1. The Importance of Aggregation*

Robbert van Renesse

Cornell University, Ithaca, NY 14853
email: rvr@cs.cornell.edu

1.1 Introduction

In this paper, we define *aggregation* as the ability to summarize information. In the area of sensor networks [1.1] it is also referred to as *data fusion*. It is the basis for scalability for many, if not all, large networking services. For example, address aggregation allows Internet routing to scale. Without it, routing tables would need a separate entry for each Internet address. Besides a problem of memory size, populating the tables would be all but impossible. DNS also makes extensive use of aggregation, allowing domain name to attribute mappings to be resolved in a small number of steps. Many basic distributed paradigms and consistency mechanisms are based on aggregation. For example, synchronization based on voting requires votes to be counted.

Aggregation is a standard service in databases. Using SQL queries, users can explicitly aggregate data in one or more tables in a variety of ways. With so many examples of aggregation in networked systems, it is surprising that no standard exists there as well. On the contrary, each networked service uses implicitly built-in mechanisms for doing aggregation. This results in a number of problems. First, these mechanisms often require a fair amount of configuration, which is not shared and needs to be done for each service separately. Second, the configuration is often quite static, and does not adapt well to dynamic growth or failures that occur in the network. Finally, the implementations are complex but the code cannot be reused.

There are only few general services available for aggregation in networked systems. "Mr. Fusion" [1.2] is a recent aggregation service intended for use with CORBA. Based on a voting framework [**?**], the Fusion Core collects ballots that are summarized when enough ballots have been collected. The output of the aggregation can be multi-dimensional, and represented as a hierarchical data cube. Cougar [1.3] is a sensor database system that supports SQL aggregation queries over the attributes of distributed sensors. Some other sensors network systems, like Directed Diffusion [1.4] have limited support for data aggregation as well.

We have developed an aggregation facility as well, called Astrolabe [1.5], for use by networked services. It resembles DNS in that it organizes hosts in a domain hierarchy and associates attributes with each domain. Different

* This research was funded in part by DARPA/AFRL-IFGA grant F30602-99-1-0532 and in part by the AFRL/Cornell Information Assurance Institute.

from DNS, the attributes of a non-leaf domain are generated by SQL aggregation queries over its child domains, and new attributes are easily introduced. Astrolabe can be customized for new applications by specifying additional aggregation queries. The implementation of Astrolabe is peer-to-peer, and does not involve any servers. To date, we have used Astrolabe to implement a scalable publish/subscribe facility and are currently developing a sensor network application with Astrolabe. Such a service may have many uses:

Leader Election. A group of cooperating processes often requires a coordinator to serialize certain decisions. Leader election algorithms are well known. Using an aggregation service, finding a deterministic leader is trivial, say by calculating the minimum (integer-valued) identifier. When this leader fails, the aggregation service should detect this and automatically report the next lowest identifier. Similarly, when a new member joins with a lower identifier, the aggregation service should report the new identifier.

Voting. In case a more decentralized synchronization is desired, voting algorithms are often used. These, too, can be implemented using aggregation. The aggregation service would simply count the total number of yes votes, the total number of no votes, and the total number of processes so the application can determine if there is a quorum. Weighted voting is a trivial generalization, and barrier synchronization can be done in a similar fashion.

Multicast Routing. In order for a message to be routed through a large network, it is necessary to know which parts of the network contain subscribers. An aggregation service should be able to aggregate data not only for the entire network, but also for parts of the network (domains). Now the aggregation service can count the number of subscribers in a domain, allowing a multicast service to route messages efficiently to only the necessary domains. As members join, leave, and/or crash, the aggregation service has to update these counts reliably.

Resource Location. Distributed applications often need to be able to locate the nearest-by resource, such as, for example, an object cache. If an aggregation service reports the number of such resources in each domain, a location-sensitive recursive search through a domain hierarchy will allow a process to find a nearby resource.

Load Balancing/Object Placement. A problem that faces many distributed applications is the placement of tasks or objects among the available hosts. Sometimes simple ad hoc solutions, such as round robin, are applied. Sometimes sophisticated placement and balancing algorithms are used, but they are tightly integrated with the application itself and not easily replaced with a new strategy. An aggregation service, by calculating such aggregates as the minimum, average, maximum, or total load in domains, can greatly simplify the problem of placement and separate the issue from the actual functionality of the application.

Error Recovery. When an error has occurred, say a host crash or a lost update message, it is often necessary to inspect the state of the application in order to detect and repair inconsistencies, or re-establish invariants. For example, when a host recovers, it may require a state snapshot from one or more other hosts. An aggregation service can help check invariants (akin to GPD), and provide the information necessary for the repair (e.g., the location of the most recently applied update).

1.2 Astrolabe

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

The implementation of Astrolabe is entirely peer-to-peer: the system has no servers, nor are any of its agents configured to be responsible for any particular domain. Instead, each host runs an agent process that communicates with other agents through an epidemic protocol or *gossip* [1.6]. The data structures and protocols are designed such that the service scales well:

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

In practice, even if the gossip load is low (Astrolabe agents are typically configured to gossip only once every few seconds), updates propagate very quickly, and are typically within a minute even for very large deployments [1.5]. In the description of the implementation of Astrolabe below, we omit all details except those that are necessary in order to understand the function of Astrolabe.

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

Since multiple agents may be in the same domain, the corresponding domain table is replicated on all these agents. For example, both the agents "/nl/amsterdam/vu" and "/nl/utrecht/uu" maintain the "/" and "/nl" tables. As hosts can only update the attributes of their own leaf domains, there are no concurrent updates. However, such updates do affect the aggregate values, and so the updates need to be propagated so agents can recalculate the aggregates.

