Proceedings of 2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

AUTO-DIAGNOSIS OF FIELD PROBLEMS IN AN APPLIANCE OPERATING SYSTEM

Gaurav Banga

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Auto-diagnosis of field problems in an appliance operating system

Gaurav Banga
Network Appliance Inc.,
495 E. Java Drive, Sunnyvale, CA 94089
gaurav@netapp.com

Abstract

The use of network appliances, i.e., computer systems specialized to perform a single function, is becoming increasingly widespread. Network appliances have many advantages over traditional general-purpose systems such as higher performance/cost metrics, easier configuration and lower costs of management.

Unfortunately, while the complexity of configuration and management of network appliances in normal usage is much lower than that of general-purpose systems, this is not always true in problem situations. The debugging of configuration and performance problems with appliance computers is a task similar to the debugging of such problems with general-purpose systems, and requires substantial expertise.

This paper examines the issues of *appliance-like* management and performance debugging. We present a number of techniques that enable *appliance-like* problem diagnosis. These include *continuous monitoring* for abnormal conditions, diagnosis of configuration problems of network protocols via *protocol augmentation*, path-based problem isolation via *cross-layer analysis*, and *automatic configuration change tracking*. We also describe the use of these techniques in a problem auto-diagnosis subsystem that we have built for the Data ONTAP operating system. Our experience with this system indicates a significant reduction in the cost of problem debugging and a much simpler user experience.

1 Introduction

The use of network appliances, i.e., computers specialized to perform a single function, is becoming increasingly widespread. Examples of such appliances are file servers [24, 6], e-mail servers [19, 13], web proxies [25, 5], web accelerators [25, 5, 16] and load balancers [4, 12]. Appliance computers have many potential advantages over traditional general-purpose systems, such as higher performance/cost metrics, simpler configuration and lower costs of management. With the recent growth in the use of networked systems by the non-expert, mainstream population, all of these advantages have significant importance.

A network appliance is typically constructed using off-the-shelf hardware components. The appliance's service is implemented by custom software running on top of a specialized operating system. (Often the server software is tightly integrated with the OS in the same address space.) The operating system itself is either designed and constructed from scratch, e.g., Network Appliance's Data ONTAP [26], or is a stripped-down version of a general-purpose operating system, e.g., BSDI's Embedded BSD/OS [8].

While appliance computer systems have delivered the promise of higher performance/cost vis-a-vis general-purpose systems, the same is not strictly true of their manageability aspects. While the complexity of configuration and management of appliance computers in normal circumstances is significantly lower than that of general-purpose systems, the debugging of configuration and performance problems of appliances (when they do occur) remains a task that requires substantial operating system and networking expertise. In this respect, appliance systems are similar to general-purpose systems.

This state of technology is not very surprising: Today, the term "appliance-like" is usually taken to mean *specialized to do a single coherent task well*. Specialization of this form has allowed appliance vendors to build and maintain smaller amounts of code than used on general-purpose computer systems. The narrower functionality of appliances has enabled simpler configuration, and more aggressive optimizations leading to superior performance. The ability to easily debug configuration and performance problems has however been a secondary issue so far, and has not received much attention.

Appliance operating systems often contain significant code derived from general-purpose operating systems, particularly UNIX. For instance, the BSD TCP/IP protocol code [33] is a common building block in appliance operating systems. Like general-purpose systems, appliance operating systems export a set of command interfaces that allow users to display values of various statistic counters corresponding to the various events that have

occurred during the operation of the system. Some command interfaces display system configuration parameters. As with general-purpose systems, these command interfaces are the key tools to debugging performance and configuration problems with appliance systems.

For example, the TCP/IP code of many appliance systems exports its event statistics and configuration via a variant of the UNIX *netstat* command. When a person debugging a configuration or performance problem suspects a bug or problem in the network subsystem of the target appliance, she executes the *netstat* command (possibly multiple times with different options) and analyzes the output for aberrations from expected normal values. Any deviations of these statistics from the norm provide clues to what might be wrong with the system. Using these clues, the person debugging the problem may perform additional observations of the system's statistics, using other commands, followed by further analysis and corrective actions (such as configuration changes).

The fundamental problem with this style of statistic-inspection based problem diagnosis is the need for human intervention, and specialized networking and performance debugging expertise in the intervening human. For example, consider a workstation that is experiencing poor NFS [27] file access performance. Assume that the cause of this problem is excessive packet loss in the network path between the client and an NFS server due to a Ethernet duplex mismatch at the server. To diagnose this problem today, the person debugging the problem needs to first isolate the problem to the problematic server, then check the packet drop statistics for the transport protocol in use (UDP or TCP), and correlate these statistics with excessive values for CRC errors or late-collisions maintained by the appropriate network interface device driver[1]. After this, the problem debugger has to check the appropriate switch's configuration to verify the existence of a duplex mismatch.

For any organization engaged in selling and supporting appliance computer systems, it is very expensive to provide a large number of human experts with this level of expertise for the on-site debugging of customer problems. In the absence of sufficient numbers of human experts, problem FAQs, and semi-interactive troubleshooting guides are commonly used by customers and by the (mostly) non-expert customer support staff of the appliance vendors for diagnosing field problems.

Another limitation of this style of problem debugging is that field problems are usually detected *after* they occur. Problems are first detected by unusual behavior (e.g., poor performance) at the application level and then traced back to the cause by a human expert via an exhaustive search and pattern-match through the system's statistics. While there is usually a well-understood notion of *normal* and *bad* values for the various statistics, there exists no software logic to continuously monitor the statistics, and to catch shifts in their values from normal to bad. Problems (and resulting service outages) which could otherwise be avoided by taking timely corrective actions are not avoided.

For all of these reasons, the use of an appliance system can sometimes be a somewhat frustrating experience for a non-expert customer. The subject of this paper is the problem of enabling simple and easy, i.e., *appliance-like*, debugging of the field problems of appliances. We describe four techniques, *continuous statistic monitoring*, *protocol augmentation*, *cross-layer analysis* and *configuration change tracking*, that we have developed to make the diagnosis of appliance problems easier. We also describe the application of these ideas in an auto-diagnosis subsystem of the Data ONTAP operating system.

Specifically, continuous monitoring involves periodically checking the system's collected operational statistics for potential problems, while actively analyzing and fixing whichever problems it can. Protocol augmentation allows configuration problems with a network protocol to be diagnosed using specially constructed higher-level protocol tests. Cross-layer analysis is a path-based approach [23] for isolating a problem with a multi-layered system to a specific system layer. Automatic configuration change tracking keeps track of changes in the system's configuration making it easier to pinpoint a problem to its cause.

Our discussion in the remainder of the paper is set in the context of an appliance operating system. More specifically, we focus on problems that arise with file server appliance systems built and sold by Network Appliance. However, we believe that most of the ideas that we present are directly applicable to the space of general-purpose operating systems. Indeed, the class of field problems involving general-purpose computer systems is much larger than the class of appliance field problems because of the broader functionality and services offered by general-purpose systems. It is probably just as important (and useful) to provide for easier debugging of field problems with general-purpose systems as it is with appliance systems. Later in this paper, we will briefly outline how our auto-diagnosis techniques can be used in a general-purpose operating system, such as BSD.

