Scalable Management and Data Mining using Astrolabe*

Robbert van Renesse, Kenneth Birman, Dan Dumitriu, Werner Vogels

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

Abstract. Astrolabe is a new kind of peer-to-peer system implementing a hierarchical distributed database abstraction. Although deigned for scalable management and data mining, the system can also support wide-area multicast and offers powerful aggregation mechanisms that permit applications to build customized virtual databases by extracting and summarizing data located throughout a large network. In contrast to other peer-to-peer systems, the Astrolabe hierarchy is purely an abstraction constructed by running our protocol on the participating hosts – there are no servers, and the system doesn't superimpose a specialized routing infrastructure or employ a DHT. This paper focuses on wide-area implementation challenges.

1 Introduction

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, and associates attributes with each domain. Unlike DNS, no servers are associated with domains, 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

* 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.

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 domain can be viewed as a relational table containing the attributes of its child domains, 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 domain hierarchy. Given bounds on the size and amount of information in a domain, 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.
4. *Security through certificates:* Astrolabe uses digital signatures to identify and reject potentially corrupted data and to control access to potentially costly operations.

This paper is organized as follows. Section 2 describes the Astrolabe system itself, while its use is illustrated in Section 3. To avoid overlap with work published elsewhere, we focus on wide-area communication challenges here. Accordingly, the basic low-level communication and addressing mechanisms used by Astrolabe are the subject of Section 4. Section 5 describes Astrolabe's self-configuration strategy. How Astrolabe operates in the presence of firewalls is the topic of Section 6. In Section 7 we describe various related work in the peer-to-peer area. Finally, Section 8 concludes.

2 Astrolabe

The goal of Astrolabe is to maintain a dynamically updated data structure reflecting the status and other information contributed by hosts in a potentially large system. For reasons of scalability, the hosts are organized into a domain hierarchy, in which each participating machine is a leaf domains (see Figure 1). The leafs may be visualized as tuples in a database: they correspond to a single host and have a set of attributes, which can be base values (integers, floating point numbers, etc) or an XML object. In contrast to these leaf attributes, which are directly updated by hosts, the attributes of an internal domain (those corresponding to "higher levels" in the hierarchy) are generated by *aggregating* (summarizing) attributes of its child domains.

The implementation of Astrolabe is entirely peer-to-peer: the system has no servers, nor are any of its agents configured to be responsible for any particular domain. Instead, each host runs an agent process that communicates with other agents through an epidemic protocol or *gossip* [DGH+87]. The data structures and protocols are designed such that the service scales well:

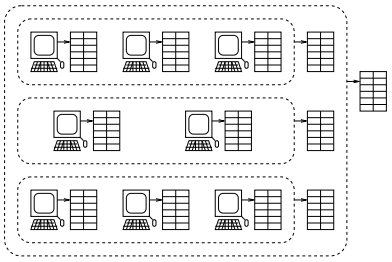

Fig. 1. An example of a three-level Astrolabe tree. The top-level *root domain* has three child domains. Each domain, including the leaf domains (the hosts), has an attribute list. Each host runs a Astrolabe agent.

- The memory used by each host grows logarithmically with the membership size;
- The size of gossip messages grows logarithmically with the size of the membership;
- If configured well (more on this in Section 5), the gossip load on network links grows logarithmically with the size of the membership, and is independent of the update rate;
- The *latency* grows logarithmically with the size of the membership. Latency is defined as the time it takes to take a snapshot of the entire membership and aggregate all its attributes;
- Astrolabe is tolerant of severe message loss and host failures, and deals with network partitioning and recovery.

In practice, even if the gossip load is low (Astrolabe agents are typically configured to gossip gossip only once every few seconds), updates propagate very quickly, and is typically within a minute even for very large deployments [vRB02].

In the description of the implementation of Astrolabe below, we omit all details except those that are necessary in order to understand the function of Astrolabe, and the issues that relate to peer-to-peer communication in the Internet. A much more detailed description and evaluation of Astrolabe appears in [vRB02].

