An Architecture for Inter-Domain Troubleshooting *

David G. Thaler and Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
The University of Michigan, Ann Arbor, Michigan 48109-2122
thalerd@eecs.umich.edu ravi@eecs.umich.edu

Abstract

In this paper, we explore the constraints of a new problem: that of coordinating network troubleshooting among peer administrative domains and untrusted observers. Allowing untrusted observers permits any entity to report problems, whether it is a Network Operations Center (NOC), end-user, or application. Our goals are to define the inter-domain coordination problem clearly, and to develop an architecture which allows observers to report problems and receive timely feedback, regardless of their own locations and identities. By automating this process, we also relieve human bottlenecks at help desks and NOCs whenever possible.

We present a troubleshooting methodology for coordinating problem diagnosis, and describe GDT, a distributed protocol which realizes this methodology.

1 Introduction

Work to date in network management has concentrated on effectively managing a single network. To our knowledge, little has been done to address the problem of coordinated network management across administrative domains, although the need for such a global coordination system has long been recognized [1, 2, 3, 4].

A recent study of routing instability [5] found that in about 10% of the problems, all parties questioned pointed to another party as the cause. Such problems strongly underscore the need for inter-domain coordination. The true cause of a problem may be distant from its effect. For example, failure to access a web page may result from a problem anywhere between the browser and the remote server. One's local help desk can not help in this case.

In this paper, we investigate issues in such inter-domain troubleshooting coordination, and address the subproblem of developing a communication protocol for detecting and reporting problems across administrative domains. We aim to provide timely feedback to (dis)affected end-users, and to relieve human bottlenecks at help desks and Network Operations Centers (NOCs) whenever possible. We leave the problems of inter-administration negotiation of repairs, and of notifying organizations of scheduled future downtime to future work. Work such as IPN [6] addresses the issue of pre-notification, but pre-notification does not help coordinate troubleshooting, since past announcements may have been lost, ignored, forgotten, and may be inaccessible during the problem.

Intra-administration management methods may not apply to the inter-administration case. For example, management entities may not be allowed to access or trust observers across administrative domains. New mechanisms are thus needed. We build inter-administration coordination on top of the management functionality existing within each administrative domain. We therefore need only concern ourselves with the task of coordinating information between domains in an effective manner.

Our goal in this paper is to clearly define the inter-domain coordination problem, and to provide a framework and protocol which allows any entity (including a user, a NOC, or an application) to report problems and receive appropriate feedback, regardless of its own location.

Our framework consists of three parts:

1. **Domain-expertise modules:** These are existing tools (such as Network Management Systems) upon which we build. They apply traditional management techniques, usually within an administrative domain.
2. **Troubleshooting Methodology:** This is the theory and algorithm behind the protocol which allows effective troubleshooting in an inter-domain environment.
3. **Coordination Protocol:** This protocol conveys information between management entities which may be in different administrative domains, and enables the methodology.

The remainder of this paper is organized as follows. Section 2 outlines our design philosophy and describes the constraints relevant to the inter-domain problem.

*This work was supported in part by National Science Foundation Grant NCR-9417032.

1095-2055/97 $10.00 © 1997 IEEE

Section 3 presents a methodology for diagnosing problems. Section 4 describes our protocol, and Section 5 covers conclusions and the future.

2 Design Philosophy

We now discuss a number of design principles and provide the motivation for the application of our methodology.

2.1 Policy principles

We present the following principles as relevant in the context of an inter-domain environment.

Principle 1 (Freedom of information): *"Outage" information should be available to those affected by it.*

Users and applications can often benefit from information such as the expected downtime, especially for problems spanning administrative domains.

Principle 2 (Privacy): *"Outage" information should be available* **only** *to those affected by it.*

Internet Service Providers (ISP's) typically do not want statistics on the number of problems observed in their network to be publicly available. Hence, information on problems is only distributed on a "need-to-know" basis.

Principle 3 (Freedom of speech): *Any entity should be able to report a problem, whether or not it is trusted.*

This principle is a consequence of the Freedom of Information principle and the fact that the cause of a problem may be distant from its effect. No assumption is made initially about the correctness (or the non-maliciousness) of the problem report.

