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 or Internet Service Providers, 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 here 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 begin by presenting a troubleshooting methodology for coordinating problem diagnosis. We then describe GDT, a distributed protocol which realizes this methodology. We show through simulation that GDT performs well as the number of observers and problems grows, and continues to function robustly amidst heavy packet loss.

1 Introduction

Work to date in network management has concentrated on effectively managing a single network. Typically, it has also been assumed that the management software and the managed devices are all owned 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].

In one recent informal study of routing instability [5], it was found that while the majority of catastrophic routing problems could be identified as software and configuration errors, about 10% of the problems could only be classified as "somebody else's problem", since all parties questioned pointed to another party as the cause. Such problems are the most difficult to resolve, and underscore the need for inter-domain coordination, so that the true causes of problems may be identified and such circular referrals detected and resolved.

A second important point is that the true cause of a problem may be distant from its effect. For example, the failure to access a web page may be the result of a problem located anywhere between the user's browser and the remote server. Contacting one's local help desk is unlikely to be of much benefit in this case.

We will investigate the problem of such inter-domain troubleshooting coordination. Our ultimate goal is to automate coordination of troubleshooting and repair efforts between administrative domains. In this paper, however, we 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.

In this paper, we will address the issues of reporting network outages and other problems, and acquiring feedback about them, but not the problem of *negotiating solutions*. We leave inter-administration negotiation as an opportunity for future work. We also leave the issue of *pre*-notification, or notifying organizations of downtime scheduled in the future, to other mechanisms. While some work currently in progress, such as IPN [6], addresses the issue of pre-notification, we observe that pre-notification does not solve the problem of coordinating troubleshooting during a problem, since past announcements (if any) may have been lost, ignored, or forgotten, and may be inaccessible during the problem.

---

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

Many of the assumptions made by intra-administration management methods do not apply to the inter-administration case. For example, management entities in one domain may not be allowed to access information in another administrative domain, and management entities in one domain cannot assume that observers in other domains are trustworthy. Hence, new mechanisms are needed to address new constraints.

Our task can be made easier by building inter-administration coordination on top of the existing management functionality available within each administrative domain. Given such functionality, we 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 upon which we build. They apply traditional management techniques, usually within an administrative domain. Example domain-expertise modules include existing Network Management Systems (NMS's), connectivity diagnosis tools such as CT [7], up-down and congestion experts as used by ANM [8], and agents employing newer techniques such as Anomaly Signature Matching [9].
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. Section 3 presents a methodology for diagnosing problems. Section 4 describes our protocol, Section 5 gives simulation results, and Section 6 covers conclusions and the future.

2 Design Philosophy

In this section, 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 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 all those affected by it.*

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 able 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. In addition, 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 having to react to each effect separately. The second part implies that 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 [10]. We summarize this philosophy 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 now adapt these constraints to the problem of network troubleshooting as discussed below.

We paraphrase our foremost constraint (from Rose [11]) as follows:

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

It is instructive to contrast the reliability requirements of troubleshooting services with those of ordinary distributed services. Whereas "reliability" of distributed systems is measured under normal operating conditions, troubleshooting must be reliable precisely when the operating conditions are harsh.

This Reliability 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.*

This means that multiple (perhaps very many) problems may exist simultaneously, and that the network spans multiple (perhaps very many) autonomous administrations. In addition, the network configuration may 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.*

In other words, the requirements for other entities to participate in the troubleshooting framework must be as simple as possible. In addition, 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.*

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

3 Troubleshooting Methodology

In this section, we describe the methodology we use to coordinate troubleshooting efforts. In the remainder of this paper, the term *object* will refer to 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

To understand how to coordinate troubleshooting efforts between administrative domains, we must first understand how faults propagate through the network. A good introduction to fault propagation can be found in [12], which describes how faults propagate both vertically, as well as horizontally through the network. For the horizontal directions, 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 [13].

