DATA SHARING IN A LARGE HETEROGENEOUS ENVIRONMENT

Rafael Alonso
Daniel Barbara
Steve Cohn

CS-TR-272-90

July 1990

DATA SHARING IN A LARGE HETEROGENEOUS ENVIRONMENT[†]

Rafael Alonso

Daniel Barbará

Steve Cohn

Department of Computer Science

Princeton University

Princeton, N.J. 08544

ABSTRACT

The increased availability of networking technology, as well as the economic pressure for inter-company cooperation have created a great deal of interest in federated (or heterogeneous) database systems. Although there has been much work recently in this area, most of it addresses (implicitly or explicitly) an environment in which there are only relatively few cooperating entities. In this paper we explore the issues involved in sharing information among a large collection of independent databases. First, we discuss some of the distinguishing features that characterize such large scale environments (such as size, autonomy, and heterogeneity). We then outline a multi-step information sharing process for those systems, and present an architecture supporting that exchange. We conclude by providing a detailed description of a working prototype based on our architecture, and present some measurements of its performance.

1. Introduction

The distributed information processing scenario has changed dramatically over the

[†] This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, by the National Science Foundation under Cooperative Agreement No. DCR-8420948, and New Jersey Governor's Commission Award No. 85-990660-6, and grants from IBM and SRI's Sarnoff Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

last few years. While individuals are ever more eager to obtain information about a wide range of topics, currently existing databases have proliferated across a variety of networks, each under the control of a different organization, and with very little standardization among them. Furthermore, the increased availability of networking technology, as well as the economic pressure for inter-company cooperation have created a great deal of interest in data sharing among independent entities. This situation motivates major changes to the way in which distributed database management architectures are conventionally defined, and has led to increased research in the area of heterogeneous databases (also denoted as federated or multidatabases).

Although there has been much work recently in this area (e.g., see the proceedings from [NSF89]), most of it addresses, implicitly or explicitly, an environment in which there are only relatively few cooperating entities. For example, consider the research done on schema integration. Since the collaborating databases are managed by different administrators, in all probability the database schemas will be very different across the various sites. A number of papers have addressed the issue of integrating all the local schemas into a single global one, thus creating the perception of a single homogeneous system. While this is an excellent idea if the number of collaborators is small, the complexity of the integration process is such that this approach does not scale well. Therefore, although the few participants case is certainly an important one to consider, we feel that some attention must be paid to the case in which the number of autonomous participants is very large (say, from hundreds to potentially many thousands of databases).

In the next section of this paper we present some of the salient features of such environments. Next, we describe a number of technical problems that must be addressed by database designers in order to build large scale systems. In Section 4, we present an architecture which we have developed which we feel can help overcome some of the technical problems described in Section 3. Finally, we provide a detailed description of a working prototype based on the architecture of Section 4, and we measure its performance.

2. Environment

The distributed information processing environment that we envision exhibits a number of essential features that were not fully explored in previous work in more traditional distributed databases. Among these features there are three very important characteristics:

Autonomy: Organizations are finding out that, in order to cooperate efficiently, they will have to share information with their business partners. However, no matter how large the benefits of this cooperation are, it seems clear that most of these organizations are not willing to surrender control over their data as the price of cooperation. The local system administrators want (and should) have autonomy in deciding how their data is stored and manipulated, as well as over deciding which software or hardware to use. Moreover, autonomy should also be major concern in software operation. For instance, a local transaction trying to update data belonging to the local organization should not be excessively delayed or stopped because of the data sharing with other organizations.

Heterogeneity: As a direct consequence of autonomy, an information network is bound to be a heterogeneous system. Since local administrators have freedom to select the local architectures, a variety of software and hardware is likely to be present in the network, making data sharing a more difficult task.

Size: The rapid growth of worldwide networks will certainly result in a large number of nodes sharing information. This issue of scale has profound consequences in system design. In particular, it calls into question the practicality of supporting network-wide sharing via a unified data schema for the entire network.

We claim that whatever information sharing solution is offered for the environment in question must be compatible with the characteristics of autonomy, heterogeneity and size described above. We believe the architecture that will be proposed in Section 4 is well suited for these type of environments. However, before describing it, we will sketch the typical steps involved in the information sharing process.

