Scalable Data Fusion Using Astrolabe

Kenneth P. Birman and Robbert van Renesse[1]
Cornell University and Reliable Network Solutions Inc.
{rvr,ken}@cs.cornell.edu

Abstract – *The dramatic growth of computer networks creates both an opportunity and a daunting distributed computing problem for users seeking to perform data fusion and data mining. The problem is that data often resides on large numbers of devices and evolves rapidly. Systems that collect data at a single location scale poorly and suffer from single-point failures. Astrolabe performs data fusion in real-time, creating a virtual system-wide hierarchical database, which evolves as the underlying information changes. A scalable aggregation mechanism offers a flexible way to perform data mining within the resulting virtual database. Astrolabe is secure, robust under a wide range of failure and attack scenarios, and imposes low loads even under stress.*

Keywords: Data fusion, data mining, distributed computing, peer-to-peer communication, aggregation, hierarchical databases

1 Introduction

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

Astrolabe is a powerful technology for data mining and data fusion. The system continuously computes summaries of the data using on-the-fly aggregation. The aggregation mechanism is controlled by SQL queries, and operates by extracting summaries of data from each zone, then assembling these into higher-level database relations.

Aggregation is analogous to computing a dependent cell in a spreadsheet. When the underlying information changes, Astrolabe will automatically and rapidly recompute the associated aggregates and report the changes to applications that have registered their interest. Even in huge networks, any change is soon visible everywhere. For example, Astrolabe aggregation can be used to identify sets of sensors which have made related observations – biothreat sensors reading low levels of toxins, coastal traffic sensors reporting vessels matching an Interpol profile of potential concern, and so forth. As a user's needs evolve, Astrolabe can be reconfigured on the fly by changing the set of aggregation queries.

We are not aware of any prior system offering the mixture of scalability, robustness and security seen in Astrolabe. A basic premise of our effort is that all of these properties will be important in emerging data fusion and data mining systems, because these systems often play a critical role in management and administration of mission-critical applications, detecting apparent intrusions or denial of service attacks, and may also represent a key piece of application functionality. In traditional, centralized, implementations of data fusion functionality, these issues can emerge as impediments to the user. For example, if computers are tracking state that changes once every second, and we have 10,000 such computers, the centralized database would be expected to keep up with 10,000 updates per second, a massive load. If that central system is unavailable, data fusion becomes impossible. And there are many security issues raised by such architectures. This paper will show how Astrolabe handles all of the analogous problems through its peer-to-peer protocols and hierarchical structuring of data.

This paper starts with a technology review. We then present the data fusion and data mining features of Astrolabe, which are based on its aggregation mechanisms. Astrolabe gains scalability and robustness at the price of generality, and we spend some time looking at the limitations of the system and their implications for developers and users. In particular, while Astrolabe is a database, it doesn't allow the user to do arbitrary database-style transactions, and it is important to understand the reasons and the degree to which one can work around these limits. For reasons of brevity, this paper omits a detailed scalability analysis, but we do summarize prior work on this problem.

[1] The authors were supported by DARPA/AFR-IF under contract F30602-99-1-0532 and by the Information Assurance Institute at Cornell University and AFRL-IF. Astrolabe is available from Reliable Network Solutions, www.rnets.com.

2 The Astrolabe System

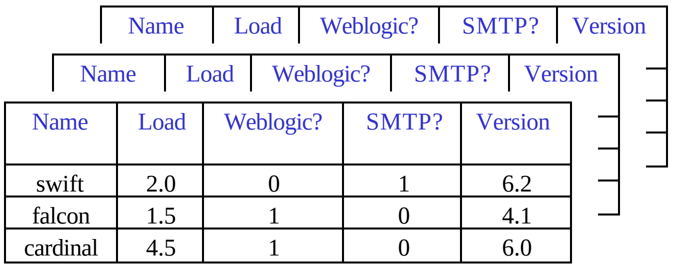
Astrolabe is best understood as a relational database built using a *peer-to-peer protocol*[2] running between the applications or computers on which Astrolabe is installed. Like any relational database, the fundamental building block employed by Astrolabe is a tuple (a row of data items) into which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

The data stored into Astrolabe can be drawn from the management information base (MIB) of a computer, extracted directly from a file, database, spreadsheet, or fetched from a user-supplied method associated with some application program. Astrolabe obtains flexibility by exploiting a recent set of standards (ODBC, JDBC) whereby a system like ours can treat the objects on a computer much like databases. Astrolabe is also flexible about data types, supporting the usual basic types but also allowing the application to supply arbitrary information encoded with XML. The only requirement is that the total size of the tuple be no more than a few k-bytes; much larger objects can be by identified by a URL or other reference, but the data would not be replicated in Astrolabe itself.

