Scalable Management and Data Mining using Astrolabe*

Robbert van Renesse, Kenneth Birman

Department of Computer Science
Cornell University, Ithaca, NY 14853
{*rvr,ken*@cs.cornell.edu}

To a growing degree, applications are expected to be self-configuring and self-managing, and as the range of permissible configurations grows, this is becoming an enormously complex undertaking. Indeed, the management subsystem for a distributed system is often more complex than the application itself. Yet the technology options for building management mechanisms have lagged. Current solutions, such as cluster management systems, directory services, and event notification services, either do not scale adequately or are designed for relatively static settings.

In this paper, we describe a new information management service called Astrolabe. Astrolabe monitors the dynamically changing state of a collection of distributed resources, reporting summaries of this information to its users. Like DNS, Astrolabe organizes the resources into a hierarchy of domains, which we call *zones* to avoid confusion, and associates attributes with each zone. Unlike DNS, the attributes may be highly dynamic, and updates propagate quickly; typically, in tens of seconds.

Astrolabe continuously computes summaries of the data in the system using on-the-fly aggregation. The aggregation mechanism is controlled by SQL queries, and can be understood as a type of data mining capability. For example, Astrolabe aggregation can be used to monitor the status of a set of servers scattered within the network, to locate a desired resource on the basis of its attribute values, or to compute a summary description of loads on critical network components. As this information changes, Astrolabe will automatically and rapidly recompute the associated aggregates and report the changes to applications that have registered their interest.

The Astrolabe system looks to a user much like a database, although it is a virtual database that does not reside on a centralized server. This database presentation extends to several aspects. Most importantly, each zone can be viewed as a relational table containing the attributes of its child zones, which in turn can be queried using SQL. Also, using database integration mechanisms like ODBC and JDBC standard database programming tools can access and manipulate the data available through Astrolabe.

The design of Astrolabe reflects four principles:

1. *Scalability through hierarchy:* Astrolabe achieves scalability through its zone hierarchy. Given bounds on the size and amount of information in a zone, the computational, storage and communication costs of Astrolabe are also bounded.

2. *Flexibility through mobile code:* A restricted form of mobile code, in the form of SQL aggregation queries, allows users to customize Astrolabe on the fly.

3. *Robustness through a randomized peer-to-peer protocol:* Systems based on centralized servers are vulnerable to failures, attacks, and mismanagement. Astrolabe agents run on each host, communicating through an epidemic protocol that is highly tolerant of failures, easy to deploy, and efficient.

*This research was funded in part by DARPA/AFRL-IFGA grant F30602-99-1-0532, in part by a grant under NASA's REE program administered by JPL, in part by NSF-CISE grant 9703470, and in part by the AFRL/Cornell Information Assurance Institute. A full version of this paper is available as http://www.cs.cornell.edu/ken/Astrolabe.pdf.

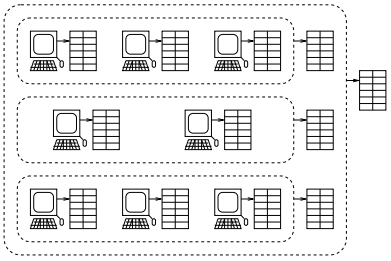

Figure 1: An example of a three-level Astrolabe tree. The top-level *root zone* has three child zones. Each zone, including the leaf zones (the hosts), has an attribute list. Each host runs a Astrolabe agent.

4. *Security through certificates:* Astrolabe uses digital signatures to identify and reject potentially corrupted data and to control access to potentially costly operations.

Zones

Astrolabe gathers, disseminates and aggregates information about *zones*. The structure of Astrolabe's zones can be viewed as a tree. The leaves of this tree represent the hosts, while the root contains all hosts (see Figure 1).

Associated with each zone is a unique identifier, a *pathname*, and an *attribute list* which contains the information associated with the zone. Borrowing terminology from SNMP, we call this attribute list a Management Information Base or MIB.