3. Information Sharing

We view the process of sharing information among a group of independent entities as consisting of a multi-step process. This process consists of first finding a set of locations that might contain the desired information. Second, undertaking a negotiating process with the databases at those sites to be allowed to query them (or to obtain the raw data from them). Third, actually generating the query or a sequence of queries to the remote sites (or querying locally if the raw data was shipped over). Finally, terminating the interaction. Clearly, this four step process can be carried out in parallel with a group of information sources or sequentially with each in turn. Furthermore, if the user queries are not answered satisfactorily, the entire process may have to be repeated a number of times.

In the sub-sections below, we discuss each of the steps mentioned above in more detail. And in Section 4, we present an architecture that provides services for each of the stages described in this section.

3.1. Finding the Information

The first step in the information sharing process entails learning what data sources exist, where they are located and what kind of information they provide. It is clear that the size of the problem precludes the obvious solutions of having a network-wide global index or posing a roving query that traverses the entire network in an attempt to find sources that have the information needed. Simpson and Alonso [SiAl89] proposed an alternative way of dealing with this problem, called *external indexing*. Essentially, the solution is modeled after the way humans find information in society. Everyone knows the location of a set of information resources, either ''well-known'' sources (such as a library), or personal ones (such as an informed friend). When a person has a question which cannot be answered by the known sources, he or she asks these sources to recommend further sources. In this way, the source set of the individual grows and eventually will (probably) contain a source capable of answering the question. Nodes in a network can build such external indexes in the same manner. (The information source pointers are called external indices because, similarly to standard indices, they speed up data retrieval.)

3.2. Access Negotiation

Once the information source is found, the node interested in the data must negotiate with the owner of the data for access to it. This must be so in order to comply with the autonomy requirement. First, any individual component of the network has sufficient local control to autonomously determine which data it will share with the rest of the nodes and in what capacity. Moreover, even if the owner decides to share the information, another decision needs to be made. Sometimes the querier will require the information only once, but many times he or she will want to repeatedly ask for the data (for example, consider the case of a user interested in the latest stock information). Thus, the owner also needs to decide whether it will (a) allow the repeated requests, (b) provide a copy of the data to the remote querier and keep that copy sufficiently up to date for the purposes of the application or (c) deny access (note that access is denied at this point not for reasons of protection, but because the repeated querying imposes an unacceptable load on the owner machine). If the owner decides on either choice (a) or (c) he need only notify the requester of that fact, (and possibly, record the decision if future requests are being allowed). If option (b) is chosen, an agreement must be reached with respect to the level of consistency of the copy. We call these ''consistent enough'' copies *quasi-copies* (see [ABGA88] for a complete description of this topic). This level of consistency is dependent on the application that the users want to run in the local site. For instance the copy be kept perfectly consistent or, say, no more than ten minutes behind the original. All these issues affect the amount of autonomy an owner has to a larger or lesser degree. A protocol for data negotiation which addresses all these issues has been developed and presented in in [AlBa89]. Note that, along with the data, the owner will have to provide the client with the semantics of the information so that the client may interpret the data. There are a number of ways in which may be done. Among others: (a) it may be obvious (e.g., you requested a food recipe and received an English language description of the recipe), (b) it may be in a standard format (e.g., perhaps all New York Stock Exchange stock price data will always be quoted in dollars per share), (c) there may arise a set of well-known standards for a given type of information, (d) a natural language description of the data is sent along with it. Notice that this schema has the flexibility of letting the individual sites involved decide how the semantics of the data involved is going to be passed from the owner to the client system. We certainly do not think that schema integration techniques will be appropriate (in practice) for this problem, since there will be far too many systems in these environments, and many interactions will be too short-lived, to warrant the expense of such a process.

3.3. Query Generation

Once the agreement is reached, the query must be posed and the actual data given to the user or application program that requested it. Notice that having a local copy of the data after the negotiation eliminates the need for further data translation. The translation is made **once**. The local DBMS can assimilate the data and future queries can be posed in the local data retrieval language.

3.4. Termination

Both sites should agree on what conditions the ensuing contract will be invalid (i.e., when to terminate the data sharing). One of the key issues to be considered here are what to do if there is a communication (or node) failure. In the particular case of an agreement to share via quasi-copies, our approach is to accompany the sharing by periodic ''I-am-alive'' messages; should those messages stop, the quasi-copy support ceases.