As there is exactly one Astrolabe agent for each leaf domain, we name Astrolabe agents by their corresponding domain names. Each Astrolabe agent maintains, for each domain that it is a member of, a relational table called the *domain table*. For example, the agent "/nl/amsterdam/vu" has domain tables for "/", "/nl", and "/nl/amsterdam" (see Figure 2). A domain table of a domain contains a row for each of its child domains, and a column for each attribute name. One of the rows in the table is the agent's *own* row, which corresponds to that child domain that the agent is a member of as well. Using a SQL aggregation function, a domain table may be aggregated by computing some form of summary of the contents to form a single row. The rows from the children of a domain are concatenated to form that domain's table in the agent, and this repeats to the root of the Astrolabe hierarchy.

Since multiple agents may be in the same domain, the corresponding domain table is replicated on all these agents. For example, both the agents "/nl/amsterdam/vu" and "/nl/utrecht/uu" maintain the "/" and "/nl" tables. A replicated table is kept approxi-

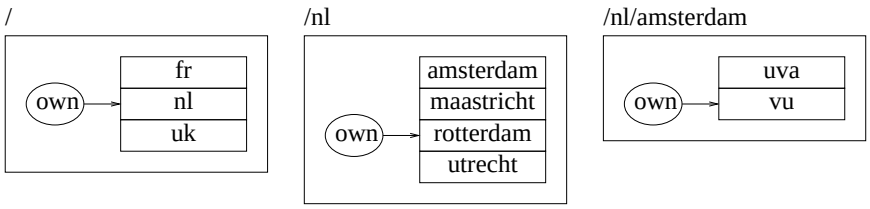

Fig. 2. A simplified representation of the data structure maintained by the agent corresponding to /nl/amsterdam/vu.

mately consistent using an epidemic protocol. Each agent in a domain calculates, using an aggregation function, a small set of representative agents for its domain. Typically, Astrolabe is configured to use up to three representative agents for each domain. The representative agents of the child domains of a parent domain run the epidemic protocol for the parent's domain table. On a regular basis, say once a second, each agent X that is a representative for some child domain chooses another child domain at random, and then a representative agent Y within the chosen child domain, also at random. X sends the parent's table to Y. Y merges this table with its own table, and sends the result back to X, so that X and Y now agree on the contents of their tables.

The rule by which such tables are merged is central to the behavior of the system as a whole. The basic idea is as follows. Y adopts into the merged table rows from X for child domains that were not in Y's original table, as well as rows for child domains that are more current than in Y's original table. To determine currency, agents timestamp rows each time they are updated by writing (in case of leaf domains) or by generation (in case of internal domains). Unfortunately, this requires all clocks to be synchronized which is, at least in today's Internet, far from being the case.

To solve this problem, each row is tagged with the *generator*: the domain name of the agent that wrote or generated the row (in addition to the timestamp). Agents also maintain, for each row in each table, the set of generators from which they received updates for that row, along with the timestamp on the last received update. The merge operation is now executed as follows. When receiving a row in a gossip message, the agent adopts it, as is, if it is created by a previously unknown generator. If it is a known generator, the row is adopted if and only if the row's timestamp is more recent than the last received timestamp from that generator. This way, only timestamps from the same agent are compared with one another, and thus no clock synchronization is necessary.

When no update has been received from a particular generator for some time period T, that generator is considered faulty, and eventually forgotten (after time $2T$ for reasons that go beyond the scope of this paper, but which are described in [vRMH98]). T should be chosen so that the probability of any *old* gossips from this generator still going around is very low [vRB02, vRMH98]. Since gossips disseminate in time $O(\log n)$, this typically is not very long, and can be determined by techniques of epidemic analysis or simulation. If there are no more known generators for a domain, the domain itself is removed from the agent's domain table.

3 Using Astrolabe

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 a ubiquitous service, somewhat analogous to the DNS.

Table 1. Application Programmer Interface.

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

In addition to its native interface, the library has an SQL interface that allows applications to view each node in the domain tree as a relational database table, with a row for each child domain 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.

New aggregation functions can be installed dynamically, and their dissemination piggybacks on the gossip protocol. This way an Astrolabe hierarchy can be customized for the applications that use it. The code of these functions is embedded in so-called *aggregation function certificates* (AFCs), which are signed certificates installed as domain attributes.