The rest of the paper is structured as follows. In the next section, we discuss the nature of common field problems of appliance computer systems. In Section 3, we describe the four techniques that we have developed to diagnose such problems automatically and efficiently. In Section 4, we describe the implementation of the NetApp Auto-diagnosis System (NADS). Section 5 describes our experience with this auto-diagnosis system. Section 6

[1] Note that the duplex mismatch cannot be simply avoided as a configuration or installation time automatic check by the server's OS; the Ethernet protocol specification does not contain sufficient logic for an end-system to detect a duplex mismatch.

covers related work. Finally, Section 7 summarizes the paper and offers some directions for future work.

2 The nature of field problems with appliance systems

Before getting into the details of what can be done to make the debugging of appliance performance and configuration problems easier, it is important to understand the nature of field problems of appliance systems. In this section, we present an overview of the common causes of field problems of appliances and try to give the reader a sense of why it is hard to debug such problems.

As mentioned earlier, for the purposes of concrete illustration, we use the example of a file server (filer) appliance. A filer provides access to network-attached disk storage to client systems via a variety of distributed file system protocols, such as NFS [27] and CIFS [15]. A useful model is to think of a filer's OS as two high-performance pipes between a system of disks and a system of network interfaces. One pipe allows for data flow from the disks to the network; the other carries the reverse flow. Field problems usually arise when something in the filer or in its environment causes one (or both) of these pipes to perform below expected levels.

The taxonomy of common field problems that we describe below was obtained from a detailed study of the call records of Network Appliance's customer service database. We examined information pertaining to customer cases that were handled in the time period February 1994 through August 1999. From this data it appears that the three most important causes of field problems are system misconfiguration, inadequate system capacity and hardware and software faults. The relative ratio of these three problem types is hard to quantify because a large number of customer cases involve more than one subproblem of each type and because the specific mix has varied from month to month and from year to year. However, between these three problem types, they cover about 98% of all field problems.

2.1 Misconfiguration

A leading cause of field problems with network appliances is system misconfiguration. This may seem somewhat paradoxial since by definition an appliance is a simple computer system that has been specially developed to perform a single coherent task. This definition is supposed to allow an appliance system to be simpler to configure and use. In reality, appliances by themselves are usually much simpler than general-purpose systems. However, the task of making appliances work correctly in a real network in a variety of application environments may still have significant configuration complexity.

One major reason for the configuration complexity associated with a appliance system is that an appliance in use is only a part of a potentially complex distributed system. For example, the perceived performance of a filer is the performance of a distributed system consisting of a client system (usually a general-purpose computer system) connected via a potentially complicated network fabric (switches, routers, cables, patch panels etc.) to the filer. These components typically come from different vendors and need to be all configured and functioning correctly for the filer to function at its rated performance. Unfortunately, this does not always happen for a variety of reasons, as discussed below.

First, the client system usually has a fairly complicated and error-prone configuration procedure. The client's configuration complexity is much more so than the filer's because the client is a general-purpose system. Often, the default configurations in which most client systems ship are simply not set for optimal performance. (This issue of default configuration is discussed in somewhat more detail later.) In many cases, the configuration controls are too coarse for any allowable setting to result in good performance for all activities that the general-purpose client may be engaged in.

Second, while most components of the network fabric are appliances (and therefore presumably easier to configure than client systems), there are numerous potential incompatibilities between them. For example, it is not uncommon for implementations of network communication protocols from different vendors to not work with each other. Usually, the corresponding vendor documentation clearly states this incompatibility, but customers try to use the incompatible implementations anyway, and the result is a field problem.

Perhaps more importantly, some commonly used standard network protocols have serious inadequacies. For example, the Ethernet standard includes an *auto-negotiation* protocol for negotiating the link speeds of the communicating entities. The standard does not provide for reliable negotiation of duplex settings. As a result, perfectly legal configuration settings for link and duplex at two communicating endpoints may result in a duplex-mismatch, a misconfiguration whose effect on a filer's throughput is disastrous.

Furthermore, network components often use protocols that are vendor-specific or ad-hoc standards. These "early" protocols work well in most situations, but not at all (or poorly) in other circumstances. In the fast moving world of network technology, there are a fair number of ad-hoc, unstandardized, or incomplete protocols in wide use at any given time. An example of this is the EtherChannel link aggregation protocol. This protocol does not specify the algorithm for performing load balancing of network traffic between the links of the EtherChannel. Vendors have their own propriety methods for this process, often with surprising interactions with how the client systems and the rest of the network elements are set up. These interactions sometimes have a significant

effect on performance and result in field problems.

A second important cause of the configuration complexity associated with appliance systems is the sub-optimal management of configuration parameters. The appliance philosophy is to expose a very small number of configuration parameters at installation. There is a second tier of parameters that are assigned default values which result in good performance in the majority of installations. For some installations with atypical workloads, these settings may not be optimal. There is usually no automatic logic to tune these second tier parameters. In these cases, these knobs may require tuning by an expert for good performance.

With the widespread increase in the variety and number of appliance users, this atypical population can become a significant overall number, potentially resulting in a large number of field problems. This problem of configuration parameter management also exists with general-purpose operating systems, including systems that are used as clients for filers. In fact, with general-purpose systems, a large number of parameters often need to be tuned for a typical user environment.

2.2 Capacity problems

A second class of field problems with appliance systems arise because of their poor handling of capacity overloads[2]. Most commonly-used general-purpose operating systems, and many appliance operating systems, perform well when the request load to which the system is being subjected lies within the capacity of system, but poorly when the offered load exceeds the capacity of the system [20, 7]. Historically, the problem of poor overload performance of computer systems is well known, but has been deemed of somewhat marginal importance. In most circumstances it is not desirable to operate a system under overload conditions for any length of time; instead, the focus so far has been to avoid overload by trying to ensure that there are always sufficient hardware resources available in order to handle the maximum offered load.

In the filer appliance market, systems are often purchased by customers with a certain client load in mind. The number and types of systems purchased is chosen based on rated capacities of the filers, by in-house benchmarking, or from knowledge based on prior-use of filers. Filers are usually assigned rated capacities based on their performance under some standardized benchmark, e.g. the SpecFS (SFS) benchmark [30]. For many customers' sites, however, the request load profile is significantly different from the SFS profile, and the real capacity of a filer in operation may be very different from its rated capacity. When offered load does exceed real capacity, poor performance and a field problem results.

[2] We use the term "capacity overload" to refer to a broad class of situations where the server system cannot handle the full client request load that it is subject to because of some system resource that is not available in sufficient quantity. These resources include "soft" system resources such as memory buffers, file system buffers etc.

2.3 Hardware and software faults

Last but not least, some field problems with appliances occur because of software and hardware faults. Unlike the other causes of field problems discussed above, faults are the result of some bug in the system's implementation, and usually result in system down-time. For a mature system made by a technically sound organization, the number of field disruptions due to faults should be very small.

Field problems due to faults are not discussed further in this paper. The techniques that we describe in this paper to enable easy debugging of field problems may have some applicability to diagnosing certain types of field disruptions due to faults, but in this paper we restrict our focus to diagnosing configuration and capacity problems.