To expand upon this notion, we propose the use of *resource dependency graphs*. We first use a resource dependency graph to describe the network, in which each node represents an object, and directed edges denote dependencies. Figure 1 shows one such example, depicting both vertical and horizontal dependencies. In such graphs, the arrows denote the direction of a *demand* for the resources of one object by another. For example, 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's performance thus depend on the extent to which its resource remands are met, and hence on the efficiency and correctness of the performance of those objects below it and downstream from it.

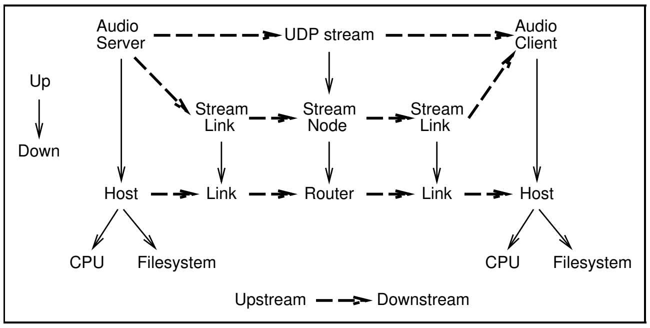


Figure 1: Sample Resource Dependency Graph

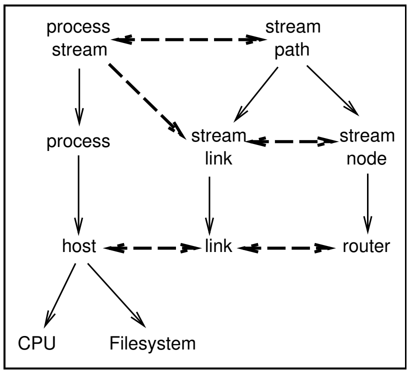


Figure 2: Static Resource Dependency Graph

When logically constructing such a resource dependency graph, it is important to distinguish between *static* relationships, and *dynamic* relationships. Relationships between *classes* of objects are typically static (see Figure 2), 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.

We now define several additional terms to better understand the relationship between fault propagation and resource dependencies. 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.

Finally, we adopt the concept of a "health function" from [14]. 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. Other literature in the field recognizes two different categories of low health. *Hard* failures (or complete lack of service) result when an element fails completely. *Soft* failures, on the other hand, represent only partial failures, such as degraded performance due to congestion. Our framework encompasses both notions, since zero health represents a hard failure, while healths in the range (0,1) represent soft failures of various degrees. This is important, since an observer may not be able to distinguish between the two, and our definition allows us to coordinate information relating to both fault management and performance management, as defined by the ISO [13].

Our methodology 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.

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, i.e., downwards and downstream.*

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 opposite direction from resource dependencies, i.e., upwards and (perhaps) upstream.*

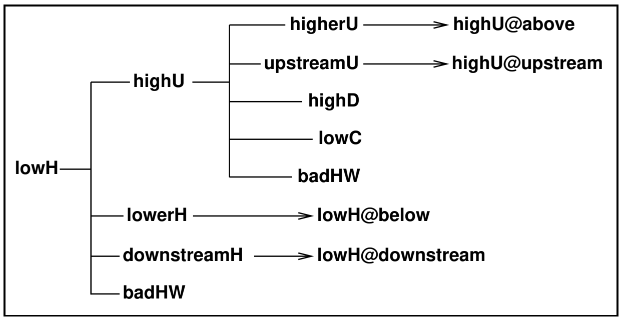
Degraded performance at some object may cause a degradation in performance of all objects which impose resource demands on it. Often, low health will not in fact propagate upstream since the degraded performance will merely result in lost packets which will be dealt with at a higher layer.

Observation 3 *High utilization can cause low 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).

3.2 Cause-effect graphs

Previous studies (e.g., [15]) 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 a cause-effect graph represents a problem, and directed edges lead from effects to causes. We begin with a taxonomy of problem types based on our discussions in Section 3.1. This taxonomy is shown in Figure 3.