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

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

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



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

Figure 1: Three Astolabe domains

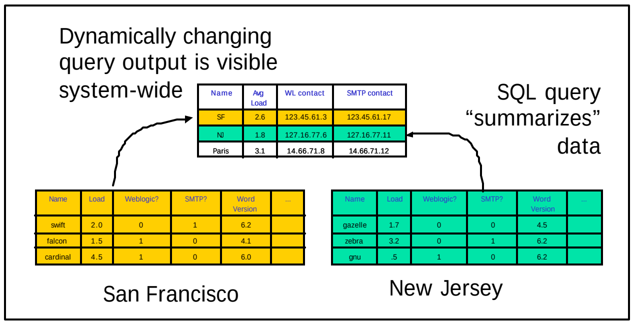
Where firewalls are present, Astrolabe employs a standard tunneling method to send messages to machines residing behind the firewall and hence not directly addressable. This approach also allows Astrolabe to deal with network address translation (NAT) filters.

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

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

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

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


Dynamically changing query output is visible system-wide
SQL query "summarizes" data
San Francisco
New Jersey

Name	Avg Load	WL contact	SMTP contact
SF	2.6	123.45.61.3	123.45.61.17
NJ	1.8	127.16.77.6	127.16.77.11
Paris	3.1	14.66.71.8	14.66.71.12

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

Name	Load	Weblogic?	SMTP?	Word Version	...
gazelle	1.7	0	0	4.5	
zebra	3.2	0	1	6.2	
gnu	.5	1	0	6.2	

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

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

The peer-to-peer protocol used for this purpose is, to first approximation, easily described [1][10]. Each Astrolabe system keeps track of the other machines in its zone, and of a subset of *contact* machines in other zones. This subset is selected in a pseudo-random manner from the full membership of the system (again, a peer-to-peer mechanism is used to track approximate membership ; for simplicity of exposition we omit any details here). At some fixed frequency, typically every 2 to 5 seconds, each participating machine sends a concise state description to a randomly selected destination within this set of neighbors and remote contacts. The state description is very compact and lists versions of objects available from the sender. We call such a message a « gossip » event. Unless an object is very small, the gossip event will not contain the data associated with it.

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

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

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

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

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

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

Our description oversimplifies. Astrolabe can actually support multiple aggregation queries, each creating its own hierarchy. The system can also be configured to accomodate heterogeneity of the leaf nodes, whereas we have presented it as if each leaf node has identical information. Moreover, the same peer-to-peer mechanisms used to propagate updates are also used to propagate new configuration certificates and new aggregation queries, hence the behavior of the system can be modified on the fly, as needs change. Details on these aspects, together with an enlarged discussion of our peer-to-peer protocol can be found in [10].

3 Consistency, Security and Expressiveness

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

3.1 Consistency

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

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

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

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

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

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

3.2 Security Model and Mechanisms

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

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

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

3.3 Query Limitations

A final set of limitations arises from the lack of a join feature in the aggregation query mechanism. As seen above, Astrolabe performs data mining by computing summaries of the data in each zone, then gluing these together to create higher level zones on which further summaries can be computed. The approach lacks a way to compute results for queries that require cross-zone joins.

For example, suppose that Astrolabe were used as the basis for a sensor network in a military targeting setting. One might want to express a data fusion query along the following lines: "for each incoming threat, report the weapons system best positioned to respond to that threat." The natural way to express this as a query in a standard database would involve a join. In Astrolabe, one would need to express this as two aggregation queries, one to compute a summary of threats and the other, using output from the first as an input, tracking down the response options. In general, this points to a methodology for dealing with joins by "compiling" them into multiple current aggregation queries. However, at present, we have not developed this insight into a general mechanism; users who wish to perform joins would need to break them up in this manner, by hand. Moreover, it can be seen that while this approach allows a class of join queries to compile into Astrolabe's aggregation mechanism, not all joins can be so treated: the method only works if the size of the dataset needed from the first step of the join, and indeed the size of the final output, will be sufficiently small.

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

4 Example

We believe that Astrolabe has data fusion and data mining applications in many settings. For the purpose of illustrating the ideas behind the system, however, we focus on a hypothetical military application involving servers and sensors operating in a networked battlefield setting.