4. Architecture

A variety of solutions have been proposed for the problem of interconnecting heterogeneous databases (for example *federated databases* [HeMc] and *multi-databases* [LiAb86]). For lack of space, we will not discuss here the virtues or the defects of any of them here, but simply pause to note that we present in this section an alternative architecture which shares some features (but not others) with previous work.

Figure 4.1 depicts the software modules that compose our system. In the figure we see the details of the interaction between two nodes, the client node that is trying to get a query answered for a user and the owner of the data which satisfies (at least partially) this query. (Notice that if the information required is only partially found at any one site , the process described here will have to be repeated a number of times.)

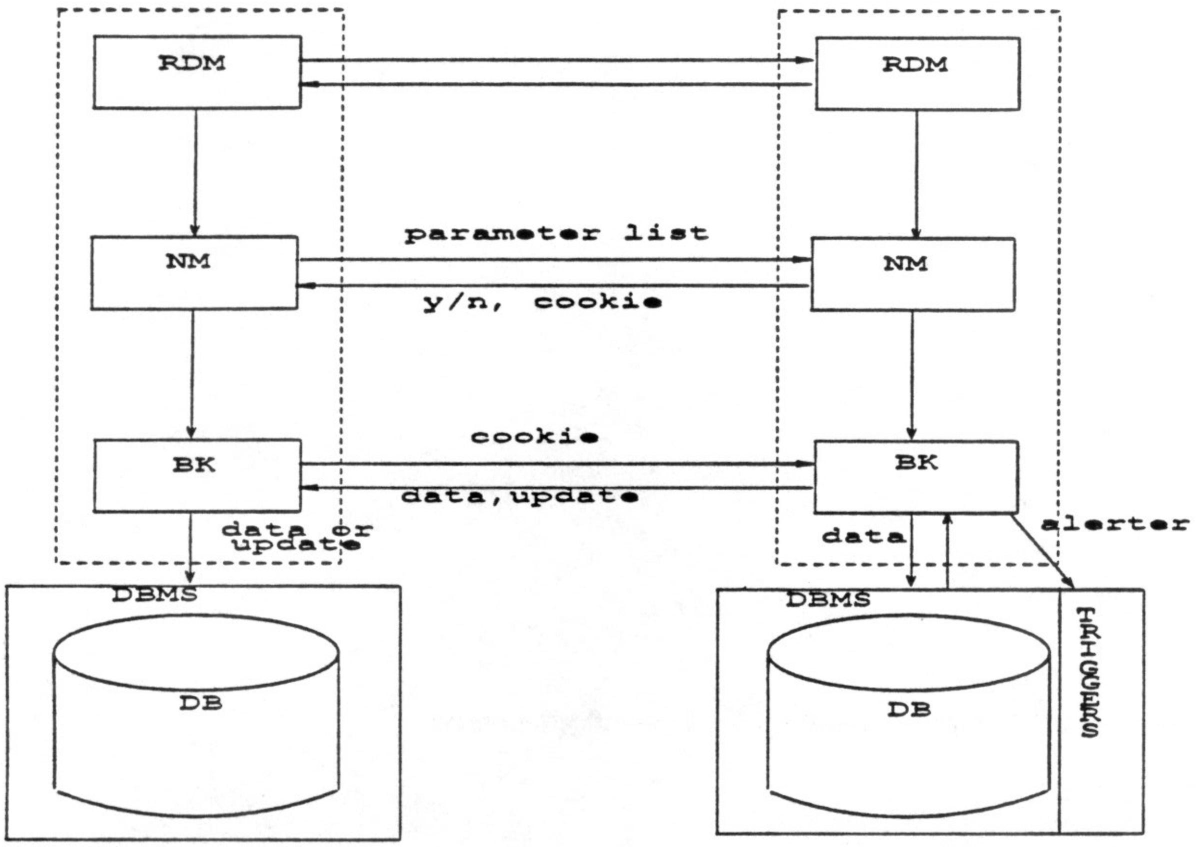

Figure 4.1 - Architecture

As the figure shows, there are three major components in the architecture. We now describe each one in turn.