2.4 Why are field problems hard to debug?

When a field problem occurs with an appliance system due to any of the reasons described above (except faults), it is often hard to debug. Consider a filer customer who observes performance that is substantially lower than the filer's rated performance. The reason for this poor performance may be a misconfiguration somewhere in the client-to-filer distributed system, i.e., in the client, in the filer, or in the network fabric. Alternately, the problem may be an overloaded filer; this particular environment may have an atypical load and the filer may have a lower capacity for this workload than for the standard SFS workload.

As the end effect of all of these potential causes is usually the same, i.e., poor file access performance as seen from the client system, it is not easy to discern the exact cause of the problem. The problem debugger is forced to perform a sanity check of *all* the components of the client-to-filer distributed system in order to ensure that each component is functioning correctly. For the filer, this implies a verification of all filer subsystems performed by invoking the various statistic commands and analyzing the output for aberrations.

This process is time-consuming, tedious and error-prone. As explained earlier, this task requires a fair amount of expertise, and a certain debugging "instinct" that comes from experience. This task is also complicated by the fact that the person debugging the field problem, being a member of the filer vendor's organization, often has no direct access to the system being debugged. In that case, the various statistic commands are executed by the customer who is in communication with the support person via email or phone. This aspect of the problem debugging process makes it slow, causing large down-time. Combined with the high expectations of *appliance-like* simplicity that most appliance

customers have, it makes the problem debugging experience frustrating for both parties involved, the customer and the support person.

The discussion above is fully applicable to general-purpose systems; appliances are usually considerably easier to debug than general-purpose systems. However, the debugging of field problems with appliances is certainly not as simple, or "appliance-like", as we would like. In the next section, we will present a new problem diagnosis methodology that attempts to apply the appliance to the debugging of field problems with appliance systems.

3 Problem auto-diagnosis methods

In this section, we describe a new methodology that we have developed to make the diagnosis of appliance field problems simpler. Our goal in designing this methodology was to enable problem diagnosis to be as automatic, precise and quick as possible. We wanted to eliminate the need for expert human intervention in the problem diagnosis process whenever possible. Furthermore, for those situations where expert manual analysis is necessary, we wanted to provide powerful debugging tools, precise and comprehensive system configuration (and configuration change) information and the results of partial auto-analysis to the human expert, allowing for fast diagnosis and smaller down-times.

Our problem diagnosis methodology is based on four specific techniques, i.e., continuous monitoring, protocol augmentation, cross-layer analysis and configuration change tracking. Each of these techniques is described in detail below. In this section, we will focus on the fundamental principles underlying these techniques; the next section will contain specific details about the application of these techniques in the auto-diagnosis subsystem of the Data ONTAP operating system.

We will also briefly discuss issues related to the extensibility of our new problem diagnosis methodology. This feature is important for the problem auto-diagnosis system to be maintainable in the field.

3.1 Continuous monitoring

As described in Section 1, current appliance operating systems maintain a large number of statistics. To help in auto-detecting and diagnosing problems, we have developed a method of continuous statistic analysis layered on top of this statistic collection procedure. Software logic in the appliance system continuously monitors the system for problems, actively analyzing and fixing whatever problems it can. Continuous monitoring has two components to it, a passive part and an active part.

The passive part of continuous monitoring is a statistic monitoring subsystem of the appliance's operating system. This subsystem periodically samples and analyses the statistics being gathered by the operating system. It automatically looks for any aberrant values in these statistics and applies a set of predefined rules on any aberrations from expected "normal" values to move the system into one of a set of error states. For example, a filer may continuously monitor the average response time of NFS requests. A capacity overload situation is flagged when the response time exceeds a high-water mark.

Some abnormal system states may correspond uniquely to specific problems; other states may be indicative of one of a set of possible problems. In the latter case, the continuous monitoring subsystem may also automatically execute specially designed tests in order to pin-point the specific problem with the system. This is the active part of continuous monitoring. For example, a large number of packet losses on a TCP connection at a filer may be indicative of, among other problems, a duplex mismatch at one of the filer's network interfaces or a high level of network congestion in the path from the relevant client to the filer. We can use the techniques described below in sections 3.2–3.4 to differentiate between these problems.

Making continuous system monitoring viable involves the following:

- Development of software logic that formally codifies the informal notion of *expected* statistic value. This activity must be performed for all of the statistics that are gathered by the system. The end-result of this activity is a set of equations that test the state of the system and return either "GOOD" or move the system into an "ERROR" state.
- Development of software logic that selects an appropriate problem pin-pointing procedure when one of several problems is suspected based on observations of aberrant system statistics.
- Development of formal procedures for pin-pointing common field problems of appliances.

Formally codifying the notion of expected values of the various statistics is a hard problem. This is because, in general, the normal values of the various system statistics and the relative sets of values that indicate error conditions depend on how a particular system is being used. For example, an average CPU utilization of 70% might be OK for a system that is usually not subject to bursts of load that greatly exceed the average. This may, however, be a big problem for a system whose peak load often exceeds the average by large factors.

To make the development of this logic tractable, it may be necessary to be somewhat conservative in the choice of the specific problems to be characterized. For any particular appliance, this logic can start from being very simple, codifying only the most obvious problems initially, and move towards more complex checks as the

appliance's vendor gains experience with how the appliance is used in the field. At any point in an appliance's life-cycle, there will be some logic that can be completely automatically executed and its results presented directly to the customer/user. Other, more complex logic may attempt to perform partial-analysis and make these results available to a support person looking at the system, should manual debugging be necessary. Still more complicated analysis may be left to the human expert.

The idea behind developing active tests for pin-pointing problems is to try to mimic the activity of problem analysis by a human expert. While debugging a field problem, this person may take a certain set of statistic values as a clue that the system is suffering from one of a certain set of problems. He may then execute a series of carefully constructed tests to verify his hypothesis and pin-point the exact problem. Continuous monitoring with active tests attempts to mimic this debugging style.

The algorithm development activity for active tests motivates the next three techniques, i.e., protocol augmentation, cross-layer analysis and configuration change monitoring that we describe below. The software logic to trigger these tests is usually straightforward, once the main logic of continuous monitoring is in place.

Of course, the continuous monitoring logic has to be lightweight. It should work with as few system resources as possible and should not impact system performance in any noticeable way. The active component of system monitoring should not affect the system's environment, e.g., the network infrastructure to which it is attached, in any adverse manner. We will discuss some practical aspects related to the user-interface of the continuous monitoring subsystem in the next section.

Once continuous monitoring is in place, it has many benefits. A sizable fraction of field problems can be auto-diagnosed without intervention of the support staff. If expert intervention is needed, all information that is normally gathered by a human expert after (potentially time-consuming) interaction with the customer is already available. Changing system behavior that slowly moves the system towards an ERROR state may be detected early, and corrected, before it results in down-time. For example, increasing average load that slowly drives a system into capacity overload can be auto-detected.

Similarly, other shifts in a system's environment, such as the load mix to which it is subjected, may be auto-detected and suitable action may be initiated. Continuous monitoring may also help an appliance vendor in tuning his product better because he now has access to more detailed information about the various customer environments in which the product operates. In essence, continuous monitoring is like having a dedicated support person attached to every appliance in the installed base, but at a very small fraction of the cost.

