Astrolabe: A Robust and Scalable Technology For Distributed System Monitoring, Management, and Data Mining*

Robbert van Renesse, Kenneth P. Birman, and Werner Vogels

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

Scalable management and self-organizational capabilities are emerging as central requirements for a generation of large-scale, highly dynamic, distributed applications. We have developed an entirely new distributed information management system called Astrolabe. Astrolabe collects large-scale system state, permitting rapid updates and providing on-the-fly attribute aggregation. This latter capability permits an application to locate a resource without knowing the machines on which it resides, and also offers a scalable way to track system state as it evolves over time. The combination of features makes it possible to solve a wide variety of management and self-configuration problems. The paper describes the design of the system with a focus upon its scalability. After describing the Astrolabe service, we present examples of the use of Astrolabe for locating resources, publish-subscribe, and distributed synchronization in large systems. Astrolabe is implemented using a peer-to-peer protocol, and uses a restricted form of mobile code based on the SQL query language for aggregation. This protocol gives rise to a novel consistency model. Astrolabe addresses several security considerations using a built-in PKI. The scalability of the system is evaluated using both simulation and experiments; these confirm that Astrolabe could scale to thousands and perhaps millions of nodes, with information propagation delays in the tens of seconds.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design – *network communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming – *distributed programming*; D.4.4 [**Operating Systems**]: Communications Management – *network communication*; D.4.5 [**Operating Systems**]: Reliability – *fault tolerance*; D.4.6 [**Operating Systems**]: Security and Protection – *authentication*; D.4.7 [**Operating Systems**]: Organization and Design – *distributed systems*; H.3.3 [**Information Systems**]: Information Search and Retrieval – *information filtering*; H.3.4 [**Information Systems**]: Information Storage and Retrieval – *distributed systems*; H.3.5 [**Information Systems**]: Online Information Services – *data sharing*.

*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, and in part by NSF-CISE grant 9703470. The authors are also grateful for support from the AFRL/Cornell Information Assurance Institute.

General Terms: Algorithms, monitoring, management, performance, reliability, security.

Additional Key Words and Phrases: membership, failure detection, epidemic protocols, gossip, aggregation, scalability, publish-subscribe.

1 Introduction

With the wide availability of low-cost computers and pervasive network connectivity, many organizations are facing the need to manage large collections of distributed resources. These might include personal workstations, dedicated nodes in a distributed application such as a web farm, or objects stored at these computers. The computers may be co-located in a room, spread across a building or campus, or even scattered around the world. Configurations of these systems change rapidly—failures and changes in connectivity are the norm, and significant adaptation may be required if the application is to maintain desired levels of service.

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 contemporary distributed system (*i.e.*, a Web Services system reporting data collected from a set of corporate databases, file systems, and other resources) 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 (see Section 8).

At the time of this writing, the most widely-used, scalable distributed management system is DNS [18]. DNS is a directory service that organizes machines into domains, and associates attributes (called *resource records*) with each domain. Although designed primarily to map domain names to IP addresses and mail servers, DNS has been extended in a variety of ways to make it more dynamic and support a wider variety of applications. These extensions include *Round-Robin DNS* (RFC 1794) to support load balancing, the *Server* record (RFC 2782) to support service location, and the *Naming Authority Pointer* (RFC 2915) for Uniform Resource Names. To date, however, acceptance of these new mechanisms has been limited.

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, zones are not bound to specific servers, 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, As-

trolabe will automatically and rapidly recompute the associated aggregates and report the changes to applications that have registered their interest.

Aggregation is intended as a *summarizing mechanism*.[1] For example, an aggregate could count the number of nodes satisfying some property, but not to concatenate their names into a list, since that list could be of unbounded size. The approach is intended to bound the rate of information flow at each participating node, so that even under worst-case conditions, it will be independent of system size. To this end, each aggregate is restricted to some scope, within which it is computed on behalf of and visible to all nodes. Only aggregates with high global value should have global scope. The number of aggregating queries active within any given scope is assumed to be reasonably small, and independent of system size. To ensure that applications do not accidentally violate these policies, nodes seeking to introduce a new aggregating function must have administrative rights within the scope where it will be computed.

Initial experience with the Astrolabe aggregation mechanisms demonstrates that the system is extremely powerful despite its limits. Managers of an application might use the technology to monitor and control a distributed application using aggregates that summarize the overall state within the network as a whole, and also within the domains (scopes) of which it is composed. A new machine joining a system could use Astrolabe to discover information about resources available in the vicinity: by exploiting the scoping mechanisms of the aggregation facility, the resources reported within a domain will be those of most likely value to applications joining within that region of the network. After a failure, Astrolabe can be used both for notification and to coordinate reconfiguration. More broadly, any form of loosely coupled application could use Astrolabe as a platform for coordinating distributed tasks. Indeed, Astrolabe uses its own capabilities for self-management.

It may sound as if designing an aggregate to be sufficiently concise and yet to have high value to applications is something of an art. Yet the problem turns out to be relatively straightforward and not unlike the design of a hierarchical database. A relatively small number of aggregating mechanisms suffice to deal with a wide variety of potential needs. Indeed, experience supports the hypothesis that the forms of information needed for large-scale management, configuration and control are generally amenable to a compact representation.

Astrolabe maintains excellent responsiveness even as the system becomes very large, and even if it exhibits significant dynamicism. The loads associated with the technology are small and bounded, both at the level of message rates seen by participating machines and loads imposed on communication links. Astrolabe also has a small "footprint" in the sense of computational and storage overheads.

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, and does not support atomic transactions. 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 [24] and JDBC [22] standard database programming tools can access and manipulate the data available through Astrolabe.

[1] Aggregation is a complex topic. We have only just begun to explore the power of Astrolabe's existing mechanisms, and have also considered several possible extensions. This paper limits itself to the mechanisms implemented in the current version of Astrolabe and focuses on what we believe will be common ways of using them.

The design of Astrolabe reflects four principles:

1. *Scalability through hierarchy:* A scalable system is one that maintains constant, or slowly degrading, overheads and performance as its size increases. Astrolabe achieves scalability through its zone hierarchy. Given bounds on the size and amount of information in a zone, the computational and communication costs of Astrolabe are also bounded. Information in zones is summarized before being exchanged between zones, keeping wide-area storage and communication requirements at a manageable level.

2. *Flexibility through mobile code:* Different applications monitor different data, and a single application may need different data at different times. A restricted form of mobile code, in the form of SQL aggregation queries, allows users to customize Astrolabe by installing new aggregation functions on the fly.

3. *Robustness through a randomized peer-to-peer protocol:* Systems based on centralized servers are vulnerable to failures, attacks, and mismanagement. Instead, Astrolabe uses a peer-to-peer approach by running a process on each host.[2] These processes communicate through an epidemic protocol. Such protocols are highly tolerant of many failure scenarios, easy to deploy, and efficient. They communicate using randomized point-to-point message exchange, an approach that makes Astrolabe robust even in the face of localized overloads, which may briefly shut down regions of the Internet.

4. *Security through certificates:* Astrolabe uses digital signatures to identify and reject potentially corrupted data and to control access to potentially costly operations. Zone information, update messages, configuration, and client credentials, all are encapsulated in signed certificates. The zone tree itself forms the Public Key Infrastructure.

This paper discusses each of these principles. In Section 2, we present an overview of the Astrolabe service. Section 3 illustrates the use of Astrolabe in a number of applications. Astrolabe's implementation is described in Section 4. We describe Astrolabe's security mechanisms in Section 5. In Section 6, we explain how Astrolabe leverages mobile code, while Section 7 describes performance and scalability. Here, we show that Astrolabe could scale to thousands and perhaps millions of nodes, with information propagation delays in the tens of seconds. Section 8 describe various related work, from which Astrolabe borrows heavily. Section 9 concludes.

2 Astrolabe Overview

Astrolabe gathers, disseminates and aggregates information about *zones*. A zone is recursively defined to be either a host or a set of non-overlapping zones. Zones are said to be non-overlapping if they do not have any hosts in common. Thus, the structure of Astrolabe's zones can be viewed as a tree. The leaves of this tree represent the hosts (see Figure 1).

[2]Processes on hosts that do not run a Astrolabe process can still access the Astrolabe service using an RPC protocol to any remote Astrolabe process.

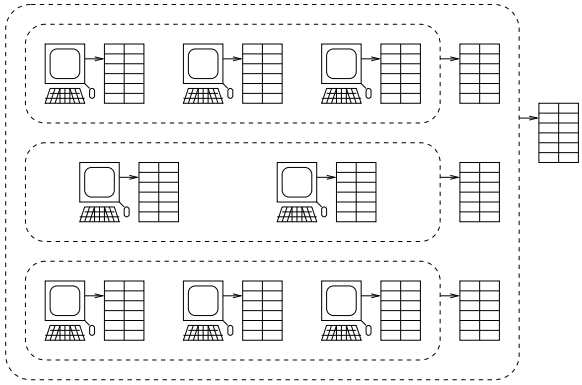


Figure 1: An example of a three-level Astrolabe zone tree. The top-level *root zone* has three child zones. Each zone, including the leaf zones (the hosts), has an attribute list (called a *MIB*). Each host runs a Astrolabe agent.

Each zone (except the root) has a local zone identifier, a string name unique within the parent zone. A zone is globally identified by its *zone name*, which is its path of zone identifiers from the root, separated by slashes. (*e.g.*, "/USA/Cornell/pc3").

Each host runs an Astrolabe agent. The zone hierarchy is implicitly specified when the system administrator initializes these agents with their names. For example, the "/USA/Cornell/pc3" agent creates the "/", "/USA", and "/USA/Cornell" zones if they did not exist already. Thus the zone hierarchy is formed in a decentralized manner, but one ultimately determined by system administrators. As we will see, representatives from within the set of agents are elected to take responsibility for running the gossip protocols that maintain these internal zones; if they fail or become unsuitable, the protocol will automatically elect others to take their places. Associated with each zone is an *attribute list* which contains the information associated with the zone. Borrowing terminology from SNMP [26], this attribute list is best understood as a form of Management Information Base or MIB, although the information is certainly not limited to traditional management information.

Unlike SNMP, 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, which 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*. The virtual child zones are local to the corresponding agent. The attributes of these virtual zones are writable, rather than being generated by aggregation functions. Each leaf zone has at least one virtual child zone called "system", but the agent allows new virtual child zones to be created. For example, an application on a host may create a virtual child zone called "SNMP" and populate it with attributes from the SNMP's MIB. The application would then be responsible for updating Astrolabe's MIB whenever the SNMP attributes change.

Astrolabe is designed under the assumption that MIBs will be relatively small objects – a few hundred or even thousand bytes, not millions. An application dealing with larger objects would not include the object itself into the MIB, but would instead export information about the object, such as a URL for downloading a copy, a time stamp, a version number, or a content summary. In our examples we will treat individual computers as the owners of leaf zones, and will assume that the machine has a reasonably small amount of state information to report.

Astrolabe can also support systems in which individual objects are the leaf zones, and hence could be used to track the states of very large numbers of files, database records, or other objects. Using Astrolabe's aggregation functions, one could then query the states of these objects. However, keep in mind that aggregation functions summarize – their outputs must be bounded in size. Thus, one might design an aggregation function to count all pictures containing images of a very tall, thin, bearded man, or even to list the three pictures with the strongest such match. One could not, however, use aggregation to make a list of *all* such pictures. In fact, it is easy to enumerate the nodes that contribute to a counting aggregate, but this would be done by the application, not the aggregation function.

We can now describe the mechanism whereby aggregation functions are used to construct the MIB of a zone.

Aggregation functions are programmable. The code of these functions is embedded in so-called *aggregation function certificates* (AFCs), which are signed and time-stamped certificates that are installed as attributes inside MIBs. The names of such attributes are required to start with the reserved character '&'.

For each zone it is in, the Astrolabe agent at each host scans the MIBs of its child zones looking for such attributes. If it finds more than one by the same name, but of different values, it selects the most recent one for evaluation. Each aggregation function is then evaluated to produce the MIB of the zone. The agents learns about the MIBs of other zones through the gossip protocol described in Section 4.1.