`lowH`	health is too low
`highU`	utilization is 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 itself is demanding too many resources
`lowC`	capacity is too low to meet normal demands
`badHW`	incorrect performance of object itself

Figure 3: Problem Taxonomy

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, 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 4 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 5). 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.)

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.

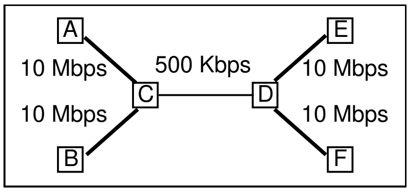


Figure 4: Sample Topology

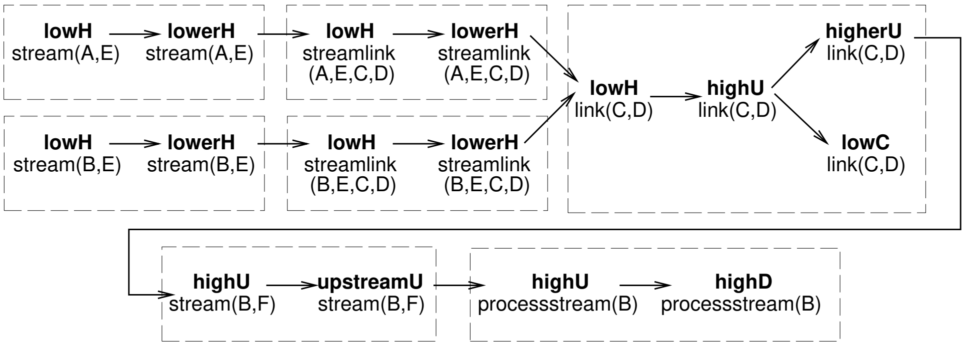


Figure 5: Cause-Effect Graph of Problems

The nine problem types shown in Figure 3 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 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 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 4.3.

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

Figure 6 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 3. 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.

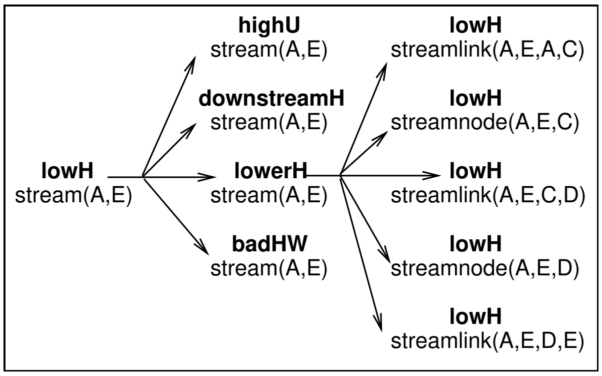


Figure 6: Generating Hypotheses

This process continues until all branches are pruned back or reach either root causes or previously-confirmed problems. Figure 5 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.

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. These agents are called *experts*, and 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 capabilities 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 area 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.

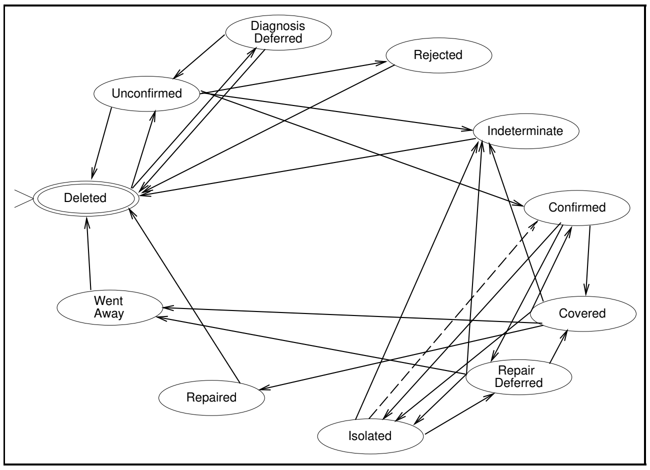


Figure 7: State Transition Table for a Problem

Each expert keeps a list of unresolved problems within its areas of expertise which have been reported to it. The state transition table seen by each problem (i.e., each node in the cause-effect graph) is shown in Figure 7. State transitions correspond to timer expirations and receipt of protocol messages. Details can be found in [16].

