Navigating in the Storm: Using Astrolabe for Distributed Self-Configuration, Monitoring and Adaptation

Kenneth P. Birman, Robbert van Renesse and Werner Vogels[1]
Dept. of Computer Science, Cornell University
{ken,rvr,vogels}@cs.cornell.edu

Abstract

The dramatic growth of computer networks creates both an opportunity and a daunting distributed computing problem for users seeking to build applications that can configure themselves and adapt as disruptions occur. The problem is that data often resides on large numbers of devices and evolves rapidly. Systems that collect data at a single location scale poorly and suffer from single-point failures. Here, we discuss the use of a new system, Astrolabe, to automate self-configuration, monitoring, and to control adaptation. Astrolabe operates by creating a virtual system-wide hierarchical database, which evolves as the underlying information changes. Astrolabe is secure, robust under a wide range of failure and attack scenarios, and imposes low loads even under stress.

Keywords: *Autonomic computing, self-configuration, distributed monitoring, system management, adaptation*

1 Introduction

In this paper, we describe a new information management service called Astrolabe, and its use in building new styles of "autonomic" computing applications. Our central premise is that large-scale distributed systems, such as data centers hosting Web Service applications and client computers accessing them over the network, are far too fragile today. We see this fragility as a direct consequence of inadequate systems support. New tools to assist such applications in self-configuration, monitoring and adaptation will promote advances in application robustness and ease of use. Although the discussion focuses on the Web Services application just described, we believe that Astrolabe will also find application in a number of other kinds of systems.

Astrolabe monitors the dynamically changing state of a collection of distributed resources, reporting summaries of this information to its users. Like the Internet Domain Name Service (DNS), Astrolabe organizes the resources into a hierarchy of domains, which we call zones, and associates attributes with each zone. Unlike DNS, the attributes may be highly dynamic, and updates propagate within seconds, even in huge networks. A novel peer-to-peer protocol is used to implement the Astrolabe system, which operates without any central servers.

Part of the power of Astrolabe stems from its ability to perform data mining and data fusion. The system continuously computes summaries of the data using on-the-fly aggregation. The aggregation mechanism is controlled by SQL queries, and operates by extracting summaries of data from each zone, then assembling these into higher-level database relations. By reprogramming these features on the fly (a task very much like asking a database to compute a dynamically materialized relation), the user can reconfigure Astrolabe within seconds. Thus, as the needs of the user change, the behavior of the system can adapt to respond to those new requirements.

Aggregation is analogous to computing a dependent cell in a spreadsheet. When the underlying information changes, Astrolabe will automatically and rapidly recompute the associated aggregates and report the changes to applications that have registered their interest. Even in huge networks, any change is soon visible everywhere. For example, suppose that a few servers in a data center come under a distributed denial of service attack. Suspecting this, an administrator might ask Astrolabe to capture some sort of statistic symptomatic of attack – perhaps, the rate of incomplete attempted connections to each server. Astrolabe has potential access to a great variety of host-maintained statistics and can also tap into data maintained by the

[1] The authors were supported by DARPA/AFRL grant RADC F30602-99-1-0532, by AFOSR/MURI grant F49620-02-1-0233, Microsoft Research BARC and the Cornell/AFRL Information Assurance Institute.

PROCEEDINGS OF THE AUTONOMIC COMPUTING WORKSHOP FIFTH ANNUAL INTERNATIONAL WORKSHOP
ON ACTIVE MIDDLEWARE SERVICES (AMS'03) ISBN 0-7695-1983-0/01 $17.00 © 2003 IEEE

application or even stored in files and databases. The protocol is extremely robust: even if those servers are under heavy load, within a short period all servers in the data center and perhaps even all clients of the system will see the situation. Healthy nodes can then respond in a coordinated way, by shifting work away from the affected servers.

We are not aware of any prior system offering the mixture of scalability, robustness and security seen in Astrolabe. These properties will be crucial in tomorrow's large, extremely critical computing applications, because the need for autonomic tools is most acute in large-scale applications that demand high availability even during disruption or distributed denial of service attacks. In traditional, centralized, implementations of system monitoring and control functionality, these issues can emerge as impediments to the user. For example, if computers are tracking state that changes once every second, and we have 10,000 such computers, the centralized database would be expected to keep up with 10,000 updates per second, a massive load. If that central system is unavailable, the autonomic support features of the system would become inaccessible. And there are many security issues raised by centralized architectures. This paper will show how Astrolabe handles the analogous problems through its peer-to-peer protocols and hierarchical structuring of data.