Thus, if one thinks of Astrolabe as a form of decentralized hierarchical database, there will be a table (a relation) for each zone. Each "column" in a leaf zone is a value extracted from the corresponding node or object. Each column in an inner zone is a value computed by an aggregating function to summarize its children. These columns might be very different from those of the children. For example, the child zones might report loads, numbers of files containing pictures of tall, thin, bearded men, etc. An inner zone could have one column giving the mean load on its children, another counting the total number of matching pictures reported by its children, and a third listing the three child nodes with the strongest matches. In the latter case we would probably also have a column giving the actual quality of those matches, so that further aggregation can be performed at higher levels of the zone hierarchy. However, this isn't required: using Astrolabe's scoping mechanism, we could search for those matching pictures only within a single zone, or within some corner of the overall tree, or within any other well-defined scope. In effect, any two leaf nodes sharing a common ancestral zone will agree on the layout of the MIB for that ancestral zone, but two sibling zones at the same level of the hierarchy could have different schemas and different kinds of data - different columns.

In addition to code, AFCs may contain other information. Two important other uses of AFCs are information requests and run-time configuration. An *Information Request AFC* specifies what information the application wants to retrieve at each participating

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.

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. We will present examples of these uses later in this paper.

Applications invoke Astrolabe interfaces through calls to a library (see Table 1). Initially, the library connects to an Astrolabe agent using TCP. The set of agents from which the library can choose is specified using `set_contacts`. Optionally, eligible agents can be found automatically using `find_contacts`. The `time` parameter specifies how long to search, while the `scope` parameter specifies how to search (*e.g.*, using a broadcast request on the local network). (In the simplest case, an Astrolabe agent is run on every host, so that application processes can always connect to the agent on the local host and need not worry about the connection breaking.)

From then on, the library allows applications to peruse all the information in the Astrolabe tree, setting up connections to other agents as necessary. The creation and termination of connections is transparent to application processes, so the programmer can think of Astrolabe as one single service. Updates to the attributes of zones, as well as updates to the membership of zones, are posted on local *event queues*. Applications can also update the attribute of virtual zones using `set_attribute`.

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. SQL does not support instant notifications of attribute changes, so that applications that need such notifications would need to obtain them using the native interface.

Astrolabe agents also act as web servers, hence information can be browsed and changed using any standard web browser instead of going through the library.

3 Examples

The foregoing overview describes the full feature set of the system, but may not convey the simplicity and elegance of the programming model it enables. The examples that follow are intended to illustrate the power and flexibility that Astrolabe brings to a distributed environment.

3.1 Example 1: Peer-to-peer Caching of Large Objects

Many distributed applications operate on one or a few large objects. It is often infeasible to keep these objects on one central server, and copy them across the Internet whenever a process needs a copy. The loading time would be much too long, and the load on the network too high. A solution is for processes to find a nearby existing copy in the Internet, and to "side-load" the object using a peer-to-peer copying protocol. In this section we look at the use of Astrolabe to locate nearby copies and to manage freshness.

Suppose we are trying to locate a copy of the file 'game.db'. Assume each host has a database 'files' that contains one entry per file. (We have, in fact, written an ODBC driver that makes a host's file system appear like a database.) To find out if a particular host has a copy of the file, we may execute the following SQL query on this host:

```
SELECT
   COUNT(*) AS file_count
FROM files
WHERE name = 'game.db'
```

If `file_count` > 0, the host has at least one copy of the given file. There may be many hosts that have a copy of the file, so we also need to aggregate this information along with the location of the files. For the purpose of this example, assume that each host installs an attribute 'result' containing its host name in its leaf MIB. (In practice, this attribute would be extracted using another query on, say, the registry of the host.) Then, the following aggregation query counts the number of copies in each zone, and picks one host from each zone that has the file. This host name is exported into the 'result' attribute of each zone[3]:

```
SELECT
   FIRST(1, result) AS result,
   SUM(file_count) AS file_count
WHERE file_count > 0
```

We now simply combine both queries in an Information Request AFC (IR-AFC), and install it. As the IR-AFC propagates through the Astrolabe hierarchy, the necessary information is collected and aggregated.

This discussion may make it sound as if a typical application might install new aggregation queries with fairly broad, even global, scope. As noted earlier, this is not the case: such a pattern of use would violate the scalability of the system by creating zones (those near the root) with very large numbers of attributes. An aggregate such as the one just shown should either be of global importance and value, or limited to a smaller zone within which many programs need the result, or used with a short lifetime (to find the file but then "terminate"). But notice, too, that aggregation is intrinsically parallel. The aggregated value seen in a zone is a regional value computed from that zone's children.

In our example, if searching for 'game.db' is a common need, each node doing so can use the aggregate to find a *nearby* copy, within the distance metric used to define

[3] `FIRST(n, attrs)` is an Astrolabe SQL extension that returns the named attributes from the first n rows. It is not necessary to specify the "FROM" clause of the SELECT statement, as there is only one input table. "FROM" is necessary in nested statements, however.

the Astrolabe hierarchy. In effect, many applications can use the same aggregation to search for different files (albeit ones matching the same query). This is a somewhat counter-intuitive property of aggregates and makes Astrolabe far more powerful than would be the case if only the root aggregation value was of interest. Indeed, for many purposes, the root aggregation is almost a helper function, while the values at inner zones are more useful.

In particular, to find the nearest copy of the file, a process would first inspect its most local zone for a MIB that lists a 'result'. If not, it would simply travel up the hierarchy until it finds a zone with a 'result' attribute, or the root zone. If the root zone's 'result' attribute is empty, there is no copy of the file.

In the example above, we did not care about the freshness of the copy retrieved. But now suppose each file maintains a version attribute as well. Now each zone can list the host that has the most recent version of the file as follows:

```
SELECT
   result, version
WHERE version ==
   (SELECT MAX(version))
```

Any process may determine what the most recent version number is by checking the 'version' attribute of the root zone. A process wishing to obtain the latest version can simply go up the tree, starting in its leaf zone, until it finds the right version. A process that wants to download new versions as they become available simply has to monitor the root zone's version number, and then find a nearby host that has a copy.

This example illustrates the use of Astrolabe for data mining and resource discovery. Before we move on to the next example, we will explore how the "raw" information may actually be collected.

ODBC is a standard mechanism for retrieving data from a variety of sources, including relational databases and spreadsheets from a large variety of vendors. The language for specifying which data is to be accessed is SQL. The data itself is represented using ODBC data structures.

We have developed another agent that can connect to ODBC data sources, retrieve information, and post this information in the Astrolabe zone tree. Each host runs both an Astrolabe and an ODBC agent. The ODBC agent inspects certificates in the Astrolabe tree whose names start with the prefix `&odbc:`. These certificates specify which data source to connect to, and what information to retrieve. The agent posts this information into the virtual *system* zone of the local Astrolabe agent. The `&odbc:` certificates may also specify how this information is to be aggregated by Astrolabe. Applications can update these certificates on the fly to change the data sources, the actual information that is retrieved, or how it is aggregated.

With this new agent, the host of an Astrolabe zone hierarchy is no longer a true leaf node. Instead, the host can be understood as a zone whose child nodes represent the objects (programs, files, etc) that reside on the host. Astrolabe now becomes a representation capable of holding information about every object in a potentially worldwide network: millions of hosts and perhaps billions of objects.

Of course, no user would actually "see" this vast collection of information at any one place. Astrolabe is a completely decentralized technology and any given user sees only its parent zones, and those of its children. The power of dynamic aggregation is that even in a massive network, the parent zones can dynamically report on the status

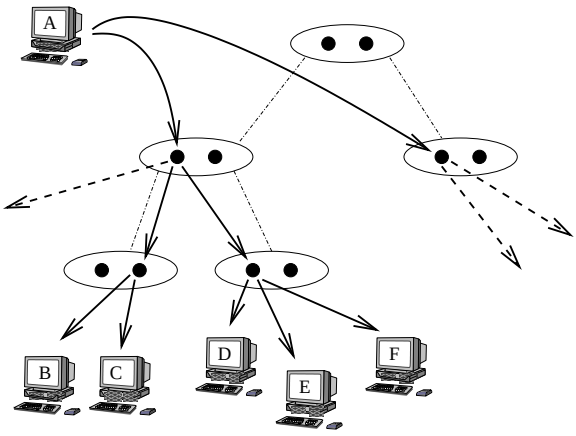


Figure 2: In this example of SelectCast, each zone elects two routers for forwarding messages, even though only one is used for forwarding messages. The sender, **A**, sends the message to a router of each child zone of the root zone.

of these vast numbers of objects. Suppose, for example, that a virus enters the system, and is known to change a particular object. Merely by defining the appropriate AFC, a system administrator would be able to identify infected computers, even if their virus detection software has been disabled.

Other agents for retrieving data are possible. We have written a Windows Registry agent, which retrieves information out of the Windows Registry (including a large collection of local performance information). A similar agent for Unix system information is also available.

3.2 Example 2: Peer-to-peer Data Diffusion

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 SelectCast, a multicast facility we have built using Astrolabe. SelectCast uses Astrolabe for control, but sets up its own tree of TCP connections for actually transporting messages. A full-length paper is in preparation on the SelectCast subsystem; here, we limit ourselves to a high-level summary.

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(2, game) AS game". That is, each zone selects two of its participants' TCP/IP addresses (see Figure 2). We call these participants the *routers* for their zone. Two routers allows for fast recovery in case one fails. If both fail, recovery will also happen, as Astrolabe will automatically select new routers, but this mechanism is relatively slow.

Participants exchange messages of the form (zone, data). A participant that wants to initiate a multicast lists the child zones of the root zone, and, for each child that has a non-empty "game" attribute, sends the message (child-zone, data) to a router for that child zone (more on this selection later). Each time a participant (router or not) receives a message (zone, data), the participant finds the child zones of the given zone that have non-empty "game" attributes and recursively continues the dissemination process.

This approach effectively constructs a tree of TCP connections spanning the set of participants. Each TCP connection is cached so long as the end-points remain active as participants, for re-use (perhaps even by a different query). The tree is updated automatically, when Astrolabe reports zone membership changes, by terminating unneeded TCP connections and creating new ones as appropriate.