Principle 4 (Conservation of effort): *One should perform the minimum repairs required to fix the problem in a timely fashion. In addition, no attempts should be made to repair non-existent problems.*

A repair should be performed close to the source of the problem, to avoid reacting to each effect separately. Also, problems reported by untrusted sources must be confirmed before being acted upon.

2.2 Architectural Constraints

Our architecture follows the Internet design philosophy described in [7], which we summarize with the following set of constraints ranked in order of importance: high availability, allowing multiple services, networks, and centers of administration, cost-effectiveness, low-effort deployment, and accountability.

We paraphrase our foremost constraint (from Rose [8]) as follows:

Constraint 1 (Reliability): *When all else fails, troubleshooting must continue to function, if at all possible.*

This constraint implies that a global troubleshooting system must require as few other services as possible to be functional. For example, it should continue to function (although not necessarily as well) if nameservice, filesystems, or TCP are not available.

Constraint 2 (Scalability): *The architecture must be scalable to span the global Internet.*

Many problems may exist simultaneously, and the network may span many autonomous administrations. The network configuration may also change over time.

Constraint 3 (Low-cost deployment): *The architecture must be both simple to implement and deploy, as well as consume resources at a reasonably low rate.*

The requirements for entities to participate in troubleshooting must be as simple as possible, and the bandwidth and memory costs required must not outweigh the benefits of a troubleshooting architecture.

Constraint 4 (Security): *The architecture must be secure and adhere to the Privacy principle.*

It must not publish information on current problems to those unaffected by them, and must prevent unnecessary "repairs" from being performed.

3 Troubleshooting Methodology

We use the term *object* to denote a logical entity in the network, such as a router or a stream of data. We will refer to *classes* of objects, and *instances* of specific classes. For example, a "TCP session" may be a class, while an instance of that class would be identified by a pair of IP addresses and port numbers.

3.1 Fault-propagation model

Faults may propagate both vertically as well as horizontally through the network [9]. For the horizontal direction, we use the term *downstream* to denote the direction of data flow, and *upstream* to denote the reverse direction. In the vertical direction, *up* and *down* are defined with respect to the seven-layer protocol stack defined by the ISO [10].

Our method uses *resource dependency graphs*, in which each node represents an object, and directed edges denote dependencies. The arrows show the direction of a *demand* for the resources of one object by another. In Figure 1 the audio client and server applications demand resources from the host CPU and the filesystem, and as the router forwards packets, it imposes a resource demand upon the downstream link. The efficiency and correctness of an object thus depend on whether its resource remands are met, and hence on the efficiency and correctness of those objects below and downstream of it.

Relationships between *classes* of objects are typically static, whereas relationships between specific *instances* of objects are often dynamic. For example,

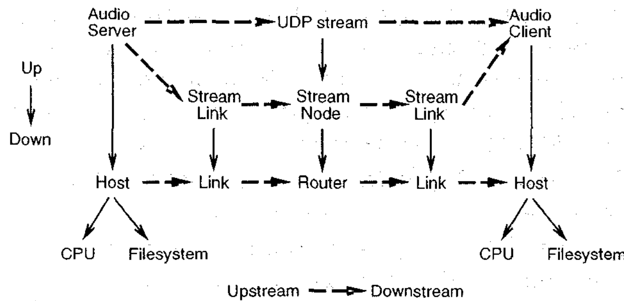

Figure 1: Sample Resource Dependency Graph

a UDP stream will always depend upon links and routers, but the specific instances may change over time. Our goal will be to statically know static relationships, and to dynamically resolve dynamic relationships.