4.1. Resource Discovery Manager

The first module, the Resource Discovery Manager (RDM), has the task of finding where the information needed resides. A detailed description of this module's work appears in [SiAl89]. Roughly, the requester provides the RDM with a set of keywords which describe what data is being sought (for example, ''doctor heart phone'' indicates a request for the phone number of a cardiologist). The RDM looks in its external index for any source matching these requirements (sources are also described in the external indices by a series of keywords). Any sources that match the query are potential candidates for containing the information of interest. If no sources appear in the local index, the RDM looks for sources that contain information about who might know the answer to the query. Thus, any sources that know about ''doctor'' or ''phone'' might be good candidates to be asked whether they know any sites that can answer the query of interest.

Clearly, the process described above is recursive. Furthermore, it is easy to imagine the role of well-known location servers in this process (e.g., the yellow pages can always be consulted in parallel to the above process). The actual descriptions that our system uses also involve weights as well as keywords (so that we might distinguish the most important aspects of the data we desire). There is also a mechanism for ensuring that we will not be inundated with data from any one site, another for dealing with hosts that lie, a third for discarding obviously irrelevant data, etc.

4.2. Negotiation Manager

The second module invoked is the negotiation manager (NM). After the source of the desired information is found, the RDM invokes this module to enter the process of negotiation between the two sites. To uniquely identify the negotiation, the respective RDMs pass an ID to the NMs. (This ID will be used by all the modules in the system to identify a given information sharing process.) The operation of the NM module is fully described in [AlBa89], and we will only summarize its operation in this paper.

The client site will send the owner a list of parameters that allow that site to evaluate whether to give the client repeated access to the data, send it a copy, or deny access. It is understood that any copy sent would be a **quasi-copy**. As we explain in [ABGA88], a quasi-copy is a cached value that is allowed to diverge from the original in a controlled way, which is determined by the actual user of the data. For instance, a user may specify that a copy should not diverge by more than 10% from the original, or that the information be no more than one hour old. Quasi-copies provide a very natural way of dealing with distributed data in very large systems, and one with reasonable performance overhead. The two principal characteristics of quasi-copies are its selection and coherency conditions. The first refers to *which* objects are to be copied. The second establishes the degree of inconsistency that the user is willing to tolerate.

The parameters provided by the client NM are the coherency condition and the estimated querying rate for the data being negotiated. Using these figures, as well as a set of internal parameters of the owner load and estimates of the update frequency of the items in question, the owner site decides whether to grant the request or not. It computes the load that repeated queries would generate, estimates the computational cost of maintain the quasi-copy, and considers whether the load imposed by the least expensive of the two will, when added to the average owner site load, exceed the maximum load that the owner site is willing to tolerate. Clearly, the higher the load that a node is willing to incur on behalf of another site the lower its autonomy will be (in the sense of losing a measure of control over its own cycles).

4.3. The Bookkeeper

The last module, the Bookkeeper (BK), is encharged of the actual transfer, interpretation, and maintenance (in the case of a quasi-copy) of the shared data. The NMs will pass the interaction ID to the BKs, so that they can initiate the actual data transfer. When contacted by the client BK, the owner BK will send the data over to the client, along with a the necessary information to understand the semantics of the data being sent.

The management of a quasi-copy's coherency will be performed in general by the owner BK, although in some instances, it is possible to unload this task to the client BK. For instance, if the coherency condition specify maximum delay time, the client BK can wake up when the condition is expiring, invalidate the current copy and request a fresh copy from the owner. Regardless of which site maintains the consistency, the actions taken are the same. When the coherency condition are violated, the BK must intervene and either send or request a fresh copy.

In principle, if the coherency condition is other than a delay condition, this means intercepting each update to the relevant data and checking whether the condition is violated, clearly not an acceptable task. However, this requirement can be circumvented if the quasi-copy facilities can be built on top of databases supporting triggers. That is, we install a trigger with the same start condition as the negation of the coherency condition of the quasi-copy, and have as the trigger's action the refreshing of the quasi-copy.

Finally, the client BK is responsible for instructing the user's DBMS to install the copy of the data locally, so it can be accessed by his or her applications.

5. Implementation

In this section we describe the implementation of a prototype information sharing system, based on the architecture just described (although the current prototype does not have the complete functionality allowed by the architecture). Necessarily, our prototype is very simple and small in scale, but we feel that it can help us shed some light on the problem we are addressing. While we will make several assumptions and create a few fictions, we hope the differences between this implementation and the theoretical world-wide model are only in detail and scale.