To make sure that the dissemination latency does not suffer from slow routers or connections in the tree, some measures must be taken. First, each 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 routers.

```
SELECT
   FIRST(2, 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 all routers for a zone concurrently, thus constructing a "fat tree" for dissemination, but then care should be taken to drop duplicates and reconstruct the order of messages. We are currently investigating this option in our implementation. These mechanisms together effectively route messages around slow parts of the Internet, much like Resilient Overlay Networks [3] accomplishes for point-to-point traffic.

SelectCast does not provide an end-to-end acknowledgement mechanism, and thus there are scenarios in which messages may not arrive at all members (particularly if the set of receivers is dynamic). Additional reliability can be implemented on top of SelectCast. For example, Bimodal Multicast [5] combines a message logging facility with an epidemic protocol that identifies and recovers lost messages, thereby achieving end-to-end reliability (with high probability). Running Bimodal Multicast over SelectCast would also remove Bimodal Multicast's dependence on IP multicast, which is poorly supported in the Internet, and provide Bimodal Multicast with a notion of locality, which may be used to improve the performance of message retransmission strategies.

Notice that each distinct SelectCast instance — each data distribution pattern — requires a separate aggregation function. As noted previously, Astrolabe can only support bounded numbers of aggregation functions at any level of its hierarchy. Thus, while a single Astrolabe instance could probably support as many as several hundred SelectCast instances, the number would not be unbounded. Our third example explores options for obtaining a more general form of Publish/Subscribe by layering additional filtering logic over a SelectCast instance. So doing has the potential of eliminating the restriction on numbers of simultaneously active queries, while still benefiting from the robustness and scalability of the basic SelectCast data distribution tree.

Although a detailed evaluation of SelectCast is outside of the scope of this paper, we have compared the performance of the system with that of other application-level

router architectures and with IP multicast. We find that with steady multicast rates, SelectCast imposes message loads and latencies comparable to other application-level solutions, but that IP multicast achieves lower latencies and lower message loads (a benefit of being implemented in the hardware routers). Our solution has not yet been fully optimized but there is no reason that peak message forwarding rates should be lower than for other application-level solutions, since the critical path for SelectCast (when the tree is not changing) simply involves relaying messages received on an incoming TCP connection into some number of outgoing connections. There is an obvious tradeoff between fanout (hence, work done by the router) and depth of the forwarding tree (hence, latency), but this is not under our control since the topology of the tree is determined by the human administrator's assignment of zone names.

3.3 Example 3: Publish-Subscribe

In Publish/Subscribe systems [19], receivers subscribe to certain topics of interest, and publishers post messages to topics. As just noted, while the SelectCast protocol introduced in the previous section supports a form of Publish/Subscribe (as well as the generalization that we termed *selective multicast*), the mechanism can only handle limited numbers of queries. Here, we explore a means of filtering within a SelectCast instance to obtain a form of subset delivery in which that limitation is largely eliminated.

In the extended protocol, each message is tagged with an SQL condition, chosen by the publisher of the message. Say that a publisher wants to send an update to all hosts having a version of some object less than 3.1. First she would install a SelectCast query that calculates the minimum version number of that object in each zone, and call it, say "`MIN(version`". We call this the *covering query*. Next she would attach the condition "`MIN(version) < 3.1`" to the message. The message is then forwarded using SelectCast.

Recall that in SelectCast, the participants at each layer simply forward each message to the routers, which are calculate for each zone using the SelectCast query. In the extended protocol, the participants in the SelectCast protocol first apply the condition as a filter (using a `WHERE` clause added to the SelectCast query that calculates the set of routers), to decide to which routers to forward the message.[4] Topic-based Publish/Subscribe can then be expressed by having the publisher specify that a message should be delivered to all subscribers to a particular topic. Our challenge is to efficiently evaluate this query without losing scalability. For example, while a new attribute could potentially be defined for each SQL condition in use by the system, doing so scales poorly if there are many conditions.

A solution that scales reasonably well uses a Bloom filter to compress what would otherwise be a single bit per query into a bit vector[8].[5] This solution associates a fixed-size bit map with each covering query. Assume for the moment that our goal is simply to implement Publish/Subscribe to a potentially large number of topics. We define a hashing function on topics, mapping each topic to a bit value. The condition tagged to the message is "`BITSET(HASH(topic))`", and the associated attribute can be aggregated using bitwise OR. In the case of hash collisions, this solution may

[4]The result of the query is cached for a limited amount of time (currently, 10 seconds), so that under high throughput the overhead can be amortized over many messages, assuming they often use the same condition or small set of conditions.

[5]Bloom filters are also used in the directory service of the Ninja system.[14]

lead to messages being routed to more destinations than strictly necessary, which is safe, but inefficient. Thus the size of the bitmap and the number of covering SelectCast queries should be adjusted, perhaps dynamically, so that the rate of collisions will be acceptably low.

Notice that the resulting protocol needs time to react when a condition is used for the first time, or when a new subscriber joins the system, since the aggregation mechanism will need time to update the Bloom filter. During the period before the filter has been updated in Astrolabe (a few tens of seconds), the new destination process might not receive messages intended for it. However, after this warmup period, reliability will be the same as for SelectCast, and performance limited only by the speed at which participants forward messages. As in the case of SelectCast, gossip-based recovery from message logs can be used to make the solution reliable.[6]

With a more elaborate filtering mechanism, this behavior could be extended. For example, the Siena system provides *content-based* Publish/Subscribe[10]. In Siena, subscribers specify information about the content of message they are interested in, while publishers specify information about the content of message they send. Subscribers' specifications are aggregated in the internal routers of Siena, and then matched against the publishers' specifications. By adding such aggregation functionality to Astrolabe's SQL engine, we could extend the above solution to support expressions (rather than just single topic at a time matching) or even full-fledged content addressing in the manner of Siena.

3.4 Example 4: Synchronization

Astrolabe may be used to run basic distributed programming facilities in a scalable manner. For example, barrier synchronization may be done by having a counter at each participating host, initially 0. Each time a host reaches the barrier, it increments the counter, and waits until the aggregate minimum of the counters equals the local counter.

Recall that Astrolabe is currently configured to impose very low background loads at the expense of somewhat slower propagation of new data. More precisely, although gossip rate is a parameter, most of our work uses a gossip rate of once per five seconds.[7] The delay before barrier notification occurs scales as the gossip rate times the log of the size of the system (the logarithmic base being the size of an average zone). If we assume zones of size 64, a system with 500,000 nodes would update global aggregations in about twenty seconds – quite acceptable for settings such as grid computing, where latencies are high in any case.

Similarly, voting can be done by having two attributes, *yes* and *no*, in addition to the *nmembers* attribute, all aggregated by taking the sum. Thus, participants have access to the total number of members, the number that have voted in favor, and the number that have voted against. This information can be used in a variety of distributed algorithms, such as commit protocols.

[6] At the time of this writing, an implementation of reliable Publish/Subscribe over SelectCast was still under development, and a systematic performance evaluation had not yet been undertaken.

[7] It is tempting to speculate about the behavior of Astrolabe with very high gossip rates, but doing so leads to misleading conclusions. Gossip convergence times have a probabilistic distribution and there will always be some nodes that see an event many rounds earlier than others. Thus, while Astrolabe has rather predictable behavior in large-scale settings, the value of the system lies in its scalability, not its speed or real-time characteristics.

Again, the value of such a solution is that it scales extremely well (and can co-exist with other scalable monitoring and control mechanisms). For small configurations, Astrolabe would be a rather slow way to solve the problem. In contrast, traditional consensus protocols are very fast in small configurations, but have costs linear in system size (they often have a 2-phase commit or some sort of circular token-passing protocol at the core). With as few as a few hundred participants, such a solution would break down.

A synchronization problem that comes up with Astrolabe applications is that AFC propagation not only takes some time, but that the amount of time is only probabilistically bounded. How long should a process wait before it can be sure every agent has received the AFC? This question can be answered by summing up the number of agents that have received the AFC, as a simple aggregation query within the AFC itself. When this sum equals the *nmembers* attribute of the root zone, all agents have received the AFC.

4 Implementation

Each host runs an Astrolabe agent. Such an agent runs Astrolabe's gossip protocol with other agents, and also supports clients that want to access the Astrolabe service. Each agent has access to (that is, keeps a local copy of) only a subset of all the MIBs in the Astrolabe zone tree. This subset includes all the zones on the path to the root, as well as the sibling zones of each of those. In particular, each agent has a local copy of the root MIB,[8] and the MIBs of each child of the root. As stated before, there are no centralized servers associated with internal zones; their MIBs are replicated on all agents within those zones.

However, this replication is not lock-step: different agents in a zone are not guaranteed to have identical copies of MIBs even if queried at the same time, and not all agents are guaranteed to perceive each and every update to a MIB. Instead, the Astrolabe protocols guarantee that MIBs do not lag behind using an old version of a MIB forever. More precisely, Astrolabe implements a probabilistic consistency 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. We call this *eventual consistency* and discuss the property in Section 4.3. First we turn our attention to the implementation of the Astrolabe protocols.

4.1 Gossip

Astrolabe propagates information using an epidemic peer-to-peer protocol known as *gossip* [11]. As we will see later, this protocol is scalable, fast, and secure. The basic idea is simple: periodically, each agent selects some other agent at random 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 zone. In this manner, the states of Astrolabe agents will tend to converge as data ages.

[8] Because the root MIB is calculated locally by each agent, it never needs to be communicated between agents.

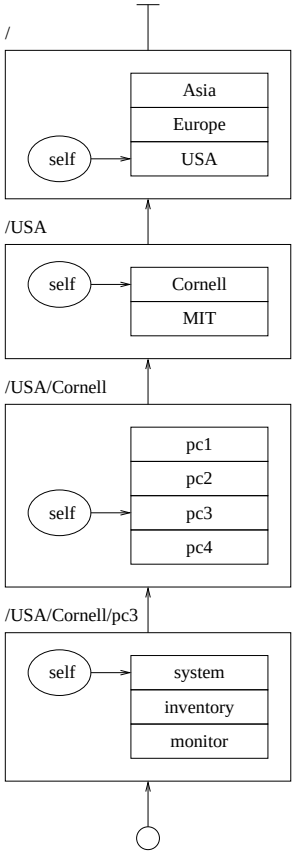


Figure 3: A simplified representation of the data structure maintained by the agent corresponding to /USA/Cornell/pc3.

Conceptually, each zone periodically chooses another sibling zone at random, and the two exchange the MIBs of all their sibling zones. After this exchange, each adopts the most recent MIBs according to the *issued* timestamps. The details of the protocol are somewhat more complicated, particularly since only leaf zones actually correspond to individual machines, while internal zones are *collections* of machines that collectively are responsible for maintaining their state and gossiping this state to peer zones.

As elaborated in Section 4.2, Astrolabe gossips about membership information just as it gossips about MIBs and other data. If a process fails, its MIB will eventually expire and be deleted. If a process joins the system, its MIB will spread through its parent zone by gossip, and as this occurs, aggregates will begin to reflect the content of that MIB.

The remainder of this section describes Astrolabe's protocol in more detail.

Each Astrolabe agent maintains the data structure depicted in Figure 3. For each level in the hierarchy, the agent maintains a record with the list of child zones (and their attributes), and which child zone represents its own zone (*self*). The first (bottom) record contains the local virtual child zones, whose attributes can be updated by writing

them directly (through an RPC interface). In the remaining records, the MIBs pointed to by *self* are calculated by the agent locally using the aggregation functions. The other MIBs are learned through the gossip protocol.

The MIB of any zone is required to contain at least the following attributes:

- *id*: the local zone identifier;
- *rep*: the zone name of the *representative* agent for the zone—the agent that generated the MIB of the zone;
- *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.
- *servers*: a small set of TCP/IP addresses for (representative agents of) this zone, used by applications to interact with the Astrolabe service.[9]
- *nmembers*: the total number of hosts in the zone. The attribute is constructed by taking the sum of the *nmembers* attributes of the child zones. It is used for pacing multicast location mechanisms, as well as in the calculation of averages.

id and *rep* are automatically assigned, that is, their values are not programmable. The *id* attribute is set to the local identifier within the parent zone, while *rep* is set to the full zone name of the local agent. The AFCs can provide values for each of the other attributes. If the AFCs do not compute a value for *issued*, the local wall clock time is used.

The *contacts* attribute is dynamically computed based on an aggregation function, much like the routers in SelectCast (Section 3.2). In effect, each zone *elects* the set of agents that gossip on behalf of that zone. The election can be arbitrary, or based on characteristics like load or longevity of the agents.[10] Note that an agent may be elected to represent more than one zone, and thus run more than one gossip protocol, as described below. The maximum number of zones an agent can represent is bounded by the number of levels in the Astrolabe tree.

Each agent periodically runs the gossip algorithm. First, the agent updates the *issued* attribute in the MIB of its virtual *system* zone, and re-evaluates the AFCs that depend on this attribute. Next, the agent has to decide at which levels (in the zone tree) it will gossip. For this decision, the agent traverses the list of records in Figure 3. An agent gossips on behalf of those zones for which it is a contact, as calculated by the aggregation function for that zone. The rate of gossip at each level can be set individually (using the `&config` certificate described in Section 6.2).

When it is time to gossip within some zone, the agent picks one of the child zones, other than its own, from the list at random. Next the agent looks up the *contacts* attribute for this child zone, and picks a random contact agent from the set of hosts in

[9] Typically these refer to the same agents as *contacts*, but applications can choose other agents by installing the appropriate AFC.

[10] The latter may be advised if there is a high rate of agents joining and leaving the system. Many peer-to-peer systems suffer degraded performance when network partitioning or a high rate of churn occurs. We are currently focused on using Astrolabe in comparatively stable settings, but see this topic as an interesting one deserving further study.

this attribute. (Gossips always are between different child zones, thus if there is only one child zone at a level, no gossip will occur.) The gossiping agent then sends the chosen agent the *id*, *rep*, and *issued* attributes of all the child zones at that level, and does the same thing for the higher levels in the tree up until the root level. The recipient compares the information with the MIBs that it has in its memory, and can determine which of the gossiper's entries are out-of-date, and which of its own entries are. It sends the updates for the first category back to the gossiper, and requests updates for the second category.

There is one important detail when deciding if one MIB is newer than another. Originally, we simply compared the *issued* timestamps with one another, but found that as we scaled up the system we could not rely on clocks being synchronized. This lack of synchronization is the reason for the *rep* attribute: we now only compare the *issued* timestamps of the same agent, identified by *rep*. For each zone, we maintain the most recent MIB for each *representative*, that is, the agent that generated the MIB. until it times out (see Section 4.2). We expose to the Astrolabe clients only the MIB of one of these representatives. Since we cannot compare their *issued* timestamps, we select the one for which we received an update most recently.

We are currently in the process of evaluating various compression schemes for reducing the amount of information that has to be exchanged this way. Nevertheless, gossip within a zone spreads quickly, with dissemination time growing $O(\log n)$, where n is the number of child zones of the zone (see Section 7.1). Gossip is going on continuously, its rate being independent of the rate of updates. (The sole impact of a high update rate is that our compression algorithms will not perform as well, and hence network loads may be somewhat increased.) These properties are important to the scalability of Astrolabe, as we will discuss in Section 7.

4.2 Membership

Up until now we have tacitly assumed that the set of machines is fixed. In a large distributed application, chances are that machines will be joining and leaving at a high rate. The overall frequency of crashes rises linearly with the number of machines. Keeping track of membership in a large distributed system is no easy matter [32]. In Astrolabe, membership is simpler, because each agent only has to know a relatively small subset of the agents (logarithmic in the total size of the membership.) These are its own gossip contacts, and those for its parent and child zones.

There are two aspects to membership: removing members that have failed or are disconnected, and integrating members that have just started up or were previously partitioned away.

The mechanism used for failure detection in Astrolabe is fairly simple. As described above, each MIB has a *rep* attribute that contains the name of the representative agent that generated the MIB, and an *issued* attribute that contains the time at which the agent last updated the MIB. Agents keep track, for each zone and for each representative agent of the zone, the last MIBs from those representative agents. When an agent has not seen an update for a zone from a particular representative agent for that zone for some time T_{fail}, it removes its corresponding MIB. When the last MIB of a zone is removed, the zone itself is removed from the agent's list of zones. This algorithm will always detect and remove failed participants and empty zones within T_{fail} seconds.

T_{fail} should grow logarithmically with membership size (see [32]), which in turn can be determined from the *nmembers* attribute.

The other part of membership is integration. Either because of true network partitions, or because of setting T_{fail} too aggressively, it is possible that the Astrolabe tree splits up into two or more independent pieces. New machines, and machines recovering from crashes, also form independent, degenerate, Astrolabe trees (where each parent has exactly one child). We need a way to glue the pieces together.

