An Architecture for Inter-Domain Troubleshooting

David G. Thaler and Chinya V. Ravishankar*

Microsoft Corporation — University of California—Riverside

dthaler@microsoft.com — ravi@cs.ucr.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. Our approach 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 approach for coordinating problem diagnosis, and describe Global Distributed Troubleshooting (GDT), a distributed protocol which realizes this approach. We show through simulation that GDT scales well as the number of observers and problems grows.

Keywords: Network management agents, fault diagnosis, fault repair.

1 Introduction

With the explosive growth of the Internet, the problem of troubleshooting across administrative domains is becoming increasingly important. Troubleshooting within a single domain is a far easier problem, since it can be assumed that the management software and the managed devices are all controlled by the same administration, or that network management entities are mutually trusted. 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].

Such assumptions are generally invalid because it is often impossible to troubleshoot networking problem without some information from outside one's local domain. For example, a 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

*Computer Science and Engineering Department, University of California–Riverside, Riverside, CA 92521, (909) 787-2451 (Tel), (909) 787-4643 (Fax)

problem anywhere between the browser and the remote server. In such cases, one's local help desk is likely to be of no help at all.

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. It is also possible to extend our work to address the issues of inter-administration negotiation for repairs, and of notifying organizations of scheduled future downtime, but we do not deal with them in this paper. Work such as the Inter-Provider Notification scheme IPN [6] addresses the issue of pre-notification, but pre-notification is of no help in coordinating troubleshooting, since past announcements may have been lost, ignored, forgotten, or may be inaccessible during the problem. Work also exists on models for fault isolation [7] and on analyzing the effects of error detection and recovery times in networks [8]. However, such work does not address architectural or other practical issues that arise when building real-world network troubleshooting systems.

While methods do exist for network troubleshooting and management within a single administrative domain, such methods may not be applicable to the inter-administration case. In particular, management entities may not be allowed to access or trust observers across administrative domains. New mechanisms are thus needed for intra-domain troubleshooting. We do not address the specifics of troubleshooting within administrative domains in this paper. Our approach here is to build inter-administration coordination on top of the management functionality existing within each administrative domain. We therefore concern ourselves primarily with the task of coordinating information exchange between domains in an effective manner.

1.1 Our approach

The general approach we adopt to network troubleshooting is for entities (which may be users, programs, or organizations) to report problems by sending problem reports (or "hypotheses") to the appropriate "experts". These experts will be agents with jurisdictions limited to specific classes of problems in specific administrative domains. However, an expert troubleshooting a problem reported to it may detect or hypothesize the presence of other network problems with a bearing on the problem at hand. In such a case, the expert can use our framework and protocol to refer such problems to other experts inside or outside its administrative domain. Such troubleshooting proceeds recursively, in stages, as each expert refines the diagnosis and generates "hypotheses" that it forwards, for confirmation and action, to the appropriate experts inside or outside its own domain. The first step in this process is to confirm the original problem report (or hypothesis), generating the appropriate follow-ups. In the final step the originator of the problem report receives either a diagnosis of the problem, or, in the ideal case, a confirmation that the problem has been fixed.

We have two goals in this paper. First, it is to clearly define the characteristics of the inter-domain information coordination problem for network troubleshooting. Second, we aim to provide a framework

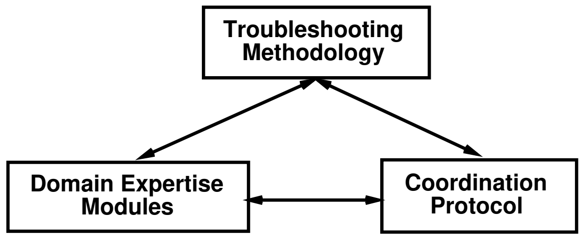


Figure 1: Components of troubleshooting scheme

and protocol that any entity (including a user, a NOC, or an application) may use to report network problems and receive appropriate feedback, regardless of its own location.

1.2 The GDT framework

We present a framework called GDT (Globally-Distributed Troubleshooting) that provides a basis for such problem reporting and troubleshooting. As Figure 1 shows, our framework comprises three components:

1. *GDT Methodology:* This constitutes the framework, or theory behind the protocol we present for performing effective troubleshooting in an inter-domain environment. The problem and domain characteristics, network problem classifications, and troubleshooting model are defined in this component.
2. *Domain-expertise modules (DEMs):* These are existing tools (such as Network Management Systems) implementing management functionalities that we assume within individual domains. These tools may apply traditional management techniques Examples of domain-expertise modules include existing Network Management Systems (NMS's), connectivity diagnosis tools such as CT [9], up-down and congestion experts as used by the Automated Network Management System ANM [10], and agents employing newer techniques such as Anomaly Signature Matching [11], Bayesian belief nets and Non-deterministic reasoning [12].
3. *GDT Coordination Protocol:* This is the protocol used by management entities, possibly resident in different administrative domains, for transfer of troubleshooting information. The protocol implements the theory and approach.

Figure 2 gives an overview of our inter-domain troubleshooting approach. The large ovals represent domains, and one or more inter-domain troubleshooting agents, called "experts", are shown in each domain. These experts have DEMs, implement the troubleshooting approach, and use the inter-domain coordination protocol to communicate with each other.

In this example, an attacker ("Bad Source BS") is sending a high rate of data ("stream B") to a destination, and this stream is adversely affecting a "good" stream A. When the destination (GD) of stream A notices heavy loss, it reports a problem with stream A to its local expert using the coordination protocol. Using its DEMs and troubleshooting approach, this expert learns that the problem with stream

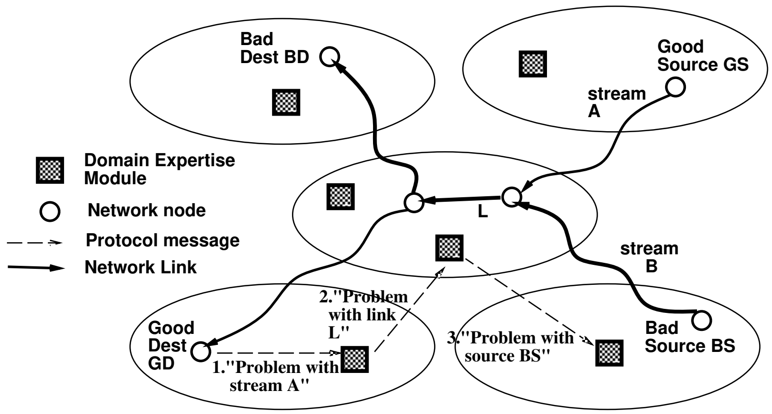


Figure 2: Troubleshooting Example

A is caused by loss on some link L in another domain. It then reports a problem with link L to an expert in that domain. Similarly, the next expert finds and reports a problem with the source of stream B to an expert in that domain, where the problem may be ultimately corrected.

