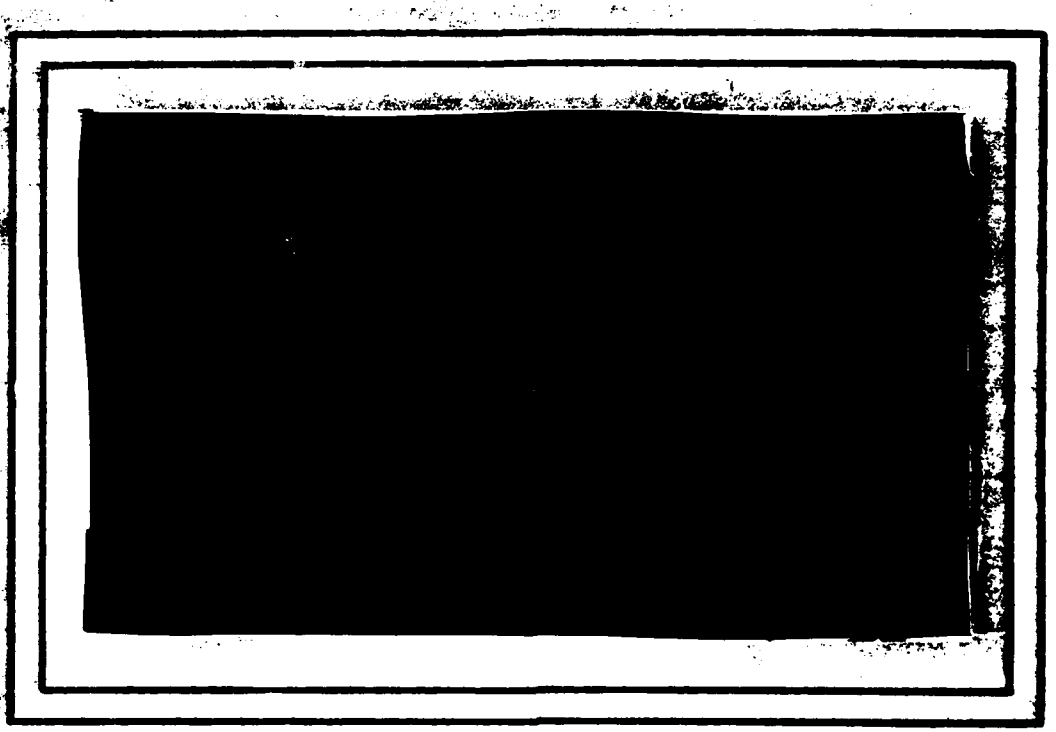
DTIC FILE COPY

(1)

AD-A225 362



COMPUTER SCIENCE TECHNICAL REPORT SERIES

DTIC ELECTE AUG 16 1990 S D

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

90 08 14 142

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

90 05 29 139

90-3586

Best
Available
Copy

(1)

UMIACS-TR-90-35
CS-TR -2424

March 1990

A Distributed User Information System

Steven D. Miller, Scott Carson, and Leo Mark

Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

DTIC
ELECTE
NOV 0 8 1990
S D
D

Abstract

Current user information database technology within the DARPA/NSF Internet is adequate to deal with hundreds of hosts and a few thousand users. However, recent size estimates of the Internet indicate that its host population is now closer to one hundred thousand machines, and that its user population numbers around one million users. The current centralized technology cannot scale to accomodate such a large information base, and provides no facilities for the distribution or replication of information. This paper presents the design of a distributed, scaleable user information database.

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and/or Special
A-1

1 User Information Server

The DARPA/NSF Internet [Qua86] provides its users with ways to communicate that are almost unparalleled in the world today. Users on hosts within the Internet can send mail, log into other machines, transfer files, do distributed computations, and interact in a large number of other ways. Having such a wide number of communications alternatives makes it very easy for users to collaborate on research and to communicate in general... once they know how to get in touch with one another in the first place.

Within the current Internet, tracking down a colleague can prove difficult. There are at this writing more than 114,000 hosts in the Internet, and a user population at least several times that size. In a single city of that size, the phone book, along with operator assistance, would provide at least a measure of aid to someone trying to reach a friend or colleague. In the Internet, there is essentially no phone book, and there is not even a way to find out where the operators are. The centralized user information service for the Internet – the WHOIS service run by the DDN Network Information Center [HSF85] – contains no more than 70,000 users, and probably many fewer. Furthermore, while the NIC attempts to keep its user data up-to-date, the highly centralized administration of this database makes it hard to update this database easily or frequently. While WHOIS is still a useful service, it is by no means the user information authority within the Internet.

Even for a large organization, such as a university, keeping track of all the people within that organization can prove to be a chore. Many organizations don't even do as well as the NIC. Few organizations have any sort of online user database, much less one that is kept up-to-date. Even those who do have problems keeping their own users informed about the existence of such a database; telling others within the Internet about the resource (and then getting them to use it) is more difficult still.

Consider how a searcher at Organization A might try to find the email address of a user somewhere at Organization B. All that the searcher knows is his colleague's name, and that his colleague has an account somewhere within Organization B. The first step might be to query the NIC WHOIS service, looking for information on users with the colleague's name. If that doesn't work, the likely next step will be to use the FINGER service [Har77] with the names of some likely machines within Organization B, and some likely user names on those machines, in an attempt to get lucky. Next, the searcher might send mail to a postmaster on one of those "well-known" machines, asking for more information. As likely as not, the postmaster doesn't work for the same department as the person being searched for, and will forward the query to other postmasters at Organization B. Eventually, if the searcher is lucky, an answer will turn up. This search procedure has a high probability of failure, and wastes a lot of time on the part of the searcher and on the part of the postmasters he asks to help in his search.

What is needed to help searchers find the information they search for is a distributed,

powerful, Internet-wide user directory service. In this paper we first discuss what one might want from such a service, borrowing heavily from the goals presented in [Sol89]. We then present the outline for such a service, discuss some of the implementation issues involved, and provide a detailed example of how a searcher might take advantage of such a service. We also discuss security issues and provide a comparison between our proposed service and other user information servers.

Overall System Goals

Before designing a user information database, we must consider the features users will want from such a database. This section enumerates these features.

First, the database must be able to store large quantities of information of multiple types for multiple users. Some possible types might be (but should not be limited to):

- office and home phone (and more than one of each)
- work and home address (with room number)
- department affiliation
- "home" machine
- general text comments
- electronic mail address

Second, the User Information Server (UIS) should support the aliasing of one key name to another. For example, it should not be necessary to know that the primary key for all of the author's information is `steve`; lookups on `Steven D. Miller` should work (or be able to be made to work) just as well. The UIS should also contain within itself a description of how the database is organized, so that the addition of attributes defined locally within a particular organization can be done easily (and so that others can learn about, and take advantage of, such additional information).

Third, the UIS should support (or, at least, be capable of supporting) more than one type of query. Queries should be accepted in a number of well-defined formats (i.e., phonetic, UNIX [1] pattern match, SQL, and so on), and multiple results should be allowed.