Let an object's *capacity* denote the total amount of resources it makes available. For example, a link's capacity might be measured in Mbps, and a file system's capacity might be measured in Gbytes. Let an object's *utilization* denote the total amount of its resources in use. We adopt the concept of a "health function" from [11]. We let an object's *health* be a measure of its performance and its ability to adequately meet imposed demands. Low health is thus an indication of degraded performance. We will use the term *problem* to denote an object experiencing low health. The precise meaning of this depends on the definition of the health function, and may be different for each class of objects. For example, the health of a TCP session may be measured by latency (possibly in addition to other factors), while the health of a filesystem might be measured by average read and write access times. This definition allows us to coordinate information relating to both fault management and performance management, as defined by the ISO [10].

With the above definitions, we are ready to analyze fault propagation in more detail. We begin with the following observations:

Observation 1 *High utilization propagates in the direction of resource dependencies.*

Any object which is highly utilized may consequently impose higher demands on those objects on which it depends.

Observation 2 *Low health propagates in the direction opposite to resource dependencies.*

Degraded performance at some object may degrade the performance of all objects depending on it.

Observation 3 *High utilization can cause low health, as utilization approaches the object's capacity.*

Low health may arise from soft failures (congestion) or hard failures (hardware or software faults).

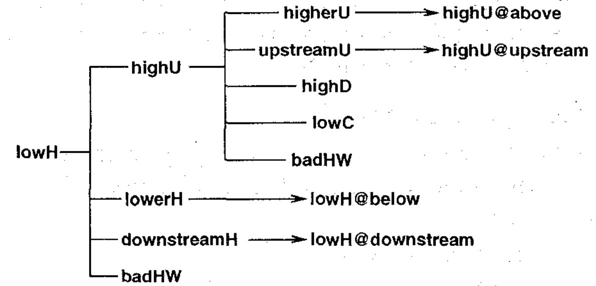

Figure 2: Problem Taxonomy

3.2 Cause-effect graphs

Previous studies (e.g., [12]) have typically only looked at one direction of fault propagation (i.e., "up"). We introduce cause-effect graphs as a more comprehensive model for representing fault propagation. Each node in such a graph represents a problem, and directed edges lead from effects to causes. We begin with a taxonomy of problem types (Figure 2) based on our discussions in Section 3.1.

Degraded performance (`lowH`) of an object can be caused by congestion (`highU`, Observation 3), by degraded performance at a lower or downstream object (`lowerH`, `downstreamH`, Observation 2), or by an actual hardware or software problem (`badHW`) with the object itself. Similarly, congestion (`highU`) can be caused by high utilization above or upstream (`higherU`, `upstreamU`, Observation 1), by the object itself generating an unusually high demand (`highD`), by the object having insufficient capacity to meet normal demands (`lowC`), or by an actual hardware or software problem (`badHW`) with the object itself.

Each of `lowerH`, `higherU`, `downstreamH`, and `upstreamU` refer to specific problems at objects below, above, downstream, and upstream from the affected object, respectively. The taxonomy thus represents a recursive method to trace problems back to one or more *root* causes (i.e., those which are not effects of other problems). A root cause can only be one of `highD`, `lowC`, or `badHW`.

Figure 3 shows a sample network topology. In this topology, nodes A, B, E, and F are connected via 10 Mbps full duplex links to nodes C and D which connect to each other via a 500 Kbps full duplex link. Using the taxonomy we described above, we can now construct directed cause-effect graphs, where each node represents a problem, and directed edges lead from an effect to a cause (see Figure 4). The dotted lines in this figure simply group all problems with the same object, since the taxonomy refers to problem types at a given object. (Note that, as we will see in Section 4, this graph will be distributed in practice to provide scalability and support privacy.)

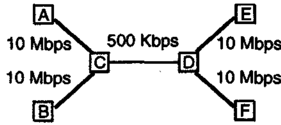

Figure 3: Sample Topology

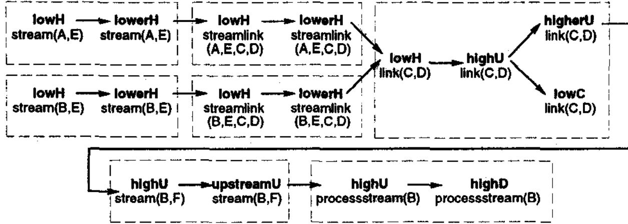

Figure 4: Cause-Effect Graph of Problems

We call a symptom from which cause-effect graph construction begins a "leaf effect" (such as **lowH** of stream(A,E)), since diagnosis always proceeds from an effect to its cause. A problem such as **highD** of processstream(B) which is not an effect of any other problem is a "root cause." If problems occur often enough that the capacity is insufficient to support normal demand, **lowC** will be another root cause, as shown. Since each cause may have multiple effects, and vice versa, the superposition of the trees constructed by tracing back from each leaf effect forms the complete cause-effect graph.