We also use AFCs for purposes other than aggregation. An *Information Request AFC* specifies what information the application wants to retrieve at each participating host, *and* how to aggregate this information in the domain hierarchy. (both are specified using SQL queries). A *Configuration AFC* specifies run-time parameters that applications may use for dynamic on-line configuration.

Example: Peer-to-peer Multicast

In [vRB02] we present a number of possible uses for Astrolabe, such as for locating ”nearby” resources by using ODBC to query the local domain, monitoring the dynamically evolving state of a subset of hosts in a large network using an aggregation function, or tracking down desired resources in very large settings. To avoid repeating that material here, we now present a different example of how Mariner might be used. 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 to track the set of multicast recipients, but then but sets up a separate tree of TCP connections for actually transporting multicast messages.

Each multicast group has a name, say "game". A participant expresses its interest in receiving messages for this group by installing its TCP/IP address (or addresses) in the attribute "game" of its leaf domain's MIB. This attribute is aggregated using the query

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

That is, each domain selects three of its participants' TCP/IP addresses. (FIRST is an often-used Astrolabe extension to SQL.)

Participants exchange messages of the form (domain, data). A participant that wants to initiate a multicast lists the child domains of the root domain, and, for each child that has a non-empty "game" attribute, sends the message (child-domain, data) to a selected participant for that child domain (more on this selection later). Each time a participant receives a message (domain, data), it finds the child domains of the given domain 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 domain 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 domain. 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 [ABKM01] accomplishes for point-to-point traffic. Notice that this makes our multicast "TCP-friendly", in the sense that if a router becomes overloaded and starts dropping messages, the multicast protocol will reduce the load imposed on that router. This property is rarely seen in Internet multicast protocols.