Fourth, the UIS should support some sort of delegation-of-authority mechanism. Large, centrally-administered databases tend to get out-of-date; bringing the administration of these databases closer to those with information in them (and possibly into the hands of the user) will most likely result in more accurate information.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Fifth, as an outgrowth of the above, the UIS should allow users to specify very general, ill-aimed queries, and get enough information via referrals to formulate tighter, better-aimed queries. It should also allow users outside the organization the same retrieval access as is provided to those within the organization.

Sixth, the UIS should interoperate well with other network entities. The UIS should support a number of different low-level protocols, such as TCP/IP [Pos81], OSI [Ros89], DECNET [McN88], and so on. Since the OSI X.500 Directory Services [Int88a] will soon be a fact of life, the UIS must interoperate well with X.500 users and implementations. The UIS should also support a variety of reasonable user and programmatic interfaces.

Seventh, the UIS should be highly distributed. Making this database distributed greatly increases its reliability and robustness in the face of the failure of individual hosts or parts of the Internet. Also, such a database is likely to have to support a high query rate, and is likely to have to transfer a substantial amount of data across the Internet. Making the UIS distributed helps to distribute the workload across a number of machines and network links. By providing some way to offload queries to entities near the data being searched, we can also reduce the amount of raw data being passed around the Internet.

Finally, the system should, like all systems, be as easy to use, administer, and implement as possible.

Now that our goals are more clear, let us examine first one possible abstract model of a user information system, and then a possible implementation of that abstract model.

A High-Level View of the UIS

The conceptual schema of the UIS is essentially one of a simple relational database. The relational model provides a number of benefits:

- The model is straightforward, intuitive, and one with which many people deal everyday.
- It provides a number of opportunities and methods for optimization.
- It places the UIS on a sound theoretical footing.

The UIS can be viewed as containing a number of primary types (i.e., **user**). It is tags of this type upon which queries are most often done, though we should not prohibit queries on other types within the database. Within each primary type, we have a number of attributes defined on each primary key within that type. In the case of the **user** primary type, we might define attributes such as **work-phone-number**, **postal-address**, and so on.

In the abstract sense, we treat each primary type *in the network*, along with all its possible attributes *as defined anywhere within the network*, as a single relation. We

name the relation with the name of the primary type, and define an attribute in that relation (or, if one prefers, a column in the table for that relation) with the name of the primary type. We define additional attributes (or columns) for each type to which we map the primary type. We emphasize that this view of the UIS is but a *fiction* of a relational database [Sno88]. The distinction between this view and the information it contains will be made more clear in later sections.

One very important attribute present within all relations in the network is the **organization-tag** attribute. The organization-tag attribute is a formal organization name (i.e., a domain name or an X.500 organization name) that can be used as some indicator where the information for that organization resides. This attribute is essentially a tag through which the UIS implementation can locate the servers associated with a real-life organization. Its contents are opaque to the user, though they should be human-readable, and should even be fairly easy to remember (or to guess). Thus, an organization tag of **umiacs.umd.edu** or (to use the notation of [RKT89]) "**@c=US @o=University of Maryland @ou=Institute for Advanced Computer Studies**" would be reasonable to associate with the author's organization, while a tag such as **XYZ1138** would be less desirable.

For the moment, at least, we treat the organization-tag attribute specially: all organization-tags on all relations referenced within a specific query must be fully-specified, even if only in terms of other organization-tag attributes within a relation. By leaving the organization-tag information unspecified, one is in effect stating that all organizations are to be queried, and thus that all organizations within the Internet *must* be polled. This is a laudable goal, but is likely to consume enough network bandwidth as to make it impractical within the current Internet.

In distributed database terms, all relations within the database are horizontally fragmented on the **organization-tag** attribute. Each fragment is then allocated to one or more sites. The UIS provides a close simulation of fragmentation transparency: users need not know where a fragment is located, or even how it is fragmented, but users (or the applications serving as their proxies) do need to have some knowledge of organization tags to allow the system to work.

Let us consider a sample relation. We will, in all likelihood, have a **user** primary type, as people will often pass in user names and ask for information on the basis of those names. We thus define a relation **user**, with attributes **user**, **organization-tag**, **postal-address**, **full-name**, and so on. If we depict this in column form, we have:

Relation USER

User	Organization-Tag	Postal-address	Full-name	...
steve	umiacs.umd.edu	UMIACS...	Steve Miller	...

In this case, attribute lookups become simple projections. If we use SQL as our query language, we might have:

```
select user, full-name
from user
where full-name = Steve Miller
and organization-tag = umiacs.umd.edu
```

The result might look like:

User	Full-name
steve	Steve Miller

We also define an `alias` relation. The contents of this relation are fairly simple: an actual alias, the name to which that alias should be mapped, the type of the new name, and an organization tag with which that name might be looked up. For example:

Alias	Name	Type	Name-organization-Tag	Organization-Tag
Steven D. Miller	steve	user	umiacs.umd.edu	root

If we then allow relational joins, we could look up the author's postal address with:

```
select user.postal-address
from user, alias
where alias.alias = Steven D. Miller
and alias.organization-tag = root
and alias.name = user.user
and user.organization-tag = alias.name-organization-tag
```

Since the organization-tags for a given organization may be hard to remember, we also suggest that a `friendly-organization` relation be defined. This relation would contain "user-friendly" strings corresponding to real-world organization names (i.e., "University of Maryland"), along with their corresponding organization-tags (i.e., `umd.edu`). Using this relation, users who can't remember organization-tags have some easy way to choose likely candidates for further queries.

In addition, we define a `uis-catalog` relation. This relation holds information about the shape and form of the UIS itself. Within such a catalog, we could insert a list of attributes defined within a particular relation within a particular organization, a list of relations defined within a particular organization, access control information, integrity constraints, and other information of the sort usually kept within the system catalog of a relational database system. By performing queries on this relation, users and applications can discover more about relations within the UIS system, and in particular can see what local extensions may have been implemented within a particular organization.

As an example of data that might be stored within the `uis-catalog` relation, consider the case where Organization X (with organization tag `xyz.com`) has a need to maintain information on the eye color of their employees. They could enter information into their part of the `uis-catalog` relation indicating that, within their organization, the `user` relation has this extra column. Outside users who wish to see what additional information might be available could do a query on the `uis-catalog` relation to discover that this information is present, and how to reference it.

We could also define other tables (i.e, `inventory`), and carry out some fairly powerful queries even with only fairly simple SQL-like join, select, and project statements. So long as we have these operators, it should be possible to answer most queries of the sort likely to be posed against the UIS. It may be worth including the union, Cartesian product, and set difference operators (making the system relationally complete); time and experience will tell. These latter operators can almost certainly be deleted in a prototype implementation.

So, conceptually, we have a user query agent which takes some sort of input from a user, casts it into relational form, and then queries the database as appropriate to return the desired information to the user. We now look at how we might decompose into pieces a system that will allow this sort of query, while meeting all our goals.

