Replication in an Information Filtering System

Douglas B. Terry

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
415-812-4427
terry@xerox.com

In the Tapestry system developed at Xerox PARC [5], users provide queries to filter incoming streams of documents. These queries run continuously over a growing database of electronic mail messages, news articles, and other textual documents. In the current implementation, all filter queries for all users run on a single database server. Those documents that match a user's filter query (or queries) are queued up for the user and can be retrieved directly via an RPC interface or, as is typically the case, can be sent to the user via electronic mail. A version of the system has been in use by a handful of researchers for over a year, and an improved service is being made available to a larger user community. Clearly, having a single server machine is undesirable for scalability, performance, and fault-tolerance reasons. Thus, we are starting to explore the design of a distributed and replicated filtering service.

An information filtering system, such as Tapestry, has the unfortunate, and unusual, property that every unit of computation, i.e., filter query, must access every unit of data, i.e., document. There is no natural distribution or locality of reference to exploit. Therefore, the design of an information filtering service that is distributed across several servers must necessarily involve replication. Either the documents or the queries must be replicated on some or all servers. This paper discusses the replication requirements and some ideas for how to accomplish this replication.

One approach to improve scalability, i.e., to increase the number of users that can be supported by the service, is to partition the user community. Suppose each server is responsible for managing and executing the filter queries of a subset of users; each user's queries are executed on a single server. In such an arrangement, an arbitrary number of database servers can be added to the system to accommodate new users. The drawback is that all documents must be fully replicated on all servers. Full replication of voluminous data streams is generally a bad idea because of (1) the difficulty of maintaining the consistency of the databases in the presence of failures, (2) the disk space required to store the data, and (3) the network load utilized to propagate updates. Let's consider each of the issues in turn in the context of the Tapestry system.

When replicating documents in an information filtering system, eventual consistency is sufficient. The main requirement is that each filter eventually see each document. Existing replicated data algorithms that guarantee eventual consistency, such as epidemic algorithms [2], ensure that multiple updates to the same data item are applied at each site in the same order (otherwise the databases would not be mutually consistent), and they assume that the order of updates to different data items is unimportant. The ordering of conflicting updates is not germane to the Tapestry service since documents are read-only. On the other hand, the order in which documents are added to the database is important since this can affect the results of a continuous query [6]. Each site must apply the updates in an agreed upon order. This requires global agreement. Having a single site assign sequence numbers is a simple way of achieving this ordering, but is not very fault-tolerant. Using atomic ordered multicast [1] to send new documents to all servers is an expensive way to provide the ordering guarantees.

In typical information filtering systems, a filter examines individual data items as they arrive and decides whether or not the data is selected by the filter. In these systems, disk space is not an issue since the information is simply flowing through the filters and need not be stored for any period of time. The Tapestry system differs from these systems in that filter queries can refer to multiple documents. For example, one can add a filter like "select all messages to which Hector Garcia-Molina has sent a reply" or "select all messages that have a favorable review written by Hector." Thus, Tapestry filters can select old documents as well as new ones. This means that Tapestry does need to store all documents in a database. Fully
replicating this database on all machines requires a large amount of disk space. One solution is to partition the database and allow the execution of a query to involve multiple servers (e.g., cross-machine joins). This is costly and complicated. Another way of reducing the disk space requirements is to keep the database on some small number of servers. The replicated data algorithm would send all documents to all servers, but only some of these servers would provide long-term storage for the documents. Where would filter queries run? These queries could be divided into two classes: those that are simple predicates over a single document (e.g., "select messages from Hector") and those that involve multiple messages (i.e., database joins). The first class is by far the most common. These queries can be executed on any server. The second class of filter queries are restricted to those servers that store the complete database. This permits arbitrary replication for scalability along with bounded storage costs.

The third problem with fully replicated data is the large load imposed on the network for sending new documents to all servers. Processing 5000 or more new documents per day is not unexpected in a system like Tapestry. In the general case, each filter, and hence each server, must see each new document. In practice, however, it may be possible to take advantage of the semantics of the filtering process to reduce the number of documents sent to any given server. Specifically, a server need not receive documents that are guaranteed to not match any filters executed on that server. Filter-based replication can use the existing filtering mechanisms and filter queries to drive the replication process. A super filter query is produced from the individual queries that are run at a particular server (and updated whenever new filters are installed). The super filter query for server 1, $SFQ_1$, selects those documents that are of interest to server 1. For each server $i$, $SFQ_i$ can be installed as a regular filter query at every other server. Suppose that documents being added to the system can be submitted to any server. When a server receives a new document from some source other than another server, it sends this document to each site $i$ whose $SFQ_i$ selects the document. This requires little, if any, new mechanisms on each server. Whether or not this method effectively reduces the number of updates sent on the network is uncertain at this point. Nevertheless, filter-based replication is an interesting avenue for exploration.

Thus far, we have considered replication in the Tapestry system solely as a means for serving more users. Replicated document bases can also be used for increased fault-tolerance. In order to continue the execution of filter queries in spite of server crashes, the queries need to be replicated on two or more server machines. These replicas can be maintained using a conventional replication algorithm, such as weighted voting [4] or available copies [3]. One promising scheme is to have each filter query executing on a single server at any point in time. The other replicas of these filter queries are cold standbys. When a server fails, a standby for that server notices and restarts executing the failed queries. It is not strictly necessary to continue execution at the place in the document stream where the previous server ended since running a filter more than once over the same document does not produce different results [6].

Finally, consider the alternative strategy of fully replicating filter queries and partitioning the database of documents. In this case, the document database need not be replicated in any way. This approach is almost certain to save disk space because filter queries are much smaller and less numerous than documents. However, there are also several disadvantages. First, it is likely to use more CPU cycles since each server is running each filter query, albeit over a smaller database of new documents. Second, filter queries are not immutable, and so a replicated update algorithm is required. Third, each client of the filtering service must communicate with each server to retrieve selected documents. Fourth, constraints must be placed on the distribution of documents in order to guarantee that a query involving multiple documents can be completely processed on a single server.

This paper shows that the design of a distributed information filtering service involves challenges not faced in other distributed applications. Replication is needed, not for fault-tolerance or performance but simply for scalability. Of course, once replication is provided, it can be used to increase the fault-tolerance of the system (with some additional work). A new technique called filter-based replication is proposed for deciding what to replicate and where. The detailed design, evaluation, and implementation of these ideas remain future work for the Tapestry project.


67