The Astrolabe attributes are not directly writable, but generated by so-called *aggregation functions*. Each zone has a set of aggregation functions that calculate the attributes for the zone's MIB. An aggregation function for a zone is an SQL program that takes a list of the MIBs of the zone's child zones, and produces a summary of their attributes.

Leaf zones form an exception. Each leaf zone has a set of *virtual child zones*, local to the corresponding agent. These attributes of these virtual zones are updated directly from some local data source. Standard data sources are available from a virtual child zone called "system", and the agent also allows new virtual child zones to be created.

The MIB of any zone is required to contain at least the following attributes: *id*: the zone identifier; *issued*: a timestamp for the version of the MIB, used for the replacement strategy in the epidemic protocol, as well as for failure detection; *contacts*: a small set of addresses for representative agents of this zone, used for the peer-to-peer protocol that the agents run; *nmembers*: the total number of hosts in the zone, constructed by taking the sum of the *nmembers* attributes of the child zones.

Aggregation Function Certificates

Aggregation functions are programmable. The code of these functions is embedded in so-called *aggregation function certificates* (AFCs), which are signed certificates installed as attributes inside MIBs. The evaluation of these AFCs produces values for the corresponding attributes. However, we also use AFCs for other purposes. An *Information Request AFC* specifies what information the application wants to retrieve at each participating host, *and* how to aggregate this information in the zone hierarchy. (both are specified using SQL queries). A *Configuration AFC* specifies run-time parameters that applications may use for dynamic on-line configuration. In the future, we are considering adding *Action AFCs* which might specify actions to be taken on hosts managed by Astrolabe, but before doing so many security implications would need to be addressed.

Agents and Gossip

Each host runs an Astrolabe agent, which keeps a local copy of a subset of the Astrolabe zone tree. These include the zones on the path to the root, as well as their sibling zones. In particular, each agent has a local copy of the root MIB, and the MIBs of each child of the root. Note that there are no centralized servers associated with internal zones; their MIBs are replicated on all agents within those zones.

Although replicated, zone information is not replicated in lock-step: different agents in a zone are not guaranteed to have identical copies of MIBs even if queried at the same time, and if updates to a leaf attribute are very rapid, some agents might even miss updates that other agents perform. However, the Astrolabe protocols guarantee that MIBs do not lag behind using an old version of a MIB forever. Instead, Astrolabe implements a probabilistic consis-

tency model under which, if updates to the leaf MIBs cease for long enough, an operational agent is arbitrarily likely to reflect all the updates that has been seen by other operational agents.

Astrolabe propagates information using an epidemic peer-to-peer protocol known as *gossip* [3]. The basic idea is simple: periodically, each agent selects some other agent and exchanges state information with it. If the two agents are in the same zone, the state exchanged relates to MIBs in that zone; if they are in different zones, they exchange state associated with the MIBs of their least common ancestor. In this manner, the states of Astrolabe agents converge as data ages. Similarly, AFCs are disseminated using the gossip protocol.

Communication

We have tacitly assumed that Astrolabe agents have a simple way to address each other and exchange gossip messages. Unfortunately, in this age of firewalls, Network Address Translation (NAT), and DHCP, many hosts have no way of addressing each other, and even if they do, firewalls often stand in the way of establishing contact. One solution would have been to e-mail gossip messages between hosts, but we rejected this, among others, for efficiency considerations. We also realized that IPv6 may still be a long time in coming, and that IT managers are very reluctant to create holes in firewalls.

We currently offer two solutions to this problem. Both solutions involve HTTP as the communication protocol underlying gossip, and rely on the ability of most firewalls and NAT boxes to set up HTTP connections from within a firewall to an HTTP server outside the firewall, possibly through an HTTP proxy server. One solution deploys Astrolabe agents on the *core Internet* (reachable by HTTP from anywhere), while the other is based on *Relay Servers* (or *Rendez-Vous Servers*) such as used by AOL Instant Messenger, Groove, and JXTA. The two solutions are mutually compatible and can be used at the same time.