This paper starts with a brief discussion of our vision of how autonomic computing might be accomplished using tools like Astrolabe. Next, we provide a technology review focused on the data mining features of Astrolabe, which are based on its aggregation mechanisms. Astrolabe gains scalability and robustness at the price of generality, and we spend some time looking at the limitations of the system and their implications for developers and users. In particular, while Astrolabe is a database, it doesn't allow the user to do arbitrary database-style transactions, and it is important to understand the reasons and the degree to which one can work around these limits. For reasons of brevity, this paper omits a detailed scalability analysis, but we do summarize prior work on this problem.

2 Autonomic Computing

The phrase "autonomic computing" has different meanings in the eyes of different kinds of users. Some imagine a new era of self-aggregating computing systems: a PDA, for example, that can automatically discover input and display devices in each room the user enters and dynamically configure a kind of virtual PC on-the-fly. Some imagine data centers with the kinds of serviceability and management features common on RAID file servers. And some imagine a completely self-managed distributed system in which parameter setting and configuration is automated, problem diagnosis and repair are common-place and standardized, and hence the total cost of ownership is dramatically reduced even for ambitious configurations.

Our work falls loosely into this third area, although Astrolabe could be useful in any of these settings. Consider a large data center hosting a cluster-style load-balanced web service application that communicates to multiple backend legacy services. A client system is performing a high-value transaction on this system when a request suddenly times out. What should that client do? If the problem is a failure within the data center, how should it diagnose and repair the problem?

A simple and somewhat facile answer would point to the transactional standard (WS_TRANSACTION) and suggest that merely by using transactions, the client's problems will magically vanish. Yet any pragmatist will recognize that such responses overlook a tremendous number of hard problems. Setting the poor performance of web service transactions to the side, how should the data center sense failures? A timeout, after all, might originate at many levels of the web service architecture. Can this type of state sensing guarantee consistency, so that all nodes monitoring the server in question simultaneously sense the failure and do so only if it really crashes? How should the data center reallocate tasks to repair any functional gaps caused by the failure? What services need to be restarted?

Our client also faces difficult problems. The server to which it was communicating may have failed during the commit stage of a transaction, or during a so-called "business transaction", which will not be automatically aborted when the connection is lost. Which server should the client reconnect to? How long should it wait for the backup server to "catch up" with the failed primary server?

In the introduction we mentioned that a distributed denial of service attack could bring down a server. Yet that server may not have "failed" – it could simply be extremely overloaded. Thus monitoring goes well beyond the mere detection of crash failures and must also encompass the discovery of all sorts of other problems capable of disrupting smooth operations.

These kinds of questions extend to much more mundane settings. For example, most modern systems have a tremendous dependency on the Internet's DNS.

Yet in an era of network address translation and firewalls, the DNS is increasingly hard pressed to simply track the bindings of machine names to IP addresses. Developers complain of unacceptable tradeoffs between slow performance (because DNS records have such short lifetimes that they cannot even be cached), and unacceptable staleness of dynamically updated data (because DNS uses a pull, not a push, architecture). Moreover, DNS is poorly suited for managing other kinds of parameters that may need to change over time. We need better tools for monitoring the state of complex distributed systems, representing that state in a way that multiple applications can access concurrently, updating state as conditions change, and triggering appropriate reconfigurations in a consistent way. By offering solutions to such problems, Astrolabe offers a major advance in system support for autonomic computing.

3 The Astrolabe System

Astrolabe is best understood as a relational database built using a *peer-to-peer protocol*[2] running between the applications or computers on which Astrolabe is installed. Like any relational database, the fundamental building block employed by Astrolabe is a tuple (a row of data items) into which values can be stored. For simplicity in this paper, we'll focus on the case where each tuple contains information associated with some computer. The technology is quite general, however, and can be configured with a tuple per application, or even with a tuple for each instance of some type of file or database.

The data stored into Astrolabe can be drawn from the management information base (MIB) of a computer, extracted directly from a file, database, spreadsheet, or fetched from a user-supplied method associated with some application program. Astrolabe obtains flexibility by exploiting a recent set of standards (ODBC, JDBC) whereby a system like ours can treat the objects on a computer much like databases. Astrolabe is also flexible about data types, supporting the usual basic types but also allowing the application to supply arbitrary information encoded with XML. The only

[2] The term « peer-to-peer » is often used in conjunction with scalable file systems for sharing media content. Here, we refer only to the communication pattern seen in such systems, which involves direct pairwise communication between « client » computers, in contrast to a more traditional star-like client-server architecture where all data passes through the centralized servers.

Name	Load	Weblogic?	SMTP?	Version
swift	2.0	0	1	6.2
falcon	1.5	1	0	4.1
cardinal	4.5	1	0	6.0

Figure 1: Three Astolabe domains

requirement is that the total size of the tuple be no more than a few k-bytes; much larger objects can be by identified by a URL or other reference, but the data would not be replicated in Astrolabe itself.