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 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 be used in diagnosing 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[17] 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. This means that the work of locating particular experts is wasted if we then find that we cannot reach them. 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 [18], X.500 [19], 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., [20]) 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 expert location server is responsible for knowing about namespace regions (areas of expertise) in the subtree rooted at itself. While we do not describe the hierarchy formation procedure in this paper, we anticipate that it will be formed as the result of a self-configuring process. This process is one area of current research. A trivial heuristic would be to manually configure the hierarchy, as is done with DNS and X.500.

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 local leaf server. Each server then reports to its parent server, 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. Bowman, et al. [21] describe a framework for reasoning about naming systems and describe how ordering rules can be expressed in terms of a "preference hierarchy". The constraints listed above lead us to the following preference hierarchy for ordering the list of experts.

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 [22]. 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.

Information distribution schemes may follow either a "push" model, where new information is sent out to everyone, or a "pull" model, where information is sent out only on request. We concluded that expert location service best fits the pull model due to our constraints. For example, we do not want the world to be flooded with messages conveying areas of expertise, as might be the case if the entire database were replicated at numerous sites. We also do not want the world to be flooded with requests for experts, as is done in the Contract Net protocol [23]. In addition, we did not want to require multicast capability due to the Reliability constraint; this decision allows us to diagnose problems with multicast routing, while not precluding the use of multicast as an optimization.

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 [24]. 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 [24, 25] 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 perhaps 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.

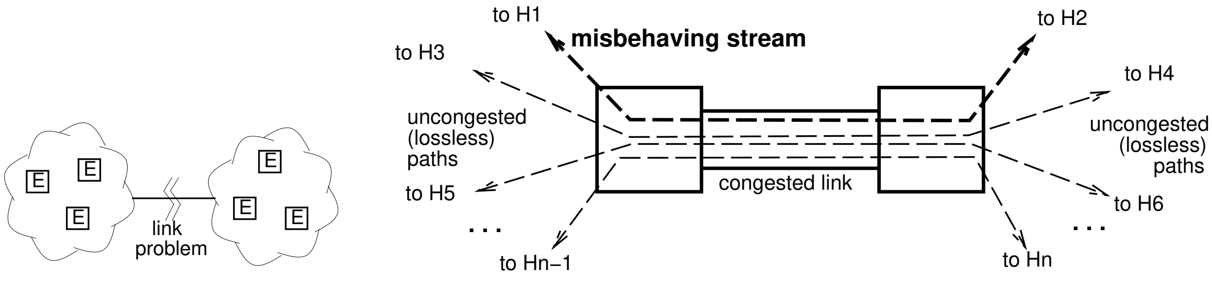
4.3 Protocol Overview

In this section we give a brief overview of the GDT protocol. A detailed specification can be found elsewhere [16].

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 with appropriate areas of expertise 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. Such a test may consist of any of: a simple table lookup if information were periodically polled and kept in memory, an automated test, or alerting an operator to request a manual test. 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



(a) Network Partition

(b) Simulated Congestion Problem

Figure 8: Network Problems

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.

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.

5 Simulation

In this section, we evaluate the performance of the GDT protocol. To do this, we implemented GDT clients and experts using "`ns`", the LBNL Network Simulator [26]. Our goal will be to test its performance and reliability under conditions particularly adverse to GDT.