2 UIS Components

For any database, one needs essentially only three pieces: the interface through which users query the database, the software with which that interface communicates, and the actual data store itself. In this section, we discuss each of these three pieces as they apply to the UIS, and how they interact.

We define the *Query User Agent* (or *QUA*) as the part of the UIS system with which users (and applications) usually interact. The QUA presents an interface which users may use to look up information within the UIS. It may also store in its own memory some record of past queries and their results, for reasons that will become clear later. A QUA may take many forms. It may look like a line-oriented SQL interface. It may look like a forms interface, either on ordinary character-based terminals, or on a window system such as the X Window System [GSN88]. Other than the information the QUA gathers by doing queries against the UIS, the QUA has no direct access to the information in the UIS. To get at such information, the QUA must communicate with an intermediate entity.

We define the *Query Service Agent* (or *QSA*) as the part of the UIS system that sits between the QUA and the actual data store itself. The QSA is thus responsible for presenting the external schema, and for implementing part of the internal schema. Placing an intermediate agent between the data store and the QUA offers a number of

benefits. In particular, the type of data store can be changed easily, completely, and transparently, so long as the QSA is changed to know about the new type of data store. One can change the data store from (say) a replicated relational database to a database implemented atop the Internet Domain Name System [Moc87a, Moc87b], to a database implemented atop the ISO X.500 Directory Services, to something else entirely, even changing the transport-level protocols by which the QSA communicates with the actual data store, without affecting any user or application using the UIS. Since we can change the data store easily, we can be very flexible in terms of the queries we support. If we want to provide the user with a SQL interface, the QSA can do so... even though the data store is not a relational database. It is the QSA which provides the bulk of the "glue" required to provide the fiction of the DS as a relational database.

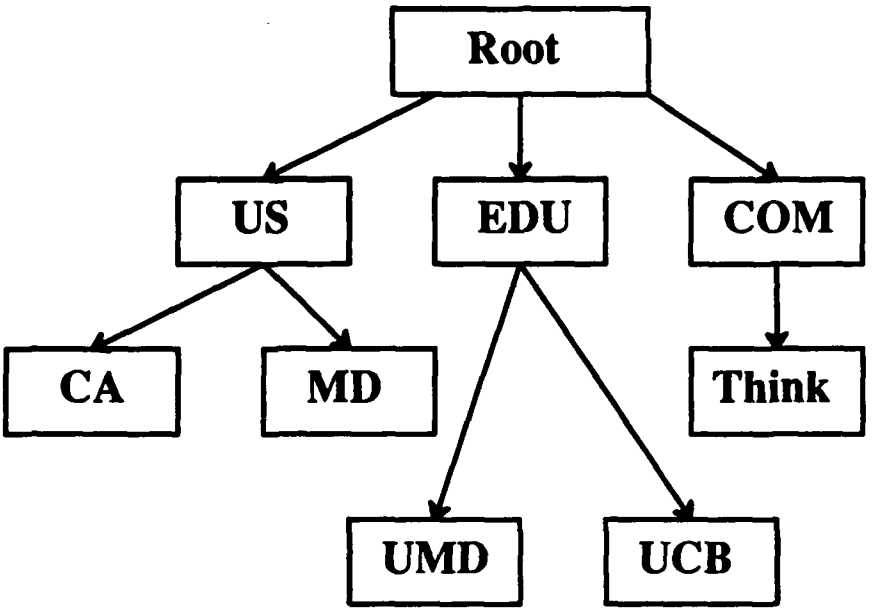
We define the *Data Store* (or *DS*) as the part of the UIS system responsible for holding UIS data. The DS is responsible for the small part of the internal schema not implemented by the QSA.The actual format in which the DS stores its data (or even makes it available to the QSA) is unimportant. We will discuss a few possibilities in this paper.

The overall structure of the UIS system is a hierarchical one. Organizationally, the UIS tree consists of a distinguished organization, which is considered to be the root of the tree. Underneath this organization (which we will call the *root*) are other high-level organizations. These high-level organizations may be true organizations (i.e., countries) or general groupings of related organizations (i.e., commercial versus educational organizations). Beneath these organizations are "normal" organizations (i.e., The University of Maryland, Thinking Machines Inc., etc.), which can have further sub-organizations to any depth desired. The organization tree is traversed only from the top.

With each node in the organization tree, we associate a QSA and its DS. Each node also has associated with a machine-friendly organization tag. We also assume the existence outside the UIS of a directory service which the UIS software can use to translate organization tags into the network addresses of the QSA associated with that organization tag. The process of deriving a network address from an organization tag may be a multi-step one.

To give an example, say that we want to look up information with an organization tag of **umd.edu**. The first thing we must do is determine where within the network the QSA for **umd.edu** resides. To do this, we ask the root directory servers, "where are the QSAs for **umd.edu**?" The root might say, "I don't know, but talk to these directory servers, which represent the **edu** grouping." We then ask the **edu** servers about **umd.edu**; since these directory servers are supposed to know either that this organization tag does not exist, or where the directory servers for that organization reside. We then finally ask the directory servers for the organization (here, the University of Maryland) our question, and receive an answer, which we cache for future use.

Figure 1 shows a simple diagram of part of the UIS tree.



3 UIS Implementation Notes

What Do We Need From Databases?

The UIS system, as described previously, mimics a distributed relational database (or *multidatabase* [LMR89]) to a large extent. However, there are some significant differences between the UIS and a "real" database. In this section, we examine some database features, and determine which ones we truly need in our implementation.

Like any database system, the UIS is only as good as the user interfaces it supports. Since SQL is such a standard in the database marketplace, a UIS design that would preclude SQL would be undesirable. SQL is much more of an expert's interface than an interface for the average user. As such, support for visually-oriented interfaces (perhaps like Query-By-Example) is also necessary. Given the basic SQL interface, it should be possible to build QBE or other similar graphical query tools by having such tools transform visual queries into SQL. A visual interface will greatly facilitate "ad-hoc" queries... which are, after all, what this system is all about.

Since the system is so distributed, and since it may support a high query rate, performance issues are crucial. As such, query optimization is extremely important. The existence and use by the system of a semijoin-style operator is important, as such an operator will allow both for highly distributed computation and for substantial reductions in network traffic. Within the servers for the system, a number of caching and

indexing schemes will almost certainly be necessary.

No successful worldwide distributed database can be administered centrally. This has been amply proven a number of times in the past, perhaps most visibly in the transition within the Internet from a centrally-administered host table to the use of the distributed domain name system. Because highly centralized administration is inadequate, the autonomy of individual sites within the UIS must be maintained.

Security is a desirable feature, but not an absolutely necessary one. In many cases, trivial security will be sufficient. By making updates moderately hard (i.e., supported only by logging into a particular machine within an organization) and by storing within the UIS only data that can be made public without risk, trivial security can be provided. More substantial security may be worth adding to the system; this topic is discussed more fully later in this paper.