Our solution can also be used to implement the Publish/Subscribe paradigm. In these systems [OPSS93], receivers subscribe to topics of interest, and publishers post messages to topics. The multicast protocol described above can easily implement Publish/Subscribe, as well as 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 should receive the message. In the example used above, when receiving a message (domain, data, condition), a participant executes the following SQL query to find out which participants to forward the message to:

```
SELECT game
FROM domain
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 accomplish this type of routing 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, such an approach would scale poorly if a system has large number of possible topics, since it requires one bit each.

A solution that scales much better is to use a Bloom filter [Blo70].[1] This solution uses a single attribute 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, a message may reach some non-subscribing destinations, but would be filtered and ignored at the last hop.

4 Low-level Communication

The preceding example showed how a multicast protocol could be layered over Astrolabe, running on TCP channels. However, for its own communication, Astrolabe uses UDP/IP, HTTP (on top of TCP/IP or SSL), or both. To support HTTP, Astrolabe agents act both as HTTP servers and clients. To enable communication through firewalls, Astrolabe makes use of Application Level Gateways, but for scalability concerns this is only done as a last resort (see Section 6). This section discusses some of the challenges that arose in running Astrolabe in wide-area settings. Before we discuss the actual communication in more detail, we will first describe the concept of *realms*, and how addressing is done in Astrolabe.

A *realm* is a set of hosts and a communication protocol. For example, the tuple ("Cornell Computer Science Department", UDP) forms a realm, as does ("Core Internet", HTTP). ("Core Internet" is the set of those hosts on the main Internet that do not reside behind firewalls.) Each realm has a unique identifier of the form name:protocol, for example "cornellcs:udp" and "internet:http". The hosts in a realm form an equivalence class, in that they can all be accessed using the realm's protocol in the same way. A host can be in more than one realm, and can have more than one address in the same

[1] This idea is also used in various other distributed systems, including the directory service of the Ninja system [GWVB+01].

realm. No two hosts in the same realm can have the same address, but the same address may be used in different realms.

UDP addresses are of the form “IP-address:port” (e.g., “10.0.0.4:6422”) or “DNS-name:port” (e.g., “rome.cs.cornell.edu:6422”). HTTP addresses are of the form “agent-name@TCP-address”, where “agent-name” is the Astrolabe domain name of the agent, and “TCP-address,” as in UDP addresses, consists of a port and either an IP address or a DNS name. For example, “/usa/ny/ithaca/cornell/cs/rome@10.0.0.4:2246”.

We define an *extended address* to be the triple (realm identifier, address, preference). For example, (“cornellcs:udp”, 10.0.0.4:6422, 5). A host has a set of these addresses, and can indicate its preference for certain addresses using the *preference* field. We call the set of extended addresses of a host the *contact* for that host. The contact is dynamic as addresses may appear and disappear over time as the administrator of the host connects to, or disconnects from, ISPs and VPNs.

Unlike agent’s contacts, agent’s names are constant. In order to simplify configuration, we observed that realms often coincide with Astrolabe domains, and thus named realms using their corresponding domain name. Thus, rather than “cornellcs:udp”, we would use “/usa/ny/Ithaca/cornell/cs:udp”. The core Internet coincides with the root domain, and is thus called “/:http”.

Each domain in Astrolabe has an attribute called *contacts*, which contains the contacts of those agents in the domain that have been elected as representatives. (This uses the FIRST function that was described in Section 3.) The *contacts* attribute of a leaf domain contains the singleton set with the contact of the agent of that leaf domain.

We will now briefly revisit Astrolabe’s gossip protocol to show how this works in practice. When an agent wants to gossip the table of some domain, it has to come up with an address. First, the agent picks one of the table’s rows at random and retrieves the *contacts* attribute from that row. The agent then picks one of the contacts at random. The resulting contact is a set of extended addresses. The agent removes the addresses of realms that it cannot reach (more on this below). If there is more than one remaining address, the agent has to make one more choice.

In order to make intelligent choices, each Astrolabe agent maintains statistics about addresses. This is simple to do, as each gossip message is followed by a response. Currently, an agent maintains for each extended address the following three values:

1. `outstanding`: the number of gossips sent since the last response was received;
2. `last_sent`: time of last send of a gossip;
3. `last_received`: time of last reception of a response.

If there is more than one extended address to choose from, the agent *scores* each address:

1. If there is no outstanding gossip, the score is the preference;
