument is based on a fact and we cannot find other arguments to defend the first argument, the first argument is eliminated and the system needs to find a better theory to explain the symptom. An argument can then be formed based on the assumption "the connection to the ISP is broken". An argument which attacks the new argument can be "the connection to the ISP is not broken because trace route can reach the 2nd hop". This time the test fails and trace route gets no response from the 2nd hop, i.e. the ISP's router. Then the attack fails and the failed test also confirms the problem is with the connection to the ISP.

3.2 Fault Diagnosis Rules

In assumption-based argumentation, arguments are deductions using rules in an underlying logic language. The rules for diagnosing faults can be represented as logic programming rules [15] of the form:

$$L :- L_0, ..., L_n, (n \geq 0).$$

Here $L, L_0, ..., L_n$ are ground literals, i.e. atoms a or negation of atoms $\sim a$. L is the head of the rule and the conjunction $L_0, ..., L_n$ is the body of the rule. In fault diagnosis rules, we have 3 different types of literals:

- Symptom: A symptom literal represents an abnormality in the home network perceived by the user. For example, "cannotConnectInternet" or "cannotUseEmail". It serves as the goal of the reasoning and can only be used in the head of a rule.
- Assumption: Assumption literals can be used in the head and the body of rules. Given a rule $L : -L_0, ..., L_n, (n \geq 0)$, if L is a symptom and $L_m, 0 \leq m \leq n$ is an assumption, then the assumption L_m is a possible cause of the symptom. If L and L_m are both assumptions, then L_m is a specialisation of L. For example, L can be $physicalLayerFault$ and L_m can be $networkMediaDisconnected$.
- Test: A test literal represents a test function to be performed in order to confirm or eliminate an assumption. It can only be used in the body of a rule in which the head is an assumption literal. The truth value of a test literal is determined by the result of the linked test function.

The diagnosis rules in the example in Section 3.1 can be represented as the following:

$$\begin{aligned}&cannotConnect(X) :- physicalLayerFault(X).\\&cannotConnect(X) :- networkLayerFault(X).\end{aligned}$$

$$\begin{aligned}&physicalLayerFault(\_) :- networkMediaDisconnected().\\&{\sim}networkMediaDisconnected(\_) :- canPingNeighbour().\end{aligned}$$

$$\begin{aligned}&networkLayerFault(X) :- linkToISPBroken(X).\\&{\sim}linkToISPBroken(\_) :- canReachHop(2).\\&linkToISPBroken(\_) :- \ {\sim}canReachHop(2).\end{aligned}$$

3.3 Dependency Rule

Modern networks follow a layered model. On each layer, an instance provides services to the instances at the layer above and requests services from the layer below, which means a service at the higher layer is functionally dependent on the lower layer services. Therefore a fault that appears to be at a higher layer could be actually caused by one or more faults at a lower layer. For example, a failed DNS query could be the result of a disconnected cable.

When diagnosing network faults, we need to work in a bottom-up approach, i.e. start from the lowest layer of the protocol stack. In our framework, we use dependency rules to specify the order of the inference rules to be taken into the reasoning. A dependency rule is of the form $L_1 < L_2$, where L_1 and L_2 are different assumptions. Intuitively, $L_1 < L_2$ means L_2 is dependent on L_1, therefore L_2 should only be taken into account after L_1 has been ruled out. For example, when a computer cannot connect to the Internet, the fault could be at the physical layer or the data link layer, and we should always consider the physical layer fault first. The rules can be expressed as:

$$\begin{array}{l} cannotConnect(X) : -physicalLayerFault(X). \\ cannotConnect(X) : -dataLinkLayerFault(X). \\ \\ physicalLayerFault < dataLinkLayerFault. \end{array}$$

If $L_1 < L_2$ and in the reasoning process, an acceptable argument supported by L_1 can be derived, then the reasoning process will stop. By focusing on one layer and one problem each time, it also makes it easier for fixing multiple faults. If a fault has been detected and fixed but the symptom remains, then another iteration can start and other faults at a higher layer can be found subsequently.

3.4 Distributed Diagnosis

When a user observes a certain symptom on a device, the cause of the symptom can be local or external. To identify the fault, local knowledge may not enough. For example, many problems caused by network contention are hard to identify with only local knowledge. Sometimes, external knowledge may also help make the reasoning process more efficient.

We can obtain external knowledge from neighbours running our diagnosis framework. For distributed diagnosis, we attach the following labels to the test literals:

- $@external$: the test needs to be run on all the neighbours which are currently available. The local device does not need to run this test.
- $@all$: the test needs to be run on all neighbours and the local device.
- $all- >$: the truth value of the test literal is the conjunction of the test results from all the neighbours. The truth value will be true if all the neighbours return true, will be false otherwise.
- $one- >$: the truth value of the test literal is the disjunction of the test results from all the neighbours. The truth value will be false if all the neighbours return false, will be true otherwise.