The specific data pulled into Astrolabe is specified in a *configuration certificate*. Should the needs of the user change, the configuration certificate can be modified and, within a few seconds, Astrolabe will reconfigure itself accordingly. This action is, however, restricted by our security policy, as discussed in Section 3.

Astrolabe groups small sets of tuples into relational tables. Each such table consists of perhaps 30 to 60 tuples containing data from sources physically close to one-another in the network. This grouping (a database administrator would recognize it as a form of schema) can often be created automatically, using latency and network addresses to identify nearby machines. However, the system administrator can also specify a desired layout explicitly.

Where firewalls are present, Astrolabe employs a standard tunneling method to send messages to machines residing behind the firewall and hence not directly addressable. This approach also allows Astrolabe to deal with network address translation (NAT) filters.

The data collected by Astrolabe evolves as the underlying information sources report updates, hence the system constructs a continuously changing database using information that actually resides on the participating computers. Figure 1 illustrates this: we see a collection of small database relations, each tuple corresponding to one machine, and each relation collecting tuples associated with some set of nearby machines. In this figure, the data stored within the tuple includes the name of the machine, its current load, an indication of whether or not various servers are running on it, and the "version" for some application. Keep in mind that this selection of data is completely determined by the configuration certificate. In principle, any data

available on the machine or in any application running on the machine can be exported. In particular, spreadsheets and databases can easily be configured to export data to Astrolabe.

The same interfaces which enable us to fetch data so easily also make it easy for applications to use Astrolabe. Most commonly, an application would access the Astrolabe relations just as it might access any other table, database or spreadsheet. As updates occur, the application receives a form of event notifying it that the table should be rescanned. Thus, with little or no specialized programming, data from Astrolabe data could be « dragged » into a local database, spreadsheet, or even onto a web page. As the data changes, the associated application will receive refresh events.

Astrolabe is intended for use in very large networks, hence this form of direct access to local data cannot be used for the full dataset : while the system does capture data throughout the network, the amount of information would be unweildy and the frequency of updates excessive. Accordingly, although Astrolabe does provide an interface whereby a remote region's data can be accessed, the normal way of monitoring remote data is through *aggregation queries.*

An aggregation query is, as the name suggests, just an SQL query which operates on these leaf relations, extracting a single summary tuple from each which reflects the globally significant information within the region. Sets of summary tuples are concatenated by Astrolabe to form summary relations (again, the size is typically 30 to 60 tuples each), and if the size of the system is large enough so that there will be several summary relations, this process is repeated at the next level up, and so forth. Astrolabe is thus a hierarchical relational database. Each of the summaries is updated, in real-time, as the leaf data from which it was formed changes. Even in networks with thousands or millions of computers, updates are visible system-wide within a few tens of seconds. (Figure 2).

A computer using Astrolabe will, in general, keep a local copy of the data for its own region and aggregation (summary) data for region above it on the path to the root of this hierarchy. As just explained, the system maintains the abstraction of a hierarchical relational database. Physically, however, this hierarchy is an illusion, constructed using a peer-to-peer protocol, somewhat like a jig-saw puzzle in which each computer has ownership of one piece and read-only replicas of a few others. Our protocols permit the system to assemble the puzzle as a whole when needed. Thus, while the user thinks of Astrolabe as a somewhat constrained but rather general database, accessed using conventional programmer APIs and development tools, this abstraction is actually an illusion, created on the fly.

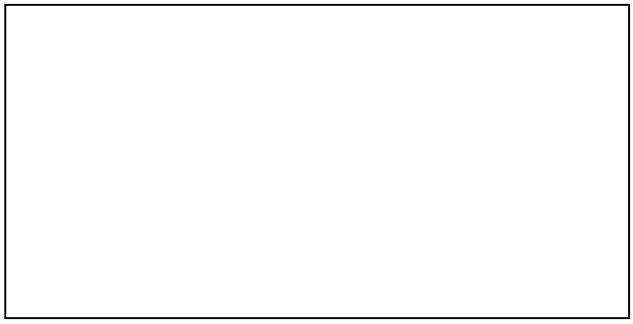

Figure 2: Hierarchy formed when data-mining with an aggregation query fuses data from many sources.

The peer-to-peer protocol used for this purpose is, to first approximation, easily described [7]. Each Astrolabe system keeps track of the other machines in its zone, and of a subset of *contact* machines in other zones. This subset is selected in a pseudo-random manner from the full membership of the system (again, a peer-to-peer mechanism is used to track approximate membership ; for simplicity of exposition we omit any details here). At some fixed frequency, typically every 2 to 5 seconds, each participating machine sends a concise state description to a randomly selected destination within this set of neighbors and remote contacts. The state description is very compact and lists versions of objects available from the sender. We call such a message a « gossip » event. Unless an object is very small, the gossip event will not contain the data associated with it.