2. If it has been more than a minute since the last gossip was sent, the score is the preference;
3. If there is just one outstanding gossip, the score is the preference minus one;
4. In all other cases, the score is zero.

This results in the following behavior. In the normal case, when gossips are followed by responses, the address of the highest preference is always used. If a single response

got lost, the score becomes only slightly smaller. The intention is that if there is more than one address of the same preference, the ones that are only somewhat flaky become less preferential. If there are more losses, the score becomes such that the address is only used as a last resort. Once a minute, the score is, for a single send operation, reset to the original preference. This allows addresses to be occasionally re-tested.

5 Configuration

In order for Astrolabe to scale well, the domain hierarchy has to be set up with care. Each domain in Astrolabe runs an instance of the gossip protocol among the representatives of its child domains. The first concern to think about is the size of domains, that is, the number of child domains in a domain. If very large, the size of gossip messages, as well as the generated gossip load, will be large as well (they both grow linearly with the domain size). If chosen to be very small, the hierarchy becomes very deep, and latency will suffer. In practice, we find that a size of 25-100 child domains in a domain works well. Smaller sizes are possible too, at the cost of some additional latency, but larger sizes make the load unacceptably large.

Locality is a second important consideration. Domains should be constructed (if possible) so that the number of network hops between its child domains' representatives is minimized, and so that independent domains (one domain is not an ancestor of the other) do not share any network links. If the Internet were a tree topology, the Astrolabe hierarchy should be preferably identical to this tree. In reality the edges of the Internet often resemble a tree topology, but the internal Internet is a complicated mesh of links that defies any resemblance to a tree.

In practice, this means that there is considerable freedom in designing the higher levels of the Astrolabe hierarchy, within the limits of the branching factor, but the lower levels should be mapped closely to the topology of the Internet edge. If we ignore the branching factor, this is not much different from the DNS hierarchy design. In DNS, too, the low levels often correspond closely to the network topology, while the high levels of the hierarchy have little correspondence to the actual network topology. Thus the main difference between DNS and Astrolabe configuration is the constrained branching factor of the Astrolabe hierarchy.

Astrolabe supports two forms of configuration: manual and automatic. The manual configuration supports various notions of security, including an integrated PKI infrastructure for the Astrolabe service. The automatic configuration is not secure. In order to foil all but the simplest forms of compromise, the communication is scrambled and signed using secret keys. Below, we will focus on the insecure automatic configuration. More on Astrolabe security can be found in [vRB02].

In an insecure version of Astrolabe, all an agent needs to know is

- Its domain name;
- The set of realms that it can send messages to;
- How to find peer agents to gossip with.

In the remainder of this section, we will look at the automatic configuration of domain names and realms.

Currently, we generate the Astrolabe domain name of an agent from the DNS domain name of the host, and the process identifier of the agent. We will first explain how this is done, and then provide the rational for this design. Say that the DNS domain name is $C_0.C_1.....C_k$, and the process id of the agent is P. We use a one-way hash function on $C_1...C_k$ (all but the first component of the domain name) to construct three 6-bit integers, A_1, A_2, and A_3. Finally, the Astrolabe domain name is constructed to be $/C_k/A_1/A_2/A_3/C_{k-1}/.../C_0/P$.

For example, say an agent runs as process 4365 on host "rome.cs.cornell.edu". By hashing "cs.cornell.edu" onto three 6-bit integers, we have effectively split the ".edu" domain up into 2^{18} pieces, as the ".edu" domain itself is much too large for a single Astrolabe domain. Using this construction, the Astrolabe "/edu" domain itself has at most 64 child domains. Say the three generated integers in our example are 25, 43, and 4 respectively. Then the domain name of the agent is "/edu/25/43/4/cornell/cs/rome/4365". (The three generated domains can be hidden from view if so desired.)

The hope is that each of the domains following "/edu/25/43/4" are of relatively limited size that can be supported by the Astrolabe protocol, and that these domains reflect the network topology to a close enough approximation. If in the future this turns out to be insufficient, we can update the downloadable executable to use more levels, or perhaps come up with an adaptive scheme. The addition of the process identifier makes it possible to run multiple agents on the same host.

Next we have to determine the set of realms that the agent can reach. We assume that any agent can communicate to the "/:http" realm, that is, any agent can use HTTP to reach another agent on the core Internet. (Agents may use WPAD to determine the existence and location of an HTTP proxy automatically.) Furthermore, we assume that $/C_k/A_1/A_2/A_3/C_{k-1}/.../C_1$:udp ("/edu/25/43/4/cornell/cs:udp" in our example) is a realm, and that all agents within this realm can communicate with one another.