For example, you have ruled out all the faults at lower layers and now you suspect you cannot connect to a website because the remote server is down. This assumption can easily be proved wrong if one of the devices in the home network can ping the remote server. These rules can be captured as the following:

$$\begin{array}{l} cannotConnect(X) : -remoteServerDown(X). \\ \sim remoteServerDown(X) : -\{@all, one- >\}ping(X). \end{array}$$

Another example is that you may suspect your network is slow because someone is downloading using Bittorrent. You can run a check across all neighbours to see whether they have a Bittorrent client running:

$$\begin{array}{l} networkSlow : -bittorrent. \\ \sim bittorrent : -\{@all, all- >\} \sim bittorrentClient. \end{array}$$

Another source of external knowledge which is more specific is from the information plane architecture [18] we have developed in the Homework project [2]. The information plane architecture provides network measurements in real-time, correlates these measurements and low level network events to generate high-level events which drive management and makes the measurement data persist for offline analysis. To utilise the information plane, we use the following label.

- $@gw$: the test needs to be run with the data from the information plane architecture. “gw” means gateway. This is because the current implementation of the architecture runs on a Homework router which is the gateway of the home network.

The information plane architecture can potentially provide a huge amount of information which is helpful in network fault diagnosis. For example, a common issue in home networks which can cause connectivity problems is the firewall. Most home routers have built-in firewalls. If certain rules have been enabled to block traffic, it is usually hard to find out the problem with only information collected on the device being blocked. However this would be relative easy to check in the information plane whether there has been traffic blocked events recently, e.g. within the last 1 minute, with matching source/destination addresses. The rules can be captured as in the following (in the rule the local address is implicit):

$$\begin{array}{l} cannotConnect(X) : -blockedByFirewall(X). \\ blockedByFirewall(X) : -\{@gw\}blockEvents(X, 60). \end{array}$$

3.5 Diagnosis Example

We have written rules for diagnosing all the faults in Table 1. The full set of rules can be found in the Appendix. Here we describe how the system diagnoses a connectivity problem.

The diagnosis system first builds a dispute tree as shown in Fig 1. The dispute tree is built automatically using the rules. In the dispute tree, white nodes are put forward by the proponent and the grey nodes are put forward by the opponent. The initial argument proposed by the proponent is that the problem is caused by a physical layer problem. This argument is chosen because $physicalLayerFault$ is at the lowest layer as specified by the dependency rules. The opponent then proposes an attacking argument that the problem is not at the physical layer. The argument is supported by the fact that the error code gathered from the config manager suggests that each network card works properly. Being attacked and there is no counter-attack, the proponent's first argument is not acceptable. The proponent then proposes the second argument that the problem is caused by a data link layer problem. This argument is also attacked by the opponent and not acceptable.

The third argument by the proponent is that the problem is caused by a network layer problem. The opponent tries to form an attacking argument. However, the ping test fails which means the attacking argument is not supported by ground fact. Since the opponent's argument is not able to attack the proponent's argument, the proponent can go on develop his argument. There are several more specific rules about network layer faults. The proponent first tries to extend his argument by assuming the fault is caused by conflict IP addresses. Although the opponent cannot propose an attacking argument for this assumption, this assumption is