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. Section 3 discusses domain-expertise modules. Section 4 presents a method for diagnosing problems. Section 5 describes our protocol, Section 6 gives simulation results, and Section 7 covers conclusions and the future.

2 Design Philosophy

Network troubleshooting is a complex process since it must resolve both technical and policy issues. The technical difficulties are generally network- and failure-related, and do not vary much across domains. In contrast, there is likely to be greater variability in policies among domains. We therefore set forth criteria and rules to guide us in the design of our troubleshooting method. We discuss some social and administrative policy issues arising in the context of a global network composed of a large number of autonomous administrations.

We will illustrate a number of design principles which provide the motivation for the application of our approach. Many of these are based on principles that are recognized to govern the design of networks in general.

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 everyone affected.* Users (and even applications) can often benefit from information such as the expected downtime. It is also important to provide enough information to allow troubleshooting problems which span several administrative domains. To obtain such information quickly, it is desirable to remove human intervention (e.g., help desks) from the loop whenever possible.

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 allowed to report a problem, whether or not it is trusted.*
This principle is a direct 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. No attempts should be made to repair non-existent problems.*
The first part requires that a repair be performed as near as possible 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

To ensure that we do not introduce new constraints, our architecture will follow the design philosophy described for the Internet itself in [13]. This philosophy may be summarized in the following keywords, ranked in order of importance: high availability, provision for mutiple services, networks, and centers of administration, cost-effectiveness, low effort for deployment, and accountability. These constraints do not directly reflect the requirements of network troubleshooting, but we adapt them as discussed below.

Network troubleshooting and diagnosis is performed primarily for restoring service. Thus, the primary criterion that we set for ourselves is high availability. To give this idea more concrete form, we paraphrase (Rose [14]) as follows:

Constraint 1 (Reliability): *When all else fails, troubleshooting must continue to function, if at all possible.*
Since our focus is specifically on troubleshooting, this constraint is of even greater importance. "Reliability" in the context of troubleshooting services has a different meaning than is usually associated with the term. The backdrop to determining the reliability of ordinary distributed systems are normal operating conditions. In contrast, troubleshooting must be reliable precisely when the operating conditions are harsh. As an analogy, a car steering system is expected to be reliable under normal driving conditions, but not when a collision occurs. On the other hand, an airbag deployment system must be guaranteed to operate in collisions, but never under normal conditions. 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 name service, file systems, or TCP are not available.

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

This constraint is essential since many problems may exist simultaneously, and the network may span many autonomous administrations. The network configuration may also change over time. If the architecture automatically adapts to network changes, the need for manual configuration of the troubleshooting system is minimized.

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 obstacles for network entities to participate in troubleshooting must be as low as possible, and the bandwidth and memory costs required for troubleshooting must not outweigh its benefits.

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

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

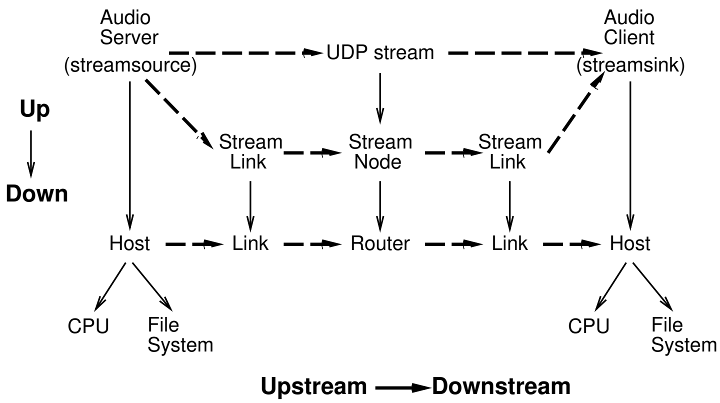
3 Domain-Expertise Modules

Domain-expertise modules are components which provide the basic troubleshooting capabilities that the rest of our scheme builds on. A domain-expertise module (DEM) is any software agent which has, or can obtain, information useful for diagnosing a problem within the agent's domain. Domain expertise modules are used by "experts", which are higer-level entities organized into a society, and which communicate using the protocol described in Section 5.4 to cooperatively troubleshoot problems. We give several examples of DEMs below:

NMSs: A Network Management System (NMS) typically uses protocols such as SNMP and `ping` (i.e., echo request/reply messages) to manage and monitor the state of all devices within an administrative domain. An NMS is typically run from a console in the domain's NOC, where it may have access to current and historical data on devices and problem reports. Examples of NMS's include HP Openview [15], MView [16], and ANM [10].

MbD agents: To address some of the scalability concerns with having a single NMS polling all devices in a large domain, there has been recent interest in distributed (intra-domain) management. For example, Goldszmidt and Yemini have proposed a Management by Delegation (MbD) [17] approach whereby an NMS can push scripts out to management agents around its domain. Each of these agents could potentially be viewed as a DEM. An overview of MbD and other approaches to distributed intra-domain management can be found in [18].

NetNow daemons: As part of the Internet Performance Measurement and Analysis (IPMA) project, NetNow daemons [19] measure latency and loss across major Internet Service Providers and Internet exchange points. Such a daemon thus has current and historical data on problems seen in each of several domains.



`link`	Connection between nodes
`stream`	End-to-end traffic session
`streamlink`	Traffic on link from one stream
`streamsource`	Source of a given stream
`streamsink`	Receiver of a given stream

Figure 3: Sample Resource Dependency Graph

DEMs also include tools which diagnose problems end-to-end and help narrow down the cause to a given domain. Since there are many existing techniques (although ad hoc) to do such things, we wish to incorporate them into our framework as well. Examples of such DEMs include:

CT: The Connectivity Tool (CT) [9] is a rule-based system which uses well-known utilities such as `traceroute` (which traces the network path between two endpoints) and `ping`, as well as SNMP queries to identify the causes of end-to-end connectivity problems in the Internet.

mtrace daemon: An agent may passively listen in on multicast traceroute [20] messages to form a picture of Internet connectivity and loss rates. In [21], we describe our experience with using such agents in our architecture.

As we will see, DEMs will provide the mechanisms to obtain specific information which is used in our approach. In Section 5, we will return to the discussion of how DEMs are used in our architecture.