Upon receiving such a gossip message, an Astrolabe system is in a position to identify information which may be stale at the sender's machine (because timestamps are out of date) or that may be more current at the sender than on its own system. We say *may* because time elapses while messages traverse the network, hence no machine actually has current information about any other. Our protocols are purely asynchronous : when sending a message, the sender does not pause to wait for it to be recieved and, indeed, the protocol makes no effort to ensure that gossip gets to its destinations.

If a receiver of a gossip message discovers that it has data missing at the sender machine, a copy of that data is sent back to the sender. We call this a *push* event. Conversely, if the sender has data lacking at the receiver, a *pull* event occurs : a message is sent requesting a copy of the data in question. Again, these actions are entirely asynchronous ; the idea is that they will usually be successful, but if not (e.g. if a message is lost in the network, received very late, or if some

other kind of failure occurs), the same information will probably be obtained from some other source later.

One can see that through exchanges of gossip messages and data, information should propagate within a network over an exponentially increasing number of randomly selected paths among the participants. That is, if a machine updates its own row, after one round of gossip, the update will probably be found at two machines. After two rounds, the update will probably be at four machines, etc. In general, updates propagate in log of the system size – seconds or tens of seconds in our implementation. In practice, we configure Astrolabe to gossip rapidly within each zone (to take advantage of the presumably low latency) and less frequently between zones (to avoid overloading bottlenecks such as firewalls or shared network links). The effect of these steps is to ensure that the communication load on each machine using Astrolabe and also each communication link involved is bounded and independent of network size.

We've said that Astrolabe gossips about *objects*. In our work, a tuple is an object, but because of the hierarchy used by Astrolabe, a tuple would only be of interest to a receiver in the same region as the sender. In general, Astrolabe gossips about information of *shared interest* to the sender and receiver. This could include tuples in the regional database, but also aggregation results for aggregation zones that are ancestors of both the sender and receiver.

After a round of gossip or an update to its own tuple, Astrolabe recomputes any aggregation queries affected by the update. It then informs any local readers of the Astrolabe objects in question that their values have changed, and the associated application rereads the object and refreshes its state accordingly.

For example, if an Astrolabe aggregation output is pulled from Astrolabe into a web page, that web page will be automatically updated each time it changes. The change would be expected to reach the server within a delay logarithmic in the size of the network, and proportional to the gossip rate. Using a 2-second gossip rate, an update would thus reach all members in a system of 10,000 computers in roughly 25 seconds. Of course, the gossip rate can be tuned to make the system run faster, or slower, depending on the importance of rapid responses and the available bandwidth.

Our description oversimplifies. Astrolabe can actually support multiple aggregation queries, each creating its own hierarchy. The system can also be configured to accomodate heterogeneity of the leaf nodes, whereas we have presented it as if each leaf node has identical information. Moreover, the same peer-to-peer mechanisms used to propagate updates are also used to propagate new configuration certificates and new aggregation queries, hence the behavior of the system can be modified on the fly, as needs change. Details on these aspects, together with an enlarged discussion of our peer-to-peer protocol can be found in [7].

4 Consistency, Security and Expressiveness

The power of the Astrolabe data mining mechanisms is limited by the physical layout of the Astrolabe database and by our need, as builders of the system, to provide a solution which is secure and scalable. This section discusses some of the implications of these limitations for the Astrolabe user.

4.1 Consistency

Although Astrolabe is best understood as a form of hierarchical database, the system doesn't support transactions, the normal consistency model employed by databases. A transaction is a set of database operations (database read and update actions) which are performed in accordance with what are called ACID properties. The consistency model, *serializability,* embodies the guarantee that a database will reflect the outcome of committed transactions, and will be in a state that could have been reached by executing those transactions sequentially in some order.

In contrast, Astrolabe is accessible by read-only operations on the local zone and aggregation zones on the path to the root. Update operations can only be performed by a machine on the data stored in its own tuple.

If Astrolabe is imagined as a kind of replicated database, a further distinction arises. In a replicated database each update will be reflected at each replica. Astrolabe offers a weaker guarantee: if a participating computer updates its tuple and then leaves the tuple unchanged for a sufficiently long period of time, there is a very high probability that the update will become visible to all non-faulty computers. Indeed, this probability converges to 1.0 in the absence of network partitioning failures. However, if updates are more frequent, a "new" value could overwrite an "older" value, so that some machines might see the new update but miss the prior one.