Astrolabe relies on IP multicast to set up the initial contact between trees. Each *tree* multicasts a gossip message at a fixed rate ρ which is typically on the order of ten seconds. The collective members of the tree are responsible for this multicasting, and they do so by each tossing a coin every ρ seconds that is weighted by the *nmembers* attribute of the root zone. Thus each member multicasts at an average rate of $\rho/nmembers$.

The current implementation of Astrolabe also occasionally *broadcasts* gossips on the local LAN in order to integrate machines that do not support IP multicast. In addition, Astrolabe agents can be configured with a set of so-called *relatives*, which are addresses of agents that should occasionally be contacted using point-to-point messages. This strategy allows the integration of Astrolabe trees that cannot reach each other by any form of multicast. These mechanisms are described in more detail in [31].

Astrolabe assumes that the administrators responsible for configuring the system will assign zone names in a manner consistent with physical topology of the network. In particular, for good performance, it is desirable that the siblings of a leaf node be reasonably close (in the network). Since zones typically contain 32 to 64 members, the vast majority of messages are exchanged between sibling leaf nodes. Thus, if this rather trivial placement property holds, Astrolabe will not overload long-distance network links.

4.3 Eventual Consistency

Astrolabe takes snapshots of the distributed state, and provides aggregated information to its users. The aggregated information is replicated among all the agents that were involved in taking the snapshot. The set of agents is dynamic. This raises many questions about consistency. For example, when retrieving an aggregate value, does it incorporate all the latest changes to the distributed state? When two users retrieve the same attribute at the same time, do they obtain the same result? Do snapshots reflect a single instance in time? When looking at snapshot 1, and then later at snapshot 2, is it guaranteed that snapshot 2 was taken after snapshot 1?

The answer to all these questions is no. For the sake of scalability, robustness, and rapid dissemination of updates, a weak notion of consistency has been adopted for Astrolabe. Given an aggregate attribute X that depends on some other attribute Y, Astrolabe guarantees with probability 1 that when an update u is made to Y, either u itself, or an update to Y made after u, is eventually reflected in X. We call this *eventual consistency*, because if updates cease, replicated aggregate results will eventually be the same.

Aggregate attributes are updated frequently, but their progress may not be monotonic. This is because the issued time in a MIB is the time when the aggregation was calculated, but ignores the times at which the leaf attributes that are used in the calculation were updated. Thus it is possible that a new aggregation, calculated by a different

agent, is based on some attributes that are older than a previously reported aggregated value.

For example, perhaps agent **a** computes the mean load within some zone as 6.0 by averaging the loads for MIBs in the child zones known at **a**. Now agent **b** computes the mean load as 5.0. The design of Astrolabe is such that these computations could occur concurrently and might be based on temporarily incomparable attribute sets: **a** might have more recent data for child zone **x** and yet **b** might have more recent data for child zone **y**.

This phenomenon could cause computed attributes to jump around in time, an effect that would be confusing. To avoid the problem, Astrolabe can track the (min, max) interval for the *issued* attribute associated with the inputs to any AFC. Here, *min* is the issued time of the earliest updated input attribute, and *max* the issued time of the most recently updated input attribute. An update with such an interval is not accepted unless the minimum issued time of the new MIB is at least as large as the maximum issued time of the current one. This way we can guarantee monotonicity, in that all attributes that were used in the aggregation are strictly newer than those of the old aggregated value.

Rather than seeing an update to an aggregate result after each received gossip, an update will only be generated after a completely new set of input attributes has had a chance to propagate. As we will see later, this propagation may take many rounds of gossip (5 — 35 rounds for practical *Mariner* hierarchies). The user thus sees fewer updates, but the values represent a sensible progression in time. The trade-off can be made by the applications.

4.4 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 solution, 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 *Application Level Gateways* (ALGs) such as used by AOL Instant Messenger (www.aol.com/aim) and Groove (www.groove.net). The solutions can both be used simultaneously.

In the solution based on ALGs, a host wishing to receive a message sends an HTTP POST request to an ALG (see Figure 4). The ALG does not respond until a message is available. A host wishing to send a message sends an HTTP POST request with a message in the body to the ALG. The ALG forwards the message to the appropriate receiver, if available, as a response, to the receiver's POST request. The ALG has limited capacity to buffer messages that arrive between receiver's POST requests. When

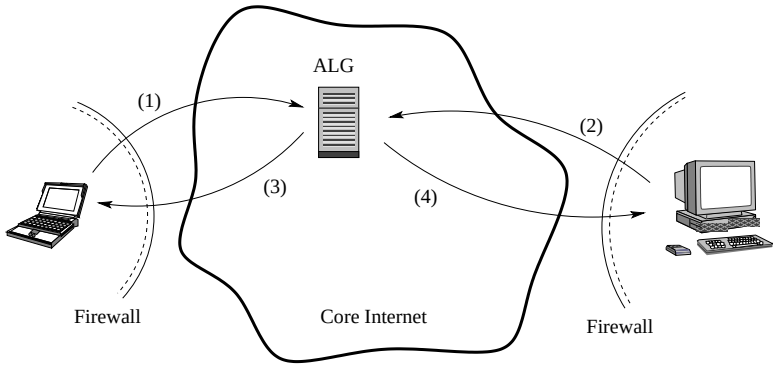


Figure 4: Application Level Gateway. (1) Receiver sends a RECEIVE request using an HTTP POST request; (2) Sender sends the message using a SEND request using an HTTP POST request; (3) ALG forwards the message to the receiver using an HTTP 200 response; (4) ALG sends an empty HTTP 200 response back to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

using persistent HTTP connections, the efficiency is reasonable if the ALG is close to its connected receivers. (It turns out that no special encoding is necessary for the messages.)

The ALGs can either be deployed stand-alone on the core Internet, or as servlets within existing enterprise web servers. For efficiency and scalability reasons, hosts preferably receive through a nearby ALG, which requires that a sufficient number of servers be deployed across the Internet. Note also that machines that are already directly connected to the core Internet do not have to receive messages through an ALG, but can receive them directly.

This solution even works for mobile machines, but for efficiency we have included a redirection mechanism inspired by the one used in cellular phone networks. A mobile machine, when connecting in a new part of the Internet, has to set up a redirection address for a nearby ALG with its "home" ALG. When a sender tries to connect to the (home) ALG of the mobile host, the sender is informed of the ALG closer to the mobile host.

Another solution is to deploy, instead of ALGs, ordinary Astrolabe agents in the core Internet. Astrolabe agents can gossip both over UDP and HTTP. One minor problem is that gossip cannot be initiated from outside a firewall to the inside, but updates will still spread rapidly because once a gossip is initiated from inside a firewall to the outside, the response causes updates to spread in the other direction. A larger problem is that the Astrolabe hierarchy has to be carefully configured so that each zone that is located behind a firewall, but has attributes that should be visible outside the firewall, has at least one sibling zone that is outside the firewall.

To increase scalability and efficiency, we designed a new way of addressing endpoints. In particular, we would like to have machines that can communicate directly through UDP or some other efficient mechanism to do so, instead of going through HTTP.

We define a *realm* to be a set of peers that can communicate with each other using a single communication protocol. For example, the peers within a corporate network

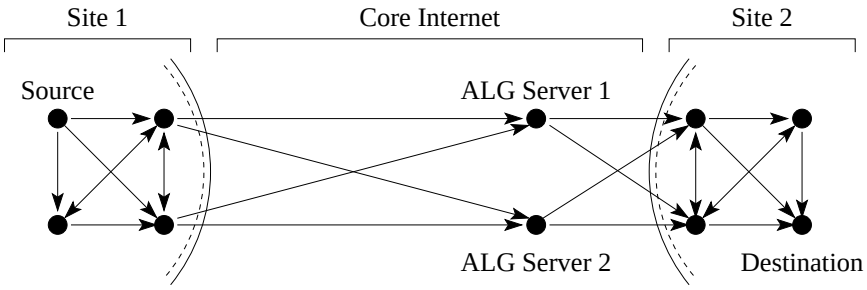


Figure 5: The many ways gossip can travel from a source host in Site 1 to a destination host in Site 2. Each site has four hosts, two of which are representative agents for their associated sites, behind a firewall. The representatives of Site 2 connect to two different ALG servers to receive messages from outside their firewall.

typically form a realm. In fact, there are two realms: a UDP and an HTTP realm. Peers that use the ALG also form a realm. The peers with static IP addresses on the core Internet that are not behind firewalls also form a UDP and an HTTP realm. Thus, a peer may be in multiple realms. In each realm, it has a local address. Two peers may have more than one realm in common.

We assign to each realm a globally unique identifier called *Realm ID*. The ALG realm is called "internet/HTTP". A corporate network realm may be identified by the IP address of its firewall, plus a "UDP" or "HTTP" qualifier (*viz* "a.b.c.d/UDP" resp. "a.b.c.d/HTTP").

We define for each peer its *Address Set* to be a set of triples of the form (Realm ID, Address, Preference), where

- *Realm ID* is the globally unique identifier of the realm the peer is in,
- *Address* is the address within the realm (and is only locally unique),
- *Preference* indicates a preference sorting order on the corresponding communication protocols. Currently, UDP is preferred over HTTP.

A peer may have multiple triples in the same realm with different addresses ("multi-homing"), typically for fault-tolerance reasons, as well as, the same address in distinct realms. For fault-tolerance purposes, a receiver registers with multiple ALGs, and thus has multiple addresses in the Chat realm (see Figure 5).

When peer X wants to send a message to peer Y, X first determines the common realms. Next, it will typically use a weighted preference (based on both X and Y's preferences) to decide which address to send the message to. Astrolabe agents randomize this choice in order to deal with permanent network failures in particular realms.

More detail on wide-area networking in Astrolabe is described in [31].

4.5 Fragmentation

Messages in Astrolabe grow in size approximately as a function of the branching factor used in the hierarchy. The larger the branching factor, the more zones that need to be

gossiped about, and the larger the gossip messages. For this reason, the branching factor should be limited. In practice, we have found that Astrolabe requires about 200 to 300 bytes per (compressed) MIB in a gossip message. The concern is that UDP messages are typically limited to approximately 8 Kbytes, which can therefore just contain about 25 - 40 MIBs in a message. This limit is reached very easily, and therefore Astrolabe has to be able to fragment its gossip messages sent across UDP.

The Astrolabe agents use a simple protocol for fragmentation. Rather than including all updated MIBs in a gossip message, an agent will just include as many as will fit. In order to compensate for lost time, the agent speeds up the rate of gossip accordingly. For example, if on average only half of the updated MIBs fit in a gossip message, the agent will gossip twice as fast. It turns out that a good strategy for choosing which MIBs to include in a message is random selection, as described in Section 7.4.

5 Security

Each Astrolabe zone is a separate unit of management, each with its own set of policy rules. Such policies govern child zone creation, gossip rate, failure detection time-outs, introducing new AFCs, etc. These policies are under the secure control of an administrator. That is, although the administration of Astrolabe as a whole is decentralized, each zone is centrally administered in a fully secure fashion. Each zone may have its own administrator, even if one zone is nested within another.

We believe that an important principle of achieving scale in a hierarchical system is that children should have a way to override policies enforced by their parents. This principle is perhaps unintuitive, since it means that managers of zones with only a few machines have more control (over those machines) than managers of larger encapsulating zones. This creates an interesting tension: managers of large zones control more machines, but have less control over each machine than managers of small zones. Astrolabe is designed to conform to this principle, which guarantees that its own management is decentralized and scalable.

Security in Astrolabe is currently only concerned with integrity and write access control, not confidentiality (secrecy).[11] We wish to prevent adversaries from corrupting information, or introducing non-existent zones.

Individual zones in Astrolabe can each decide whether they want to use public key cryptography, shared key cryptography, and no cryptography, in decreasing order of security and overhead. For simplicity of exposition, in what follows we will present just the public key mechanisms, although experience with the real system suggests that the shared key cryptography option often represents the best trade-off between security and overhead.

5.1 Certificates

Each zone in Astrolabe has a corresponding Certification Authority (CA) that issues certificates. (In practice, a single server process is often responsible for several such CAs.) Each Astrolabe agent has to know and trust the public keys of the CAs of its

[11] The problem of confidentiality is significantly harder, as it would involve replicating the decryption key on a large number of agents, an inherently insecure solution.

ancestor zones. Certificates can also be issued by other principals, and Astrolabe can autonomously decide to trust or not trust such principals.

An Astrolabe certificate is a signed attribute list. It has at least the following two attributes:

- *id*: the issuer of the certificate;
- *issued*: the time at which the certificate was issued. Astrolabe uses this attribute to distinguish between old and new versions of a certificate.

Optionally, a certificate can have an attribute *expires*, which specifies the time at which the certificate will be no longer valid. (For this mechanism to work well, all agent's clocks should be approximately synchronized with real time.)

Each zone in Astrolabe has two public/private key pairs associated with it: the CA keys (the private key of which is kept only by the corresponding CA), and the zone keys. These are used to create four kinds of certificates:

1. a *zone certificate* binds the id of a zone to its public zone key. It is signed using the private CA key of its parent zone. Note that the root zone cannot have (and will turn out not to need) a zone certificate. Zone certificates contain two attributes in addition to the *id* and *issued*: *name*, which is the zone name, and *pubkey*, which is the public zone key. As only the parent CA has the private CA key to sign such certificates, adversaries cannot introduce arbitrary child zones.

2. a *MIB certificate* is a MIB, signed by the private zone key of the corresponding zone. These are gossiped along with their corresponding zone certificates between hosts to propagate updates. The signature prevents the introduction of "false gossip" about the zone.

3. an *aggregation function certificate* (AFC) contains the code and other information about an aggregation function. An agent will only install those AFCs that are issued directly by one of its ancestor zones (as specified by the *id* attribute), or by one of their clients (see next bullet).

4. a *client certificate* is used to authenticate clients to Astrolabe agents. A *client* is defined to be a user of the Astrolabe service. The agents do not maintain a client database, but if the certificate is signed correctly by a CA key of one of the ancestor zones of the agent (specified by the *id* attribute), the connection is accepted. Client certificates can also specify certain restrictions on interactions between client and agent, such as which attributes the client may inspect. As such, client certificates are not unlike capabilities. A client certificate may contain a public key. The corresponding private key is used to sign AFCs created by the client.

In order to function correctly, each agent needs to be configured with its zone name (or "path"), and the public CA keys of each zone it is in. (It is important that agents do not have access to private CA keys.) As we saw when discussing the network protocols (Section 4.1), some agents in each zone (except the root zone) need the private zone keys and corresponding zone certificates of those zones for which they are allowed to post updates. In particular, each host needs the private key of its leaf zone.

The root zone is an exception: since it does not have sibling zones, it never gossips updates and therefore only needs CA keys, which it uses when signing certificates on behalf of top-level child zones. There is the common issue of the trade-off between fault-tolerance and security: the more hosts that have the private key for a zone, the more fault-tolerant the system, but also the more likely the key will get compromised. (Potentially this problem can be fixed using a threshold scheme; we have not yet investigated this option.)

Note that *zone certificates are not chained*. Although each is signed by the private CA key of the parent zone, the agents are configured with the public CA key of each ancestor zone, so that no recursive checking is necessary. Chaining would imply transitive trust, which does not scale and violates Astrolabe's governing principle of zones being more powerful than their ancestor zones.

The CA of a zone, and only that CA, can create new child zones by generating a new zone certificate and a corresponding private key, and thus preventing impersonation attacks. The private key is used to sign MIB certificates (updates of the MIB of the new zone), which will only be accepted if the zone certificate's signature checks using the public CA key, and the MIB certificate's signature checks using the public key in the zone certificate. Similarly, a zone CA has control over what AFC code is installed within that zone. This is described in more detail in the next section, which treats the question of which clients are allowed which types of access.

5.2 Client Access

As mentioned above, Astrolabe agents do not maintain information about clients. The CA of a zone may choose to keep such information, but it is not accessible to the Astrolabe agents themselves, as we do not believe this solution would scale (eventually there would be too many clients to track). Instead, Astrolabe provides a mechanism that has features of both capabilities and access control lists.

A client that wants to use Astrolabe has to obtain a client certificate from a CA. The client will only be able to access Astrolabe within the zone of the CA. When it contacts one of the agents in the CA, the agent will use the *id* attribute in the certificate to determine the zone, and if it is in fact in this zone, will have the public key of the zone. It uses the public key to check that the certificate is signed correctly before allowing any access to the agent. Thus, in some sense, the client certificate is a capability for the zone of the CA that signed it.

Additional fine-grained control over clients is exercised in (at least) two ways:

1. the client certificate can specify certain constraints on the holder's access. For example, it may specify which attributes the client is allowed to read or update.

2. each zone can specify (and update at run-time) certain constraints on access by holders of client certificates signed by any of its ancestor zones.

That is, the client is constrained by the union of the security restrictions specified in its certificate, and the restrictions specified by all zones on the path from the leaf zone of the agent it is connected to, to the zone that signed its certificate.

In general, client certificates issued by larger zones have different security rules than client certificates issued by its child zones. The former will have more power in the larger zones, while the latter will have more power in the smaller zones. For this reason, users may require a set of client certificates.

Security will not scale if policies cannot be changed on the fly. The zone restrictions can be changed at run-time by installing an "&config" AFC for the zone (see Section 6.2). But client certificates share a disadvantage with capabilities and certificates in general: they cannot be easily revoked until they expire. We are considering two possible solutions. One is to have client certificates with short expiration times. The other is to use Certification Revocation Lists (CRLs). Both have scaling problems.

The mechanisms discussed so far take care of authentication and authorization issues. They prevent impersonation and spoofing attacks, but they do not prevent authorized agents and clients from lying about their attributes. Such lies could have significant impact on calculated aggregates. Take, for example, the simple program:

```
SELECT MIN(load) AS load;
```

This function exports the minimum of the *load* attributes of the children of some zone to the zone's attribute by the same name. The intention is that the root zone's *load* attribute will contain the global minimum load. Both clients and agents, when holding valid certificates, can do substantial damage to such aggregated data. Clients can lie about load by installing an unrealistically low or high load in a leaf zone's MIB, while an agent that holds a private key of a zone can gossip a value different from the computed minimum of its child zones' loads. To make an application robust against such attacks, we recommend removing outliers as much as possible. Unfortunately, standard SQL does not provide support for removing outliers. We are planning to extend Astrolabe's SQL engine with such support. More problematically, when applied recursively, the result of aggregation after removing outliers may have unclear semantics.

6 Aggregation Functions

The aggregation function of a zone reads the list of MIBs belonging to its child zones, and produces a MIB for its zone as output.[12] The code for an AFC is provided in attributes of its child zone MIBs whose name starts with the character '&'. AFCs themselves are attribute lists. An AFC has at least the following attributes in addition to the standard certificate attributes:

- *lang*: specifies the language in which the program is coded.
- *code*: contains the SQL code itself.
- *deps*: contains the input attributes on which the output of the function depends. Astrolabe reduces overhead by only re-evaluating those AFCs for which the input has changed.
- *category*: specifies the attribute in which the AFC is to be installed. As we will see later, this explicit specification prevents rogue users from misusing correctly signed AFCs.

[12] Astrolabe is not secured against faulty aggregation functions. For example, suppose that an aggregation function is expected to report the highest load in some zone, but sometimes incorrectly reports zero. The aggregation value seen by users would seem to bounce around, sometimes reflecting the correct value, and sometimes reflecting this erroneous input. It may be possible to use a form of voting to overcome such failures, but at present, this is simply a known security deficiency of the initial system.

Function	Description
MIN(attribute)	Find the minimum attribute
MAX(attribute)	Find the maximum attribute
SUM(attribute)	Sum the attributes
AVG(attribute [, weight])	Calculate the weighted average
OR(attribute)	Bitwise OR of a bit map
AND(attribute)	Bitwise AND of a bit map
FIRST(n, attribute)	Return a set with the first n attributes
RANDOM(n, attribute [, weight])	Return a set with n randomly selected attributes

Table 2: Extended SQL aggregation functions. The optional weight corrects for imbalance in the hierarchy, and is usually set to $nmembers$.

As described later, an AFC may also have the following attributes:

- *copy*: a Boolean that specifies if the AFC can be "adopted." Adoption controls propagation of AFCs into sibling zones.
- *level*: an AFC is either "weak" or "strong." Strong AFCs cannot be replaced by ancestor zones, but weak AFCs can if they have more recent *issued* attributes.
- *client*: in case of an AFC issued by a client, this attribute contains the entire client certificate of the client. The client certificate may be checked with the CA key of the issuer, while the AFC may be checked using the public key in the client certificate.

In this section, we will discuss how AFCs are programmed, and the propagation rules of AFCs.

6.1 Programming

We have extended SQL with a set of new aggregation functions that we found helpful in the applications that we have pursued. A complete table of the current set of functions appears in Table 2.

The following aggregation query is installed by Astrolabe per default (but may be changed at run-time):