4 Troubleshooting Methodology in GDT

We now present our model for fault location and propagation in networks, and the framework that embeds the troubleshooting approach. 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.

Relationships between *classes* of objects are typically static, whereas relationships between specific *instances* of objects are often dynamic. For example, a UDP stream will always depend upon links and routers, but the specific instances may change over time (even during the lifetime of the stream). Our goal in practice will be to statically know static relationships, and to dynamically resolve dynamic relationships.

An object's *capacity* denotes the total amount of resources it makes available. For example, a link's capacity might be measured in megabits per second, and a file system's capacity might be measured in gigabytes. An object's *utilization* denotes the total amount of its resources in use.

We adopt the concept of a "health function" from [22], so that an object's *health* is 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 term 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 file system 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 [23].

4.1 Fault-propagation Model

As in [24], we use the terms "vertical propagation" of faults to refer to the effects a fault at some level of the networking hierarchy has on levels above it, and "horizontal propagation" to refer to the effects it has on peer entities across the network. These ideas are illustrated in Figure 3. In referring to 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 [23].

Our approach is to model fault propagation using *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 3, the audio client and server applications demand resources from the host CPU and the file system, 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 consequently on the efficiency and correctness of objects below and downstream of it.

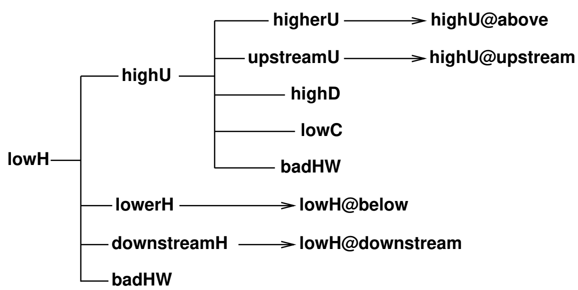
Our approach requires creating and maintaining, for each class of objects, means for determining an instance's capacity, utilization, and health, and methods for determining the set of instances above, below, upstream, and downstream from a given instance. This implies that if there are several methods of determining whether an object is healthy, for instance, then these may represent separate classes of objects. For example, a link may look congested to a single stream, but uncongested when looking at the aggregate traffic. In our examples, the object class `streamlink` refers to the former concept, and `link` to the latter.

We are now 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.*

An object which is highly utilized may consequently impose higher demands on the objects upon which it depends.

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



`lowH/lowC`	health/capacity too low
`highU/highD`	utilization/resource demand too high
`lowerH`	an object below has low health
`downstreamH`	an object downstream has low health
`higherU`	an object above has high utilization
`upstreamU`	an object upstream has high utilization
`highD`	object's resource demand too high
`lowC`	capacity too low for normal demands
`badHW`	object's hardware/software broken

Figure 4: Problem Taxonomy

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

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

That is, low health may arise from soft failures (congestion) in addition to hard failures (hardware or software faults).

4.2 Cause-effect Graphs

Previous studies (e.g., [25, 26]) have typically only looked at only the "up" direction for fault propagation. 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. In Figure 4, we present a taxonomy of problem types based on our discussions in Section 4.1.

As shown in this figure, degraded performance (`lowH`) of an object can be caused by congestion (`highU`, Observation 3 above), by degraded performance at a lower or downstream object (`lowerH`, `downstreamH`, Observation 2 above), 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 above), 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 5 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. This graph is shown in Figure 6. SInce problem types in our taxonomy are always qualified by an object name, we use dotted

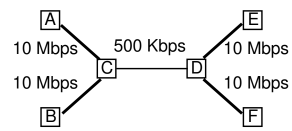


Figure 5: Sample Network Topology

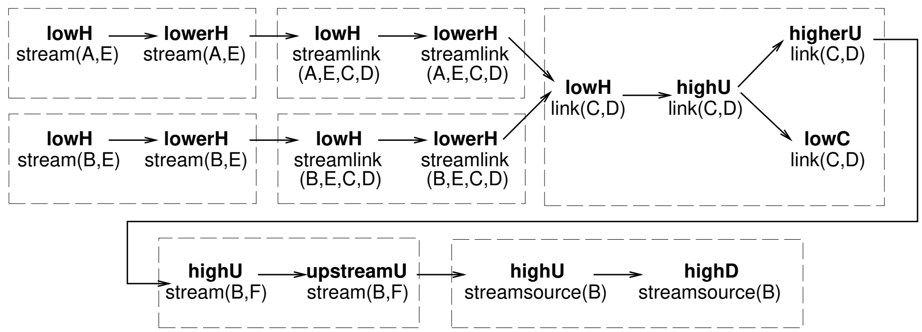


Figure 6: Cause-Effect Graph for Figure 5. (See Figures 3 & 4 for explanations of terms.)

lines in Figure 6 to group all problems associated with the same object. As we will see in Section 5, this graph will be distributed in practice to provide scalability and support privacy.

We call a symptom from which cause-effect graph construction begins a "leaf effect", since diagnosis always proceeds from an effect to its cause. An example in this case would be `lowH` of stream(A,E). A problem such as `highD` of streamsource(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 the figure shows. 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.

4.2.1 Adequacy of model

The nine problem types shown in Figure 4 are both 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 additional subdivisions will be specific to the classes 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. We have no need, therefore, to classify problems at any finer granularity.

This observation is important, since we want to make use of existing techniques developed for specific classes of objects or within specific administrative domains. This is the purpose of the Domain-Expertise Modules.

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

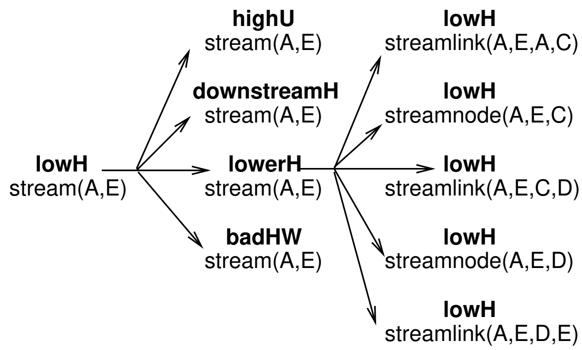


Figure 7: Generating Hypotheses. (See Figures 3 & 4 for explanations of terms.)

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 5.5.

4.2.2 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 4) and the Resource Dependency Graph (Figure 3) which includes the affected object.
3. Repeat from step 1 for each new hypothesis generated.

Figure 7 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 4. 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 6 shows the results of applying this process to two leaf effects: low health of both 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.

5 Problem Report Coordination in GDT

We outlined our approach and philosophy to developing a network troubleshooting model in Section 2, and described our troubleshooting framework and model more concretely in Section 4. In this section, we address the issue of how to apply this approach in a scalable manner to a distributed architecture meeting the requirements stated in Section 2.