The problem types shown in Figure 2 are necessary and sufficient for inter-domain coordination since they enumerate and distinguish the different directions in which fault and performance problems can be propagated. Problems in real networks will simply be instances of these types, and further subdivisions will be specific to each class of objects. Since we are interested in efficient coordination of troubleshooting efforts, rather than the details of the efforts themselves, the high-level classifications will suffice for our purposes.

In addition, we observe that since problems can only be propagated across resource dependencies, cycles can occur in cause-effect graphs only if cycles are present in the resource dependency graph. Cycles in cause-effect graphs are particularly important. They may lead all administrations involved to conclude it is "somebody else's problem", as observed in the informal routing instability study [5], resulting in no action taken at all. We will return to this issue in Section 4.4.

3.2.1 Constructing cause-effect graphs

Given an initial problem (leaf effect) report, a cause-effect graph can be constructed according to the following procedure: (1) Run a test to confirm whether the problem exists (this is done by a domain-expertise module). If none exists, stop. Note that this step is necessary when the origin of the report is either untrusted or unsure. In cases where the origin is both trusted and sure of the problem's existence, this step can be omitted. (2) Generate hypotheses about possible causes by referring to the Problem Taxonomy (Figure 2) and the Resource Dependency Graph (Figure 1) which includes the affected object. (3) Repeat from step 1 for each new hypothesis generated.

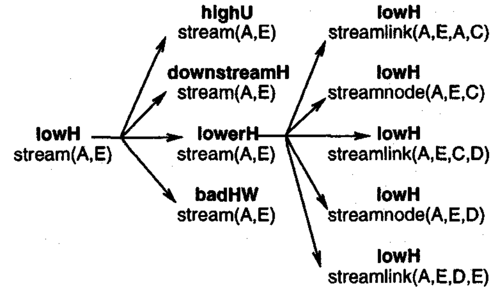

Figure 5: Generating Hypotheses

Figure 5 gives an example, starting with a low-health report for a stream object. If the report is confirmed, hypotheses of **highU**, **lowerH**, and **badHW** are generated in accordance with Figure 2. If **lowerH** is confirmed, a **lowH** hypothesis will be generated for each object below the stream object in the resource dependency graph. We prune back branches for all hypotheses which are rejected by tests or whose tests were indeterminate, leaving only confirmed problems.

This process continues until all branches are pruned back or reach either previously-confirmed problems or root causes. Figure 4 shows the results of applying this process to two leaf effects: low health of stream (A,E) and stream (B,E). These effects are traced to a common cause: low health of the C-D link, which is in turn a result of B generating too much traffic.

4 Problem report coordination

In this section, we describe how the methodology outlined in Section 3 can be applied in a scalable manner to a distributed architecture which meets the requirements in Section 2.

To scale to a global network composed of a large number of administrative domains, we propose a society of troubleshooting coordination agents which we call *experts*. Experts communicate with each other and with *clients*, which are agents acting on behalf of the end-user, application, or NOC observing a problem. Each expert has one or more *areas of expertise*. An area of expertise is defined as knowledge about problems with a specific class of objects, and the capabil-

ities and permissions necessary to diagnose a specific set (e.g., a range or list) of instances of that class.