3.2 Protocol augmentation

The technique of protocol augmentation refers to the process by which a higher-level protocol in a stacked modular system configures and operates a lower-level protocol through a series of carefully chosen configurations and operating loads. The goal of protocol augmentation is to determine the optimal configuration of the lower-level protocol when it is impossible to determine this setting within the protocol itself. This is necessary because the lower-level protocol is either *inadequate*, *incompletely specified* or if one of the communicating entities has a *broken* protocol implementation.

As briefly mentioned in the previous section, some network protocols are *inadequate* in that it is impossible to detect configuration problems of the communicating entities within the protocol itself. An example of this is Ethernet auto-negotiation, which does not always allow for the correct negotiation of the duplex settings of the communicating entities.

Some network protocols are *incompletely specified*. For instance, the algorithms for congestion control were not specified as part of the original TCP protocol standard. Congestion control was incorporated by most TCP implementations much later from a de-facto standard published by the researchers who developed these algorithms. Often, such de-facto standards involve areas of the protocol that are not necessary for correctness, and are therefore not enforced. A TCP implementation that does not perform congestion control correctly may still be able to communicate adequately with other TCP implementations; however, correct congestion control is imperative for system-wide stability and performance.

A number of protocol implementations, especially where unofficial de-facto standards are involved, are *broken*. For example, some commonly used auto-negotiating Gigabit Ethernet devices detect link only if the peer entity is also set to auto-negotiate.

When a problem occurs due to any of the three reasons mentioned above, the continuous monitoring subsystem detects this situation and flags an error condition. If an active test has been associated with the equations that triggered this error state, this active test is executed. The active test will use protocol augmentation to mimic a human expert in the debugging process. For example, a test designed to detect an Ethernet duplex mismatch may try all legal settings of speed and duplex coupled with initiation of carefully constructed Ethernet traffic. It may analyze the resulting change in system behavior to determine the correct settings for speed and duplex.

Protocol augmentation is a powerful technique that can be used as a guiding framework to formalize many ad-hoc problem debugging techniques used by human experts. Any manual debugging technique that involves a series of steps where network configuration changes alternate with functionality or performance tests (to val-

idate the configuration) is really a form of protocol augmentation. Using this technique as a design guide, we can come up with problem diagnosis procedures that are more precise and systematic than the ad-hoc techniques normally used in manual diagnosis. In the next section, we will describe some examples of the use of this technique in designing automatic problem diagnosis tests for commonly occurring filer problems.

3.3 Cross-layer analysis

Many subsystems of appliance operating systems are implemented as stacked modules. For example, the TCP/IP subsystem consists of the link layer, the network layer (IP), the transport layer (TCP and UDP) and the application layer organized as a protocol stack. Each layer of a stacked set of modules maintains an independent set of statistics for error conditions and performance metrics. When a problem occurs, it may manifest itself as aberrant statistic values in multiple layers in the system. In classical systems, there is no logic that correlates these aberrant statistic values across different system layers.

Cross-layer analysis is a new technique whereby statistic values in different layers of a subsystem are linked together, and co-analyzed. Essentially, we identify paths [23] in OS subsystems and link together the statistic values in the various layers that each path crosses. When continuous monitoring detects a problem in a path, the various layers of the path can be quickly examined to isolate the specific problem.

As a debugging technique, cross-layer analysis is a formalization of the ad-hoc technique used by human experts in manual problem debugging where an observation of an aberrant statistic value in one layer triggers a study of the statistic values of an adjacent layer. Considering the pipeline analogy of an appliance operating system, cross-layer analysis guides the debugging process by tracing through and ensuring the health of the various layers that implement the disk-to-network pipes.

As a guiding framework, cross-layer analysis can aid the design of logic that causes the continuous monitoring subsystem to trigger the various active tests. For example, the logic to perform a check for a duplex mismatch on a network interface may be triggered by an observation of excessive TCP level packet loss in a connection that goes through this interface. Cross-layer analysis can also guide the design of the statistic data and its collection logic so as to allow problem debugging to be easier. For example, the need to do cross-layer analysis may require a modification of the BSD *tcpstat* and *ipstat* structures so as to keep some statistics on a per flow basis.

3.4 Automatic configuration change tracking

Many field problems with appliance systems are caused by changes in the system's environment. These include system configuration changes and changes in the offered load. As described earlier, there is a lot of value in continuous monitoring of system statistics to notice shifts in metrics like average system load. Likewise, it is useful to track changes in the system's configuration, both explicit as well as implicit.

Automatic tracking of configuration changes is useful in finding the cause of appliance problems that occur after a system has been up and running correctly for some time. This technique also helps in prescribing solutions for the problems found by other auto-diagnosis methods. In many organizations, there are multiple administrators responsible for the IT infrastructure. Configuration change tracking allows for actions of one administrator that result in an appliance problem to be easily reversed by another administrator. This is also useful where administrative boundaries partition the network fabric and the clients from the filer.

The fundamental motivation behind automatic configuration change tracking is to automatically gather information that is asked for by human problem debuggers in a large majority of cases. Anyone familiar with the process of field debugging probably knows that one of the first questions that a customer reporting a problem gets asked by the problem solving expert is: "What has changed recently?" The answer to this is often only loosely accurate (especially in a multi-administrator environment), or even incorrect, depending on the skill level of the customer/user. Automatic configuration change tracking makes precise and comprehensive state change information available to the problem solver, i.e., the auto-diagnosis logic or a human expert.

Configuration changes are tracked by a special module of the appliance OS. As hinted above, configuration changes are of two types: the first type of changes are explicit, and correspond to state changes initiated by its operator. The second type of changes are implicit, e.g., an event of link-status loss and link-status regain when a cable is pulled out and re-inserted into one of a filer's network interface cards. The system logs both explicit and implicit changes. The amount of change information that needs to be kept around is a system design parameter, and may require some experience in getting to optimal for any particular appliance.

Given comprehensive configuration change information, when a problem occurs the various events between the last instance of time which was known to be problem free to the current event are examined and analyzed. The software logic to do this analysis, like the logic for continuous monitoring, is system specific and may need to be evolved over time. In some cases, the auto-diagnosis system can directly infer the cause for the field problem, and report this. In other cases, the set of all applicable configuration changes can be made available to the human debugging the system.

Note that it is not absolutely imperative to log *all* relevant configuration change information. (In fact, some

configuration changes may not be easily visible to the appliance. For example, the path between a client and a filer may involve multiple routers, and it may be possible to change/re-configure one of these without the filer noticing any changes in its environment.) State change information is however only a set of hints that guide the problem diagnosis process and make it easier. If some relevant state change information is not logged, diagnosing the cause of a specific problem may become harder, but not necessarily impossible. In our experience, logging even a modestly-sized, carefully chosen set of configuration change information, is extremely valuable in the problem diagnosis process.