We have now listed many database features needed by the UIS. These features are all easy to moderately difficult to implement. However, there is one very important difference between the UIS and a traditional database: while queries may be frequent, updates are extremely infrequent, as most of the information within the database is of a type that changes only rarely. This difference has important consequences in the areas of integrity, concurrency control, and recovery.

Since updates are so infrequent, we can get away with allowing the data in the UIS to be somewhat inconsistent without causing major problems. The UIS is much like a telephone book; people use it to get the information they need to contact people, but if the telephone book is not absolutely current, there are other methods, including searching elsewhere in the telephone book, which may be used to track someone down. Thus, we need not worry about one piece of information appearing completely consistently within two relations. If, for example, an entry in the `alias` relation points to a user who has moved or who no longer exists, the inconsistency is merely annoying, not crippling. After all, it may be possible to search elsewhere within the UIS to find that person, and it may be possible to contact people at the place to which the alias points in the hopes that a human will have more information.

Because inconsistencies pose only minor problems, we do not need any sort of integrity rules to prevent them from happening. We also do not need to worry much about locking or other forms of concurrency control; the lost update and uncommitted update problems disappear in a system without frequent updates. This does not mean that we should tolerate inconsistencies (i.e., half-updated records) within a particular part of the UIS. However, the UIS *as a whole* may have parts which are inconsistent. We must still have local consistency within each copy of a piece of data; it is just that we allow copies to be somewhat out of sync with one another.

Similarly, most of the recovery procedures within the database field are designed to make sure that updates are done (or not done) properly. Again, since updates are infrequent and inconsistencies are tolerated, these recovery procedures are largely

superfluous.

UIS Data Store

Because of the use of the QSA to hide the details of information storage from those looking for information, the form of the UIS Data Store (DS) is actually the least important part of a whole UIS implementation. Nonetheless, if the DS provides certain features, the QSAs will be somewhat easier to implement, use, and administer. We will discuss these features, and then give some details on two possible DS implementations.

The first of the features the UIS desires in a DS is distributed operation. If the DS is itself distributed, the loss of a single copy of the DS should not significantly decrease the service the QSAs are providing. Ideally, the DS should be able to accept updates from one source and automatically push those updates out to other DSes without human intervention. This does not preclude in any way a master/slave approach; one DS for an organization can be the source of all updates, with the backup DSes for that organization retrieving a copy of the updated information as needed. This update scheme is similar to that discussed in [Moc87b], among other places.

The second feature is residence near (in a network sense) the QSAs for which the DS provides data. In some implementations, the DS will be part of the QSA (i.e., an on-disk relational database). However, a QSA implementation can conceivably use the DS only as the source of all data, making a private copy of the DS data in some format more appropriate for its own manipulation. In such a case, the potential for large and frequent data transfers between the DS and the QSA exists. Ideally, such transfers would not be across slower, long-haul networks.

One potential DS which could satisfy both these needs is the Internet Domain Name System. The DNS already provides many of the features we want in the UIS. It provides redundancy, allowing for multiple servers for a given domain, and for automatic updates between master and slave servers. It is hierarchical, and provides a mechanism for delegation of authority which, if mirrored in the QSA organization, meets our needs for allowing information to be spread throughout the Internet. It also supports multiple query types.

There are a number of different ways in which we could encode UIS information within the DNS. One possibility would use the naming scheme of [Moc89] to specify the relation, the value for the primary key for that relation, and the value of the organization-tag. The values returned could be plain text records (i.e., the TXT resource record type defined in [Moc87b]). In this scheme, we would take a multi-column relation and re-express it as a number of ternary relations between the primary key for the original relation, the name of the original relation, and each attribute within the original relation. Then, to represent within the DS a value within one of these ternary relations within a particular organization, one looks for the TXT record (or records)

associated with the domain name:

primary-key-value.*original-rel-name*.*attr-name*.UIS.*organization-tag*.

Thus, the author's work phone number information might be stored as a TXT record associated with the domain name:

steve.user.work-phone-number.UIS.umiacs.UMD.EDU.

There are some undesirable restrictions with this scheme, however. First, since no domain name can be longer than 255 octets, the relation name, key value, attribute, and organization-tag must all fit within 255 octets. Second, no TXT record can be longer than 255 octets. This severely limits the amount of data that can be placed at the intersection of a row and a column in a relation (though perhaps some clever scheme could be worked out using multiple TXT records concatenated together to avoid this restriction). Third, the use of a single octet to represent a character within the DNS somewhat restricts the names of data that can be stored within the DNS. This may pose some problem for international alphabets and for the cases where a primary key value might be a binary value. Fourth, since the DNS has only very primitive pattern-match lookup facilities, the QSA will have to copy all the DS information from the DNS; no possibility of query translation exists. (On the plus side, this keeps the nameservers for an organization from being loaded too heavily.) Fifth, the DNS provides no access control, thus allowing any UIS access controls to be defeated by going directly to the underlying DS. Sixth, the DNS provides no standard means by which individual data records may be updated over the network. These restrictions are not enough to stand in the way of a prototype implementation.

Another existing hierarchical directory service implementation (and one that may serve better as an DS for the UIS) is the OSI X.500 Directory Services system. This system provides the same sort of hierarchy as does the DNS. Server redundancy, automatic updates, delegation of authority, and multiple query types are also supported, as in the DNS. While providing all of the same benefits as a DNS-based DS implementation, an X.500-based implementation also takes care of many of the problems with a DNS-based DS. The naming scheme is more flexible than that of the DNS. Arbitrary data can be stored within the DNS. Internationalization was considered when X.500 was developed, so international character strings can be represented. A flexible query interface is supported. Access controls and updates are also supported. Furthermore, if all data is stored in a reasonable manner within the X.500 database, interoperability with X.500 implementations is provided transparently.

The changes to the DNS DS model to support a X.500 DS are relatively straightforward. Of course, the same relations and the same basic attribute types within those relations are supported as in the DNS DS model. The organization-tag names given

as part of queries become X.500 organization names (i.e., **@c=US@o=University of Maryland**) rather than domain names. The attributes within relations would be those attributes associated with common classes within the X.500 directory. For example, the **person** relation might contain attributes for common name, surname, telephone number, or description, as defined in the "person" object class in X.521 [Int88b], along with the attributes associated with subclasses of persons, such as organizational persons or residential persons.

Query Service Agents

The Query Service Agents are easily the most important part of the whole UIS system. Since the QSAs decouple the actual information from its users, a multitude of sins in the DS implementation can be covered over with a good QSA implementation. The QSAs are responsible for generating (and holding) most of the UIS data that traverses the Internet, and thus small implementation efficiencies here can make the entire system perform much better. In this section, we discuss the features we want to see in a QSA, and then look at some implementation details.

First, a QSA must be able to process queries handed to it, and to return only the results of those queries across the network. By doing as much manipulation of the data near where the data actually resides, we can try to minimize network traffic and thus improve performance.