5.1. Overall organization of the prototype

The physical network in our prototype is composed of a set of Sun workstations connected by an Ethernet [MeBo76]. There are a number of UNIX processes in each machine, each modeling an autonomous (logical) site. The various processes maintain an individual Ingres [St76] database, and they communicate with each other via TCP/IP

[Postel80A,B] sockets.

The activity at each site is handled by a multi-tasking controlling program (in essence, a simplified operating system). There are a number of *database processes* under its control, each of which may be in a variety of execution states. These database processes share memory with each other, and use it to communicate.

We create a new database process every time we begin a search for information, or receive a request for data from another site. These processes are placed on various queues and handled one at a time. For instance, a process attempting resource discovery may send out several requests, and then place itself on a wait queue until it receives a response. Until that response comes, the site can handle requests from outside, or process other local queries. If a response comes in, the waiting process is pulled off the queue and allowed to handle the message. If no response comes in within a certain time, the waiting process will reach the head of the wait queue, time out, and act accordingly. There are four queues in each controller: a run queue (processes waiting to receive a time slice), a wait queue (processes waiting for a response message or a time out), a resource discovery queue (processes who are looking for information), and a dead queue (processes which have either failed to succeed in resource discovery, or have timed out waiting for a reply).

Since we believe that communication should be connectionless in the systems we are studying (mostly for autonomy reasons), we use one-way messages to send our requests and responses. Thus, sites can handle requests from multiple remote sites at once, and continue to operate asynchronously while waiting for other databases to respond. All messages have a standard header which contains the address of the database process (not just the site) for which it is intended.

5.2. Resource discovery

5.2.1. Requesting information

As we mentioned in Section 3, we must have a mechanism for finding the location of data. Thus, we have implemented the RDM module described in Section 4.1. As pointed out there, we want to be able to search for information based on a set of simple keywords, such as **(shoes white golf)** to indicate a desire for data about a particular item.

We implemented various keyword-based algorithms for different versions of our RDM. The simplest method was to associate a list of keywords with each relation, and then specify user requests with an unordered set of keywords. If any of the keywords were in the list of words for the relation, we would report a match. Clearly, this method could also have been expanded to include logical operations like AND and OR (i.e., **shoes and (white or golf)**), but we did not implement this extension. The main problem with this method is its simplicity, which leads to a lack of expressive power. We cannot indicate which keywords are more important, nor can we easily build up a history of our requests to make future similar requests faster.

A much more effective way of dealing with the keywords is to have each site maintain a tree of the keywords. In this tree we can put references to the information that the local site knows, and the names of sites that we know of that might be good sources for the information. For example, consider the tree in figure 4.1.

```
      SHOES
      /   /
  WHITE   GOLF
```

Figure 4.1 - An example of a keyword tree

If our local site knew about **white shoes**, then the **WHITE** node would have a reference to the local relation that contained the information we had on white shoes. The **GOLF** node however, might have a list of sites that we have heard of that might be good sources for golf shoes. The **SHOES** node would refer to our relations and our sources for any kinds of shoes (perhaps including, redundantly, white shoes and golf shoes).

We attach weights to each keyword in the tree and in our request, and then take the dot product to determine how close a match we may have. For instance, if site A had a lot of information on shoes, but very little of it was specifically on white shoes, its entry in its own tree might be **shoes (0.8), white (0.1)**. Site B may not know all that much about shoes, but most of what it knows is about white shoes, so it would have **shoes (0.3), white (0.7)**.

If we are requesting information on white shoes, we would want to consider how particular we want to be. If we request **shoes (0.7), white (0.3)**, indicating that we want shoes, and we would like them to be white, then site A will give us a dot product of 0.59, and site B will give us 0.42. If, however, we really emphasize that they be *white shoes* (**shoes (0.1), white (0.9)**) then site B will be preferred with 0.67 over site A with 0.17. Along with the weights on the keywords, we can specify a minimum threshold for the dot product, thus limiting ourselves only to sites with a reasonable chance of having the information we are looking for.

Maintaining a tree also has the advantage that once we find a source for a specific request, we can store that information in our tree. Then when we want to handle a similar request, we have a better set of starting points by looking at the nodes near the one we are requesting. If our tree were three levels, and our third level beneath **WHITE** had **TAP** and **PUMPS**, then we would consider any possible source for white tap shoes as a possible source for white pumps.