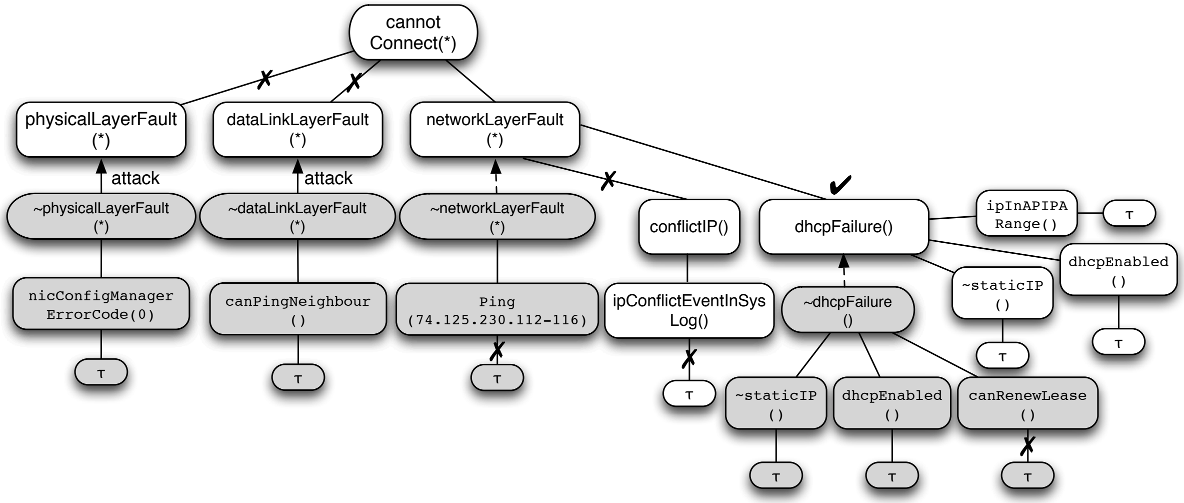

Figure 1: The Dispute Tree of a Fault Diagnosis Example

not supported by the test result. So the proponent chooses another assumption that the fault is because of a failure on the DHCP server. The opponent can propose an attacking argument, but the following test shows that the computer cannot renew DHCP lease. Therefore the opponent's attacking argument is not supported. The proponent goes on to test his assumption. This time all tests confirm his assumption. In this case, the system makes the decision that the problem is very likely to be caused by a DHCP failure.

4. IMPLEMENTATION

We have developed a prototype of the argumentation-based fault diagnosis tool. The prototype has a simple user interface. The user selects the symptom he has observed in his home network, and when necessary, a domain name or IP address of the external website/service he was accessing. Currently we focus on connectivity issues, but we are also extending to performance and other problem domains.

The prototype has two main components: the rule engine and the test functions.

The rule engine is written in Java. It takes as input a set of rules and reasons about the faults according to the rules. The rules are written using the syntax we have shown in Section 3. The rules are parsed and fed into an interpreter. The interpreter then starts building arguments from the symptom input by the user using a backward chaining algorithm. When a local test is needed as part of the reasoning process, it invokes the corresponding test functions. For remote tests, it can communicate with neighbours and the information plane residing on the homework router with UDP.

The test functions are platform dependant. They are scripts written in script languages, therefore can be created or modified easily. For example, on Windows platforms, the test functions are implemented as batch files or Windows Management Instrumentation (WMI) scripts. The file name of a test function must correspond to one used in the diagnosis rules. When the rule engine encounters a leave node in the argument trees, it will try to invoke a test function using the name of the atom.

5. CONCLUSION AND FUTURE WORK

We have presented an argumentation-based fault diagnosis framework. It represents knowledge as logic rules and the reasoning process is very similar to what network administrators do when diagnosing faults. The rule-based approach also makes it highly extensible. Rules can be modified or added without needing to rebuild the system. In addition, the framework supports distributed fault diagnosis when external knowledge is available.

A critical question all rule-based systems, including ours, need to solve is how to break the knowledge bottleneck. To be useful, the system needs high quality rules which cover all faults or those which are most likely to be encountered in a home network. This requires significant amount of knowledge regarding network devices, operating systems and applications. We hope to tackle the problem in a community-based approach by leveraging shared information across a large user population. Rules can be submitted and reviewed by community experts. This development model has been proven to be more responsive and efficiently than a "closed" model. Several successful rule-based open source systems, e.g. Snort intrusion-detection system, are developed using this model. Since the rules in our system are loosely coupled, a rule writer doesn't need comprehensive knowledge about the whole rule set. He just needs to construct arguments for and against certain assumptions. This makes it easier for people to add and modify rules.

Currently the system supports only simple collaborative tasks, we would like to investigate what additional functionalities are needed for distributed diagnosis and to improve it accordingly in the future.

Acknowledgement

This research was supported by the UK's EPSRC research grant EP/F064446/1.

6. REFERENCES

[1] Homenet manager.
http://www.softpedia.com/get/Network-Tools/Network-Tools-Suites/HomeNet-Manager.shtml.

[2] The homework project. http://www.homenetworks.ac.uk/.

[3] Net work magic pro. http://www.purenetworks.com/product/pro.php.

[4] B. Agarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, pages 349–364, 2009.

[5] L. Amgoud and H. Prade. Using arguments for making and explaining decisions. *Artif. Intell.*, 173(3-4):413–436, 2009.

[6] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artif. Intell.*, 93:63–101, 1997.

[7] A. Brush. IT@home: Often best left to professionals. In *CHI 2006 Workshop: IT@Home*, 2006.

[8] K. L. Calvert, W. K. Edwards, N. Feamster, R. E. Grinter, Y. Deng, and X. Zhou. Instrumenting home networks. In *Proceedings of the 2010 ACM SIGCOMM workshop on Home networks*, HomeNets '10, pages 55–60, New York, NY, USA, 2010. ACM.