Second, the QSAs should, like the DS, be highly distributed. Having a large number of QSAs for an organization allows different queries to be handled by different QSAs, thus keeping any one QSA from becoming saturated by its workload. Distributing the QSAs also helps make the system more capable of dealing with system or network failures.

Third, the QSAs should be able to deal with a large number of queries, and be able to do so quickly and efficiently. Data structures and algorithms should be chosen with this constraint in mind. The amounts of data stored within a QSA could conceivably be quite large; the system should be engineered in such a way that the QSA does not drown in its own data.

Fourth, the QSAs should be flexible in terms of the queries they support, and in terms of the protocols used to move those queries (and their results) across the network. By being flexible, the QSAs can evolve with the Internet and with international standards, and continue to serve the Internet user community for many years to come.

Fifth, the QSAs should provide the same level of security and access control as does the underlying DS implementation. Providing more security is useless and unnecessary, as the raw data can always be accessed by going directly to the DS.

Sixth, while most entries in the UIS are likely to be text-based, it is possible that other entries will be based on some sort of binary format (i.e., G3 fax-encoded pho-

tographs, as described in [RKT89]). Such objects should be passed between parts of the UIS in a standard format such as ASN.1. When passing such data between QSAs, it is up to the sending side to translate binary objects from their local representations into their canonical representation on the network, and up to the receiving side either to ignore such objects or to translate them back into the appropriate local representations.

Let us now move on to some implementation issues. It seems reasonable to divide QSAs into two types: authoritative QSAs and caching QSAs. We say that a QSA is authoritative for an organization if that QSA gets its data for that organization directly from the DS for that organization. A caching QSA is one which gets its data only from other QSAs. There is no reason why an authoritative QSA cannot also be a caching QSA, but all cached information must be carefully segregated from any authoritative information within the QSA. By knowing that a QSA is authoritative (and that the answers coming from that QSA are based on authoritative information), we can know that the information provided is the best possible, rather than old information that may be outdated. Similarly, because information from somewhere other than an authoritative source may be forged, such information should be acknowledged to be slightly untrustworthy.

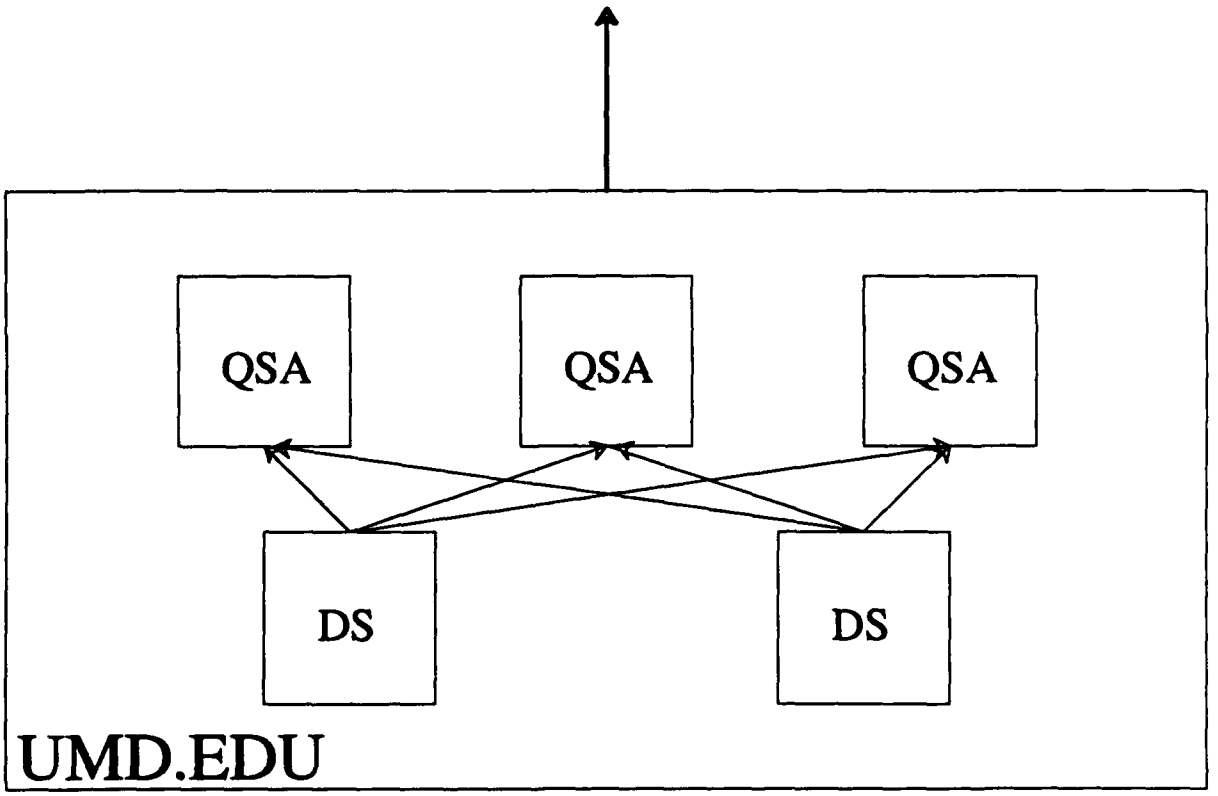
The data flow between QSAs and other parts of the UIS system is fairly straightforward. Authoritative QSAs fetch information from their DSes, according to whatever protocol is specified by the DSes. Authoritative QSAs must periodically refresh their information so that changes in the DS are tracked with some arbitrary degree of closeness. It is assumed that checking for new information is an inexpensive operation, and that checks can be performed frequently. The actual update is likely to be relatively expensive.

Unlike the authoritative QSAs, all other QSAs fetch information only from other QSAs, and only in response to queries. All user or application queries pass through a caching QSA, which breaks the queries down into pieces based on organization and relation, hands them to the QSAs for the organizations involved, collects answers, does any final joining, selection, and projection, and returns the answer to the entity that posed the query. QSAs for other organizations can be located through a variety of means; a new resource record type within the DNS or information stored within an X.500 directory could be used to locate QSAs on a per-organization basis. Caching QSAs should cache the information they get back from queries posed to other QSAs, along with the query performed, so that duplicate queries within a certain time-to-live (associated, perhaps, with each piece of cached information, and determined by the authoritative QSA) are not redone.[2] User interfaces and applications are of course free to do whatever they wish with the data they get back from the caching QSAs.

[2]The effectiveness of such a caching scheme, and possible alternative caching schemes, needs investigation once a prototype is up and in use.

Within an organization, QSAs should be deployed as follows. More than one authoritative QSA should be deployed, preferably on different hosts on different power supplies on different networks to maximize availability. These QSAs should be placed close (in a network bandwidth sense) to the machines providing DS service, and even on the same machine as an DS if possible. At least one caching QSA should be deployed, and one caching QSA should be deployed on any host which interacts frequently with the UIS.