Administrative control

Although the administration of Astrolabe as a whole is decentralized, each zone is centrally administered. Each zone may have its own administrator, even if one zone is nested within another. The administrator for a zone is responsible for creating its child zones, signing AFCs that are distributed within the zone, setting protocol parameters such as the gossip rate for that zone, and managing its keys.

API

Applications invoke Astrolabe interfaces through calls to a library (see Table 1). The library allows applications to peruse all the information in the Astrolabe tree, setting up new connections as necessary. The creation and termination of connections is transparent to application processes, so the programmer can think of Astrolabe as one single service.

Besides a native interface, the library has an SQL interface that allows applications to view each node in the zone tree as a relational database table, with a row for each child zone and a column for each attribute. The programmer can then simply invoke SQL operators to retrieve data from the tables. Using selection, join, and union operations, the programmer can create new views of the Astrolabe data that are independent of the physical hierarchy of the Astrolabe tree. An ODBC driver is available for this SQL interface, so that many existing database tools can use Astrolabe directly, and many databases can import data from Astrolabe.

Example: Peer-to-peer Multicast

Many distributed games and other applications require a form of multicast that scales well, is fairly reliable, and does not put a TCP-unfriendly load on the Internet. In the face of slow participants, the multicast protocol's flow control mechanism should not force the entire system to grind to a halt. This section describes such a multicast facility. It uses Astrolabe for control, but sets up its own tree of TCP connections for actually transporting messages.

Each multicast group has a name, say "game". The participants notify their interest in receiving messages for this group by installing their TCP/IP address in the attribute "game" of their leaf zone's MIB. This attribute is aggregated using the query

```
SELECT FIRST(3, game) AS game
```

Method	Description
find_contacts(time, scope)	search for Astrolabe agents in the given *scope*
set_contacts(addresses)	specify addresses of initial agents to connect to
get_attributes(zone, event_queue)	report updates to attributes of *zone*
get_children(zone, event_queue)	report updates to zone membership
set_attribute(zone, attribute, value)	update the given attribute

Table 1: Application Programmer Interface.

That is, each zone selects three of its participants' TCP/IP addresses.

Participants exchange messages of the form (zone, data). A participant that wants to initiate a multicast lists the child zones of the root domain, and, for each child that has a non-empty "game" attribute, sends the message (child-zone, data) to a selected participant for that child zone (more on this selection later). Each time a participant receives a message (zone, data), it finds the child zones of the given zone that have non-empty "game" attributes and recursively continues the dissemination process.

The TCP connections that are created are cached. This effectively constructs a tree of TCP connections that spans the set of participants. This tree is automatically updated as Astrolabe reports zone membership updates.

To make sure that the dissemination latency does not suffer from slow participants in the tree, some measures must be taken. First, a participant could post (in Astrolabe) the rate of messages that it is able to process. The aggregation query can then be updated as follows to select only the highest performing participants for "internal routers."

```
SELECT
   FIRST(3, game) AS game
ORDER BY rate
```

Senders can also monitor their outgoing TCP pipes. If one fills up, they may want to try another participant for the corresponding zone. It is even possible to use more than one participant to construct a "fat tree" for dissemination, but then care should be taken to reconstruct the order of messages. These mechanisms together effectively route messages around slow parts of the Internet, much like Resilient Overlay Networks [1] accomplishes for point-to-point traffic.

In Publish/Subscribe systems [5], receivers subscribe to certain topics of interest, and publishers post messages to topics. The multicast protocol described above can be easily modified to implement Publish/Subscribe, and a generalized concept that we call *selective multicast* or *selective Publish/Subscribe*.