Clients are the entities in our system that report problems, acting on behalf of the end-user, application, or NOC that observed the problem. To ensure scalability of our scheme to a global network composed of a large number of administrative domains, we propose a society of troubleshooting coordination agents, which we call *experts*. These are higher-level entities than the Domain Expertise Modules described in Section 3. Experts avail themselves of the capabilities of DEMs for troubleshooting, but also communicate with each other and with clients using the troubleshooting protocols our scheme defines. 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 capabilities and permissions necessary to diagnose a specific set (e.g., a range or list) of instances of that class. Figure 8 illustrates how experts may be distributed across five domains A, B, C, D, and E.

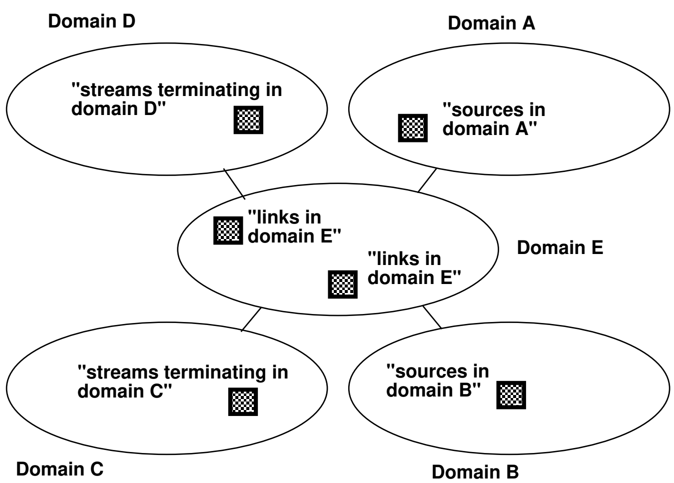


Figure 8: Assigning Capabilities to Experts

5.1 Capabilities and operation of experts

Each expert must posess certain capabilities in order to perform its designated functions. 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 approach of Section 4 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 (see Principle 2, Section 2).

5.2 Expert Naming and discovery

We begin with the observation that an expert location service for troubleshooting will be required precisely when faults and problems may have compromised the environment and infrastructure. Reliability considerations (see Constraint 1 of Section 2.2) mandate that the expert location service should *not* make use of any existing system functionality for service location. For example, it should not *require* multicast, or else it cannot diagnose multicast routing problems. 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 name space. 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 given as "`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.

The jurisdiction of an expert, however, may cover a number of different objects, and thus its jurisdiction extends over multiple points from the name space. Thus, areas of expertise correspond to regions in the name space. 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 [27] 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 name service below, in order of importance:

1. Allow availability during network partitions (i.e., locate reachable experts during partitions).
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 name servers corresponding to a hierarchy in the name space (as is used by DNS [28], X.500 [29], etc) will not work, since we must have successful queries even when we are partitioned from a large part of the network. Replicating such name servers everywhere will not keep the bandwidth overhead low. We also want to avoid mandating a hierarchical name space to preserve domain autonomy and class independence. On the other hand, we desire some structure to the server scheme so that expert location scheme has higher availability, and be easily adapted to changing conditions without manual reconfiguration. Many existing attribute-based naming schemes (e.g., [30]) provide no structure to name servers and hence rely on manual configuration.

The solution we adopt is depicted in Figure 9. Expert location servers (ELS's) are organized into a hierarchy according to their *location*. Informally, each ELS is responsible for knowing the name space 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 [31].

Experts form the actual leaves of the tree so defined, with each expert's parent being the closest ELS. We will refer to an ELS whose children are all experts (and not ELS's) as a *leaf ELS*. As in all hierarchical systems, the specificity of naming information is higher at lower levels. Each expert periodically advertises its areas of expertise to its parent ELS. Each ELS, in turn, reports to its own parent, either the bounding box enclosing the regions that have been reported to it, with its own address as the location authority (or contact ELS), or preferably, the true union of the regions. This choice will be made based on tradeoffs that exist between the bandwidth and state required, 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 execute a point query (i.e., locate an object), one presents it to one's local leaf server. If any matches occur, the query completes and returns. Otherwise, the next higher server in the hierarchy is consulted

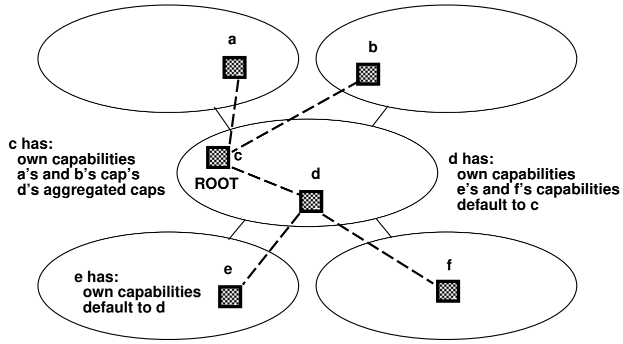


Figure 9: Using a Hierarchy for Expert Location

to determine if any knowledgeable experts exist in its (wider) area of jurisdiction. This procedure ensures that closer experts will be found before more distant experts, and helps ensure availability of experts matched, and minimizes latency and bandwidth used.

A second issue is the *ordering* of the list of experts returned. 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 preferences rules listed below thus are used by a level-i ELS 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 ELS must not contain any experts whose areas of expertise are known not to include the given object. Within an ELS, spatial database techniques such as R-trees may be used to implement this rule. The preferences rules below describe how a single ELS 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. The capability level is a required attribute when 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 [32]. If two requests for the same object retrieve the same set S of ELS's, HRW generates the same ordering of ELS's 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 ELS's. However, when a request is sent to an "expert" which is actually a level-i ELS, the ELS responds by redirecting the request to a list of level-$(i-1)$ ELS (or experts). This means that a request sent to any ELS can actually 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 the area of expertise of some expert in the cache covers that object, then no network messages are needed.

5.3 Security Issues

Integrity, privacy, and denial-of-service are three important security issues. We discuss them below.

5.3.1 Integrity

It is important, albeit not required, that an entity reporting a problem be able to trust the feedback from an expert. If a malicious expert violates the Freedom of Information principle by rejecting a true hypothesis, the entity reporting the problem can simply attempt to cope with the symptoms. Similarly, if a malicious expert confirms a false hypothesis, the reporting entity 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. In the following discussion, we use the term "capability" to refer to an expert's area of expertise and its authority over it, jointly. We provide a mechanism for identifying malicious 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 DNSsec model [33] recommended by the IETF for use with nameservice-like applications. 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 [33, 34] for a more detailed discussion of the security model and associated concerns.