For each class of objects in its areas of expertise, an expert must have the ability to determine the set of objects immediately above, below, upstream, and downstream from a given object, and the ability to test for (at least) `lowH`, `highU`, `highD`, and `lowC`. The results of each test should either confirm or deny the existence of the problem, or report that the test was indeterminate. Any test which is indeterminate is later considered to be confirmed if an immediate effect was confirmed, and all other potential hypotheses are rejected. For example, `badHW` is frequently difficult to test because of the many ways in which hardware and software may be faulty. If no test is available, then `badHW` is considered to be confirmed if `lowH` was confirmed but `highU`, `lowerH`, and `downstreamH` were rejected.

Intermediate problem types (`lowerH`, `downstreamH`, `upstreamU`, and `higherU`) are considered confirmed if any hypothesis they generate is confirmed, rejected if all hypotheses they generate are rejected (or if no hypotheses are generated), and are indeterminate otherwise. Note that the same strategy could be used for any other indeterminate problem by testing the hypotheses it would have generated if confirmed.

The cause-effect graph spans the society of experts, with all nodes in the graph for the same object being located at the same expert. The resource dependency graph is likewise distributed, since each expert knows the static relationships between classes in its areas of expertise, and is able to dynamically determine which objects relate to any given problematic object within its areas of expertise. Finally, the same taxonomy is used by all experts in generating hypotheses.

Each expert keeps a list of unresolved problems within its areas of expertise which have been reported to it. Each expert then locally follows the methodology of Section 3 for the nodes of the global cause-effect graph which it holds, and GDT protocol messages are exchanged between experts to create and maintain the distributed cause-effect graph.

We reiterate that each expert only keeps information on its own problems, and only receives information about problems which directly affect it or its problems. This provides scalability as well as meets the privacy requirement from Section 2.

4.1 Expert Location

The following considerations are important in designing a scalable mechanism for locating appropriate experts to which to submit problem reports. First, an expert location service will be used precisely when problems exist. The Reliability constraint (see Section 2.2) thus mandates that the expert location service should *not* make use of any existing system for service location. For example, it should not *require* multicast, or else it cannot diagnose problems with multicast routing. Therefore, we must construct an expert location service tailored specifically to our needs. Note that we do not preclude the use of multicast as an optimization when it is available. We simply present a method below which is able to function without the use of multicast. An analysis of optimizations which introduce such additional dependencies is a topic for future work.

In our scheme, object names are attribute-based and correspond to individual points in the namespace. The name of an object consists of two sets of *attribute=value* pairs: a mandatory set which uniquely identifies the instance, and an optional set to provide additional information. For example, the name of a specific UDP stream might be "`class=UDPstream, sourceAddr=141.213.10.41, sourcePort=1234, destAddr=204.140.133.4, destPort=5678, application=vat`", where `application` is an optional attribute. To report a problem with a specific object, all required attributes must be specified.

Areas of expertise, on the other hand, correspond to regions in the namespace. The description of a region contains *attribute=set* or *attribute=range* pairs, and need not specify required attributes. For example, one area of expertise might be "`capability=diagnosisOnly, class=UDPstream, sourceAddr=141.213/16`". To submit a hypothesis, one must be able to map the name of a problematic object to one or more experts whose areas of expertise include the given object. This problem is analogous to that of performing a *point query* in a spatial database to get a list of regions covering the given point.

Traditional spatial database techniques such as R-trees [13] are not directly applicable, however, since scalability requires that the database of regions be physically distributed. In addition, it doesn't matter whether a region is matched if the associated expert isn't reachable. Thus, there are fundamental differences imposed by our constraints which make traditional approaches less applicable.

We summarize our design requirements for nameservice below, in order of importance:

1. Allow availability during network partitions (i.e., locate reachable experts).
2. Minimize point query time by minimizing the number of exchanges of network messages.
3. Maintain low bandwidth and memory overhead (thus trying not to exacerbate congestion problems, and interfering as little as possible with other objects).

The first constraint suggests that a hierarchy of servers corresponding to a hierarchy in the namespace (as is used by DNS [14], X.500 [15], etc) will not work, since we must have successful queries even when we are partitioned from a large part of the network. Replicating such servers everywhere will not keep the bandwidth overhead low. We also want to avoid mandating a hierarchical namespace to preserve domain autonomy and class independence. On the other hand, we desire some structure to the servers so that expert location can provide higher availability, and be easily adapted to changing conditions without manual reconfiguration. Many existing attribute-based naming schemes (e.g., [16]) provide no structure to servers and hence rely on manual configuration.