The idea is to tag messages with a SQL condition, chosen by the publishers. For example, a publisher that wants to send an update to all hosts that have a version of some object that is less than 3.1, this condition could be "`MIN(version) < 3.1`". The participants in the multicast protocol above use this condition to decide to which other participants to forward the message. In the example used above, when receiving a message (zone, data, condition), a participant executes the following SQL query to find out which participants to forward the message to:

```
SELECT game
FROM zone
WHERE condition
```

In this idea, the publisher specifies the set of receivers. Basic Publish/Subscribe can then be expressed as the publisher specifying that a message should be delivered to all subscribers to a particular topic.

The simplest way to do this is to create a new attribute by the name of the topic. Subscribers set this attribute to 1, and the attribute is aggregated by taking the sum. The condition is then "`attribute` > 0". However, this does not scale well in case there are many topics.

A solution that scales much better is to use a Bloom filter [2].[1] This solution uses a single attribute

[1] This idea is also used in the directory service of the Ninja system.[4]

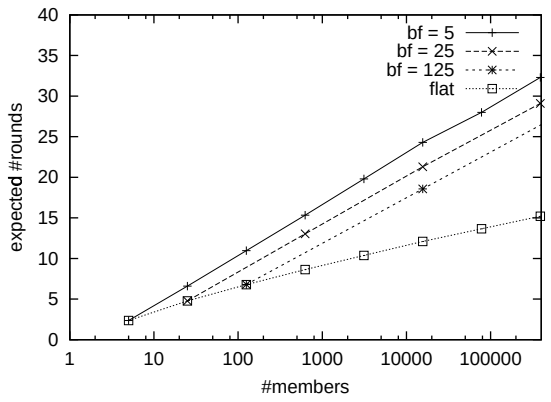

Figure 2: The average number of rounds necessary to infect all participants, using different branching factors. In all these measurements, the number of representatives is 1, and there are no failures.

that contains a fixed-size bit map. The attribute is aggregated using bitwise OR. Topic names are hashed to a bit in this bit map. The condition tagged to the message is "`BITSET(HASH(topic))`". In the case of hash collisions, this may lead to messages being routed to more destinations than strictly necessary.

Scalability

We know that the time for gossip to disseminate in a "flat" population grows logarithmically with the size of the population, even in the face of network links and participants failing with a certain probability [3, 6]. The question is, is this also true in Astrolabe, which uses a hierarchical protocol? The answer appears to be yes, albeit somewhat slower. To demonstrate this, we conducted simulated experiments. The results of one such experiment is described here.

We simulated up to 5^8 (390,625) members. Gossip occurred in rounds, with all members gossiping at the same time. We assumed that successful gossip exchanges complete within a round. (Typically, Astrolabe agents are configured to gossip once every 2-5 seconds, so this assumption seems reasonable.) Each experiment was conducted at least ten times. (For small numbers of members much more often than that.) In both experiments, the variance observed was low.

In this experiment, we varied the branching factor of the tree. We used branching factors 5, 25, and 125 (that is, 5^1, 5^2, and 5^3). There was one representative (contact) per zone, and no failures. We measured the average number of rounds necessary to disseminate information from one node to all other nodes. We show the results in Figure 2 (on a log scale), and compare these with flat (non-hierarchical) gossip. Flat gossip is impractical in a real system, as the required memory grows linearly, and network load quadratically with the membership, but it provides a useful baseline.

Flat gossip provides the lowest dissemination latency. The corresponding line in the graph is slightly curved, because Astrolabe agents never gossip to themselves, which significantly improves performance if the number of members is small. Hierarchical gossip also scales well, but is slower than flat gossip. Latency improves when the branching factor is increased, but this also increases overhead.

For example, at 390,625 members and branching factor 5, there are 8 levels in the tree. Thus each member only has to maintain and gossip only $8 \times 5 = 40$ MIBs. (Actually, since gossip messages do not include the MIBs of the destination agent, the gossip message only contains 32 MIBs.) With a branching factor of 25, each member maintains and gossips $4 \times 25 = 100$ MIBs. In the limit (flat gossip), each member would maintain and gossip an impractical $390,625$ MIBs.