An alternative method to ensure integrity of capability advertisements and authentication of their origin entails the use of one or more "trusted third parties", or signature authorities. The idea here is that such an authority would sign the public key of an organization according to a binding, legal agreement. Any intentional misuse by a party with a signed key could then be followed up with legal action. Each ELS may then include with its capability advertisements, its signed credentials, which can be verified by anyone with the authority's public key. Such credentials can also be passed along with capabilities in Redirect messages. Each client may then be configured with one or more public keys of signature authorities which it trusts, and prefers experts whose credentials it can validate (i.e., are signed by an authority it trusts) over those it cannot. This allows communication to occur even when no keys are used and does not require any online interaction with authorities (which may not be available during a problem). Finally, revoking a key is as simple as periodically (until the lifetime of the key expires) multicasting a public key revocation message which is signed with the associated private key.

Comparing the two above schemes, the DNSsec-like method assumes transitivity of trust, which may be dangerous, but does not require any special authorities. Further investigation of these tradeoffs and solutions is an opportunity for future work.

5.3.2 Privacy

The second security issue is privacy. It is important to ensure that experts maintain control over the amount and nature of information that they make available to others, especially over information that crosses domain boundaries.

We first observe that our protocol does not itself generate new information. Thus, it does not provide any information to users which would not be available to them in its absence. The protocol simply provides a way to automate the exchange of such information without human intervention.

Preventing leakage of information is then largely a question of restricting the detail in the responses that experts provide to reporters. The most straightforward approach to addressing this issue is to recognize multiple privacy classes, each defining a level at which information may be shared with reporters of problems. An expert can now use a local mechanism to assign a privacy class to each reporter, and thus directly control what information it is willing to share with it. Experts may assign reporters to privacy classes purely via static configuration, based on IP address, or based on a security relationship.

The examples below illustrate how an expert may provide graded reponses to reporters, arranged in order of decreasing privacy.

1. No response. This can be useful in combating denial-of-service attacks, discussed below. However, this extreme level of privacy can be detrimental to troubleshooting, since lack of response could itself become a reportable problem symptom.

2. An `indeterminate` response is always returned. This allows a reporter's state machine to continue immediately, possibly initiating local repairs, but without divulging any information.

3. The correct response is returned, one of `confirmed`, `rejected`, or `indeterminate`. This allows the reporter's state machine to work best, but does not divulge any information other than admitting the existence of a problem if the response was `confirmed`.

4. The correct response is returned, along with additional information which may be useful to the reporter. For example, a description of a workaround, or additional clarification of the problem, may be sent, such as which would be placed on a public web site in the absence of a troubleshooting protocol.

5. Additional privacy levels may be added if varying degrees of extra information are warranted.

5.3.3 Denial of service

A third security issue is denial-of-service attacks by observers reporting non-existent problems. Such attacks can be addressed 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 5.4), the tests need not be performed. In extreme cases where resources are inadequate to keep such state, the expert may respond to the origin that the test was indeterminate. No state is required in this case. In other words, when a denial-of-service attack is suspected, the privacy level can be increased.

5.4 GDT Protocol Overview

In this section we give an overview of the GDT protocol. A detailed specification can be found elsewhere [35].

Hypotheses may originate either from a client (in which case it is a problem report), of from an expert hypothesizing about potential causes of a known problem. To report a problem, an ordered list of experts with appropriate areas of expertise is first obtained using the method outlined in Section 5.2. 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). Since a problem can have multiple effects, there can be multiple origins, one per reported effect.

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. Each `hypothesis` is periodically (at low frequency) resent to its expert to indicate that the sender is still interested in receiving status reports for the problems reported.

When an expert receives a `hypothesis` about a new problem, it invokes a domain-expertise module, which applies known domain-specific tests to confirm or deny the existence of the reported problem. Such a test may consist of any of: a simple table lookup if information is periodically collected and stored, an automated test, or involve alerting an operator to request a manual test. The expert acts as a supervisor for these tests, but lets the domain expertise module (or a human) conduct the actual test using methods it deems best. 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.

Once a problem is confirmed, the expert generates hypotheses about potential causes according to the procedure described in Section 4. This may entail employing a resolution procedure to determine the list of objects above, below, upstream, or downstream from the problematic object. An appropriate expert is located for each such hypothesis, and a corresponding `hypothesis` message sent to it.

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. At this point, we either report the diagnosis to the originator and to other concerned entitites, or initiate repairs, 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 performed 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.

5.5 Breaking Cycles in the Graph

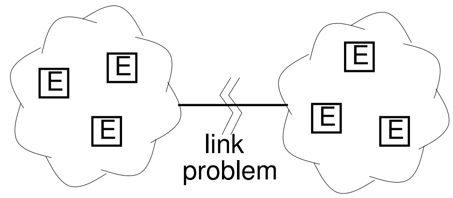
As discussed in Section 4.2, cycles in cause-effect graphs are an important concern. To prevent deadlock, cycles are detected by propagating selected Status Report messages down to leaf effects in the cause-effect graph when potential for a cycle exists (namely, when a `hypothesis` message is received 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.

6 Simulation

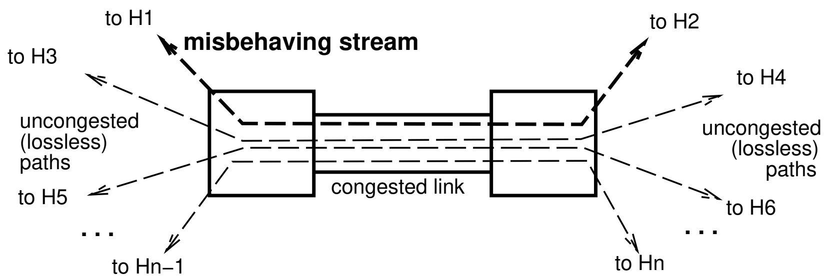
In this section, we evaluate the performance of the GDT protocol. For this purpose, we implemented GDT clients and experts using "`ns`", the LBNL Network Simulator [36]. Our goal will be to test its performance and reliability under conditions particularly adverse to GDT.