The solution we adopt is as follows. Expert location servers (ELS's) are organized into a hierarchy according to their *location*. Informally, each ELS is responsible for knowing the namespace regions (areas of expertise) in the subtree rooted at itself. To avoid manual configuration, such a hierarchy may be formed by a self-configuring process such as TDH [17].

Experts form the actual leaves of the tree so formed, with each expert's parent being the closest expert location server. We will refer to a server whose children are experts (as opposed to other servers) as a *leaf server*. Each expert periodically advertises its areas of expertise to its parent. Each server then reports to its own parent, either the bounding box covering the regions it has, with its own address as the "owner" (or expert to contact), or preferably, the union of the regions. Tradeoffs exist between the amount of bandwidth and state used, and the speed of queries. In general, we prefer to keep a greater amount of more accurate state, so as to minimize the query time.

To perform a point query, one starts at one's local leaf server. If any matches occur, the query is completed and returns. Otherwise, the next higher server in the hierarchy is consulted to determine if any knowledgeable experts exist in a wider area. This procedure ensures that closer experts will be found before more distant experts. This approach both helps to ensure availability of experts matched, and minimizes latency and bandwidth used.

A second issue is the *ordering* of the list of experts. The constraints listed above lead us to the following ordering method.

1. Prefer closer experts first to achieve availability. This heuristic corresponds to using the levels of the hierarchy, starting at the bottom and working upwards. The remaining preferences (below) thus correspond to rules employed by a level-i server to construct a list of experts in response to a query it receives.
2. Required attributes must match exactly between the requested object and matched areas of expertise, and any optional attributes must not be in conflict. This means that the list constructed by a level-i server must not contain any experts whose areas of expertise are known not to include the given object. Note that within the server, spatial database techniques such as R-trees may be used to implement this rule. The remaining preferences (below) then describe how a single server should order the regions it finds.
3. Experts with more advanced capability are preferred (e.g., ones that can repair, not just diagnose) over less advanced experts. Note that the capability level is a required attribute in describing regions (but not objects) to enable this rule.
4. Prefer experts which match more of the optional attributes. If a region's description does not specify an optional attribute contained in the request, it is not considered to match when counting matched attributes.
5. Finally, to break ties, we use the Highest Random Weight (HRW) algorithm described in [18]. If two requests for the same object retrieve the same set S of servers, HRW generates the same ordering of servers in S for both requests. However, requests for different objects generate different orderings on S, so HRW both helps to balance the load on equivalent experts, and reduces duplication of work in accordance with the Conservation of Effort principle. For classes of objects which are popular, but not confined to a small area, all experts found in step 2 will often be tied and hence HRW will determine expert selection.

According to the algorithm described above, the list of "experts" returned by a query may actually be location servers. However, when a request is sent to an "expert" which is actually a level-i server, the server responds by redirecting the request to a list of level-$(i-1)$ servers (or experts). This means that any non-expert can be resolved to a list of actual experts.

Finally, we optimize the lookup procedure and enhance availability by caching the results of previous lookups. That is, if an entity wants to locate an expert on a particular object, and any experts' areas of expertise in the cache include that object, then no network messages are needed to resolve the list of experts.

4.2 Security Issues

It is important, though not required, that a reporter be able to trust the feedback from an expert. If a malicious expert rejects a true hypothesis (violating

the Freedom of Information principle), the reporter will simply attempt to cope with the symptoms.

If a malicious expert confirms a false hypothesis, the reporter may choose either to wait for repairs to complete if the expert's expected time to repair is acceptable, or to simply cope with the symptoms. An expert that frequently provides wrong feedback must be isolated. We provide a mechanism for identifying such experts by requiring that the originators of capability advertisements be authenticated. In practice, only short-term intruders are a concern, since experts wishing to establish a long-term presence would have no incentive to become known as unreliable.