Currently, these are all the assumptions we make about realms. In practice this is sometimes conservative, and in that case agents that can communicate directly using UDP will use an Application Level Gateway (ALG) instead. In the next section, we describe how ALGs are configured and used.

6 Communication through an ALG

An Application Level Gateway (ALG) may be the only possibility for two agents to communicate (see Figure 3). Significant care should be taken in deploying and configuring ALGs. The number of ALGs is likely to be small compared to the number of agents using them, and thus they should be used judiciously in order not to overload them or the network links that connect them. Also, care should be taken that the peer-to-peer network remains tolerant of failures, and does not get partitioned when a single ALG server crashes or otherwise becomes unavailable, and that network security is not compromised. Finally, in order for the system to scale, it should be possible to add new ALG servers dynamically to the system as the number of Astrolabe agents grows. These new servers should automatically be discovered and used by the existing agents as well as the new ones.

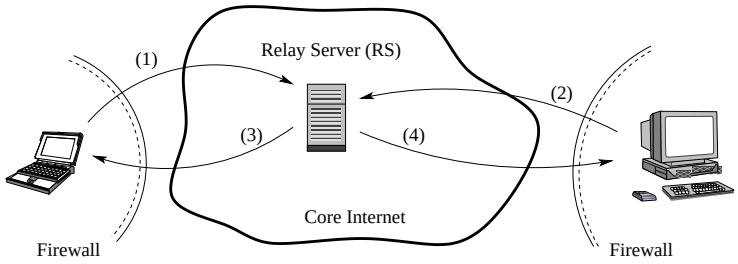

Fig. 3. Application Level Gateway. (1) Receiver sends a RECEIVE request using an HTTP POST request; (2) Sender sends the message using a SEND request using an HTTP POST request; (3) ALG forwards the message to the receiver using an HTTP 200 response; (4) ALG sends an empty HTTP 200 response back to the sender.

Ideally, an ALG is located on the network path between a sender and a receiver, so that the number of hops that a message has to travel is not severely affected by the presence of an ALG. Since many senders may send messages to the same receiver, it follows that the ALG should be located as close to the receiver as possible. Thus, ideally, each firewall or NAT box has a companion ALG that serves receivers behind the firewall. In practice, we suspect that far fewer ALGs will be deployed, but it is still important for receivers to connect to the nearest-by ALG or ALGs.

Each receiver that wishes to receive messages through an ALG has to use HTTP requests to the ALG. In practice, this happens over a persistent TCP connection. In order to reduce the number of such connections to an ALG, not every host behind a firewall has to connect to the ALG. In Astrolabe, only representatives for the realm corresponding to the firewalled site gossip beyond the firewall boundaries, and only these agents (typically, two or three), need to receive through an ALG. The other agents learn indirectly of updates outside the realm through gossip with the representatives (see Figure 4).

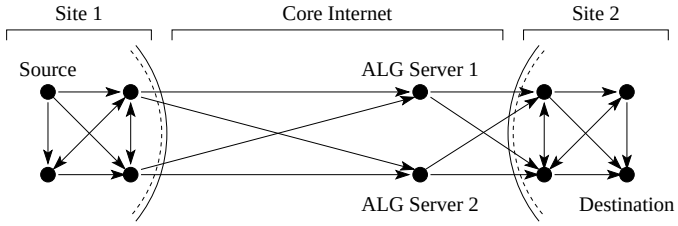

Fig. 4. The many ways gossip can travel from a source host in Site 1 to a destination host in Site 2. Each site has four hosts, two of which are representatives, behind a firewall. The representatives of Site 2 connect to two different ALG servers to receive messages from outside their firewall.

In order for agents to locate ALGs, the ALGs themselves are situated in the Astrolabe hierarchy itself. Each ALG has a companion Astrolabe agent with a configured domain name. The *relays* attribute of the corresponding leaf domain is set to the singleton set containing the TCP/IP address of the ALG. This attribute is aggregated into internal domains in the same way as the *contacts* attribute (*i.e.,* using the FIRST aggregation operator).

An agent determines whether it is a representative for a firewalled site by monitoring the *contacts* attribute of the corresponding realm domain and noticing whether its contact is in there. When this becomes the case, the agent finds ALGs by traveling up the Astrolabe hierarchy starting in its realm and finding the *relays* attributes, stopping when it has located k ALGs or when it reaches the root domain. To ensure fault tolerance, k is typically chosen to be a small integer such as 2 (as in Figure 4). If no ALGs are found, agents resort to using a set of built-in addresses of ALG servers that we deployed for this purpose.

For each ALG in the set, the agent generates a new extended address of the form ("domain-name@ALG", "/:http", preference), and adds this address to its contact set. The preference is chosen to be relatively low compared to its other addresses, so as to discourage its use. Finally, the agent sends an HTTP request to the ALG to receive the first message on this address.

7 Related Work

The most popular peer-to-peer systems such as Chord [SMKK95], Pastry [RD01], and Tapestry [ZKJ01] implement distributed hash tables (DHT), then use these tables to locate desired objects. Just as a conventional hash table maps a key to a value, a DHT maps a key to a location in the network. The host associated with that location stores a copy of the value associated with the key. The hosts that implement the DHT maintain routing tables of $O(\log N)$ size that allow messages to be routed in $O(\log N)$ steps to one of these locations.

Astrolabe also implements a distributed data structure, although it is neither a DHT nor a distributed file system. Instead, Astrolabe most closely resembles a spreadsheet, particularly because updates of an attribute causes other attributes to change. However, the Astrolabe interface is more like that of a database. Athough Astrolabe uses a peer-to-peer protocol with scalability properties similar to those of DHTs, it differs in important ways from the best known DHT solutions.