It is first useful to understand the effects on GDT of hard and soft failures. If a hard failure is present, the network may be partitioned, potentially making some experts unreachable (Figure 8(a)). In this case, if any experts able to diagnose the problem and supervise repairs are reachable, then troubleshooting will succeed. If no reachable experts exist, then repairs will be requested at effects whenever possible, and either the leaf effects' problems will be repaired or nothing further can be done. (This is less likely if the client itself has areas of expertise). In any case, the situation is relatively time-invariant compared with a soft failure. Most of the Internet is also "self-correcting" since the routing protocols automatically adapt (in time) to hard failures, masking them from higher layers. In

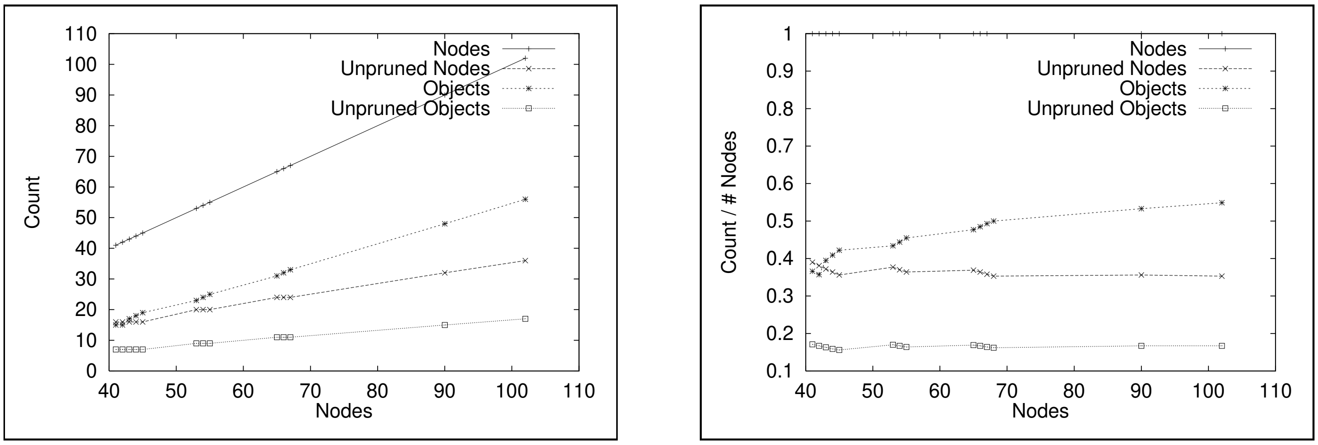


Figure 9: Cause-Tree Statistics

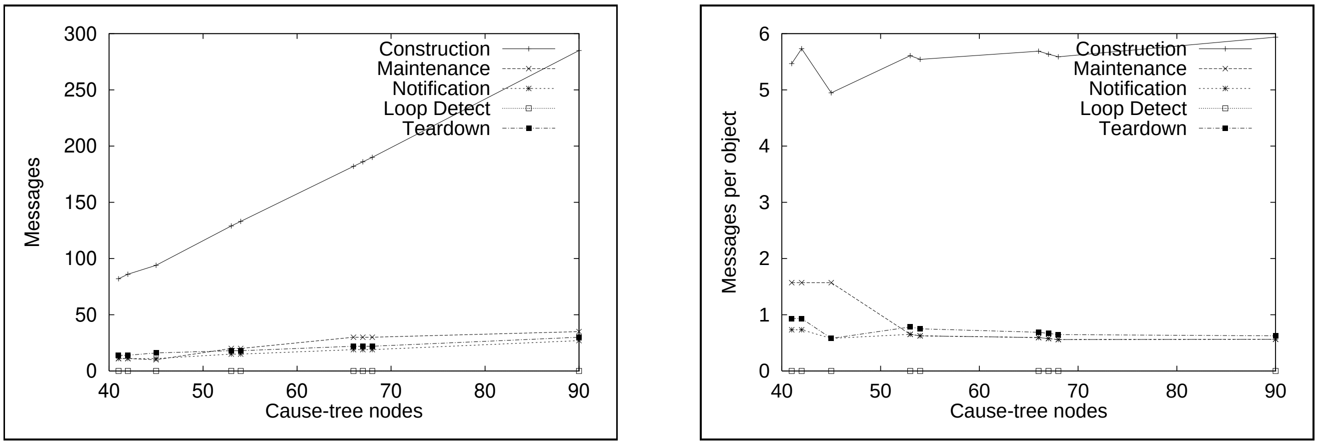


Figure 10: Messages Sent

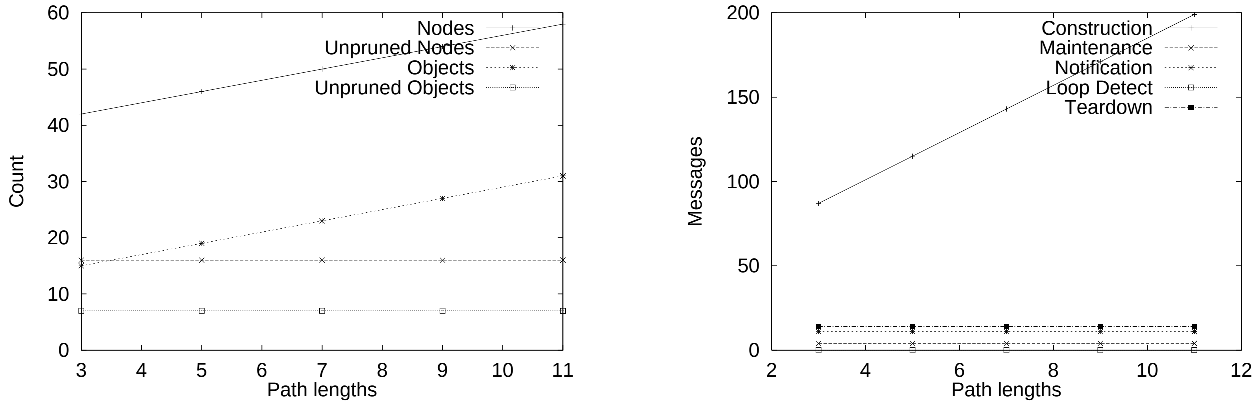
our methodology, this is equivalent to automatically modifying the dynamic resource dependency graph such that objects do not depend upon failed nodes or links. This means that, unlike effects of soft failures, many symptoms of hard failures will go away after some convergence time without the need to 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 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.