To ensure integrity of capability advertisements and authentication of their origin, we adopt the current model recommended by the IETF for use with nameservice-like applications, which is known as DNSsec [19]. Briefly, a public/private key pair is associated with each domain, and all capabilities are signed with a domain key. To reliably learn the public key of a domain, the key itself must be signed. A resolver must therefore be configured with at least the public key of one domain that it can use to authenticate signatures. It can then securely read the public keys of other domains if the intervening domains in the ELS tree are secure and their signed keys accessible. See [19, 20] for a more detailed discussion of the security model and associated concerns.

A second security issue is denial-of-service attacks by observers reporting non-existent problems. Such attacks can be combatted in GDT by deferring tests and repairs once such an attack is suspected. For example, if a large number of reports arrive from the same origin, and the first few are rejected, the rest may be deferred (and a "GDT denial-of-service" problem report generated). If the client never refreshes the deferred state (see Section 4.3), the tests need not be performed.

4.3 Protocol Overview

In this section we give a brief overview of the GDT protocol. A detailed specification can be found elsewhere [21].

Any entity may report a problem, whether the entity is a client perceiving a problem, or an expert hypothesizing about potential causes of a known problem. To report a problem, an ordered list of experts is first obtained using the method outlined in Section 4.1. A Hypothesis message describing the potential problem is then sent to each of these experts in turn until one responds (i.e., until one is found to be reachable).

GDT is designed to be a soft state protocol, meaning that all state held in experts and servers will eventually expire and be deleted unless explicitly refreshed by receiving relevant messages. Significant events cause experts to return a status report to each entity which sent it a Hypothesis message for that problem. These status reports are not acknowledged, but are periodically resent to allow for lost messages and to keep state alive at the origin so it need not try another expert. Hypotheses are periodically (at low frequency) resent to experts to indicate that the sender is still interested in receiving status reports for reported problems.

When an expert receives a Hypothesis about a new problem, a domain-expertise module applies known domain-specific tests to confirm or deny the existence of the reported problem. The expert merely acts as a supervisor for these tests, letting the domain expertise module (or a human) conduct the actual test using its own methods. This confirmation step may be skipped if the Hypothesis is received from a trusted source and indicates that the problem has already been confirmed. When a test completes, all origins are informed that the hypothesis was confirmed, rejected, or that it was indeterminate. (Since a problem can have multiple effects, there can be multiple origins, one per reported effect.)

Once a problem is confirmed, the expert generates hypotheses about potential causes according to the procedure described in Section 3. This may entail employing a resolution procedure to determine the list of objects above, below, upstream, or downstream from the problematic object. Each hypothesis is then sent to an appropriate expert as described above.

If no potential causes were found for a confirmed problem, or if all hypotheses have been rejected or are indeterminate, then a root cause has been reached, and repairs may begin whenever possible.

To conserve effort and ensure that repairs are conducted as close to the root cause as possible, no actions will be initially requested for problems with confirmed causes. When a root cause cannot be repaired immediately, its status is set to Repair-Deferred, and all entities from which a hypothesis of the cause has been received are informed. When all confirmed causes of a problem have had repairs deferred, then repairs may begin (or be deferred) for the effect where possible. This process continues down the tree of effects until repairs are begun immediately, or until the origins of problem reports for leaf effects are reached.

We emphasize that the specific tests and repairs to be done are domain-specific and hence are outside the scope of the coordination protocol. This also allows each administration or even each expert to have its own troubleshooting procedures, while still allowing coordination between heterogeneous experts.

4.4 Breaking cycles in the graph

As discussed in Section 3.2, cycles in cause-effect graphs are an important concern. To prevent deadlock, cycles are detected by propagating selected Status Report messages down to leaf effects in the cause-effect graph when potential for a cycle exists (namely, when a Hypothesis message is received for a previously-confirmed problem). If a Status Report about a specific problem is relayed down to the same problem, then a cycle must exist, and the cycle is broken by treating its cause as indeterminate, forcing the problem to be treated as a root cause.