Astrolabe gains a great deal by accepting this weaker probabilistic consistency property: the system is able to

scale with constant loads on computers and links, and is not forced to stop and wait if some machine fails to receive an update. In contrast, there is a well-known impossibility result that implies that a database system using the serializability model may need to pause and wait for updates to reach participating nodes. Indeed, a single inopportune failure can prevent a replicated database from making progress. Jointly, these results limit the performance and availability of a replicated database. Astrolabe, then, offers a weaker consistency property but gains availability and very stable, predictable performance by so doing.

Aggregation raises a different kind of consistency issue. Suppose that an aggregation query reports some property of a zone, such as the least loaded machine, the average humidity in a region, etc. Recall that aggregates are recomputed each time the Astrolabe gossip protocol runs. One could imagine a situation in which machine A and machine B concurrently update their own states; perhaps, their loads change. Now suppose that an aggregation query computes the average load. A and B will both compute new averages, but the values are in some sense unordered in time: A's value presumably reflects a stale version of B's load, and vice versa. Not only does this imply that the average computed might not be the one expected, it also points to a risk: Astrolabe (as described so far) might report aggregates that bounce back and forth in time, first reflecting A's update (but lacking B's more current data), then changing to reflect B's update but "forgetting" A's change. The fundamental problem is that even if B has an aggregation result with a recent timestamp, the aggregate could have been computed from data which was, in part, more stale than was the data used to compute the value it replaces.

To avoid this phenomenon, Astrolabe tracks minimum and maximum timestamp information for the inputs to each aggregation function. A new aggregate value replaces an older one only if the minimum timestamp for any input to that new result is at least as large as the maximum timestamp for the one it replaces. It can be seen that this will slow the propagation of updates but will also ensure that aggregates advance monotonically in time. Yet this stronger consistency property also brings a curious side-effect: if two different Astrolabe users write down the series of aggregate results reported to them, those sequences of values could advance very differently. Perhaps, A sees its own update reflected first, then later sees both its own and B's; B might see its update first, then later both, and some third site, C, could see the system jump to a state in which both updates are reflected. Time moves forward, but different users see events in different order and may not even see the identical events! This tradeoff seems to be fundamental to our style of distributed data fusion.

4.2 Security Model and Mechanisms

A related set of issues surround the security of our system. Many peer-to-peer systems suffer from insecurity and are easily incapacitated or attacked by malfunctioning or malicious users. Astrolabe is intended to run on very large numbers of machines, hence the system itself could represent a large-scale security exposure.

To mitigate such concerns, we've taken several steps. First, Astrolabe reads but does not write data on the machines using it. Thus, while Astrolabe can pull a great variety of data into its hierarchy, the system doesn't take the converse action of reaching back onto the participating machines and changing values within them, except to the extent that applications explicitly read data from Astrolabe.

The issue thus becomes one of trustworthiness: can the data stored in Astrolabe be trusted? In what follows, we assume that Astrolabe instances are non-malicious, but that the computers on which they run can fail, and that software bugs (hopefully, rare) could corrupt individual systems. To overcome such problems, Astrolabe includes a public-key infrastructure (PKI) which is built into the code. We employ digital signatures to authenticate data. Although machine B may learn of machine A's updates through a third party, unless A's tuple is correctly signed by A's private key, B will reject it. Astrolabe also limits the introduction of configuration certificates and aggregation queries by requiring keys for the parent zones within which these will have effect; by controlling access to those keys, it is possible to prevent unauthorized users from introducing expensive computations or configuring Astrolabe to pull in data from participating hosts without appropriate permissions. Moreover, the ODBC and JDBC interfaces by means of which Astrolabe interfaces itself to other components offer additional security policy options.

4.3 Query Limitations

A final set of limitations arises from the lack of a join feature in the aggregation query mechanism. As seen above, Astrolabe performs data mining by computing summaries of the data in each zone, then gluing these together to create higher level zones on which further summaries can be computed. The

approach lacks a way to compute results for queries that require cross-zone joins.

For example, suppose that Astrolabe were used to detect distributed denial of service attacks along the lines suggested earlier. One might want to express a data mining query along the following lines: "detect servers under attack and, for each such server, find a healthy server best positioned to take over its workload." The natural way to express this as a query in a standard database would involve a join. In Astrolabe, one would need to express this as two aggregation queries, one to compute a summary of apparent attacks and the other, using output from the first as an input, tracking down the best backup machines. In general, this points to a methodology for dealing with joins by "compiling" them into multiple current aggregation queries. However, at present, we have not developed this insight into a general mechanism; users who wish to perform joins would need to break them up in this manner, by hand. Moreover, it can be seen that while this approach allows a class of join queries to compile into Astrolabe's aggregation mechanism, not all joins can be so treated: the method only works if the size of the dataset needed from the first step of the join, and indeed the size of the final output, will be sufficiently small.