```
SELECT
   SUM(nmembers) AS nmembers,
   MAX(depth) + 1 AS depth,
   FIRST(3, contacts) AS contacts,
   FIRST(3, servers) AS servers
```

Here, *nmembers* is the total number of hosts in the zone, *depth* is the nesting level of the zones, *contacts* are the first three "representative" gossip addresses in the zone, and *servers* the first three TCP/IP addresses, used by clients to interact with this zone. *nmembers* is often used to weigh the values when calculating the average of a value, so that the actual average is calculated (rather than the average of averages).

Astrolabe's SQL engine also has a simple exception handling mechanism that produces descriptive error messages in a standard output attribute.

6.2 Propagation

An application introduces a new AFC by writing an attribute of a virtual child zone at some Astrolabe agent. This Astrolabe agent will now automatically start evaluating this AFC. The Astrolabe architecture includes two mechanisms whereby an AFC can propagate through the system. First, the AFC can include another AFC (usually, a copy of itself) as part of its output. When this propagation mechanism is employed, the output AFC will be copied into the appropriate parent MIB and hence will be evaluated when the MIB is computed for the parent. This approach causes the aggregation process to recursively repeat itself until the root MIB is reached.

Because of the gossip protocol, these AFCs will automatically propagate to the other agents just like normal attributes do. However, since the other agents only share ancestor zones, we need a second mechanism to propagate these AFCs down into the leaf MIBs. This second mechanism, called *adoption*, works as follows. Each Astrolabe agent scans its ancestor zones for new AFC attributes. If it detects a new one, the agent will automatically copy the AFC into its virtual "system" MIB (described above in Section 2). This way, an introduced AFC will propagate to all agents within the entire Astrolabe tree.

Jointly, these two AFC propagation mechanisms permit an application to introduce a new AFC dynamically into the entire system. The AFC will rapidly spread to the appropriate nodes, and, typically within tens of seconds, the new MIB attributes will have been computed. For purposes of garbage collection, the creator of an AFC can specify an expiration time; unless the time is periodically advanced (by introducing an AFC with a new issued and expiration time), the aggregation function will eventually expire and the computed attributes will then vanish from the system.

It is clear that a secure mechanism is necessary to prevent clients from spreading code around that can change attributes arbitrarily. The first security rule is that certificates are only considered for propagation if

- the AFC is correctly signed by an ancestor zone or a client of one, preventing "outsiders" from installing AFCs. In case of an AFC signed by a client, the client has to be granted such propagation in its client certificate.
- the AFC has not expired.
- the *category* attribute of the AFC is the same as the name of the attribute in which it is installed. This prevents a rogue client from trying to use an AFC for an attribute it was not intended for.

We call such AFCs *valid*.

It is possible, and in fact common, for an AFC to be installed only in a subtree of Astrolabe. Doing so requires only that the corresponding zone sign the certificate, which signature is easier to obtain than the CA-signature of the root zone.

The adoption mechanism allows certificates to propagate "down" the Astrolabe tree. Each agent continuously inspects its zones, scanning them for new certificates, and installs one in its own MIB if the AFC satisfies the following conditions:

- if another one of the same *category* is already installed, the new one is *preferable*.
- its *copy* attribute is not set to "no" (such AFCs have to be installed in each zone where they are to be used).

"Preferable" is a partial order relation between certificates in the same category: $x > y$ means that certificate x is preferable to certificate y. $x > y$ iff

- x is valid, and y is not, or
- x and y are both valid, the level of x is strong, and $x.id$ is a child zone of $y.id$, or
- x and y are both valid, x is not stronger than y, and $x.issued > y.issued$.

Note that these rules of propagation allow the CAs to exercise significant control over where certificates are installed.

This controlled propagation can be exploited to do run-time configuration. Astrolabe configures itself this way. The `&config` certificate contains attributes that control the run-time behavior of Astrolabe agents. Using this certificate, the gossip rate and security policies can be controlled at every zone in the Astrolabe tree.

7 Performance

In this section, we present simulation results that support our scalability claims, and experimental measurements to verify the accuracy of our simulations. Finally, we describe various implementation details with the gossip protocol in the current Internet.

7.1 Latency

Since sibling zones exchange gossip messages, the zone hierarchy has to be based on the network topology so that load on network links and routers remains within reason. As a rule of thumb, if a collection of machines can be divided into two groups separated by a single router, these groups should be in disjoint zones. On the other hand, as we will see, the smaller the branching factor of the zone tree, the slower the dissemination. So some care should be taken not to make the zones too small in terms of number of child zones. With present day network packet sizes, CPU speeds, and memory sizes, the number of child zones should be approximately between 5 to 50. (In the near future, we hope that through improving our compression techniques we can support higher branching factors.)

As the membership grows, it may be necessary to create more zones. Initially, new machines can be added simply to leaf zones, but at some point it becomes necessary to divide the leaf zones into smaller zones. Note that this re-configuration only involves the machines in that leaf zone. Other parts of Astrolabe do not need to know about the re-configuration, and this is extremely important for scaling and deploying Astrolabe.

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 [11]. The question is, is this slow growth in latency also true in Astrolabe, which uses a hierarchical protocol? The answer appears to be yes, albeit that the latency grows somewhat faster. To demonstrate this, we conducted a number of simulated experiments.[13]

[13] Although Astrolabe is currently deployed on approximately 60 machines, this is not a sufficiently large system to evaluate its scalability.

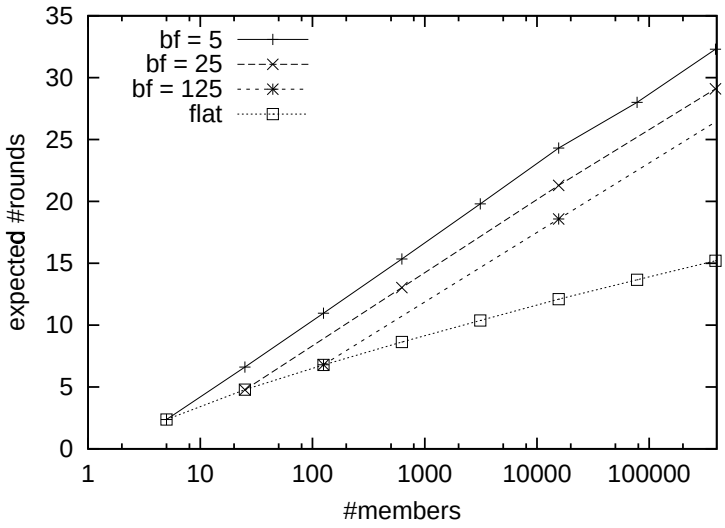


Figure 6: The average number of rounds necessary to infect all participants, using different branching factors. In all these measurements, the number of representatives is 1, and there are no failures.

In the experiments, we varied the branching factor of the tree, and the number of representatives in a zone, the probability of message loss, and the ratio of failed hosts. In all experiments, we used a balanced tree with a fixed branching factor. We simulated up to 5^8 (390,625) members. In the simulation, gossip occurred in rounds, with all members gossiping at the same time.[14] We assumed that successful gossip exchanges complete within a round. (Typically, Astrolabe agents are configured to gossip once every two to five 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 all experiments, the variance observed was low.

In the first experiment, we varied the branching factor. We used branching factors 5, 25, and 125 (that is, 5^1, 5^2, and 5^3). In this experiment there was just one representative per zone, and there were no failures. We measured the average number of rounds necessary to disseminate information from one node to all other nodes.[15] We show the results in Figure 6 (on a log scale), and compare these with flat (non-hierarchical) gossip. Flat gossip would be impractical in a real system, as the required memory grows linearly, and network load quadratically with the membership [32], 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

[14] In more detailed discrete event simulations, in which the members did not gossip in rounds but in a more randomized fashion, we found that gossip propagates faster, and thus that the round-based gossip provides useful "worst-case" results.

[15] We used a non-representative node as the source of information. Representatives have an advantage, and their information disseminates significantly faster.

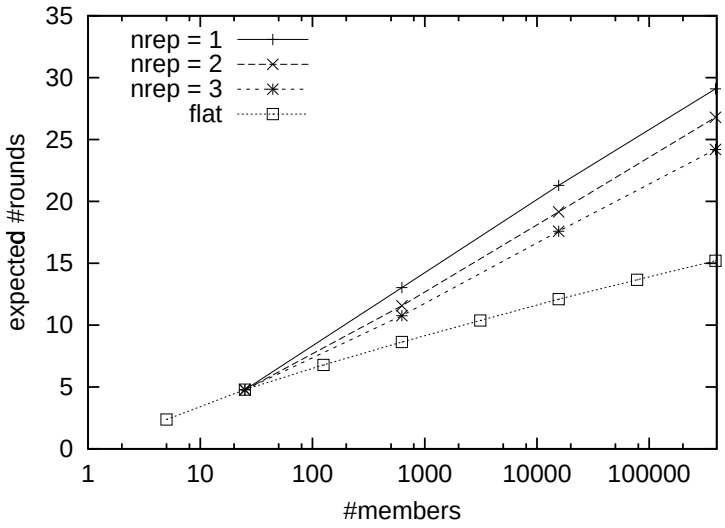


Figure 7: The average number of rounds necessary to infect all participants, using a different number of representatives. In all these measurements, the branching factor is 25, and there are no failures.

gossip also scales well, but is significantly slower than flat gossip. Latency improves when the branching factor is increased, but doing so 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.

With only one representative per zone, Astrolabe is highly sensitive to host crashes. The protocols still work, as faulty representatives are detected and replaced automatically, but this detection and replacement takes time and leads to significant delays in dissemination. Astrolabe is preferably configured with more than one representative in each non-leaf zone. In Figure 7, we show the average number of rounds necessary to infect all participants in an Astrolabe tree with branching factor 25. In the experiments that produced these numbers, we varied the number of representatives from one to three. Besides increasing fault-tolerance, more representatives also decrease the time to disseminate new information. But three times as many representatives also leads to three times as much load on the routers, so the advantages come at some cost.

In the next experiment, we determined the influence of message loss on the dissemination latency. In this experiment we used, again, a branching factor of 25, but this time we fixed the number of representatives at three. Gossip exchanges were allowed to fail with a certain independent probability $loss$, which we varied from 0 to $.15$ (15%). As can be seen from the results in Figure 8, loss does lead to slower dissemination, but, as in flat gossip [32], the amount of delay is surprisingly low. In a practical setting, we would probably observe dependent message loss due to faulty or

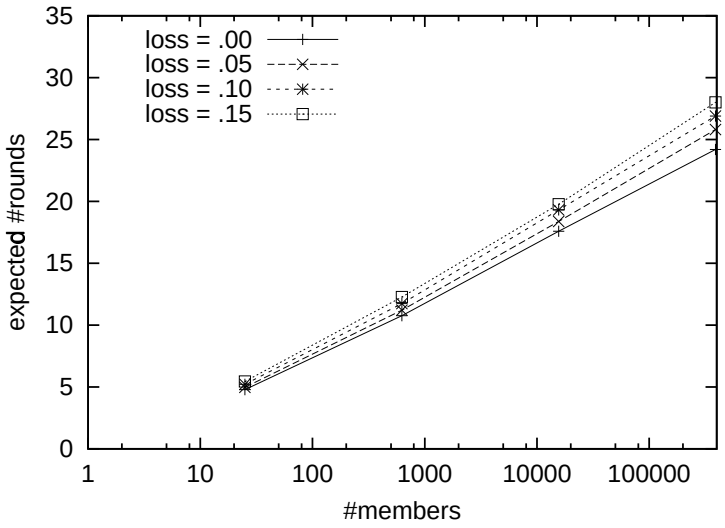


Figure 8: The average number of rounds necessary to infect all participants, using different message loss probabilities. In all these measurements, the branching factor is 25, and the number of representatives is three.

overloaded routers and/or network links, with more devastating effects. Nevertheless, because of the randomized nature of the gossip protocol, updates can often propagate *around* faulty components in the system. An example of such dependent message loss is the presence of crashed hosts.

In this final simulated experiment, we stopped certain agents from gossiping in order to investigate the effect of host crashes on Astrolabe. Again we used a branching factor of 25, and three representatives. Message loss did not occur this time, but each host was down with a probability that we varied from 0 to $.08$ (8%). (With large numbers of members, doing so made the probability that all representatives for some zone are down rather high. Astrolabe's aggregation functions will automatically assign new representatives in such cases.) As with message loss, the effect of crashed hosts on latency is quite low (see Figure 9).

If we configure Astrolabe so that agents gossip once every two to five seconds, as we normally do, we can see that updates propagate with latencies on the order of tens of seconds, rather than hours, as can be the case with DNS if TTL settings are large.

7.2 Load

We were also interested in the load on Astrolabe agents. We consider two kinds of load: the number of received messages per round, and the number of signature checks that an agent has to perform per round. The average message reception load is easily determined: on average, each agent receives one message per round for each zone it represents. Thus, if there are k levels, an agent that represents zones on each level with have the worst average load of k messages per second. Obviously, this load grows $O(\log n)$.

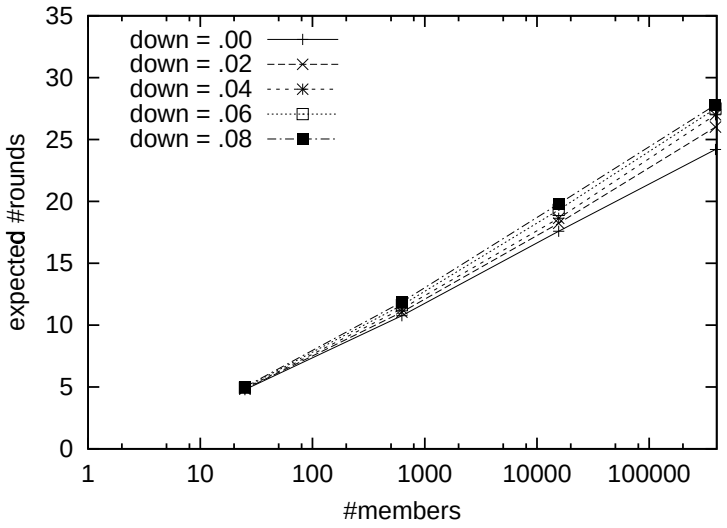