4.5 Performance

We simulated the performance of GDT by implementing clients and experts using "ns", the LBNL Network Simulator [22]. Because of space constraints, we could not include these results here, but they are available in an extended version of this paper [23]. Briefly, we simulated the performance of GDT in terms of troubleshooting time, number of messages, and amount of state required, while varying the number of clients, the network size, and the degree of message loss. These simulations showed that GDT performs well as the number of clients and problems grows, and continues to function amidst heavy packet loss.

5 Conclusions

In this paper, we have explored the constraints of a new problem: that of coordinating troubleshooting information among peers and untrusted observers. Allowing untrusted observers removes any restrictions on who may be an observer, and may include NOCs, end-users, and even applications.

As part of our architectural framework, we have presented a methodology for coordinating problem diagnosis under these contraints. We then described a protocol, GDT, which realizes our methodology.

We believe that our architecture scales well, and is potentially suitable for the global Internet. Our vision is to allow troubleshooting to proceed automatically so that end users and applications can get accurate and timely feedback on problems, such as obtaining the expected time until repair. We believe this is especially important (and potentially difficult) when the cause of observed problems is very distant.

References

[1] Shri Goyal and Ralph Worrest. Expert systems in network maintenance and management. In *IEEE International Conference on Communications*, June 1986.

[2] Makoto Yoshida, Makoto Kobayashi, and Haruo Yamaguchi. Customer control of network management from the service provider's perspective. *IEEE Communications Magazine*, pages 35–40, March 1990.

[3] Kraig R. Meyer and Dale S. Johnson. Experience in network management: The Merit network operations center. In *Integrated Network Management, II.* IFIP TC6/WG6.6, April 1991.

[4] Alan Hannan. Inter-provider outage notification. *NANOG*, May 1996.

[5] Craig Labovitz. Routing stability analysis. *North American Network Operator's Group*, October 1996.

[6] Merit/ISI. Inter-provider notification. `http://compute.merit.edu/ipn.html`.

[7] David D. Clark. The design philosophy of the DARPA Internet protocols. *Proc. of ACM SIGCOMM '88*, pages 106–114, 1988.

[8] Marshall T. Rose. *The Simple Book.* Prentice Hall, 2nd edition, 1994.

[9] Zheng Wang. Model of network faults. In *Integrated Network Management, I.* IFIP TC6/WG6.6, April 1989.

[10] ISO. Information processing systems - open systems interconnection - basic reference model - part 4: Management framework, 1989. ISO 7498-4.

[11] Germán Goldszmidt and Yechiam Yemini. Evaluating management decisions via delegation. In *Integrated Network Management, III.* IFIP TC6/WG6.6, April 1993.

[12] Willis Stinson and Shaygan Kheradpir. A state-based approach to real-time telecommunications network management. In *NOMS*, 1992.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, June 1984.

[14] Paul Mockapetris. Domain names - concepts and facilities, November 1987. RFC-1034.

[15] Gerald Neufeld. Descriptive names in X.500. In *Proceedings of the ACM SIGCOMM*, pages 64–70, 1989.

[16] Larry L. Peterson. The profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[17] D. Thaler and C.V. Ravishankar. Distributed top-down hierarchy construction. Submitted to *IEEE INFOCOM'98.*

[18] D. Thaler and C.V. Ravishankar. Using name-based mappings to increase hit rates. *ACM/IEEE Transactions on Networking*, to appear.

[19] D. Eastlake and C. Kaufman. Domain name system security extensions, January 1997. RFC-2065.

[20] D. Eastlake. Secure domain name system dynamic update, April 1997. RFC-2137.

[21] D. Thaler. Globally-distributed troubleshooting (GDT): Protocol specification. *Work in progress*, November 1996.

[22] Lawrence Berkeley National Labs. ns software. `http://www-nrg.ee.lbl.gov/ns/`.

[23] David Thaler and Chinya V. Ravishankar. An architecture for inter-domain troubleshooting (extended version). Technical Report CSE-TR-344-97, University of Michigan, July 1997.