The main problem we found in using full n-level trees to store the information is that there is usually not a natural hierarchy in which to place the keywords. For instance, if one were indexing white golf shoes, one could put **SHOES** in the first level, but it is not clear what determines whether **GOLF** is a specific kind of white shoe, or **WHITE** is a type of golf shoe.

Our experience seems to indicate that the trees for most natural requests are actually only two levels high, usually with the subject in the first level and modifiers in the second (i.e. **SHOES - <WHITE GOLF PUMPS>, STOCKS - <NYSE AMEX NASDAQ>**). Third level qualifiers would more likely become part of the query once the relation is found (i.e. **SHOES - WHITE : SIZE 11, STOCKS - NASDAQ : Micsft**). For generality, full n-level trees were implemented in our system.

5.2.2. Candidate sites

Given the request in some form, we create a list of sites we think might have the information and send them a message asking them if they do. These initial sites are called *primary candidates*. If an initial site does not have the information, we can ask them to tell us the names of other sites who might know (i.e., which primary candidates would they choose if the query had originated at their site). Sites that do not have the desired information but may know about primary sites who do are called *secondary candidates*; we make each site on the list they give us primary candidates.

The process of finding candidates is iterative, and continues until the user is satisfied with the answer to the query. For instance, if we are looking for white golf shoes we would build up a list of primary sites by looking in our tree under **shoes golf** and **shoes white**. If this does not yield a sufficient number of primary candidates, then we might generalize and take anybody who simply knows about shoes.

We enter these primary sites in a table associated with the database process that lists the sites we have tried or will try for this particular request, and their current status (primary, secondary, waiting, finished, etc). Then each time the process reaches the top of the queue, we send out a request to one or more of the sites and then wait for replies. If a primary request (''Do you know about white golf shoes?'') comes back ''no,'' we change that site's status in the table to secondary so we will eventually send it a secondary request. When we get a response from a secondary request, we add any new sites as primary and mark the secondary site as finished.

5.2.3. Results of Resource Discovery

Eventually (unless there is no data pertinent to the query in the entire network) we will get a site that believes it has the information we want. When it replies ''yes'' to the primary request, it will also tell us the name of the relation in its database it thinks we want, and a *datatype skeleton.*

The datatype skeleton is a list of the relation's field names and their types, and from this it is expected that the requester can tell whether or not the relation is what they are looking for. (We made this assumption not because we expect that it will be true in reality, but because we wanted to keep the prototype as simple as possible; in practice, unless the keywords are standardized, some form of semantic information must be sent along with the skeletons so that the data may be interpreted.) If, for example, you want to get the current price for IBM's latest bond offering, you might specify you request as **bonds** and receive **company_name=string, price=dollars** from some brokerage house. Then you can form you query using these fields (**retrieve (r.price) where (r.company_name="IBM")**). On the other hand, if the site tells you it has a relation like **valence=shell_number, electrons=integer**, then you are probably on the wrong track and should continue to look elsewhere.

For simplicity sake, our model was based on a list of global datatype skeletons, indexed by a number. This saved us the trouble of sending the entire list of relation field names, and allowed us to automatically continue if the data we find is not of the exact skeleton type.

5.2.4. Learning

We already stated that maintaining a tree makes it easier for us to remember what we have found out so we can improve our searching the next time around. Unfortunately, it is next to impossible to save all the information we amass during the search. The list of sites, for instance, that *did not* have information on our subject will be quite large. We also must be careful not to write them off for good, as they may have been temporarily down, or they may acquire new information.

It seems best, then, to remember which sites had the information we wanted, and to later use them as starting points for identical or similar requests. This alone is a great advantage, as we can bypass the multiple levels of referrals (A told us to try B; B said try C; C had what we wanted) and go directly to the source (try C first). And if any of our collaborating databases ever ask for that same information, we can recommend C to them and save them the multiple-level requests altogether.

In our experiments, we observed that, after a while, the network starts to get ''smaller'' as each site builds up a list of who actually has what. Thus, if all the sites in the network were to continually request information on all known subjects, then eventually each site would build up a complete RDM tree which, for each subject, would refer directly to a site containing the information.