Figure 2 shows an exploded view of the QSAs and the DSes for one organization. Since these pieces of the UIS are all heavily redundant, any one (or more) can fail, but leave the system still up and running.



Query User Agents

It is desirable to make the user and application interface to the UIS as simple as possible, so that applications that use the system may be provided on a large number of systems. Since so much of the query processing is dealt with by local caching QSAs in conjunction with remote authoritative QSAs, the UIS need only be able to pass its queries to a

caching QSA, and be able to retrieve and deal with responses. Let us present an extended example of a user's query, and demonstrate how such a query is dealt with by the QUA. Such an example may also help to clarify how some other parts of the UIS system work.

Let us consider a query (coming, say, from outside UMCP) that might be posed by someone who knows that he wants to send mail to the author of this document. All this person knows is the author's name; he does not even know where the author works. So, he fires up his QUA:

```
% quash
Welcome to the User Information System Shell!
>
```

The QUA uses some implementation-dependent method (i.e., reading from a file or broadcasting on the local network) to determine the location of one or more caching QSAs it can use.

The user gives a query to list all user alias entries known to the root of the UIS tree with the substring "miller" in them:

```
> select alias, name, name-organization-tag
from alias
where name ~= miller
and organization-tag = .
and type = user
```

The QUA takes this information, arranges it in accordance with the protocol being used between the QUA and its caching QSAs, and passes the query to the QSA for processing.

The caching QSA determines the location of the QSAs for the root of the UIS tree, and (since this query deals only with the organization "root", represented by a single dot in this example) hands the entire query off to one of the QSAs authoritative for the root. The authoritative QSA hands back some number of responses, formatted according to the QSA-QSA protocol:

Alias	Name	Name-Organization-Tag
Mark Miller	miller	stanford.edu
Steven D. Miller	Steve Miller	umiacs.umd.edu
Joe Miller	Joe	mit.edu
...	...	...

The caching QSA stores this response, along with the query that retrieved the information in the first place. The information is then handed back to the user's QUA via the QSA-QUA protocol.

On the basis of this information, the user does a more exact query:

```
> select mail-address
from user
where user = Steve Miller
and organization-tag = umiacs.umd.edu
```

This query gets handed to the local caching QSA for processing. The local QSA finds the QSAs for umiacs.umd.edu, hands the query off to them, gets an answer, and returns that to the QUA:

```
No information available.
```

The user tries the lookup again, checking the alias table first:

```
> select user.user, user.mail-address
from user, alias
where alias.alias = Steve Miller
and alias.name = user.user
and alias.organization-tag = umiacs.umd.edu
and user.organization = alias.name-organization-tag
```

This query is somewhat more complex. So far as the QUA is concerned, it is just like any other query, and it gets handed off to the local caching QSA as normal. The caching QSA has more work to do this time, as it has to split out the **alias** part of the query, do that, then use the organization information there to select against the **user** relation.

The alias query becomes:

```
select name, name-organization-tag
from alias
where organization-tag = umiacs.umd.edu
and alias = Steve Miller
```

The result handed back to the caching QSA is:

Name	Name-Organization-tag
steve	umiacs.umd.edu

This information is cached. We then do the rest of the query on the user relation:

```
select user, mail-address
from user
where user = steve
and organization-tag = umiacs.umd.edu
```

The result handed back to the caching QSA is:

User	Mail-address
steve	steve@umiacs.umd.edu

This result is cached, and then handed off to the QUA, which displays it.

Also note that if we couldn't find anything in the root query, and we could remember (eventually) that the author is at the University of Maryland, we could look within organization-tag `umd.edu` for alias entries containing the substring "miller", and use the results of that query to get closer to the answer. Also, if that didn't work, we could ask for sub-organizations of the University of Maryland (perhaps by queries on the `uis-catalog` relation), and take a look through their databases (either by pattern-match queries, or by asking for a list of all records of a particular type, or by asking for a list of all records in the organization) as we look for more clues.

Interfaces

In this section, we consider the functions provided by each part of the UIS, with the intent of indicating how all the UIS pieces fit together.

User Agent Services

The piece of the UIS which is least important to standardize is the user interface. There will be many user interfaces available; they may all look substantially different, but all will use the services provided by the QSAs to perform queries. The mandatory QUA services are:

- Some means by which the user can input a query.
- The ability to turn a query into its canonical form (SQL) within the system. This does not imply an understanding of SQL. For example, if the user inputs queries directly in SQL form, no translation is necessary, while if the user uses a QBE-style interface, the visual actions must be translated into SQL. Also note that not all of SQL need be supported; update and deletion operations happen so infrequently, and potentially complicate the implementation so much, that they could be handled by a separate interface if necessary.

- The ability to locate QSAs. In the simplest case, the QUA is simply given the network address of a single local QSA to use for forwarding queries.
- The ability to pass a query in SQL form to a QSA for processing. Again, in the simplest case, the QUA accepts SQL input and passes it to a single local QSA, which does the query processing.
- The ability to accept results in tabular form from a QSA. A possible "on-the-wire" representation of tabular results might include an indication of the number of rows and columns returned, an indication of the names associated with the columns, and a list of tuples.
- The ability to handle text-based attributes.
- Some way to turn the tabular results returned by the QSA into an information display presented to the user.

Optional features might include:

- The ability to break SQL down into parts to farm out to different QSAs for execution. This implies the ability to traverse the user information tree.
- The ability to perform query optimization.
- The ability to define views within the QUA, and to perform query modification to turn queries on views into queries on base relations.
- The ability to cache results locally within the QUA.
- The ability to use more than one protocol (i.e., Sun RPC [Sun88] with TCP transport versus the OSI Remote Operations Service with TP4 transport) to communicate with QSAs.
- The ability to pass authentication information to QSAs so that security checks can be performed.
- The ability to deal with arbitrary binary-format attributes (i.e., per ASN.1).

Query Service Agent Services

Since the QSA provides the bulk of the glue that holds the UIS together, it is easily the most complex piece of the system. Mandatory QSA functions include:

- The ability to accept SQL queries over the network from QUAs and other QSAs.

- The ability to translate an organization tag into a list of network addresses for that organization's QSAs.
- The ability to perform query optimization on incoming queries.
- The ability to partition (as part of query optimization, in a manner similar to the semijoin program) queries into pieces operating on different organization tags.
- The ability to farm those subqueries out to the QSAs for the organizations involved, to accept the results of those queries, and to join them together as needed to produce a final result.
- The ability to return the answers to queries from QUAs and other QSAs, formatted according to some high-level protocol.
- The ability to deal with text attributes.
- The ability either to load data from a DS into a database maintained by the QSA, or to translate SQL queries referring to data within the QSA's DS into queries to be handed to the DS for further processing. Note that the "null" network transfer is allowed, for QSAs that maintain their own DSes directly. A QSA built atop a true relational database might operate in this manner.
- The ability to detect when data within the DS has changed and to reload that data if necessary.
- The ability to operate in a multithreaded manner, so that more than one user query can be processed at a time. Similarly, a multithreaded implementation allows the system to spend the idle time while waiting for results from a remote QSA to process additional user queries, and vice versa.