6.1 Failure model

To start with, it is important to evaluate the effects of hard and soft failures on GDT. Consider hard failures first. Hard failures may cause network partitions, making some experts unreachable (see Figure 10(a)). In this case, troubleshooting will succeed if any experts able to diagnose the problem and supervise repairs are still reachable. If no reachable experts exist, the only option available is to directly request repairs whenever possible, and repair the leaf effects' problems. Nothing further can be done. (This is less likely if the client itself has areas of expertise). In any case, the hard failure conditions, unlike soft failures, are unlikely to change in the short term. Most of the Internet is also "self-correcting" since the routing



(a) Network Partition



(b) Simulated Congestion Problem

Figure 10: Sample Network Problems

protocols automatically adapt (in time) to hard failures, masking them from higher layers. In our approach, this is equivalent to automatically modifying the dynamic resource dependency graph such that objects do not depend upon failed nodes or links. This means that, unlike effects of soft failures, many symptoms of hard failures will go away after some convergence time without the need to report the symptoms.

When a soft failure is present, on the other hand, "reachability" can be a somewhat fuzzy concept, since some messages may get through and others may not. A hard failure may be indistinguishable from a soft failure during some period of time where no service is available and no demands are met. These characteristics make soft failures more difficult to diagnose, and hence we will use one in our simulations to show how GDT performs in this case.

6.2 Simulation results

We first place one "misbehaving" stream between one host in each cloud shown in Figure 10(a). This stream sends 400 Kbps of constant bit-rate data across a 500 Kbps link, adversely affecting one or more other low (but constant) bit-rate data streams (Figure 10(b)), each of which has a client at the receiving end.

We generate a varying number of reporters (clients observing and reporting problems) by varying the number and bandwidth demands of the well-behaved streams. Figure 11 summarizes various characteristics

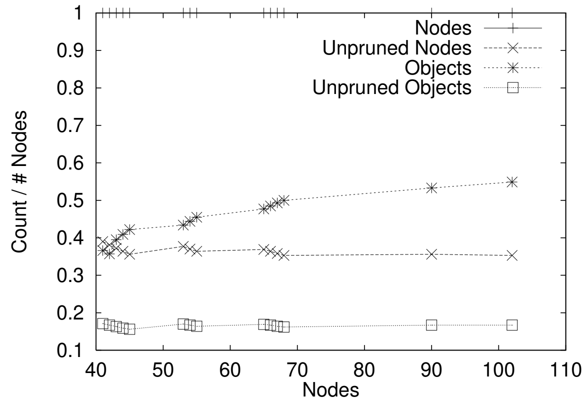


Figure 11: Cause-Tree Statistics

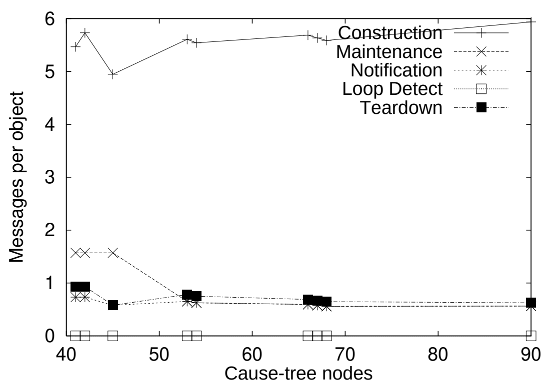
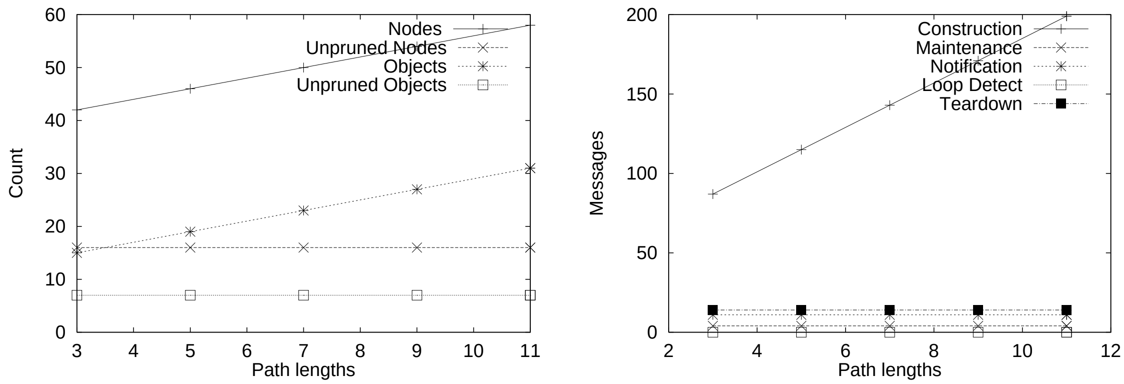


Figure 12: Messages Sent

of the resulting cause-effect graphs, showing the ratios relative to the total number of nodes in the cause-effect graph. The first set of simulations used paths of length 3, and all clients and experts were on the same side of the congested link, so that no GDT messages were lost. Each of the clusters of data points corresponds to adding another reporter. After the first reporter, only enough nodes are added to determine that the new effect is caused by a previously-confirmed problem. Finally, we observe that only about 36% of the nodes get confirmed; the rest get pruned back.

Figure 12 shows the results of this simulation in terms of the number of messages sent, and the number of messages sent per object in the cause-effect graph. Construction messages are messages (e.g., Hypothesis messages) exchanged to construct the distributed cause-effect graph. Maintenance messages are periodic retransmissions to keep state alive. Notification messages relay important information (such as expected time to repair) back to reporters. Loop detect messages are used to detect and resolve loops in the cause-effect graph. Teardown messages flush state for problems which have been resolved. Note that the message counts shown refer to *potential* network messages. If all objects in the cause-effect graph were covered



(a) Total State Generated

(b) Total Messages Generated

Figure 13: Effects of Path Length

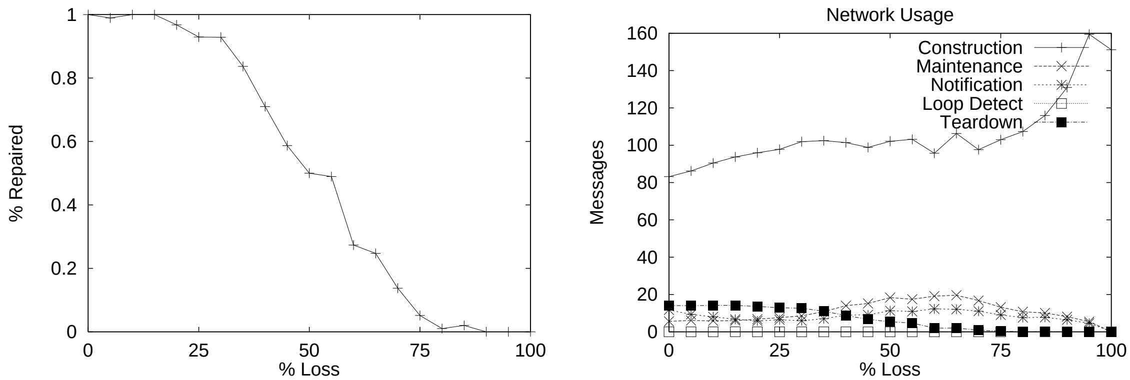
by different experts, all messages would be transmitted over the network. Thus, Figure 12 represents the *worst*-case scenario. When all objects are assigned to the *same* expert, the number of actual network messages sent is only about 5 to 10 messages *total* per reporter. In each case simulated, experts' areas of expertise were found in the reporter's cache. That is, we did not simulate expert location.