Configuring Astrolabe so that one query will use the output of another as part of its input raises a further question: given that these queries are typically introduced into the system while it is running, how does the user know when the result is "finished"? We have a simple answer to this problem, based on a scheme of counting the number of sub-zones reflected in an aggregation result. The idea is that as a new aggregate value is computed, a period passes during which only some of the leaf zones have reported values. At this stage the parent aggregation zone is not yet fully populated with data. However, by comparing a count of the number of reporting child zones with a separately maintained count of the total number of children, applications can be shielded from seeing the results of an aggregation computation until the output is stable. By generalizing this approach, we are also able to handle failures or the introduction of new machines; in both cases, the user is able to identify and disregard outputs representing transitional states. The rapid propagation time for updates ensures that such transitional conditions last for no more than a few seconds.

5 Example

In Section 2, we noted that Astrolabe has applications in many settings. For the purpose of illustrating the ideas behind the system, however, we continue to focus on a hypothetical commercial web service application.

The explosive growth of the web services market and unprecedented uptake of web services technology has caught many by surprise. Part of the success of the technology is undoubtedly a consequence of its natural evolutionary fit into settings which had already become more and more object oriented. Given that platforms such as .NET and J2EE were already relatively in their handling of objects and interfaces, extending them to use web-based standards for documenting the services offered by an application, representing requests and arguments, and communicating with services over HTTP is not a huge step. Yet precisely because the transition to web services has been so straightforward, commercial users now face a wrenching new development. Those traditional systems were often batch oriented, whereas web services are intended to be highly responsive, interactive applications. Systems that used to be live comfortably in the isolated backwaters of large commercial data centers are suddenly being interconnected to web service applications in ways never before possible. And with this sudden leap to turn all systems into spaghetti-like structures with layers and layers of interdependencies and interactions, managing those systems has become a potential nightmare.

Suppose that a client is using a web service and his request times out. The problem may be a network outage, sluggish response in the web services system itself, a crash of that platform, an internal queuing delay (many of these systems use message oriented middleware such as MQSeries or MSMQ), bursty behavior in old legacy software, or an inappropriately set parameter. The server may even be under some form of attack. We have few tools to assist in diagnosing such problems: Modern computing systems operate in the dark. A vast amount of information is potentially relevant to their correct configuration and operation, yet little of this information is ever represented or available to the application.

Consider first the problem as it arises within a data center. With Astrolabe, we can easily instrument the many platforms that comprise the center, and reconfigure this instrumentation if an unexpected change in system behavior or responsiveness compels

the site administrator to hunt for problems of a type she has never seen previously. The instrumentation mechanism taps into the full set of data items available on the various data center servers: parameters in their MIBs (such as paging and I/O rates, network statistics, etc), application-specific parameters, information in databases or files, etc. Thus the center's administrator can view all of this data as comprising a long "tuple" with one field for each potential data item. She merely selects items of interest within the set, and Astrolabe will reach into the system and, given appropriate permissions, extract the data items in question and monitor them for subsequent changes. Thus the systems administrator can think of the whole data center as a form of dynamically changing database.

Were this the end of the story, Astrolabe might be best understood as a new kind of network monitoring system. However, the goal of the technology is to offer a fundamentally new kind of operating systems service, accessible not just to human administrators but also to applications. To this end, Astrolabe offers a consistent, robust state representation that can be exploited by the application itself. When an event occurs that disrupts state – a machine crashes, or a service hangs – the deviation from the nominal state will become globally evident within seconds. Every healthy program will simultaneously notice failures or degradation. Observing the problem, the many programs that comprise the system can respond in a coordinated manner. For example, some server might take over tasks that a failed server had been responsible for, and advertise its new role. Other servers, seeing this, can establish new connections to the server in question and interrogate it about work in progress.

In practice, we would not expect the applications themselves to detect degradation in an automated manner. Instead, the administrator would do this, flagging degraded servers in the Astrolabe table by defining aggregation queries that identify such servers, with a sufficient degree of built-in delay to avoid a "flapping state" problem if a server fails intermittently only to quickly recover. The application program thus uses Astrolabe to sense overall state but relies on a human-defined notion of "degradation" to identify servers that are operational but malfunctioning.

Now, suppose that we stand on the data center and look out towards the client platforms. To what degree can Astrolabe improve the experience of the end-user? As a first step, suppose that we use Astrolabe to monitor the states of client systems. Merely by taking this step, the data center gains a completely new kind of functionality. Traditionally, we have viewed a data center as being operational and "healthy" provided that the servers seem to be working properly. Suddenly, the option of focusing on the client's *experience* of the system becomes available.