In particular, Astrolabe reflects the physical organization of the hosts in its domain hierarchy, while DHT implementations hide the hosts and present a uniform key-value mapping. This difference is also apparent in the underlying protocols. Astrolabe exploits the domain organization so that most message exchanges are between nearby hosts, while the DHT implementations require special mechanisms to avoid doing large numbers of long distance exchanges. Pastry and Tapestry come closest to exploiting proximity in their message routing, but the protocols that maintain the routing tables themselves still require mostly long distance messages. Also, these protocols treat each network link as equal, and are expected to suffer from significant problems with slow modem links.

The designers of Tapestry are investigating a two-level architecture, called Brocade [JK02], that exploits the network topology. Basically, each site (*e.g.*, an organization or campus) deploys a number of so-called supernodes. The supernodes are organized in an ordinary Tapestry network. Each site then deploys another Tapestry network locally, including its supernodes. Messages are now routed in three superhops: first to a local supernode, then to the remote supernode, and lastly to its final destination. Simulation shows dramatic improvements in performance.

Astrolabe is perhaps most similar to a multi-level Brocade architecture. Astrolabe's aggregation facilities are used to elect, for each domain, the most appropriate supernodes. Election may be done based on quality of connectivity, resource capacity, host security, etc., and these policies may in fact be changed on the fly. Another difference between Astrolabe and other peer-to-peer protocols is therefore that Astrolabe exploits the available heterogeneity in the network, rather than hiding it.

Much of Astrolabe is concerned with the details of peer-to-peer communication in the actual Internet, an environment rife with Network Address Translation and other inconveniences. Well-known technologies in this field are Groove Networks (groove.net) and JXTA (jxta.org). Groove Networks provide a peer-to-peer communications technology for distributed collaboration. Although in theory peers in Groove can communicate directly with one another (assuming they are not separated by firewalls or NAT), they heavily rely on their proprietary ALG, called the *Groove Relay Server* [Ora01]. Unless peers are explicitly configured to communicate directly with one another, they will use the Relay Server for communication. They also use the Relay Server for other functions. This includes message queuing for off-line peers and resource discovery. This makes most Groove applications heavily dependent on the Relay Server, and direct peer-to-peer interactions are rarely used.

JXTA [Gon01] is an open platform for peer-to-peer interactions in the network, intended for pervasive use from servers to workstations to PDAs and cell phones. JXTA offers a variety of services such as Peer Discovery and Peer Membership. In order to allow peer discovery and peer-to-peer communication, the notion of an ALG (*Rendez-Vous Server* in JXTA terminology) has been proposed, but this is still an ongoing research effort.

8 Conclusions

By combining peer-to-peer and gossip protocols and using the resulting mechanism to implement a scalable database abstraction, Astrolabe plugs a gap in the existing Internet infrastructure. Today, far too many applications are forced to operate in the dark: they lack a good way to sense the status of component systems and of the network, and yet need to adapt their behavior on the basis of such status. More broadly, there is an important need for better scalable computing tools addressing the communications and configuration requirements of large-scale applications. Astrolabe offers such tools, packaged in an easily used database abstraction. Because Astrolabe itself is stable under stress and extremely scalable, it promotes the development of new kinds of applications sharing these properties.

Acknowledgements

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

References

[ABKM01] 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.

[Blo70] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, July 1970.

[DGH+87] 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.

[Gon01] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, May/June 2001.

[GWVB+01] 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.

[JK02] A.D. Joseph and J.D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

[OPSS93] 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.

[Ora01] A. Oram, editor. *Peer-To-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the Middleware 2001*, November 2001.

[SMKK95] I. Stoica, R. Morris, D. Karger, and M.F. Kaashoek. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the '95 Symp. on Communications Architectures & Protocols*, Cambridge, MA, August 1995. ACM SIGCOMM.

[vRB02] R. van Renesse and K.P. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 2002. Submitted for review.

[vRMH98] 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.

[ZKJ01] B.Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, Computer Science Department, 2001.