Accordingly, suppose that we are interested in flexible detection of some class of airborne chemical and biological threats. A number of assets may have bearing on the detection and reaction to the threat : sensors within the system, but also terrain maps, predictions of weather for the affected area, etc. Should an event be suspected, we wish to report the danger to soldiers at risk.

To address the threat detection problem, we would start by configuring the system to pull threat detection reports into Astrolabe. For example, we might dedicate one column each in the Astrolabe table for measured concentration of each of a few dozen agents being monitored. Computers capable of sensing a given agent would report the associated value; computers not equipped with sensors would output a distinguishable value so that the absence of a sensor capability will not be misinterpreted as a report that no threat was detected.

Using Astrolabe’s aggregation capability, it is now a simple matter to build an aggregation that reports the overall threat status of the sensor network as a whole, highlighting those regions and sensors detecting the most serious risks or the highest levels of the monitored agents. Analysts monitoring the status of the battlefield would build web pages on which this information is dynamically superimposed; a trivial task given that Astrolabe appears to the application as a database. Indeed, most existing GPS-equipped mapping systems perform this task automatically, superimposing « sites of interest » extracted from a database on a map as the user approaches them.

An analyst who receives a report of a potentially serious threat would react by probing to confirm that the problem is real. For example, upon receiving a report that sensors in the northern quadrant of the battlefield are reporting threat detections, the analyst might «drill down » by fetching the regional Astrolabe database for that region and confirming that the pattern of detections is consistent with a chemical or biological attack

Suppose that an attack is believed to be underway. The analyst will now need to evaluate the likely consequences. Astrolabe could again play a role, for example by dynamically finding servers having data relevant to the area threatened by the event. The query here might be keyed by the coordinates from the system that made the original detection, and would look for databases and servers with information relevant to situational assessment in that area. Notice that we could either require that all servers continuously report the coordinates of regions for which they hold data or, more likely, could reconfigure

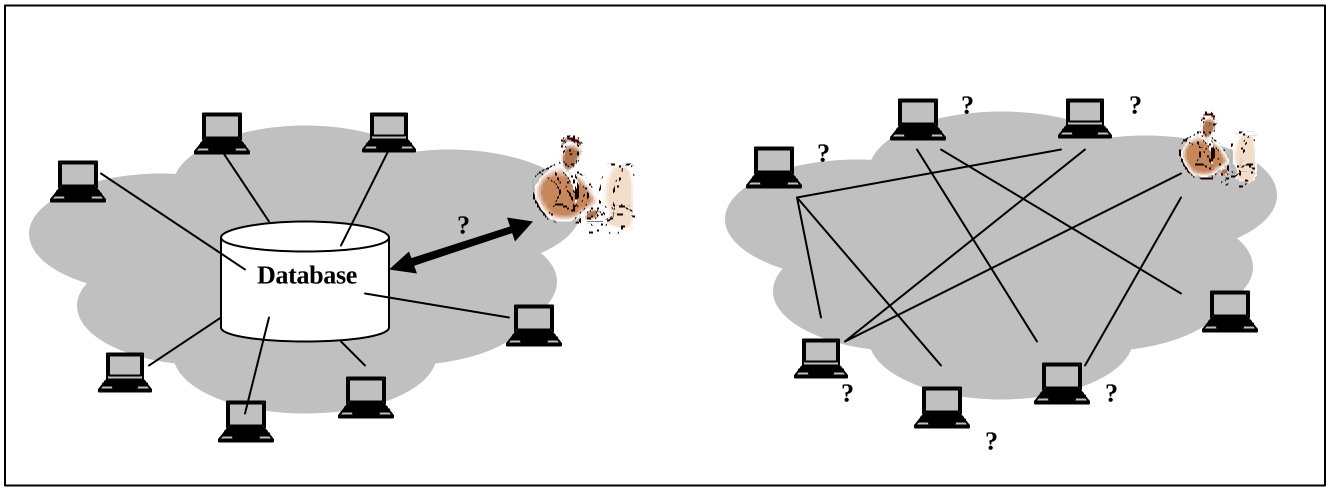


Figure 3: A conventional data fusion system gathers the data at a central site, where queries are performed (left). Performance is limited by the bandwidth to the central server and its update rate. Astrolabe’s power reflects the peer-to-peer protocol used to compute aggregates. The query is performed at the participating nodes, which experience only a low communication and computational load, independent of the size of the system.

Astrolabe on the fly to request that servers with relevant information begin to signal their availability. The latter approach introduces a delay of a few seconds, but makes much more efficient use of Astrolabe's limited-size tuples.