Perhaps our data center is one that streams media files to its clients. The mere knowledge that the servers are not aware of problems tells us relatively little about the client experience. A client connected to the closest, least loaded server may still be experiencing disrupted downloads and poor throughput because of network problems such as overloads and router or link failures. Sensing such conditions would enable responses such as redirecting that client to some other server, perhaps one that is handling a heavier load or seems to be more remote in the network, and yet that is capable of offering a better end-user experience.

Similarly, Astrolabe can offer the client system better options for connecting to the server pool. To the degree that we wish to expose such information, clients can be shown information about which servers are handling which categories of data, server load, average service response times, availability of data replicas, and so forth. These kinds of information can be used as input to a client-side decision-making process concerning the best server to handle a given kind of request. The administrator controls the configuration of Astrolabe and hence can easily select the data that clients can see. Keep in mind that the clients we have in mind are software – the client-side applications developed originally by the designers of the data center. So we are not proposing that end-users would "see" the state of the data center, but merely that the software they downloaded to use the data center might be smarter about its current state, just as it could be smarter about their states. The human user simply experiences better performance and higher availability, because problems are now sensed more rapidly and reaction is more automated.

Our scenario started with a hypothesized timeout. With Astrolabe in use, the client and data center both "see" the system state in a consistent manner. If a server has crashed, they both share this information, and the client can track the progress of the backup server in taking over, bringing itself back into sync with the state of the failed primary server, and eventually coming back online. Parameter settings advertised through Astrolabe become globally visible and updates are seen rapidly. Thus, a wide range of autonomic adaptations become relatively straightforward.

This paper has mentioned distributed denial of service attacks several time. As we conclude this

section, consider briefly the challenges of responding to such an event. Commercial data centers are bedeviled by such problems; few, if any, have escaped unscathed. However, few of these attacks target more than a small number of servers: they succeed precisely because the attacker is able to marshal the resources of large numbers of machines to overwhelm a small number of target systems. Using Astrolabe, both servers and clients can sense localized disruptions, reallocating work and redirecting clients away from the disabled systems. Even if an attack has never been seen previously, by having system administrators in the loop – for example, in a position to redefine the data-mining query used to identify "faulty" servers – the capability now exists for dynamically responding to attacks purely by recognizing their symptoms. The effect is that the data center will seamlessly and automatically shift tasks away from faulty servers and towards those not currently under attack. The DDoS attacker will lack the resources to attack all servers and thus will be frustrated. As he discovers that his attack is having little impact on performance, he is likely to abandon the effort in favor of more promising targets.

Earlier we spoke of the many ways that autonomic computing is viewed by potential developers and users. Astrolabe can support even more ambitious styles of computing, extending to the kinds of resource-location problems that need to be solved to support self-aggregating computing platforms. We see the technology as opening the door to a major advance in computing styles.

Astrolabe has been designed to interoperate comfortably in a world of Web Services and data centers. While the sorts of uses just summarized would require a substantial integration effort between the vendor of the Web Services platform and our development team, there are no obvious obstacles to undertaking such an effort.

Some of the Astrolabe uses outlined here require a degree of caution on the part of the programmer. Recall that consistency in Astrolabe is a probabilistic property. Data will converge over time (so that, given enough time – seconds or minutes – multiple viewers will see the same data) but not instantly. Thus, some caution must be taken in the way that Astrolabe is used. Actions should be triggered only after a pause to give the system time to stabilize, and applications should be designed to watch for evidence of "flapping" systems or other anomalies. However, if actions are based on stable states and delayed by long enough to give Astrolabe itself time to reach a quiescent state, the approach offers a high degree of robustness and actions taken will be coordinated with extremely high probability. We are doubtful that any technology could offer stronger guarantees.

To reiterate a point made previously, today, the developer of a sophisticated distributed computing system is asked to work in the dark. This limits availability and makes such systems far more expensive to administer than need be the case. With new services such as the Astrolabe service, we can turn on the lights, enabling a new generation of far more automated computing systems that perform well under all sorts of conditions, adapt as conditions change, and configure themselves without requiring endless human intervention.

6 Performance

The dual goals of keeping this paper brief and of avoiding repetition of material reported elsewhere led us to omit any detailed performance section from this paper. However, Astrolabe is a real system and we have evaluated it in great detail. The interested reader is referred to [7].

Broadly, this evaluation consists of four parts. In a first step, we used formal methods to develop a theoretical analysis of the scalability and propagation properties of the system. Such an analysis is interesting to the extent that it seems to confirm our observations of behavior, but also limited insofar as we are forced to simplify the real world in order to reason about the technology. The analysis predicts the logarithmic scalability properties outlined earlier, and also lets us predict the distribution of update delays. Our work suggests that the exponential wave of infection that propagates updates not only makes the protocol itself robust to failures or network disruption, but also makes our *analysis* robust to these simplifications. In effect, when simplifying the model of a network, one perhaps arrives at behavioral predictions that are overly optimistic or pessimistic. But because that behavior is so strongly dominated by the exponential spread of information, such an error only leads to a minor inaccuracy. Our experience has been that the formal analysis of Astrolabe is highly predictive of its behavior.