Optional features include:

- The ability to communicate via more than one protocol, as with the QUA.
- The ability to accept authentication information, and to act on it to provide a secure system.
- The ability to provide more than one QSA for an organization tag. This feature is optional, but ***highly*** desirable.
- The ability to load or refer to data within more than one DS. This feature, too, is optional but highly desirable.

- The ability to cache the results of queries, and to delete items from the cache either on a periodic basis (driven by a time-to-live, for example) or when information in the cache is determined to be invalid.
- The ability to deal with arbitrary binary-format attributes (i.e., as per ASN.1).

Data Store Services

The services provided by the DS are fairly simple. Mandatory services include:

- The ability to transfer across the network the contents of the data store. This is used by QSAs that copy DS data into their own databases.
- As an alternative to the above (for DS implementations that provide sufficiently rich functionality), the ability to answer moderately complex queries involving partial matches on conjunctions and disjunctions of attributes and values. As stated in the QSA section above, it is possible for a QSA to translate a query into primitive operations on the DS, to perform those operations, and to translate the results back into relational form.
- The ability to maintain text-based attributes.
- The ability to allow some sort of updates. In the degenerate case, updates can be done by directly modifying the data loaded by the DSes (i.e., with a text editor). A more sophisticated scheme might allow updates to be done across the network.

Optional DS services include:

- A means by which the time-to-live for a particular piece of data may be specified, or a means by which the DS can tell a QSA that a piece of data has changed.
- The ability to operate in a replicated form, so that multiple DSes can feed one or more QSAs for an organization. This feature is optional but highly desirable.
- The ability to function in a multithreaded manner, so that multiple QSAs can update simultaneously from the same DS, or so that a DS can handle more than one QSA-generated query at a time.
- When operating in replicated form, the ability to have one DS update other DSes maintaining the same information. This can be done in master-slave form, or through some more complicated protocol.
- The ability to pass along (or interpret directly) authentication information, so that security may be provided.
- The ability to maintain arbitrary binary-format attributes (as per ASN.1).

Some Details on Query Execution

To demonstrate how the pieces of the system work together, consider the path a sample query takes as it moves through the system. The query we will consider is:

```
select user.user, user.mail-address
from user, friendly-organization
where user.fullname = "Steve Miller"
and user.organization-tag =
    friendly-organization.name-organization-tag
and friendly-organization.org-name = UMCP
and friendly-organization.organization-tag = root
```

This query is presented in SQL form to a QUA, which (we will assume here) reads the network address of a "local" QSA from a file. The QUA hands the query to the local QSA unmodified.

The local QSA (which we will call QSA *A*) accepts the query from the QUA. It then begins the process of query optimization and in particular begins the process of splitting the query into pieces. As the QSA builds its execution plan, it notices that it cannot query the QSAs for the `user` relation until it finds the `organization-tag` it should use for that relation. To do this, the part of the query dealing with the `friendly-organization` relation must be processed first.

QSA *A* splits the query into two queries:

```
select name-organization-tag
from friendly-organization
where organization-tag = root
and org-name = UMCP
```

and:

```
select user, mail-address
from user
where fullname = "Steve Miller"
and organization-tag = X
```

We assume that the results of the first query are stored in X.

In processing the first part of the query, QSA *A* sees that the `organization-tag` is `root`. It looks up the network address(es) of the QSAs for the root and, since there is no further optimization it can perform, it passes the query to one of those QSAs. If more than one root QSA is available, and it cannot connect to the first one in the list,

it works its way down the list until it has exhausted all the possibilities. In that case, it returns an error.

The root QSAs accept the query. Any optimization that can be done purely on a physical basis (i.e., the use of indexes) is performed, and the query is executed on the information provided by the DS. [3] The result (here, perhaps `umd.edu`) is then returned to QSA *A*.

Now that QSA *A* has the result from the first part of the query, it can execute the second part. QSA *A* determines the addresses of the `umd.edu` QSAs, and passes the second query to them as described above. The `umd.edu` QSA optimizes and executes the query, and hands the result back to QSA *A*. In effect, a distributed join has now been executed. QSA *A* has the results it needs, so it puts them on the wire back to the QUA. QSA *A* also caches the original query, its subqueries, and the results of all those queries, so that a repeated request (or a close variant) will result in less network traffic and computation than the original query.

The QUA receives the result, and pretty-prints it:

User	Mail-Address
steve	steve@umiacs.umd.edu

At this point, the processing of this query is finished.

4 Security

Security within a distributed system is a hard problem, and providing security within the UIS is no exception. The first temptation when dealing with security issues is to ignore them, and state that information that should not be made freely available to all should not be placed within the UIS. That interpretation may not even be wrong; historically, though, people have wanted to store sensitive information within distributed databases, including the DNS. Thus, while we do not investigate the security issue in detail, we do provide a number of observations as to how security might be built into the UIS system.

Any UIS security scheme, in order to be effective, must rely upon the security of the information within DSes. If this information has no security controls, UIS security is merely security through obscurity, which is no security at all. For example, consider a "secure" UIS implementation built atop the DNS. While queries at the UIS level might be fully authenticated (via a public-key authentication system such as that provided by Kerberos [MNSS89]), a knowledgeable cracker can always do the same DNS queries

[3] We will assume in this case that the root QSA involved has copied its information from a DS for the root, so that translation into, say, X.500 queries against the DS is not necessary.

or zone transfers that the QSAs themselves do, and in this way access any information within the UIS.

Given the above constraint, there are two ways in which security can be added to the UIS model. The first (and, perhaps, best) scheme is possible only where a fully-developed network authentication model exists within the DS. In this case, access control information (i.e., passwords, host identifiers, encryption keys, and so forth) is passed from the QSA to its DS, and the information is passed from the QSA back to the requestor if and only if the DS would allow the access directly. Such a scheme might be possible to use in conjunction with an X.500-based DS. As an alternative, access control information for an organization could be placed in the part of the `uis-catalog` relation associated with that organization. All QSAs for that organization would then have to interpret this information in some special way, and use that information to restrict access to UIS data within that organization. Because the QSA and the DS for an organization can be under the exclusive control of that organization, and because the manner in which they transfer and store information can be changed in any way desired (so long as the QSA-QSA protocol remains the same), this second scheme stands at least a chance at being secure.

As a final observation, we state that the protocols used to transfer information between QSAs should be built with authentication in mind. Such a protocol should provide a place for authentication information to be passed, with the default authentication being null authentication. Then authentication can be added at a later time without disrupting existing implementations.

5 Other User Information Databases

A number of user information databases exist today. These servers run the gamut between lightly-used databases storing little information, to heavily-used databases storing a great deal of information. In this section we compare the UIS presented here to some typical and popular user information databases already in existence.