There is an important underlying assumption here that has not been mentioned. If a site we expect to have information does not have it, we ask them to recommend others.

This method works best if the sites who know about a subject also know other sites who deal with that subject. For instance, if the shoe store we ask for white golf shoes did not have it, we would consider them a particularly good source to tell us who might have the shoes. A real network will also be expected to contain yellow page servers: sites who know about a lot of other sites. We will want to keep track of as many of these sites as possible, using them as default secondary candidates for many requests. Even if we did not keep track of where we find information, an appropriately chosen list of these yellow page servers can reduce our searches down to only two messages (one to the server, one to the site it recommends).

5.3. Negotiating

Once the data has been found, and once we have decided that the information there is indeed what was being sought, we must then *negotiate* with the foreign site to get access to it. Real world parameters to such a protocol would include: security, price, size of the query, how often the data is needed, how current it needs to be, and how busy the providing site is at the time. For simplicity, in our prototype we ignore price (everything is free) and security (everything is public) and concentrate on cost of execution. The outcome of the negotiation is either allowed access, granting of a quasi-copy, or denial of service.

The basic premise behind our protocol is to allocate a certain portion of the machine's cycles to handling other site's requests, and then let remote sites have access on a first-come first-served basis. We approximate the cost of querying the database, and compare that with the cost of sending all the information across the network and making the requesting site do the work. If the request is not a single query, but a request for periodic updates (say, stock quotations), then we must take the consistency constraints into consideration. If a site wants to query our data every minute, but they only need it to be an hour up to date, then we may agree to maintain a quasi-copy.

We will not describe here any further the details of our protocol, since they appear in [AlBa89]; we have implemented the protocol described there.

5.4. Query Management

Once we have found a site who agrees to supply us data, the fields in the database skeleton are used to construct a RETRIEVE statement and a WHERE statement, which are then sent to the data source. The serving site executes the query on its local database, and sends the resulting tuples back to the requester.

If the query was only meant to be executed once, then when it completes both sites can dispose of the respective database processes. If the query is periodic, then both sites will put their database process on a wait queue. One of the two is responsible for waking up after δ seconds and restarting the query. The other will wait for a longer period (say 2δ), and timeout if the request has not been executed by then. The question of who should be responsible for waking up is an implementation detail; we have chosen to place the burden on the requester.

Since the query may take a relatively long time, and there might be many resulting tuples, it can be expected that the requesting process cannot wait for all the tuples to come in. It must therefore be able to handle them in any order (messages are not guaranteed to be sequential), and recognize when all of them have arrived.

It would also be the responsibility of the query manager to insure that the requester is not violating the terms of the negotiations. That is, if the sender agreed to update the requester every δ seconds, it should not notice if too many requests have come in a give time period. We have not implemented this feature.

Finally, since the interaction may be taking place via quasi-copies, the query manager for the requester must be prepared to send the updating tuples it receives to a temporary relation, and then query that relation locally whenever a local user queries that information.

5.5. Performance

We made some performance measurements of our prototype to gauge the usefulness of our architecture. Obviously, any results must be judged in context of the simplicity ot the prototype, and are to be taken as qualitative, not quantitative indications of success. The strongest statement we can make is that in running the system we did not see any obvious bottlenecks that would impede scaling it up another order of magnitude.

The tests were run on a single machine (a Vax 8650) to minimize the effects on network speed on the results. As we already mentioned, the databases used UNIX sockets to communicate, and (university) Ingres databases to store the (synthetically derived) information.

We were only able to run tests with about 10 sites, since multiple instantiations of Ingres caused our Vax to crash. However, Ingres itself ran ''relatively'' fast, although no comparable database was tested. Tests were made for files of different sizes (number of tuples (N) ranging from 10 to 100,000), and different proportions of the query succeeding (the fraction γ, ranging from 1/N to 1.0). Let us call the number of tuples retrieved k, which would be γN. Surprisingly, Ingres was almost as fast for small values of k as for large. The largest test, $N = 100{,}000$ and $\gamma = 1.0$, resulted in a retrieval time of 1.32 milliseconds/tuple.