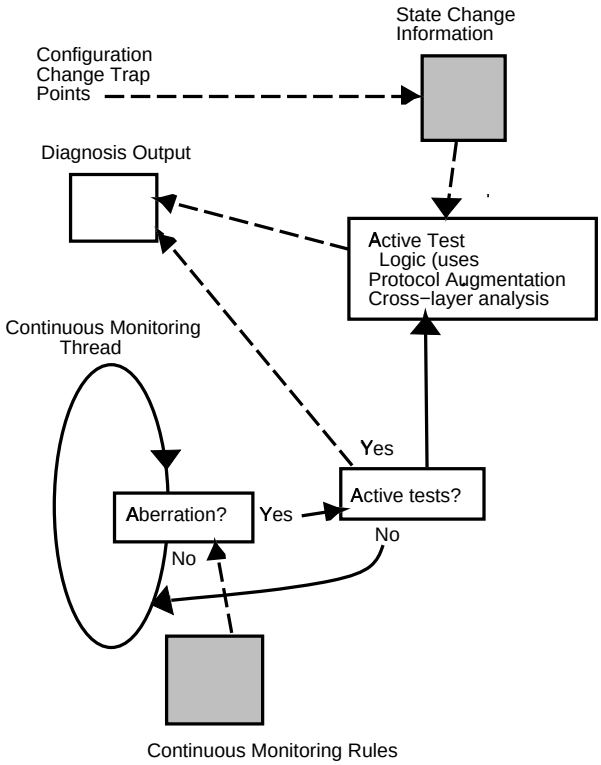

Fig. 1: Role of different auto-diagnosis techniques.

Figure 1 shows the role of the various auto-diagnosis techniques in the problem diagnosis process. In the figure, dashed lines indicate flow of data while solid lines indicate flow of control. The shaded rectangles indicate stores of data or logic rules. The unshaded rectangles indicate processing steps. Note that the problem diagnosis process uses all the techniques we described above. The techniques are complementary and designed to work with each other; they are *not* different types of procedures targeted to address disjoint problem sets.

3.5 Extensibility issues

It is important for an auto-diagnosis system built around the techniques described above to be extensible. As explained above, the checks and actions performed by the continuous monitoring logic need to be developed in a phased and conservative manner. Each time a new version of this logic is available, a vendor may want to upgrade the systems in the field with this logic, even if the customers do not wish to upgrade the rest of the system. A customer may not wish to take on the risk associated with a new software release, or may not want to pay for the release, especially if it does not contain any functionality that the customer needs. It is, however, usually in the vendor's interest to upgrade the auto-diagnosis logic because of the little associated risk and potential benefits of lower support costs.

For example, an appliance problem may have been first discovered at one customer's installation because of an environment change, e.g., the addition of a new model of some hardware in the network fabric. In some cases, significant effort by human experts may be required to debug this problem since it has not been seen before. Ideally, we would like to leverage off this effort by codifying the debugging logic used in this manual diagnosis into the appliance's auto-diagnosis logic and upgrading the auto-diagnosis subsystems of *all* the systems in the field. This may save a lot of time and effort by auto-diagnosing subsequent instances of this problem which would otherwise require significant human intervention.

Extensibility can be achieved in a variety of ways. One method is for the continuous monitoring system to use a configuration file containing equations that define the various periodic checks that the monitoring system performs and conditions that trigger the flagging of an ERROR state, or cause an active subtest to be executed. This requires a language to express the logic of the periodic checks, and an interpreter for this language to be part of the problem auto-diagnosis subsystem.

4 Implementation of the NetApp Auto-diagnosis System

We have implemented a semi-automatic problem diagnosis system, the NetApp Auto-diagnosis System (NADS), in the Data ONTAP operating system. This system applies the techniques described in the previous section to field problems with filers and NetCache appliances. Currently this auto-diagnosis system only targets problems related to the networking portion of Data ONTAP, and some of the interactions of this code with the rest of Data ONTAP. Extension of the auto-diagnosis system to other ONTAP subsystems is in progress.

An interesting social problem that we had to address while developing the auto-diagnosis system was to how not to make the auto-diagnosis logic intrusive. We did not want our expert customers to be turned-off by an overbearing problem diagnosis "assistant" and immediately disable the auto-diagnosis system. We also did not want our non-expert customers to be lead off on a sidetrack by a bug in the auto-diagnosis logic. For this rea-

son, we decided that we would make the auto-diagnosis process semi-automatic initially, and later, as both we and our customers gained experience with the system, make it fully automatic.

4.1 Core implementation

In its current form the NetApp Auto-diagnosis System consists of a continuous monitoring subsystem and a set of diagnostic commands. ONTAP's continuous monitoring logic consists of a thread that wakes up every minute and performs a series of checks on statistics that are maintained by various ONTAP subsystems. These checks may flag the system as being in an ERROR state. This logic is currently hard-coded into ONTAP (as C code tightly integrated into the kernel) and needs to be tuned with every maintenance release. Threshold values and most constants used by this logic are read from a file present root filesystem of the filer[3]. This logic does not yet perform any output for direct user consumption; nor does this logic execute any active tests. Instead this output is logged internally in ONTAP for consumption by the various diagnostic commands, which also execute any active tests that are needed. Since in ONTAP all commands are implemented in the same address space as the kernel, it is straightforward for the data gathered by continuous monitoring to be accessed by the diagnostic commands. Likewise, it is easy for the active test logic to be executed by the diagnostic commands.

When the customer or a support person debugging a field problem suspects that the problem lies in the networking portion of ONTAP, she executes the *netdiag* command. The *netdiag* command analyzes the information logged by the continuous monitoring subsystem, performing any active tests that may be called for and reports the results of this analysis, and some recommendations on how to fix any detected problems, to the user. Our plan is to have the computation of the various diagnostic commands be performed automatically after the next few releases of ONTAP.

The checks that ONTAP's continuous monitoring system performs and the various thresholds used by this logic have been defined using data from a variety of sources of collected knowledge. These include FAQs compiled by the NetApp engineering and customer support organizations over the years, troubleshooting guides compiled by NetApp support, historical data from NetApp's customer call record and engineering bug databases, information from advanced ONTAP system administration and troubleshooting courses that are offered to NetApp's customers, and ideas contributed by some problem debugging experts at NetApp.

The specific monitoring rules and the values of various constants and thresholds used by the monitoring logic and even the list of problems that ONTAP's auto-diagnosis subsystem will address when complete is fairly extensive; due to space considerations we will not cover this information in full detail. Instead, we will restrict the following discussion to some common networking problems that ONTAP currently attempts to auto-diagnose. We will describe the set of problems targeted by this logic and illustrate its operation with two examples.

At the link layer, ONTAP attempts to diagnose Ethernet duplex and speed mismatches, Gigabit auto-negotiation mismatches, problems due to incorrect setting of store and forward mode on some network interface cards (NICs), link capacity problems, EtherChannel load balancing problems and some common hardware problems. At the IP layer, ONTAP can diagnose common routing problems and problems related to excessive fragmentation. At the transport layer, ONTAP can diagnose common causes of poor TCP performance. At the system level, ONTAP can diagnose problems due to inconsistent information in different configuration files, unavailability or unreachability of important information servers such as DNS and NIS servers, and insufficient system resources for the networking code to function at the load being offered to it.

To see how the techniques described in the previous section are used, consider the link layer diagnosis logic. The continuous monitoring system monitors the different event statistics such as total packets in, total packets out, incoming packets with CRC errors, collisions, late collisions, deferred transmissions etc., that are maintained by the various NIC device drivers. Assume that the continuous monitoring logic notices a large number of CRC errors. Usually, this will also be noticed as poor application-level performance.