A second style of evaluation focuses on two kinds of simulation. First, using network simulation systems (NS/2) we have simulated Astrolabe to understand its behavior in a variety of network topologies and under a variety of loads and scales. Second, we have looked at the behavior of our Astrolabe implementation by running the real software over a simulated network. We

do this by injecting packet loss or delays so as to emulate conditions that might be encountered in the field.

Finally, we have worked with Astrolabe in real world settings, and evaluated its behavior as it runs. While such an approach has the benefit of being an evaluation of a real system in a real setting, one also has less control over competing applications which share resources, less ability to reproduce scenarios to understand precisely how they gave rise to an observed behavior, and less opportunity to systematically vary parameters which determine behavior.

Jointly, these studies have confirmed that Astrolabe indeed exhibits the predicted logarithmic growth in update propagation latency, and that the system has stable, low, computing and communication loads. We have subjected Astrolabe to a variety of stresses (failures, packet loss) and found it to be robust even under rather severe attacks. In particular, conditions similar to those seen during distributed denial of service (ddos) attacks slow Astrolabe down, but not very much, and do not trigger any substantial growth in message rates or loads associated with the technology. This suggests that Astrolabe may remain useful even when a network is experiencing severe disruption. The possibility of using Astrolabe for distributed detection of such episodes and to trigger a coordinated response appears to be very promising.

7 Related Work

Our work draws heavily on prior research in peer-to-peer computing and databases. In the database area, the idea of building replicated databases using gossip communication dates to the Xerox Clearinghouse server, a flexible directory service for large networks. Discussion and analysis of the protocols used in this system appears in [2]. Subsequent Xerox work on a database system called Bayou takes the idea even further [6], and also includes a formal analysis of the scalability of push and pull gossip. The idea of building large-scale information systems hierarchically is an old one; many elements of our approach were anticipated by Lampson [5] and Golding [3]. Work on treating large sensor networks as databases can be found in [1]. The Ninja system replicates data using a peer-to-peer protocol similar to the one we use in Astrolabe, but lacks an aggregation mechanism [4].

8 Conclusions

The Astrolabe system creates a new option for developers of ambitious autonomic computing applications which run in large networks. Whereas traditional approaches collect data in a centralized server, Astrolabe implements a novel peer-to-peer protocol whereby queries can be computed directly in the network by the participating computers themselves. Although the loads imposed on participating computers are very small (and independent of the size of the system), the aggregated computing capability may be huge, hence we are able to solve problems that would be infeasible in a centralized solution. Moreover, the approach scales much better than centralized ones, is robust against failures and attack, and propagates updates within seconds or tens of seconds even in networks with huge numbers of computing nodes.

References

[1] P. Bonnet, J.E. Gehrke, P. Seshadri. Towards Sensor Database Systems. In *Proc of the 2nd Intl. Conf. On Mobile Data Management.* Hong Kong, Jan. 2001.

[2] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry. Epidemic Algorithms for Replicated Database Management. In *Proc. Of the Sixth ACM Symp. On Principles of Distributed Computing,* 1-12, Vancouver BC, Aug. 1987.

[3] R.A. Golding. A Weak-Consistency Architecture for Distributed Information Services. *Computing Systems,* 5(4) : 379-405, Fall 1992.

[4] S.D. Gribble, M. Welsh, R. Von Behren, E.A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. To appear in a special issue of *Computer Network* on the topic of Pervasive Computing. 2001.

[5] B.W. Lampson. Designing a Global Name Service. In *Proc. Of the 5th ACM Symposium on Principles of Distributed Computing.* Calgary, Alberta, Aug. 1986.

[6] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible Update Propogation for Weakly Consistent Replication. In *Proc. Of the 16th ACM Symposium on Operating Systems Principles,* 288-301, Sant-Malo, France, Oct. 1997.

[7] R. van Renesse and K.P. Birman. *Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management and Data Mining.* To appear,

ACM Transactions on Computer Systems, May 2003.
http://www.cs.cornell.edu/ken/Astrolabe.pdf

PROCEEDINGS OF THE AUTONOMIC COMPUTING WORKSHOP FIFTH ANNUAL INTERNATIONAL WORKSHOP
ON ACTIVE MIDDLEWARE SERVICES (AMS'03) ISBN 0-7695-1983-0/01 $17.00 © 2003 IEEE