In the next simulation, we vary the path lengths of the simulated streams to see how the size of the network topology affects GDT. Figure 13(a) shows that as the path length increases, the total number of objects in the distributed graph increases by 2 per additional link traversed, reflecting the additional nodes in the resource dependency graph. These additional objects all get pruned since they aren't problematic.

Figure 13(b) shows that the total number of messages needed for construction scales linearly with the path length, since they are a function of the total number of nodes on the graph, while all message types relating only to unpruned nodes are unaffected.

To assess how well GDT functions under harsh network conditions, we now look at the effects of message loss on GDT performance. We assume that clients start with a warm cache of expert capabilities, and vary the loss rate of GDT messages by artificially injecting loss. Our simulations assumed that all messages had an equal probability of being lost when the aggregate bandwidth demand is constant (e.g., as with Random Early Detection (RED) [37] gateways). When routers with a drop-tail queueing policy exist in the network, this assumption is invalid; the policy is biased against bursty streams. Since GDT is somewhat bursty (e.g., several hypotheses may be sent out when a problem is confirmed), GDT is unfairly penalized and hence performs less well. In the future, we will investigate techniques for making GDT less bursty. Another potential scheme for improving GDT performance is to send troubleshooting coordination messages at a higher priority than normal traffic. GDT messages would then see a lower loss rate than normal traffic, and thus have an increased ability to coordinate troubleshooting information.

Figures 14 and 15 give the results of this simulation, with each point representing an average over 100



(a) Probability of Repair

(b) Resource Usage

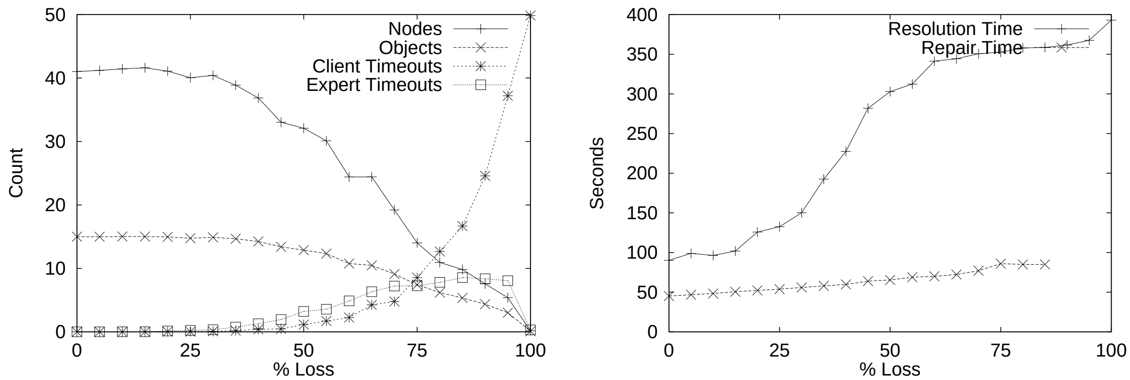
Figure 14: Effects of Loss

trials. Figure 14(a) shows the percentage of the trials in which the problem was repaired. Note that this simulation represented a worst-case scenario, since there was only one possible expert available for each problem. In practice, multiple experts would be used to provide robustness and to increase the chances of there being a reachable expert capable of determining the root cause. The chances of successful repair depend upon who must perform the repair and whether the expert can communicate with that party (or is that party). Repairs are handled by domain-expertise modules, and are outside the scope of GDT.

Figure 14(b) shows the number of messages of each type sent as the loss rate increased. We observed that the number of messages remained roughly constant as long as the repair occurred. When repairs did not occur, construction messages continued since Hypothesis messages were resent periodically in the hopes of reaching an expert.

Figure 15(a) shows that the distributed cause-effect graph is relatively unaffected until more than 50% of the GDT messages get lost. After that point, fewer clients' reports reach experts, and the cause-effect graph becomes smaller. The "Expert Timeouts" line shows how many times experts detected problems with connectivity to other experts, and began diagnosing these connectivity problems in addition to any reported by clients. GDT is therefore somewhat self-diagnosing.

Finally, Figure 15(b) shows that the time to repair the problem (when the problem was in fact repaired) increases slightly until around 75% GDT message loss (assuming no tests are deferred and the problem is repaired shortly after the root cause is found), after which point the problem was not repaired since a root cause was not found. In this simulation, no repairs to effects were performed when a cause could not be repaired. The resolution time is the time until the original reporter was notified of the repair and all cause-effect graph state was removed. If the problem was not repaired, the resolution time was the end of the simulation (400 seconds). For low loss, all times are dominated by the times required by domain-expertise modules to perform tests and repairs, whereas for higher loss, GDT's retransmission



(a) Cause-effect graph statistics

(b) Reaction Time

Figure 15: Effects of Loss

timeout intervals dominate the resolution time.

6.3 Using a Real Topology

Our final set simulations used a subset of the actual MBone topology obtained from the MWatch [38] server, which constructs a database of the actual MBone topology by periodically querying all MBone routers. (Since some routers do not respond to such queries, the MWatch database is actually a subset of the full MBone topology.) We first obtained this database from MWatch, removed all one-way links, and found the largest connected component. This final topology included 2216 routers and 7444 links[1].

Administrative divisions were difficult to obtain for the topology, so we simulated them as described below. To place experts and add areas of expertise, we first partitioned the dataset into disjoint "administrative" regions, such that each region contained only links connecting endpoints with addresses having the same first two bits. Inter-region links connected addresses whose first two bits were different. In each region, we randomly selected a single node to be an expert with areas of expertise covering objects within its region.

Since the `ns` simulator was not able to simulate multicast trees, we randomly chose pairs of senders and receivers for ten data streams with "good" rates and four more pairs for data streams with unusually high rates. As a result, four of the "good" streams experienced loss rates of up to 61%. Table 1 summarizes our simulation results.