Without auto-diagnosis, the manner in which this field problem is handled depends on the skill level and the debugging approach of the person addressing the problem. Some people will simply assume bad hardware and swap the NIC. Other people will first check for a duplex mismatch (if the NIC is an Ethernet NIC) by looking at the duplex settings of the NIC and the corresponding switch port, and if no mismatch is found may try a different cable and a different switch port in succession before swapping the NIC.

With the *netdiag* command, this process is much more formal and precise (Figure 2). The *netdiag* command first executes a *protocol augmentation* based test for detecting if there is a duplex mismatch. Specifically, the command forces some "reverse traffic" from the other machines on the network to the filer using a variety of different mechanisms in turn until one succeeds. These mechanisms include an ICMP echo-request broadcast, layer 2 echo-request broadcast and TCP/UDP traffic to well-known ports for hosts in the ARP cache of the filer. First the ambient rate of packet arrival at the filer using whatever mechanism that generated sufficient return traf-

[3] See the subsection on extensibility for how this is going to change.

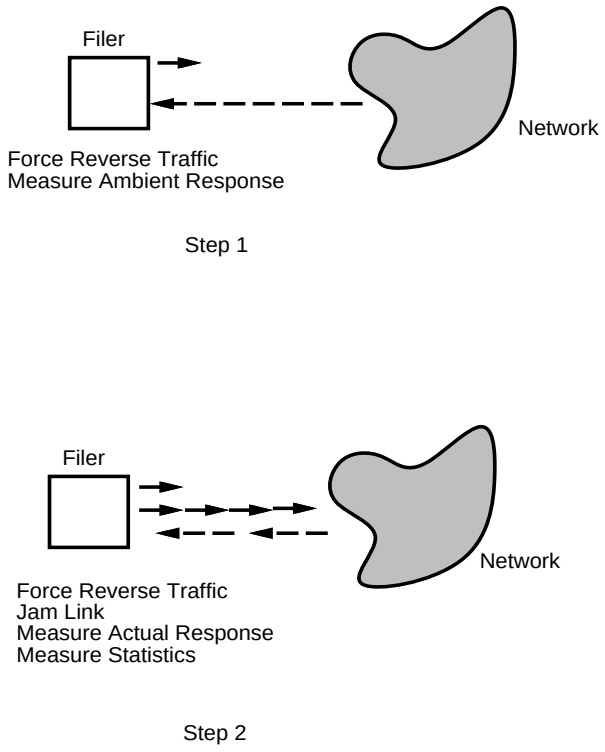

Fig. 2: Diagnosing a duplex mismatch using protocol augmentation.

fic is measured (Figure 2, Step 1). Next this reverse traffic is initiated again using the same mechanism as before and the suspect outgoing link is jammed with back-to-back packets destined to the filer itself (which will be discarded by the switch). The reverse traffic rate is then measured, along with the number of physical level errors during the jam interval (Figure 2, Step 2). If there is indeed a duplex mismatch, these observations are sufficient to discover it, since the reverse rate will interfere with the forward flow inducing certain types of errors only if the duplex settings are not configured correctly. In this case, the *netdiag* command prints information on how to fix the mismatch.

If the reason behind the duplex mismatch is a recent change to the filer's configuration parameters, this information will also be inferred by the auto-diagnosis logic and printed for the benefit of the user. If the NIC in question noticed a link-down-up event in the recent past and no CRC errors had been seen before that event, the *netdiag* command will print out this information as it could indicate a switch port setting change, or a cable change or a switch port change event which might have triggered off the mismatch. This extra information, which is made possible by automatic configuration change tracking, is important because it helps the customer discover the cause of the problem and ensure that it does not repeat. This problem may have been caused by, for example, two administrators inadvertently acting at cross-purposes.

If there is no duplex mismatch, the *netdiag* command prints a series of recommendations, such as changing the cable, switch port and the NIC, in the precise order in which they should be tried by the user. The order itself is based on historical data regarding the relative rates of occurrence of these causes.

As another example, consider the TCP auto-diagnosis logic. ONTAP's TCP continuously monitors the movement of each peer's advertised window and the exact timings of data and acknowledgment packet arrivals. A number of rules (which are described in detail in a forthcoming paper) are used to determine if the peer, or the network, or even the filer is the bottleneck factor in data transfer. For instance, if the filer is sending data through a Gigabit interface but the receiver client does not advertise a window that is large enough for the estimated delay-bandwidth product of the connection, the client is flagged as "needing reconfiguration". If the receiver did initially (at the beginning of the connection) advertise a window that was sufficiently large, but subsequently this window shrank, this indicates that the client is unable to keep up with protocol processing at the maximum rate supported by the network, and this situation is flagged.

Cross-layer analysis is used to make the TCP logic aware as to what time-periods of a TCP connection are "interesting" from the point of view of performance auto-diagnosis of the type described in the previous paragraph. For example, the beginning of a large NFS read may indicate the beginning of an "interesting" time period for an otherwise idle TCP connection. Protocol augmentation (using ICMP ping based average RTT measurement) is used to estimate the delay-bandwidth product of the path to various clients.

4.2 Extensibility

Data ONTAP contains an implementation of the Java Virtual Machine. Our approach to addressing the issue of extensibility is to write most of the auto-diagnosis system in Java. This provides us complete flexibility to change the auto-diagnosis logic in a new version, and support older versions of ONTAP. In a Java scenario, the auto-diagnosis logic ships as a collection of Java classes that reside on the root file system of the filer.

Note that the current version of the auto-diagnosis system is in C; we plan to use Java in the next major version of ONTAP. As mentioned earlier, a file containing constant and threshold values provides limited extensibility in the current implementation.

4.3 Implementation in other operating systems

Based on our knowledge of the internals of BSD-like general-purpose operating systems and our experience with the implementation of the NetApp Auto-diagnosis

System, it appears that it should be relatively straightforward to implement an auto-diagnosis subsystem based on the techniques presented in this paper in BSD-like systems. Like in ONTAP, a kernel thread can be used to implement continuous monitoring. The rules and thresholds used by the continuous monitoring logic can be chosen based on field information about the problem situations being targeted. The continuous monitoring logic can be partitioned into sub-logic blocks for each major OS subsystem.

The specific active tests to be implemented will also depend on the field problems being targeted. Protocol augmentation and cross-layer analysis can be used as guiding principles in the design of such tests. The implementation of active tests triggered automatically by the in-kernel auto-diagnosis logic should be relatively straightforward. If a command-based approach (like our diagnostic commands based approach) is to be used for user interaction, the BSD *kvm* kernel memory interface may be used for transferring information between the in-kernel continuous monitoring logic and user-land diagnostic commands. Alternatively, an approach based on the */proc* file system may be used.

Special interfaces will be needed for commands to trigger in-kernel active tests. Again, several alternative approaches are possible. The tests may be enumerated and made available as a series of system calls or different *ioctls* for a special diagnostics pseudo-device. For some active tests, it might also be possible to write the tests entirely in user-space using low level kernel interfaces (e.g. raw IP sockets). Extensibility can be provided by implementing the kernel portion of the auto-diagnosis logic as one or more loadable kernel modules.

5 Performance and experience

In this section, we will briefly discuss the performance of the NetApp Auto-diagnosis System, and our experience with its effectiveness in making the task of debugging field problems simple.