Separately for each domain, Astrolabe runs an epidemic protocol to propagate updates to all the agents contained in the domain. Not all agents in the domain are directly involved in this protocol. Each agent in a domain calculates, using an aggregation function, a small set of representative agents for its domain (typically three).

The epidemic protocol of a domain is run among the representatives of its child domains. On a regular basis, say once a second, each agent X that is a representative for some child domain chooses another child domain at random, and then a representative agent Y within the chosen child domain, also at random. X sends the domain table to Y. Y merges this table with its own table, and sends the result back to X, so that X and Y now have the latest versions of the attributes known to both of them. X and Y repeat this for all ancestor domain tables up to the root domain table. As this protocol is run for all domains, eventually all updates spread through the entire system.

The rule by which such tables are merged is central to the behavior of the system as a whole. The basic idea is as follows. Y adopts into the merged table rows from X for child domains that were not in Y's original table, as well as rows for child domains that are more current than in Y's original table. To determine currency, agents timestamp rows each time they are updated by writing (in case of leaf domains) or by generation (in case of internal domains). Unfortunately, this requires all clocks to be synchronized which is, at least in today's Internet, far from being the case. Astrolabe therefore uses a mechanism somewhat similar to Lamport timestamps so no clock synchronization is required in practice.

The aggregation functions themselves are gossiped along with other updates, so that new aggregation function can be installed on the fly to customize Astrolabe for new applications.

1.3 Research Agenda

Above we have seen important uses for aggregation. Although much of the required functionality for aggregation is provided by Astrolabe and other aggregation services, all these services have shortcomings that suggest directions for future research. Below we list eight such issues, using Astrolabe an illustration of how such issues might appear:

Weak Consistency. Astrolabe samples the local state and lazily aggregates this information, but does not take consistent snapshots. There is obviously a trade-off between consistency and performance or scalability. Aggregation services should provide multiple levels of consistency so that an application can choose the appropriate trade-off.

Staleness. The aggregated information reflects data that may no longer be current. This is not the case when, say, counting votes (assuming processes vote only once), but it is the case when aggregating continuously changing attributes such as host load. Load balancing schemes that depend on such attributes may make incorrect decisions and be prone to oscillation. Research is necessary to improve accuracy of information.

High Latency. Astrolabe uses an epidemic protocol to disseminate updates. Although this protocol has excellent scaling properties, it takes significant time to evaluate an aggregation (measured in seconds), and thus the solution given above for voting may suffer from high latency. A hybrid solution of an epidemic protocol and a multicast protocol may improve latency while maintaining good scaling properties and robustness.

Inappropriate Hierarchy. The Astrolabe hierarchy is fixed by configuration, and this hierarchy is not always appropriate for an application at hand. Many applications will not even require a hierarchy. Future aggregation services will have to decide what kind of topologies are most appropriate for particular applications.

Limited Expressiveness. Astrolabe puts limitations on what aggregations can be expressed. This is partially due to SQL's limited expressiveness, but also due to the recursive nature in which aggregations are evaluated in the Astrolabe hierarchy. Also, in order to ensure that resource use is limited, the Astrolabe SQL engine does not support join operations. Future research could determine what expressiveness is possible while observing limited resources.

Limited Attributes. Astrolabe puts a limit on the size of the attribute set of a domain in order to make sure that its epidemic protocols scale well. Currently, an attribute set for a domain cannot be larger than about one kilobyte. This places significant limitations on how many aggregations can be executed simultaneously, and thus how many applications can use Astrolabe concurrently. Better compression in gossip exchanges may improve this situation. New protocols may be able to scale better in this dimension.

Weak Security. A scalable system has to be secure. The larger a system, the more likely that it will be subject to malicious attack. These attacks may include confusing the epidemic protocols, introducing bogus attributes, stealing information, etc. Although Astrolabe does use public key cryptography to avoid such problems, there are a number of weaknesses left. For example, in the voting algorithm described above, it does not prevent a host from voting twice. Also, the computational cost of public key cryptography is very high. New protocols may increase security while decreasing computational costs.

Unpredictable Performance. Although we have performed extensive simulation of large Astrolabe networks, Astrolabe still has to prove itself in a large deployment. Concern exists that as Astrolabe is scaled up, variance in its behavior will grow until the system becomes unstable and thus unusable. It may also place high loads on particular links in the network infrastructure. More detailed simulation may identify such problems before deployment, while more proactive protocols may avoid a meltdown.

Thus there is ample opportunity for research in this area, with practical implications for the development of distributed applications, particularly those of large scale.

References

1.1 Walz, E., Llinas, J.: Multisensor Data Fusion. Artech House, Boston (1990)
1.2 Franz, A., Mista, R., Bakken, D., Dyreson, C., Medidi, M.: Mr. Fusion: A programmable data fusion middleware subsystem with a tunable statistical profiling service. In: Proc. of the International Conference on Dependable Systems and Networks, Washington, D.C. (2002)
1.3 Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Proc. of the Second Int. Conf. on Mobile Data Management, Hong Kong (2001)
1.4 Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed Diffusion: A scalable and robust communication paradigm for sensor networks. In: Proc. of the Sixth Annual Int. Conference on Mobile Computing and Networks (MobiCom 2000), Boston, MA (2000)
1.5 Van Renesse, R., Birman, K., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (2003) Accepted for publication.
1.6 Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proc. of the Sixth ACM Symp. on Principles of Distributed Computing, Vancouver, BC (1987) 1–12