We first place one "misbehaving" stream between one host in each cloud shown in Figure 8(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 8(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 9 summarizes various characteristics of the resulting cause-effect graphs, showing both the absolute values as well as 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 10 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)



(a) Total State Generated

(b) Total Messages Generated

Figure 11: Effects of Path Length

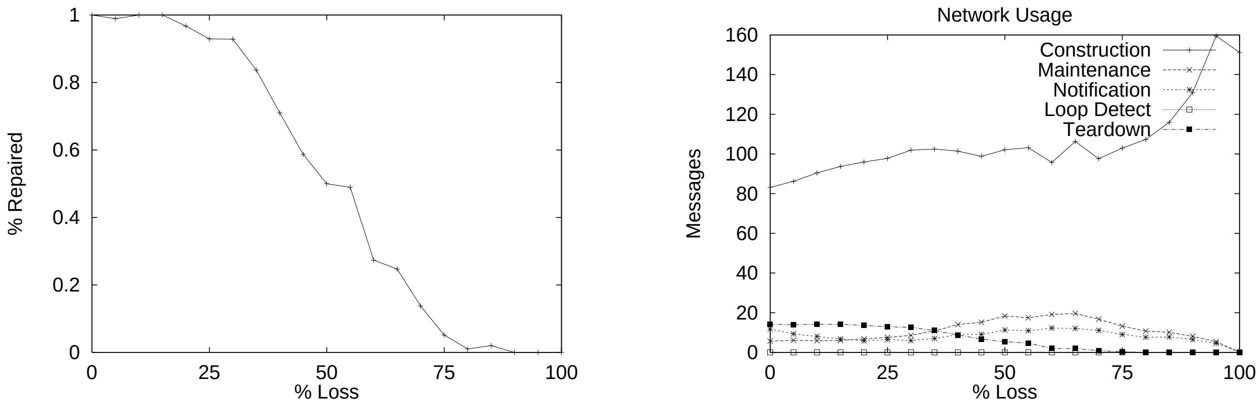
exchanged to construct the distributed cause-effect graph. Maintenance messages are periodic retransmissions to keep state alive. Notification messages relay important information (such as expected time to repair) back to reporters. Loop detect messages are used to detect and resolve loops in the cause-effect 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 by different experts, all messages would be transmitted over the network. Thus, Figure 10 represents the *worst*-case scenario. When all objects are assigned to the *same* expert, the number of actual network messages (between the clients and the expert) sent is only about 5 to 10 messages total per reporter. In each case simulated, the cache of experts' areas of expertise was warm. 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 11(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 11(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 RED [27]). 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 12 and 13 give the results of this simulation, with each point representing an average over 100 trials. Figure 12(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.