[9] M. Chetty, J.-Y. Sung, and R. E. Grinter. How smart homes learn: The evolution of the networked home and household. In J. Krumm, G. D. Abowd, A. Seneviratne, and T. Strang, editors, *Ubicomp*, volume 4717 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2007.

[10] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming. In *IJCAI*, pages 852–859, 1993.

[11] P. M. Dung, R. A. Kowalski, and F. Toni. Assumption-based argumentation. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 199–218. Springer, 2009.

[12] P. Filipovic. The future of home networks - a global perspective, 2008.

[13] R. E. Grinter, W. K. Edwards, and M. W. Newman. The work to make a home network work. In H. Gellersen, K. Schmidt, M. Beaudouin-Lafon, and W. E. Mackay, editors, *ECSCW*, pages 469–488. Springer, 2005.

[14] T. Karagiannis, C. Gkantsidis, P. Key, E. Athanasopoulos, and E. Raftopoulos. Homemaestro: Distributed monitoring and diagnosis of performance anomalies in home networks. Technical Report MSR-TR-2008-161, Microsoft Reasearch, 2008.

[15] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[16] K. Scherf. Home networks for consumer electronics, 2009.

[17] M. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Sci. Comput. Program.*, 53(2):165–194, 2004.

[18] J. Sventek, A. Koliousis, O. Sharma, N. Dulay, D. Pediaditakis, M. Sloman, T. Rodden, T. Lodge, B. Bedwell, K. Glover, and R. Mortier. An information plane architecture supporting home network management. In *The proceddings of 12th IFIP/IEEE International Symposium on Integrated Network Management*, 2011.

[19] J. Yang and W. K. Edwards. A study on network management tools of householders. In *Proceedings of the 2010 ACM SIGCOMM workshop on Home networks*, HomeNets '10, pages 1–6, New York, NY, USA, 2010. ACM.

Appendix

```
% set dependency
physicalLayerFault<dataLinkLayerFault.
dataLinkLayerFault<networkLayerFault.
networkLayerFault<transportLayerFault.
transportLayerFault<applicationLayerFault.
%%%% physical layer problems %%%%%
cannotConnect(X):-physicalLayerFault(X).
% config manager error code 0 means work properly
~physicalLayerFault(_):-nicConfigManagerErrorCode(0).
physicalLayerFault(_):-networkMediaDisconnected().
~networkMediaDisconnected(_):-canPingNeighbour().
networkMediaDisconnected(_):-
            allEnabledAdapterStatus(disconnect).
physicalLayerFault(_):-networkAdapterDisabled().
~networkAdapterDisabled(_):-canPingNeighbour().
networkAdapterDisabled(_):-noAdapterEnabled().
%%%% datalink layer problems %%%%%
% currently empty
cannotConnect(X):-dataLinkLayerFault(X).
~dataLinkLayerFault(_):-canPingNeighbour().
%%%% network layer problems %%%%%
cannotConnect(X):-networkLayerFault(X).
% 74.125.230.112-116: addresses of www.google.com
~networkLayerFault(_):-ping(74.125.230.112-116).
networkLayerFault(_):-conflictIP().
conflictIP():-ipConflictEventInSysLog().
networkLayerFault(_):-dhcpFailure().
~dhcpFailure():-~staticIP(),dhcpEnabled(),
         canRenewLease().
% APIPA address: 169.254.0.0/16, self assigned
% when DHCP server is not available
dhcpFailure():-~staticIP(),dhcpEnabled(),
       ipInAPIPARange().
networkLayerFault(_):-noDefaultGateWay().
noDefaultGateway():-noDefaultRoute().
networkLayerFault(_):-linkToISPBroken(X).
~linkToISPBroken():-canReachHop(2).
linkToISPBroken():- ~canReachHop(2).
networkLayerFault(_):-noDNS().
noDNS():-dnsNotConfigured().
networkLayerFault(_):-ipSec().
% require security means not to communicate
% with non-IPSec nodes
ipSec():-policyAssigned(),requireSecurity().
networkLayerFault(_):-wrongDNSAddress().
wrongDNSAddress():-nslookupFail().
%%%% transport layer problems %%%%%
% currently empty
cannotConnect(X):-transportLayerFault(X).
%%%% Application layer problems %%%%%
cannotConnect(X):-applicationLayerFault(X).
applicationLayerFault(X):-modifiedHostFile(X).
~modifiedHostFile(X):-~isIPAdress(X).
~modifiedHostFile(X):-~inHostFile(X,_).
modifiedHostFile(X):-nsLookup(X,Y),
         ~inHostFile(X,Y).
applicationLayerFault(X):-deadProxy(X).
~deadProxy(_):-~proxyConfigured(_).
deadProxy(_):-proxyConfigured(X),~ping(X).
```