The main bottleneck turned out to be message sending, which we feel we can attribute to the limitations of the UNIX environment. When tested with queries involving a very wide range of N, γ, and k, the total time was constant at 0.76 seconds/tuple. Moreover, it was apparent that the system was pausing between groups of eight messages. We attributed this to the tight limitations of UNIX sockets, which allow a queue length of no more that five (the eight packet statistic probably arose from being able to process three messages in the time it took for five others to back up on the queue. At that point, we believe the sending process was blocked on the send instead of dropping the packet. No message was ever lost). We could have improved the sending statistics by bundling multiple messages together when we could, which was usually only when we were sending the results of a query. This would have improved our time by a factor of M, had we put M messages in a single packet.

A real world system would use better methods. The networking needs are actually very simple (unordered, one-way packets), so there is no reason why performance should be lost in the software layers of the network. And since the transport time is far greater than the retrieval time (a factor of about 600 for our prototype), this is one area where implementation effort should be spent.

We also tested our resource discovery software. We created the synthetic databases such that each site had some information, knew sites that had a different type of information, and knew sites that had information similar to its own. We experimented with a number of factors, such as the number of sites (S), how many types of information there were (N), how many categories there were of each of the N types of information (M), and how many sites knew about each of the NM kinds of information (K). N and M were set to values around 10 each, and experiments with K showed values of 2 or 3 to be interesting. With S=100, a small number of sites were set to make random resource discovery attempts, building up their RDM trees in the process for their own future queries and secondary queries from other sites. As one would expect, in our simple network it did not take long before the searching sites found the correct sources for each of their queries. The number of ''hops'' became smaller rapidly, and the RDM tree quickly grew towards complete.

It is difficult to predict the performance of our RDM architecture in a real network, since it depends heavily on the nature and placement of the information. This much we can say about our system: if a site is continually searching for similar information, we are confident it will quickly become an ''expert'' if there is indeed any data to be shared; if, on the other hand, a site requests information on vastly different topics, it will always be starting from scratch; finally, the availability of yellow page servers will clearly have a major impact on the usefulness of our approach.

6. Conclusions

In this paper, we have listed some of the key characteristics of very large information networks. We have sketched the details of a multi-step information sharing process, and provided the salient features of an architecture supporting data sharing in large heterogeneous information networks. We have also discussed in depth our existing prototype, which enables users to query a collection of heterogeneous databases. Although the prototype is small in scale and simple in design, we feel that it has shed much light on the problem addressed by this article. We are currently expanding the functionality of our software, and expect to start running tests on real data in the immediate future.

7. References.

[ABGA88]

Alonso, Rafael, Daniel Barbará, Hector Garcia-Molina, and Soraya Abad, ''Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems,'' *Proceedings of the International Conference on Extending Database Technology,* Italy, 1988.

[AlBa89]

''Negotiating Data Access in Federated Database Systems'' *Proc. of the Fifth Conference on Data Engineering*. Los Angeles, CA, Feb. 1989.

[HeMc85]

Heimbigner, Dennis, and Dennis McLeod, ''A Federated Architecture for Information Management,'' *ACM Transactions on Office Information Systems,* vol. 3, no. 3, pp. 253-278, July 1985.

[LiAb86]

Liwin, W, Abdellatif, A., '' Multidatabase Interoperability,'' *IEEE Computer* Dec. 86.

[MeBo76]

Metcalfe, R. M. and D. R. Boggs, ''Ethernet: Distributed Packet Switching for Local Computer Networks,'' *CACM*, July 1976, pp. 395-404.

[NSF89]

Proceedings of the 1989 NSF Workshop on Heterogeneous Databases, Evanston, Illinois, December 11-13, 1989.

[SiAl89]

Simpson, Patricia, and Rafael Alonso, ''A Model for Information Exchange Among Autonomous Databases'' Technical Report, Princeton University, May 1989.

[Postel80A]

J. Postel, ''DOD Standard Transmission Protocol,'' RFC 761, Information Sciences Institute, January 1980.

[Postel80B]

J. Postel, ''DOD Standard Internet Protocol,'' RFC 760, Information Sciences Institute, January 1980.

[Ritchie78]

D. Ritchie and K. Thompson, ''UNIX Time-Sharing System,'' Bell System Technical Journal, Vol. 57, Number 6, 1978.

[St76]Stonebraker, M.R., E. Wong, P. Kreps, and G.D. Held, ''Design and Implementation of INGRES,'' *ACM Transactions on Database Systems,* vol. 1, no. 3, September 1976.