(a) Probability of Repair

(b) Resource Usage

Figure 12: Effects of Loss

Figure 12(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 13(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 13(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 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 timeout intervals dominate the resolution time.

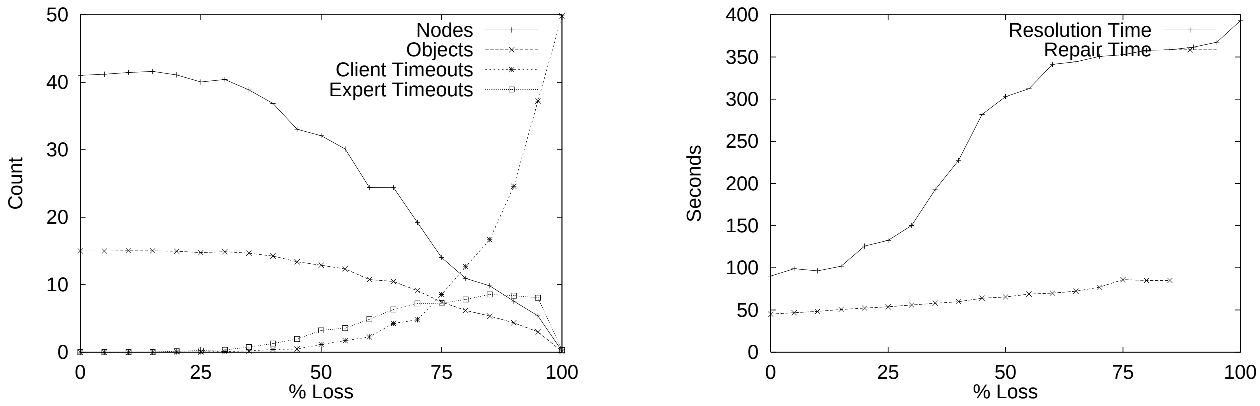
6 Conclusions

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 troubleshooting methodology for coordinating problem diagnosis under these contraints. We then described a distributed protocol, GDT, which realizes our methodology, and showed through simulation that it performs well as the number of observers and problems grows, and continues to function amidst heavy packet loss.

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.

In the future, we will investigate techniques for having expert location servers organize themselves into a hierarchy, and explore whether the architecture may be easily expanded to allow negotiating repairs. We also plan to develop optimizations to make GDT less bursty and hence see lower packet loss across drop-tail routers.



(a) Cause-effect graph statistics

(b) Reaction Time

Figure 13: Effects of Loss

References

[1] Shri K. Goyal and Ralph W. 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. *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] Metin Feridun. Diagnosis of connectivity problems in the internet. In *Integrated Network Management, II.* IFIP TC6/WG6.6, April 1991.

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

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

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

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

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

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

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

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

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

[17] 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.

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

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

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

[21] Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning about naming systems. *ACM Transactions on Programming Languages and Systems*, 15(5):795–825, November 1993.

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

[23] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *ACM Transactions on Computers*, pages 1104–1113, December 1980.

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

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

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

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