WHOIS

The WHOIS service is a popular database managed by the DDN Network Information Center. In addition to storing commonly-requested information about Internet users, WHOIS stores information on hosts, networks, and domains within the Internet system. While more than one organization may maintain a WHOIS service – the University of Maryland has one, for instance – WHOIS is not a distributed database in the sense that there is no way for one WHOIS server to chain or refer queries to another server. WHOIS provides a global search capability, but only for the users whose information is managed directly by the NIC. To change the information given for a user, the user

must ask the NIC personnel to make the change; this process often takes more than a day or so before the new information appears.

The functionality provided by our UIS can be used to emulate WHOIS. If the NIC personnel maintain the part of the `alias` relation with the root organization-tag, global search at the root is still allowed, even without the intensive network activity involved in walking the UIS tree. Maintaining this information would be easier than maintaining full information for all registered users, as only the aliases would need to be maintained, and these aliases would change less frequently than does the full user information. By modifying the NIC WHOIS server to communicate with the UIS, applications and users who rely on WHOIS may continue to use it, with results that are likely to be better than normal, and which will be no worse than they are currently.

FINGER

Another well-known user information database is the FINGER service [Har77]. FINGER is an extremely simple service, and as such is widely implemented. FINGER information is distributed, in that FINGER information is stored on many machines throughout the Internet; FINGER is not a distributed database for the same reasons that WHOIS is not distributed. Unlike WHOIS, FINGER provides no way to perform even the illusion of a global search, as it relies on the user to choose the hosts to be searched for information. The information returned by FINGER is unstructured, and varies widely from site to site.

The UIS can provide all the information provided by FINGER. To do so, a process continues to advertise FINGER service on the FINGER port, but this FINGER pseudo-server is modified to perform some straightforward query on the UIS. This information is returned to those who request it. Again, users who currently rely on FINGER may continue to use FINGER, with better results than before.

X.500

The X.500 Directory Services provides user information storage facilities that are far superior to those provided by WHOIS or FINGER [RS89]. The X.500 Directory Information Tree is a distributed database, and supports global search by walking the DIT. X.500 also provides a number of relatively sophisticated authentication features. Another important feature of X.500 is that it will be a worldwide standard.

The fundamental differences between X.500 and the UIS are slight. In an ideal world, the UIS would be little more than a sophisticated interface build atop X.500. In fact, interoperability with X.500 can be trivially achieved by implementing the UIS atop X.500. We feel that the additional abstraction of the relational model, along with some of the UIS features designed to reduce network traffic (including relational query

optimization), and along with the additional protocol and DS independence offered in the UIS model, make the UIS worth implementing even in an X.500-based world.

Multidatabases

A multidatabase is a distributed relational database built from a number of smaller databases within a network. Multidatabases emphasize the autonomy of their component databases. As such, they provide features for referencing data within more than one component database, for doing conversions between possibly varying external schemas, and for doing conversions between possibly varying data type representations across multiple databases.

The multidatabase features could certainly be used to build a distributed UIS similar to what is presented here. However, such a database would be overkill for our needs. Multidatabases uphold (to a greater or lesser extent) the traditional database requirements of integrity, concurrency control, and recovery. As we have already seen, such features are not required within the UIS. Since most data types are just character strings, and since non-text data types are encoded in some standard format, the data conversion features of a multidatabase are not really needed.

Similarly, the UIS presents only a very loose external schema, and one that is deliberately allowed to vary across organizations. In fact, because the basic UIS database is so simple both in terms of the different relations it supports and in terms of the data types of attributes in those relations, the approach of using QSAs to present a consistent interface makes more sense than providing SQL extensions to perform such conversions.

Most multidatabases provide SQL extensions to allow the referencing of specific relations on specific hosts. The UIS takes the stand that placing the name of specific QSA servers within the relations will work. This aids in presenting the fiction of the UIS as a single entity, and also avoids the work involved in extending SQL. One can argue that either approach is the better one.

6 Conclusion

In this paper, we have seen how the UIS allows the world of user information to be viewed through relational-colored glasses. We have seen how the decomposition of the UIS into distributed Data Stores, Query Service Agents, and Query User Agents can provide a great deal of functionality and protocol independence. We have seen glimpses of how security can be worked into the UIS model, and we have seen that the UIS is at least as powerful as the user information databases in existence today. The usefulness of a UIS system seems, from the vantage point of a design without an implementation, to be worth the effort. Further implementation experience will indicate whether or not the UIS can meet efficiently the user information needs of the Internet community.

References

[GSN88] James Gettys, Robert W. Scheifler, and Ron Newman. *Xlib - C Language X Interface, X Window System, X Version 11, Release 3*, 1988.

[Har77] K. Harrenstein. Name/finger (rfc 742). Technical report, DDN Network Information Center, SRI International, 1977.

[HSF85] K. Harrenstein, M. Stahl, and E. Feinler. Nicname/whois (rfc 954). Technical report, DDN Network Information Center, SRI International, 1985.

[Int88a] International Organization for Standardization and International Electrotechnical Committee. *Information Processing Systems, Open Systems Interconnection, The Directory, Overview of Concepts, Models, and Service*, 1988.

[Int88b] International Organization for Standardization and International Electrotechnical Committee. *Draft Recommendation X.521, The Directory, Selected Object Classes (Final Version)*, 1988.

[LMR89] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. ACM Computing Surveys, 1990 (to appear), 1989.

[McN88] J. McNamara. *Technical Aspects of Data Communications.* Digital Press, 1988.

[MNSS89] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Project Athena Technical Plan, Section E.2.1: Kerberos Authentication and Authorization System. Technical report, MIT Project Athena, 1989.

[Moc87a] Paul Mockapetris. Domain names - concepts and facilities. Technical report, DDN Network Information Center, SRI International, 1987.

[Moc87b] Paul Mockapetris. Domain names - implementation and specification. Technical report, DDN Network Information Center, SRI International, 1987.

[Moc89] Paul Mockapetris. Dns encoding of network names and other types. Technical report, DDN Network Information Center, SRI International, 1989.

[Pos81] Jon Postel. Transmission control protocol (rfc 791). Technical report, DDN Network Information Center, SRI International, 1981.

[Qua86] John Quarterman. Notable computer networks. *Communications of the ACM*, 29(10), October 1986.

[RKT89] Colin J. Robbins, Stephen E. Kille, and Alan Turland. *The ISO Development Environment Users Manual, Volume 5: QUIPU*, 1989.

[Ros89] Marshall T. Rose. *The Open Book.* Prentice Hall, 1989.

[RS89] Marshall T. Rose and Martin L. Schoffstall. An introduction to a nysernet white pages pilot project. 1989.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, May 1988.

[Sol89] K. Sollins. A plan for internet directory services. Technical report, DDN Network Information Center, SRI International, 1989.

[Sun88] Sun Microsystems, Inc. *Remote Procedure Calls: Protocol Specification*, 4.0 edition, 1988.