Finally, Astrolabe can be used to notify threatened soldiers. Again, we would simply configure the system to report the coordinates of soldiers, then use an aggregation query to identify that set of individuals. Here, we need to use a small trick: aggregates are summaries, not the raw data. Accordingly, while it is easy to compute the number of soldiers in each region who might be at risk, and hence to compute the total number of soldiers at risk, doing so does not yield a list of the contact information (email addresses, pager numbers) for those individuals. Additionally, as noted before, Astrolabe reads data from participating machines, but does not update them.

However, Astrolabe offers two mechanisms for overcoming this limitation. One option is to build a simple application that crawls the Astrolabe hierarchy. Starting at the root zone, this application would descend layer by layer, exploring zones with non-zero counts of soldiers at risk. Although fetching the data from these remote zones requires a remote procedure call, doing so infrequently doesn't represent a particularly high cost. Thus one can easily build an application to enumerate specific information for the soldiers at risk. Alternatively, if the goal is simply to send a message, Astrolabe offers a multicast interface that will deliver a message to a designated port number at each machine with a tuple matching some user-provided pattern.

As noted earlier, each of these steps completes within seconds or tens of seconds even in massive networks. Thus, with the exception of steps at which a human might need to intervene (for example, to verify that the detected attack is a legitimate threat), each step in the scenario described here can be completed in real-time. In contrast, traditional ways of solving such problems would require that data be gathered in a central database, and the delays associated with keeping that database reasonably current – the application we have described might involve millions of state changes per second – makes it clear that very long lags might arise between when an event occurs and when a response can be coordinated.

Notice also that because Astrolabe's queries are evaluated on the participating nodes, the system achieves massive parallelism (Figure 3). For example, a query might require that participating machines look for files containing some pattern (perhaps, images that could match a picture of a suspected terrorist). Such a task could take hours on a central server, since doing so would first require that all of these image documents be shipped to that server, and then that they be scanned one by one. Yet if millions of machines are each asked to scan a few of their own files, concurrently, the query results would be available within seconds, and no machine would be subjected to more than a transient computing load.

5 Performance

The dual goals of keeping this paper brief and of avoiding repetition of material reported elsewhere led us to omit any detailed performance section from this paper. However, Astrolabe is a real system and we have evaluated it in great detail. The interested reader is referred to [1, 9, 10, 11].

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

A second style of evaluation focuses on two kinds of simulation. First, using network simulation systems (NS/2) we have simulated Astrolabe to understand its behavior in a variety of network topologies and under a variety of loads and scales. Second, we have looked at the behavior of our Astrolabe implementation by running the real software over a simulated network. We do this by injecting packet loss or delays so as to emulate conditions that might be encountered in the field.

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

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

6 Related Work

Our work draws heavily on prior research in peer-to-peer computing and databases. In the database area, the idea of building replicated databases using gossip communication dates to the Xerox Clearinghouse server, a flexible directory service for large networks. Discussion and analysis of the protocols used in this system appears in [4]. Subsequent Xerox work on a database system called Bayou takes the idea even further [4, 8], and also includes a formal analysis of the scalability of push and pull gossip. The idea of building large-scale information systems hierarchically is an old one; many elements of our approach were anticipated by Lampson [7] and Golding [5]. Work on treating large sensor networks as databases can be found in [3]. The Ninja system replicates data using a peer-to-peer protocol similar to the one we use in Astrolabe, but lacks an aggregation mechanism [6].

7 Conclusions

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

References

[1] K.P. Birman, R. van Renesse and W. Vogels. Spinglass: Secure and Scalable Communications Tools for Mission-Critical Computing. International Survivability Conference and Exposition. DARPA DISCEX-2001, Anaheim, California, June 2001.

[2] B. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Comm. Of the ACM*, 13(7) : 422-426, July 1970.

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

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

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

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

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

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

[9] R. van Renesse, Y. Minsky, M. Hayden. A Gossip-Style Failure Detection Service. In *Proc. Of Middleware `98,* pages 55-70. IFIP, Sept. 1998

[10] R. van Renesse and K.P. Birman. *Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management and Data Mining.* Nov. 2001; submitted to ACM TOCS. http://www.cs.cornell.edu/ken/Astrolabe.pdf

[11] R. van Renesse, K.P. Birman. Scalable Management and Data Mining Using Astrolabe. Submitted to the First International Workshop on Peer-to-Peer Systems (IPTPS 2002).