Figure 9: The average number of rounds necessary to infect all participants, using different probabilities of a host being down. In all these measurements, the branching factor is 25, and the number of representatives is three.

Due to randomization, it is possible that an agent receives more than one message per round and per level. The variance of message load on an agent is expected to grow $O(\log n)$: if a process is involved in k epidemics with iid distributions, where each epidemic involves the same number of participants (the branching factor of the Astrolabe tree), then the variance is simply k times the variance of the load of each individual epidemic.

In order to evaluate the load on Astrolabe agents experimentally, we used three representatives per zone, but eliminated host and message omission failures (as these only serve to reduce load). We ran a simulation for 180 rounds (fifteen minutes in case each round is five seconds), and measured the maximum number of message received per round across all agents. The results are shown in Figure 10. This figure perhaps reflects best the trade-off between choosing small and large branching factors mentioned earlier.

If we simply checked all signatures in all messages that arrived, the overhead of checking could become enormous. As the number of MIBs in a message grows as $O(\log n)$, the computational load would grow as $O(\log^2 n)$. The larger the branching factor, the higher this load, as larger branching factors result in more MIBs per message, and can easily run in the thousands of signature checks per round even for moderately sized populations.

Rather than checking all signatures each time a message arrives, the agent buffers all arriving MIBs without processing them until the start of a new round of gossip. (Actually, only those MIBs that are new with respect to what the agent already knows are buffered.) For each zone, the agent first checks the signature on the most recent MIB, then on the second most recent MIB, etc., until it finds a correct signature (usually

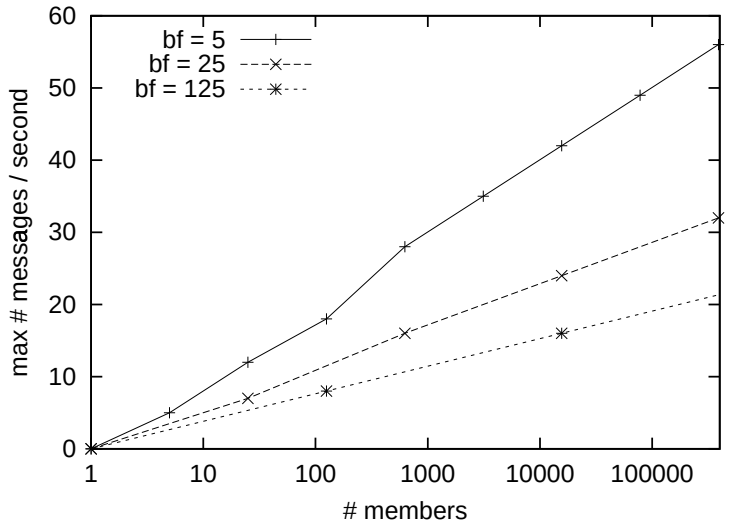


Figure 10: The maximum load in terms of number of messages per round as a function of number of participants and branching factor.

Experiment	# agents	agents/host	Description
1	48	1	(8 8 8 8 8 8)
2	48	1	((8 8) (8 8) (8 8))
3	63	1	(8 8 8 7 8 8 8 8)
4	63	1	((8 8) (8 7 8) (8 8 8))
5	96	2	(16 16 16 16 16 16)
6	96	2	((16 16) (16 16) (16 16))
7	126	2	(16 16 16 14 16 16 16 16)
8	126	2	((16 16) (16 14 16) (16 16 16))

Table 3: Hierarchies used in each experiment.

the first time). The other versions of the same MIB are then ignored. Thus we have artificially limited the computational load to $O(\log n)$ without affecting the speed of gossip dissemination. In fact, the maximum number of checks per round is at most $k \times (bf - 1)$, where k is the number of levels and bf the branching factor. Moreover, the computational load is approximately the same on all agents, that is, not worse on those agents that represent many zones.

Another approach to limit computational overhead is based on a Byzantine voting approach. It removes almost all need for signature checking [17].

7.3 Validating the Simulations

In order to verify the accuracy of our simulator, we performed experiments with up to 126 Astrolabe agents on 63 hosts. For these experiments, we created the topology of

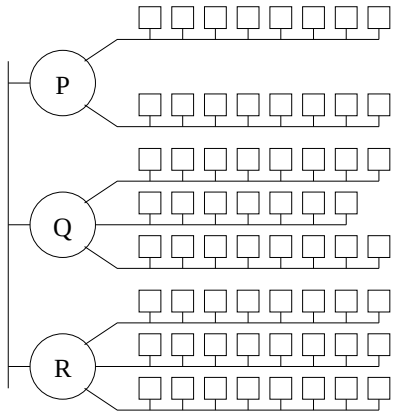


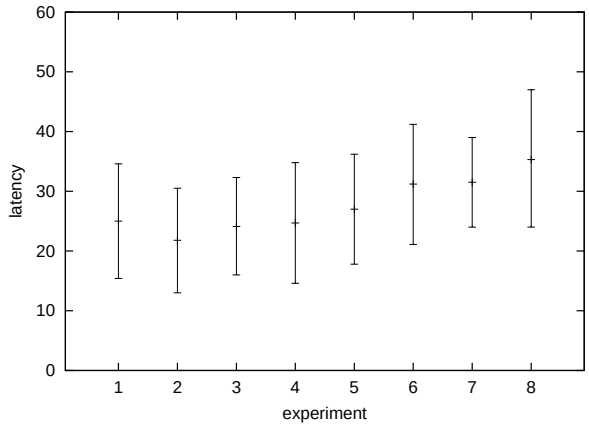
Figure 11: The experimental topology, consisting of 63 hosts, eight 10 Mbps LANs, and three routers connected by a 100 Mbps backbone link.

Figure 11 in the Emulab network testbed of the University of Utah. The set-up consists of six LANs, each consisting of eight Compaq DNARD Sharks (233 MHz StrongARM processor, 32 MB RAM, 10 Mbps Ethernet interface, running NetBSD) connected to an Asante 8+2 10/100 Ethernet switch. (As indicated in the figure, one of the Sharks was broken.) The set-up also includes three routers, consisting of 600 MHz Intel Pentium III "Coppermine" processors, each with 256 MB RAM and 4 10/100 Mbps Ethernet interfaces, and running FreeBSD 4.0. All the routers' Ethernet interfaces, as well as the Asante switches, are connected to a Cisco 6509 switch, and then configured as depicted in Figure 11.

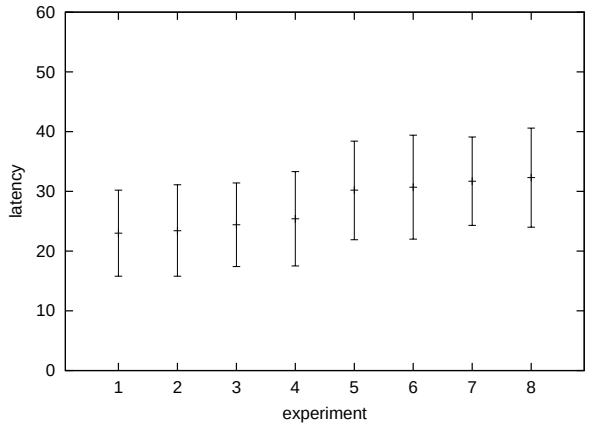
On this hardware, we created eight different Astrolabe hierarchies (see Table 3) with the obvious mappings to the hardware topology. For example, in Experiment 8 we created a three-level hierarchy. The top-level zone consisted of a child zone per router. Each child zone in turn had a grandchild zone per LAN. In Experiments 1, 3, 5, and 7 we did not reflect the presence of the routers in the Astrolabe hierarchy, but simply created a child zone per LAN. Experiments 1, 2, 5, and 6 did not use the 4th and 7th LAN. In the last four experiments, we ran two agents per host. Here, the agents were configured to gossip once every five seconds over UDP, and the (maximum) number of representatives per zone was configured to be three.

In each experiment, we measured how long it took for a value, updated at one agent, to disseminate to all agents. To accomplish this dissemination, we installed a simple aggregation function 'SELECT SUM(test) AS test'. All agents monitored the attribute 'test' in the top-level zone, and noted the time at which this attribute was updated. We updated the 'test' attribute in the virtual system zone of the "right-most" agent (the one in the bottom-right of the topology), since this agent has the longest gossiping distance to the other agents. Each experiment was executed at least 100 times. In Figure 12(a) we report the measured average, minimum, maximum, and 95% confidence intervals for each experiment.

Figure 12(b) reports the simulation results of the same experiments. In the simulations, we assumed no message loss, as we had no easy way of modeling the actual behavior of loss on the Emulab testbed. The simulations, therefore, show more regular



(a)



(b)

Figure 12: Results of (a) the experimental measurements and (b) the corresponding simulations. The x-axes indicate the experiment number in Table 3. The error bars indicate the averages and the 95% confidence intervals.

behavior than the experiments. They give sometimes slightly pessimistic results, as in the simulations the agents gossip in synchronous rounds (non-synchronized gossip disseminates somewhat faster). Nevertheless, the results of the simulations correspond well to the results of the experiments, giving us confidence that the simulations predict the actual behavior in large networks well. The experimental results show the presence of some outliers in the latencies (although never more than two rounds of gossip), which could be explained by occasional message loss.

We plan to present comprehensive experimental data for Astrolabe in some other forum. At present, our experiments are continuing with an emphasis on understanding

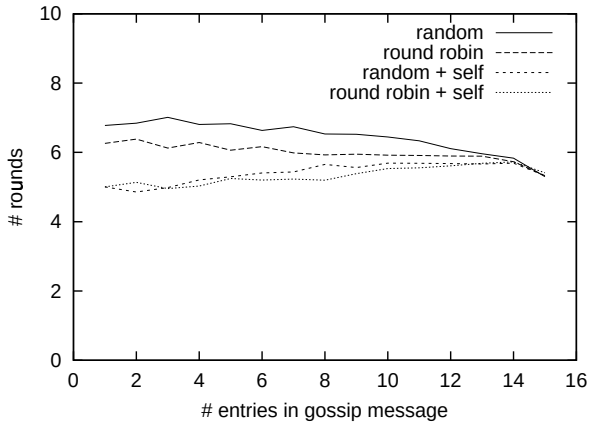


Figure 13: Cost of fragmentation.

how the system behaves on larger configurations, under stress, or when the rate of updates is much higher.

7.4 Fragmentation

As described in Section 4.5, the underlying network may enforce a *maximum transmission unit* size. In order to deal with this limitation, we reduce the number of MIBs in a message, and speed up the gossip rate accordingly. In order to decide which is a good method for choosing MIBs to include in messages, we simulated two basic strategies, with a variant for each. The two basic strategies are round robin and random. In the first, the agent fills up gossip messages in a round robin manner, while in the second the agent picks random (but different) MIBs. The variant strategy for each is that the agent's own MIB is forced to be included as exactly one of the entries.

In Figure 13 we show the results when the number of hosts here is 16, and the hierarchy is flat. Round robin performs better than random, and including the local MIB of the sending agent is a good idea. The current implementation of the Astrolabe agent actually uses random filling, but also includes the local MIB. The reason for not using round robin is that, in a hierarchical epidemic, round robin can cause the MIBs of entire zones to be excluded.

8 Related Work

8.1 Directory Services

Much work has been done in the area of scalable mapping of names of objects (often machines) onto meta-information of the objects. The best known examples are the DNS [18] and X.500 [21], and LDAP (RFC 1777) standards. Similar to DNS, and particularly influential to the design of Astrolabe is the Clearinghouse directory service [11]. Clearinghouse was an early alternative to DNS, used internally for the Xerox Corporate Internet. Like DNS, it maps hierarchical names onto meta-information. Unlike DNS,

it does not centralize the authority of parts of the names space to any particular servers. Instead, the top two levels of the name space are fully replicated and kept eventually consistent using a gossip algorithm much like Astrolabe's. Unlike Astrolabe, Clearinghouse does not apply aggregation functions or hierarchical gossiping, and thus its scalability is inherently limited. The amount of storage grows $O(n)$, while the total bandwidth taken up by gossip grows $O(n^2)$ (because the size of gossip messages grows linearly with the number of members). Clearinghouse has never been scaled to more than a few hundred servers. (Neither has Astrolabe at this time, but analysis and simulation indicate that Astrolabe could potentially scale to millions.)

More recent work applies variants of the Clearinghouse protocol to databases (*e.g.*, Bayou's anti-entropy protocol [20] and Golding's timestamped anti-entropy protocol [12]). These systems suffer from the same scalability problems, limiting scale to perhaps a few thousands of participants.

Also influential to the design of Astrolabe is Butler Lampson's paper on the design of a global name service [16], based on experience with Grapevine [7] and Clearinghouse. This paper enumerates the requirements of a name service, which include large size, high availability, fault isolation, and tolerance of mistrust, all of which we believe Astrolabe supports. The paper's design does not include aggregation, but otherwise shares many of the advantages of Astrolabe over DNS.

In the Intentional Naming System [1], names of objects are based on properties rather than location. A self-organizing resolver maps names onto locations, and can route messages to objects, even in an environment with mobile hosts. A recent extension to INS, Twineallows for partial matching of properties based on a peer-to-peer protocol[4].

The Globe system [33] is an example of a very scalable directory service that maps arbitrary object names onto object identifiers, and then onto location. Globe also supports locating objects that move around.

8.2 Network Monitoring

Network monitors collect runtime information from various sources in the network. A standard for such retrieval is the SNMP protocol (RFC 1156, 1157). A large variety of commercial and academic network monitors exist. Many of these systems provide tools for collecting monitoring data in a centralized place, and visualizing the data. However, these systems provide little or no support for dissemination to a large number of interested parties, aggregation of monitored data, or security. The scale of these systems is often intended for, and limited to, clusters of up to a few hundred machines. Of these, Argent's Guardian (www.argent.com) is closest to Astrolabe, in that it uses regional agents to collect information in a hierarchical fashion (although it does not install agents on the monitored systems themselves). Monitoring information is reported to a single site, or at most a few sites, and Guardian does not support aggregation.

Note that Astrolabe does not actually retrieve monitoring information; it just has provides the ability to disseminate and aggregate such data. Neither does Astrolabe provide visualization. The monitoring agents and visualization tools provided by these products, together with Astrolabe, could form an interesting marriage.

8.3 Event Notification

Event Notification or Publish/Subscribe services allow applications to subscribe to certain classes of events of interest, and systems to post events. Examples are ISIS [6], TIB/RendezVous™[19], Gryphon [2], and Siena [10]. Note that although Astrolabe processes events, it does not route them to subscribers, but processes them to determine some aggregate state "snapshot" of a distributed system. However, a Publish/Subscribe system has been built on top of Astrolabe and is described in Section 3.3.