The continuous monitoring subsystem of ONTAP takes very few resources. Its CPU overhead is less than 0.25% CPU, even on the slowest systems that we ship. The memory footprint is less than 400KB for a typical system. The time that the *netdiag* command takes depends on the configuration of the system and the load on the system. On our slowest filer that is configured with the maximum number of allowable network interfaces and is saturated with client load, *netdiag* takes no more than 15 seconds to execute. On most systems, it takes less than 5 seconds.

Specifically, on a F760 class filer (600 Mhz 21164 Alpha, 2GB RAM) configured with 4 network interfaces and under full client load, the CPU usage of auto-diagnosis continuous monitoring code is less than 0.1% CPU. On this system, *netdiag* takes approximately 4 seconds to execute.

The version of ONTAP that contains the NetApp Auto-diagnosis System has only recently been made available to customers. However, since this version of ONTAP has not yet shipped to our customers in volume, we have not been able to see how well the auto-diagnosis subsystem is able to deal with real-life problems in the field. Instead, we have been forced to rely on a study in the laboratory. In this study we simulated a sample of field problem cases from our customer support call record database and measured the effectiveness of the auto-diagnosis system in solving the problems. For each case, we re-created the specific problem situation in the laboratory and measured the effectiveness of the auto-diagnosis logic.

We first looked at a sample of 961 calls that came in during the month of September 1999. This set did not include calls corresponding to hardware or software faults. We also did not consider calls that were related to general information about the product asked for by the customer. All other types of calls were considered. The month of September 1999 was the first month whose call data we did *not* include in our analysis of historical call record data while designing ONTAP's auto-diagnosis logic.

Of these 961 calls, 84 had something to do with the networking code and its interactions with the rest of ONTAP. Auto-diagnosis, when simulated on these cases, was able to auto-detect the problem cause for all but 12 of these calls, at a success-rate of 84.5%. The average time that it took the *netdiag* command to diagnose the problem was approximately 2.5 seconds. We did not even attempt to quantify the secondary effect on the customer's level of satisfaction that auto-diagnosis would cause due the the dramatic reduction in average problem diagnosis time.

Of the 12 calls on which auto-diagnosis did not diagnose, 7 were related to transient problems with external networking hardware, 1 was due to a NIC that was exhibiting very occasional errors and had needed re-seating and 4 were problems for which we did not have appropriate auto-diagnosis logic.

Of the 877 calls not corresponding to networking, we performed a static manual analysis in order to figure out which of these problems could be auto-diagnosed by the complete ONTAP auto-diagnosis system. This analysis was performed against a design description of the auto-diagnosis logic for other subsystems of ONTAP. Our study indicates that about three quarters of these problems could indeed by addressed by auto-diagnosis. Another 124 (about 20%) of these calls corresponded to problems whose diagnosis could be sped-up significantly by the partial auto-diagnosis information that the diagnosis system provided.

We repeated this simulation and analysis for calls that came in during October 1999. We considered 1023

cases, 97 networking and 926 other. Simulation of the networking cases indicated that auto-diagnosis could solve 88% of these. All but 5 of the networking problems that could not be auto-diagnosed were related to misconfigured clients. The rest were problems for which we have not yet developed appropriate auto-diagnosis logic. Static manual analysis of the non-networking cases indicated a success-rate of about 70%.

We also considered 500 randomly chosen samples from the customer call data from the months of November 1999 through February 2000. We repeated the above described analysis and simulation for these 500 calls. Our results for this sample of calls were very similar to the results for September and October 1999.

In summary, our historical call data seems to indicate that our auto-diagnosis system will be hugely successful in making a lot of problems that currently require human intervention to be automatically addressed. This should lead to a big reduction in the cost of handling customer calls because of a significant reduction in the number of calls per installed system. We were unable to directly quantify the increase in simplicity of the problem diagnosis process; the only (relatively weak) metric that we could quantify was turnaround time for the problem, with and without auto-diagnosis. This metric was at least three orders of magnitude lower for auto-diagnosis.

6 Related work

To place our work in context, we briefly survey other approaches to field problem diagnosis of computer systems, and how our work relates to these techniques.

6.1 Ad-hoc monitoring of UNIX and UNIX-like systems

As briefly described before, most UNIX and UNIX-like operating systems maintain a large number of statistics corresponding to various events that have occurred in the operation of the system. Access to these statistics and other configuration information is provided by a number of command interfaces. Problem diagnosis usually consists of manually obtaining appropriate statistics and perusing them for aberrant values.

System administrators in some organizations that use a large number of UNIX systems often use a set of homegrown (or commercially available) frameworks of automated scripts to obtain information from a large number of systems and analyse these values. There is a wealth of literature describing these tools [29, 10, 9, 2]. In some ways, this is similar to our technique of continuous monitoring. The information gathered by these automated scripts, however, is at the granularity at which the various operating systems export system information. This granularity is usually too coarse for extensive auto-diagnosis of the kind that we can perform inside the operating system kernel with reasonable system overhead. These environments are also limited in the types of active tests that they can perform for pin-pointing problems.

6.2 SNMP

The Simple Network Management Protocol (SNMP) [3] allows for the management of systems in a TCP/IP network within a coherent framework. In the SNMP world, network management consists of *network management stations*, called managers, communicating with the various systems in the network (hosts, routers, terminal servers etc.), called *network elements*. SNMP based management consists of three parts: 1) a Management Information Base (MIB) [18] that defines the various variables (both standardized and vendor-specific) that network elements maintain that can be queried and set by the manager, 2) a set of common structures and an identification schema, called the Structure of Management Information (SMI) [28], that is used to reference the variables in the MIB, and 3) the protocol with which managers and elements communicate, i.e., SNMP.

The system works as follows: The network managers periodically send queries to the elements to get the state of the various elements. Elements send *traps* to managers when certain events happen. The manager may analyse the information available to it via results of queries to build a picture of the health of the network and present this information to the human network manager in a variety of ways. Plugins that extend a managers functionality in a vendor-specific manner are available to handle vendor specific MIBs. An example of a commonly used manager is HP's OpenView [11].

The problem of using SNMP is in some ways similar to the problem of defining appropriate checks for our continuous monitoring system. The various system variables that are checked by continuous monitoring equations correspond to MIB variables. The auto-diagnosis checking logic corresponds to logic in the network manager plugin handling the vendor-specific MIBs. Thus issues that arise in defining the checks that a continuous monitoring system should execute also apply to the design of SNMP logic.

SNMP is different from our system in two main ways: First, SNMP does not really have a parallel for our active tests. A manager can manipulate a network element in some limited fashion, e.g., by using setting appropriate MIB variables. However, this is not nearly as general or as powerful as what can be done by an active test executing in the concerned system itself.

Second, the fact that SNMP depends on the network connectivity to be present between the network elements and the manager limits the types of problems that can be effectively auto-diagnosed by using SNMP. In particular, problems effecting network connectivity may not be easily diagnosed by SNMP.

In some ways the use of SNMP complements our ap-

proach. A system of auto-diagnosis using the techniques that we described earlier may be responsible for the "local" health of a system and its interactions with other networking entities that it communicates with. An SNMP based network management infrastructure may provide overall information about the health of a network using information gained by communication with network elements and their auto-diagnosis subsystems.