Figure 3 shows a typical output of experiments we have conducted into the scalability of the current Astrolabe implementation. We configured three sets of 16 450 MHz Pentium machines, and one set of 16 400 MHz Xeon machines into a variety of regular trees, with branching factors of 64, 8, and 4, and three representatives per zone. The machines were connected using Gigabit Ethernet switches. Each machine ran one Astrolabe agent. The agents gossiped at a rate of one exchange every two seconds over UDP. Data was then injected at the agent with the longest gossiping distance to all other agents, and we measured worst case dissemination times. Each experiment was run 100 times, and Figure 3 shows how

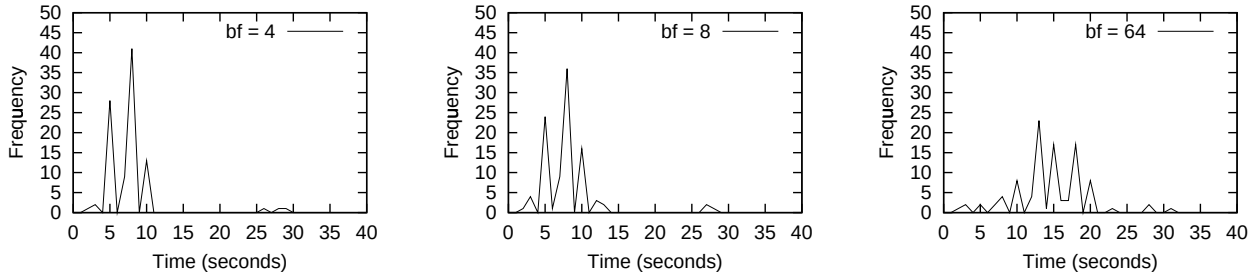

Figure 3: Latency distribution measurements in a system of 64 agents using the given branching factors.

many times each latency occurred, quantitized in 1 second bins.

The peaks seen at 2 second intervals are explained because the agents gossip in rounds; jittering the gossip interval would provide a smoother background load. Although it can be seen that most agents quickly obtain the new data, occasionally there appears to be at least one agent that takes a longer time. It turned out that these "late" values were all from a single agent that apparently has a broken network interface. The experiment thus highlights the robustness of our protocols, at least to this type of failure. Elsewhere, we plan to report a more comprehensive evaluation that looks at Astrolabe analytically, through simulation, and through a detailed experimental evaluation in larger configurations, under stress, and with high rates of updates.

For a detailed description of the Astrolabe protocols and examples of its applications, as well as an extensive discussion of related work, please read http://www.cs.cornell.edu/ken/Astrolabe.pdf.

Acknowledgements

We would like to thank the following people for various contributions to the Astrolabe design and this paper: Tim Clark, Al Demers, Dan Dumitriu, Terrin Eager, Johannes Gehrke, Barry Gleeson, Indranil Gupta, Kate Jenkins, Anh Look, Yaron Minsky, Andrew Myers, Venu Ramasubramanian, Richard Shoenhair, Emin Gun Sirer, Werner Vogels, and Lidong Zhou.

References

[1] D.G. Andersen, H Balakrishnan, M.F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. of the Eighteenth ACM Symp. on Operating Systems Principles*, pages 131–145, Banff, Canada, October 2001.

[2] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, July 1970.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, August 1987.

[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, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *To appear in a Special Issue of Computer Networks on Pervasive Computing*, 2001.

[5] B. M. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus—an architecture for extensible distributed systems. In *Proc. of the Fourteenth ACM Symp. on Operating Systems Principles*, pages 58–68, Asheville, NC, December 1993.

[6] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. of Middleware'98*, pages 55–70. IFIP, September 1998.