XMLBlaster (www.xmlblaster.com) and a system built at MIT [25] encode events in XML format, and allow routing based on queries over the XML content in the messages. The latter system's protocol for this routing is similar to SelectCast (Section 3.3), although our protocol can do routing not only based on the content of the messages, but also on the aggregate state of the environment in which the messages are routed.

Although many of these systems support both scale and security, the events are either very low-level, or generated from aggregating low-level event histories. In order to monitor the collective state of a distributed system, a process would have to subscribe to many event sources, and do the entire event processing internally. This strategy does not scale. Event notification and Publish/Subscribe services are intended to disseminate information from few sources to many subscribers, rather than integrating information from many sources.

8.4 Sensor Networks

In a sensor network, a large number of sensors monitor a distributed system. The problem is detecting certain conditions and querying the state of the network. There are many projects in this area. A couple that relate closely to our work are Cougar [9] and an unnamed project described in [15].

In Cougar, a user can express a long-running SQL query, as in Astrolabe. A centralized server breaks the query into components that it spreads among the various devices. The devices then report back to the server, which combines the results and reports the result to the user. Although much of the processing is in the network, Cougar's centralized server may prevent adequate scaling and robustness.

In [15], a sensor network for a wireless system is described. As in Active Networks [28], code is dynamically installed in the network that routes and filters events. Using a routing mechanism called "Directed Diffusion," events are routed towards areas of interest. The event matching is specified using a low-level language that supports binary comparison operators on attributes. The system also allows application-specific filters to be installed in the network that can aggregate events.

We have studied various epidemic protocols, including the ones that are used by Astrolabe, for use in power-constraint wireless sensor networks [30]. What is clear from this study is that the hierarchical epidemics used in Astrolabe cannot be used as is in such networks, as the protocol does send messages across large distances, albeit occasionally.

8.5 Cluster Management

As made clear above, directory services and network monitoring tools do not support dynamic distributed configuration. There are a number of cluster management tools

available that take care of configuration, for example, Wolfpack, Beowulf, NUCM, and the "Distributed Configuration Platform" of EKS. Unlike Astrolabe, they do not scale beyond a few dozen machines, but they do provide various problem-specific functionality for the management of distributed system. Astrolabe is not such a shrink-wrapped application, but could be incorporated into a cluster management system to support scalability.

8.6 Peer-to-Peer Routing

A *peer-to-peer routing protocol* (P2PRP) routes messages to locations, each determined by a location-independent key included in the corresponding message. Such a protocol may be used to build a so-called *Distributed Hash Table*, simply by having each location provide a way to map the key to a value. Well-known examples of P2PRPs include Chord [27], Pastry [23], and Tapestry [34] These protocols have been used to implement distributed file systems and application-level multicast protocols.

As in Astrolabe, each location runs an agent. When receiving a message, the agent inspects the key and, unless it is responsible for the key itself, forwards the message to an agent that knows more about the key. The number of hops grows $O(logn)$ for each of the P2PRPs named above, as does the size of the routing tables maintained by the agents.

The functionalities of a P2PRP and Astrolabe are orthogonal, yet, a comparison proves interesting. Although sometimes called *location protocols*, P2PRPs can't *find* an object—they can only place the object in a location where they will be able to retrieve the object later. Astrolabe, on the other hand, is able to find object based on attributes of the object. However, Astrolabe is quite limited in how many objects it can find in a short period of time, while P2PRPs can potentially handle high loads.

In their implementations, there are some interesting differences between a P2P routing protocol and Astrolabe. While Astrolabe exploits locality heavily in its implementation, P2PRPs only try to optimize the length of hops chosen when routing (so-called *proximity routing*). At each hop in the routing path, the number of eligible next hops is small, and thus the effectiveness of this approach may be variable. Worse, as agents join and leave the system (so-called *churn*), objects have to be moved around potentially across very large distances). The P2PRP protocols also employ ping-based failure detection across these large distances.

Additionally, whereas P2PRPs treat agents as identical, Astrolabe is able to select agents as representatives or routers based on various attributes such as load, operating system type, or longevity. That is, while P2PRPs assume the agents are homogeneous, Astrolabe tries to exploit the heterogeneity offered by the agents.

8.7 Epidemic Protocols

The earliest epidemic protocol that we are aware of is Gene Spafford's Usenet protocol, the predecessor of today's NNTP (RFC 977, February 1986), which gossiped news over UUCP connections starting in 1983. Although epidemic in nature, the connections were not intentionally randomized, so that a stochastic analysis would have been impossible. The first system that used such randomization on purpose was the previously described Clearinghouse directory service [11], to solve the scaling problems that the designers were facing in 1987 (initially with much success). More recently, the

REFDBMS bibliographic database system [13] uses a gossip-based replication strategy for reference databases, while Bimodal Multicast [5] (see also Section 3.2) is also based on gossip. Astrolabe itself is based directly on earlier work described in [29].

9 Conclusions and Future Work

Astrolabe is a DNS-like distributed management service, but differs from DNS in some important ways. First, in Astrolabe, updates propagate in seconds or tens of seconds, rather than tens of minutes at best (and days more commonly) in DNS. Second, Astrolabe users can introduce new attributes easily. Third, Astrolabe supports on the fly aggregation of attributes, making it possible, for example, to find resources based on attributes rather than by name. A restricted form of mobile code, in the form of SQL SELECT statements, makes it possible to change the way attributes are aggregated rapidly. This, and other aspects of our database-like presentation, build on the existing knowledge base and tool set for database application development, making it easy to integrate the technology into existing settings. Finally, Astrolabe uses a peer-to-peer architecture that is easy to deploy, and its gossip protocol is highly scalable, fault tolerant, and secure.

These properties enable new distributed applications, such as scalable system management, resource location services, sensor networks, and application-level routing in peer-to-peer services.

We are currently pursuing several improvements to Astrolabe. Currently, the amount of information maintained by the attributes of a Astrolabe zone cannot grow beyond about a kilobyte. By sending deltas rather than entire new versions, we believe we can significantly compress the amount of information send in the gossip protocol and increase the size or number of the attributes significantly. Alternatively or additionally, the set reconciliation techniques of [17] can be used.

Also aimed at increasing scale, we are also looking at ways to *direct* the flow of information towards the nodes requesting aggregation. Although, using the *copy* attribute of AFCs a certain level of control over the flow is already possible, we are looking at additional ways that would allow us to increase the number of concurrently installed aggregations considerably.

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, Lidong Zhou, and the anonymous reviewers.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the Seventeenth ACM Symp. on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[2] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, Atlanta, GA, May 1999.

[3] 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.

[4] M. Balazinska, H. Balakrishnan, and D. Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In F. Mattern and M. Naghshineh, editors, *Pervasive Computing, First International Conference, Pervasive 2002, Zürich, Switzerland, August 26-28, 2002, Proceedings*, volume 2414 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2002.

[5] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[6] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987.

[7] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: an exercise in distributed computing. *CACM*, 25(4):260–274, April 1982.

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

[9] P. Bonnet, J.E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. of the Second Int. Conf. on Mobile Data Management*, Hong Kong, January 2001.

[10] A. Carzaniga, D.S. Rosenblum, and Wolf A.L. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[11] 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.

[12] R.A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, Fall 1992.

[13] R.A. Golding, D.D.E. Long, and J. Wilkes. The REFDBMS distributed bibliographic database system. In *Proc. of Usenix'94*, pages 47–62, Winter 1994.

[14] 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. *Computer Networks, Special Issue of Computer Networks on Pervasive Computing*, 35(4):473–497, 2001.

[15] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proc. of the Eighteenth ACM Symp. on Operating Systems Principles*, pages 146–159, Banff, Canada, October 2001.

[16] B.W. Lampson. Designing a global name service. In *Proc. of the Fifth ACM Symp. on Principles of Distributed Computing*, Calgary, Alberta, August 1986.

[17] Y. Minsky. *Spreading Rumors Cheaply, Quickly, and Reliably*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, February 2002.

[18] P. Mockapetris. The Domain Name System. In *Proc. of IFIP 6.5 International Symposium on Computer Messaging*, May 1984.

[19] 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.

[20] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the Sixteenth ACM Symp. on Operating Systems Principles*, pages 288–301, Saint-Malo, France, October 1997.

[21] S. Radicati. *X.500 Directory Services: Technology and Deployment*. International Thomson Computer Press, 1994.

[22] G. Reese. *Database Programming with JDBC and Java, 2nd Edition*. O'Reilly, 2000.

[23] 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.

[24] R.E. Sanders. *ODBC 3.5 Developer's Guide*. M&T Books, 1998.

[25] A. Snoeren, K. Conley, and D.K. Gifford. Mesh-based content routing using XML. In *Proc. of the Eighteenth ACM Symp. on Operating Systems Principles*, pages 160–173, Banff, Canada, October 2001.

[26] W. Stallings. *SNMP, SNMPv2, and CMIP*. Addison-Wesley, 1993.

[27] 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.

[28] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[29] R. Van Renesse. *Distributed Processing System with Replicated Management Information Base*. United States Patent 6,411,967, June 2002.

[30] R. Van Renesse. Power-aware epidemics. In *Proc. of the International Workshop on Reliable Peer-to-Peer Distributed Systems*, Osaka, Japan, October 2002.

[31] R. Van Renesse and D. Dumitriu. Collaborative networking in an uncooperative internet. In *Proc. of the 21st Symposium on Reliable Distributed Systems*, Osaka, Japan, October 2002.

[32] 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.

[33] M. van Steen, F.J. Hauck, P. Homburg, and A.S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, January 1998.

[34] 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.