6.3 Problem diagnosis systems from the mainframe and telecommunications world

A number of papers and patents in the literature describe various components of semi-automatic problem diagnosis systems that were developed and used in the context of mainframe computer systems [14, 31, 22], other highly reliable systems [1, 17, 32] and the phone system [35, 21]. These systems used the technique of continuous monitoring of the health of the system. Events affecting the health of the system were fed into a decision tree based expert diagnosis system. The expert system used the input events to walk down its decision tree to narrow down the set of possible problematic situations that might be present.

The hardest part of building such a system was defining the set of events to be monitored and building the knowledge base (the decision tree) of the expert system. There is some literature that describes at an abstract level how such knowledge base rule-sets can be created for a specific system based on probabilistic data about events and problems [35, 34]. Presumably, in practice, these knowledge bases were created based on experience information gathered from the field.

In some ways the work that we describe in this paper is similar to this older work. We also use continuous monitoring and have a rule-set and use thresholds to trigger off further diagnosis steps including various kinds of active tests.

Our work differs from this older work in that it provides novel guiding principles and a certain structure to the problem of designing rule-sets, thresholds, causality in related diagnosis procedures and active tests. The four auto-diagnosis techniques that we present were designed based on our knowledge of common field problems and how the occurrence of such field problems effects the dynamics of modern layered operating systems. The technique of protocol augmentation directly targets problems that arise out of inadequate, incompletely specified or poorly implemented open network protocols. Such problems are much more widespread and important in today's network-centric open computing infrastructures than in older environments where communication was based on closed proprietary protocols.

7 Summary and future work

To summarize, we described some general techniques to enable *appliance-like* debugging of field problems of network appliances. These techniques formalize various ad-hoc debugging techniques that are used in manual debugging of system problems by human experts. These techniques also help in making the task of debugging hard problems manually much simpler and quicker than it currently is.

We have implemented these ideas in the Data ONTAP operating system. Our laboratory studies primed with real historical case data seem to indicate that auto-diagnosis as a methodology is very viable and has the potential of greatly reducing the complexity of problem analysis that is exposed to the customer.

In terms of future work, we would like to expand our continuous monitoring logic to encompass more complicated problems. As mentioned earlier, we are in the process of making the auto-diagnosis system extensible and easy to re-configure; this problem has a number of interesting issues. It would also be interesting to see a new user-interface paradigm linked with the ideas discussed in this paper that can vary the amount of detail and complexity in the output of the system based on the expertise of the user.

While our discussion has focused on Data ONTAP, from our experience it seems that most of the ideas described in this paper are directly applicable general-purpose operating systems. ONTAP's network code is based on BSD, and much of our auto-diagnosis logic can be directly applied to any BSD based TCP/IP subsystem. We look forward to an application of some of these ideas to general-purpose operating systems.

Acknowledgements

Lots of people at NetApp helped in the work described in this paper. Barbara Naden, Henk Bots, Devi Nagraj, Mark Smith, Diptish Datta, Brian Pawlowski, Janet Takami, Susan Whitford, Paul Norman and a large number of folks in NetApp's customer satisfaction department contributed in terms of useful ideas, discussion and feedback. We are also grateful to our shepherd Aaron Brown, and the anonymous USENIX reviewers for their valuable feedback.

References

[1] Alex Winokur and Joseph Shiloach and Amnon Ribak and Yuangeng Huang. Problem determination method for local area network system. US Patent 5539877, 1996.

[2] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software—Practice and Experience*, 27, 1997.

[3] J. D. Case, M. S. Fedor, M. L. Schoffstall, and

C. Davin. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[4] Cisco Local Director. http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/.

[5] Cobalt CacheRaQ 2. http://www.cobalt.com/products/cache/index.html.

[6] Cobalt NASRaQ. http://www.cobalt.com/products/nas/index.html.

[7] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[8] e/BSD: BSDI Embedded Systems Technology. http://www.BSDI.COM/products/eBSD/.

[9] M. Gomberg, C. Stacey, and J. Sayre. Scalable, Remote Administration of Windows NT. In *Proceedings of the Second Large Installation Systems Administration of Windows NT Conference (LISA-NT)*, Seattle, WA, July 1999.

[10] S. Hansen and T. Atkins. Centralized System Monitoring With Swatch. In *Proceedings of the Seventh Systems Administration Conference (LISA)*, Monterey, CA, Nov. 1993.

[11] HP OpenView. http://www.openview.hp.com/.

[12] IBM SecureWay Network Dispatcher. http://www.-ibm.com/software/network/dispatcher/.

[13] Intel InBusiness eMail Station. http://www.intel.-com/network/smallbiz/inbusiness_email.htm.

[14] L. Koved and G. Waldbaum. Improving Availability of Software Subsystems Through On-Line Error Detection. *IBM Systems Journal*, 25(1):105–115, 1986.

[15] P. J. Leach and D. C. Naik. A Common Internet File System (CIFS/1.0) Protocol. Internet Draft, Network Working Group, Dec. 1997.

[16] J. Liedtke, V. Panteleenko, T. Jaeger, and N. Islam. High-performance caching with the Lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[17] Luan J. Denny. Threshold alarms for processing errors in a multiplex communications system. US Patent 4817092, 1989.

[18] K. McCloghrie and M. T. Rose. Management Information Base for Network management of TCP/IP-based Internets: MIB-II. RFC 1213, Mar. 1991.

[19] Mirapoint Internet Message Server. http://www.mirapoint.com/products/servers/index.asp.

[20] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 USENIX Technical Conference*, pages 99–111, 1996.

[21] Mohammad T. Fatehi and Fred L. Heismann. Performance monitoring and fault location for optical equipment, systems and networks. US Patent 5296956, 1994.

[22] R. E. Moore. Utilizing the SNA Alert in the Management of Multivendor Networks. *IBM Systems Journal*, 27(1):15–31, 1988.

[23] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[24] Network Appliance – Products – Filers. http://www.netapp.com/products/filer/.

[25] Network Appliance – Products – NetCache. http://www.netapp.com/products/netcache/.

[26] Network Appliance Technical Library. http://ww-w.netapp.com/tech_library/.

[27] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3: Design and Implementation. In *Proceedings of the USENIX 1994 Summer Technical Conference*, Boston, MA, June 1994.

[28] M. T. Rose and K. McCloghrie. Structure and Identification of management Information for TCP/IP-based Internets. RFC 1155, May 1990.

[29] E. Sorenson and S. R. Chalup. RedAlert: A Scalable System for Application Monitoring . In *Proceedings of the Thirteenth Systems Administration Conference (LISA)*, Seattle, WA, Nov. 1999.

[30] SPEC SFS97. http://www.specbench.org/osg/sfs-97/.

[31] T. P. Sullivan. Communcations Network Management Enhancements for SNA Networks: An Overview. *IBM Systems Journal*, 22(1/2):129–142, 1983.

[32] Wing M. Chan. Data integrity checking with fault tolerance. US Patent 4827478, 1989.

[33] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[34] Yuval Lirov and On C. Yue. Technique for producing an expert system for system fault diagnosis. US Patent 5107497, 1992.

[35] Yuval Lirov and Swaminathan Ravikumar and On-Ching Yue. Arrangement for automated troubleshooting using selective advice and a learning knowledge base. US Patent 5107499, 1992.