We observe that the number of unpruned nodes and unpruned objects have increased only slightly over the numbers seen in Figure 13(a), even though there were four times as many problematic streams. This shows GDT's ability to conserve state by using cause-effect trees where multiple effects reduce to the same

[1]The MWatch database included over 3000 routers. Since it is unknown what percentage of routers are not visible to the MWatch server, the size of the actual MBone is difficult to determine.

Nodes	139	
Unpruned Nodes	20	(14%)
Objects	103	
Unpruned Objects	10	(10%)
Avg. path length	10	
Construction	570	(90%)
Maintenance	37	(6%)
Notification	20	(3%)
Loop Detection	0	(0%)
Teardown	8	(1%)
Client Timeouts	1	
Expert Timeouts	21	
Repair Time	35	
Resolution Time	190	

Table 1: MBone Topology Simulation Summary

cause, and, in the other direction, multiple causes reduce to the same effect. Since the other statistics we measured are also consistent with the results from simulations of random topologies, we believe that our earlier results and conclusions are applicable to real topologies.

7 Conclusion

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 an architecture and approach for coordinating problem diagnosis under these contraints. We then described a protocol, GDT, which realizes our approach.

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.

There reamin several open issues for future work in this area. One problem is to develop optimizations to make GDT less bursty and hence see lower packet loss across drop-tail routers. A second issue is that in practice, no faults are typically found after processing for a significant number of trouble reports. It may be useful to keep track of such reports, since they may help in tracking down intermittent or hard-to-find problems.

References

[1] Shri K. Goyal and Ralph W. Worrest. Expert systems in network maintenance and management. In *Proceedings of the 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*, pages 301–310. IFIP TC6/WG6.6, April 1991.

[4] Alan Hannan. Inter-provider outage notification. *North American Network Operator's Group*, May 1996. `http://www.academ.com/nanog/may1996/outage.html`.

[5] Craig Labovitz. Routing stability analysis. *North American Network Operator's Group*, October 1996. `http://www.academ.com/nanog/oct1996/routing-stability.html`.

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

[7] S. Kaetker and K. Geihs. A Generic Model for Fault Isolation in Integrated Management System. *Journal of Network and Systems Management*, Vol. 5, No. 2, June 1997

[8] D. Logothesis and K. Trivedi. The Effect of Detection and Restoration Times on Error Recovery in Communication Networks *Journal of Network and Systems Management*, Vol. 5, No. 2, June 1997

[9] Metin Feridun. Diagnosis of connectivity problems in the internet. In *Integrated Network Management, II*, pages 691–701. IFIP TC6/WG6.6, April 1991.

[10] M. Feridun, M. Leib, M. Nodine, and J. Ong. ANM: Automated network management system. *IEEE Network Magazine*, 2(2):13–19, March 1988.

[11] Frank Feather, Dan Slewlorek, and Roy Maxion. Fault detection in an ethernet network using anomaly signature matching. In *Proceedings of ACM SIGCOMM*, pages 279–288, September 1993.

[12] Paul Hong and Prodip Sen. Incorporating non-deterministic reasoning in managing heterogeneous network faults. In *Integrated Network Management, II*, pages 481–492. IFIP TC6/WG6.6, April 1991.

[13] David D. Clark. The design philosophy of the DARPA Internet protocols. *Proceedings of ACM SIGCOMM*, pages 106–114, 1988.

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

[15] Jill Huntington-Lee, Kornel Terplan, and Jeffrey A. Gibson. *HP Openview: A Manager's Guide.* McGraw-Hill, 1997.

[16] David G. Thaler and Chinya V. Ravishankar. NView: A visual framework for network tool integration. In *Proceedings of the IEEE International Phoenix Conference on Computers and Communications*, pages 283–289, March 1995.

[17] Germán Goldszmidt and Yechiam Yemini. Distributed management by delegation. In *Proceedings of the International Conference on Distributed Computing Systems*, June 1995.

[18] M. Kahani and H.W.P. Beadle. Decentralized approaches for network management. *Computer Communication Review*, 27(3):36–47, July 1997.

[19] Craig Labovitz. Netnow. `http://www.merit.edu/~ipma/netnow`.

[20] W. Fenner and S. Casner. A "traceroute" facility for IP multicast. *Internet Draft*, November 1997.

[21] David G. Thaler III. An architecture for inter-domain network troubleshooting, January 1998. Ph.D. Thesis, University of Michigan.

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

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

[24] Zheng Wang. Model of network faults. In *Integrated Network Management, I*, pages 345–352. IFIP TC6/WG6.6, April 1989.

[25] Willis Stinson and Shaygan Kheradpir. A state-based approach to real-time telecommunications network management. In *NOMS*, pages 520–532, 1992.

[26] Aiko Pras. *Network Management Architectures*. PhD thesis, University of Twente, Enschede, Netherlands, February 1995.

[27] 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.

[28] P. Mockapetris. Domain names - concepts and facilities, November 1987. RFC-1034.

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

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

[31] D. Thaler and C.V. Ravishankar. Distributed top-down hierarchy construction. In *Proceedings of the IEEE INFOCOM*, 1998.

[32] David G. Thaler and Chinya V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, February 1998.

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

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

[35] D. Thaler. Globally-distributed troubleshooting (GDT): Protocol specification. *Internet Draft*, January 1997. draft-thaler-gdt-00.txt.

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

[37] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[38] MICE. mwatch tools. `ftp://ftp.cl.cam.ac.uk/pub/mwatch.README`.

Biographies

David G. Thaler received the B.S. degree in computer science from Michigan State University in 1992, and the Ph.D. degree from the University of Michigan in 1998. He has been at Microsoft since 1998, where he has been heavily involved in research and deployment of both multicast and IPv6 technologies. He is currently the development lead for the TCP/IP group. He is an active participant in the Internet Engineering Task Force (IETF), where he serves as the chair of the Multicast Address Allocation (MALLOC) Working Group, and has authored numerous RFCs. He is a member of the IEEE Computer Society and of the Association for Computing Machinery (ACM).

Chinya V. Ravishankar is a Professor of Computer Science at the University of California—Riverside. He received his B.Tech. in Chemical Engineering from the Indian Insititute of Technology, Bombay, and his Ph.D. degree from the University of Wisconsin—Madison in 1987. He was on the faculty at the University of Michigan—Ann Arbor till 1999, where he founded the Software Systems Research Laboratory. His current research spans the areas of Networking, Distributed Systems, Security, and Databases.

List of Figures

List of Tables