A Name Server Database

David W. Riggle

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

May 1, 1984

ABSTRACT

A name server maps names to network addresses. It needs to maintain only a small amount of data to accomplish this feat. A special purpose database of simple design can provide much better performance than a general purpose database. This paper describes such an implementation.

A Name Server Database

David W. Riggle

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

May 1, 1984

1. Introduction†

This paper describes the design and implementation of a name server database from conception to completion. Relevant topics such as name server design and function are discussed in minor detail when appropriate.

2. The Purpose of a Name Server

A name server simplifies the task of naming objects such as mailboxes and machine addresses in a distributed environment. It must provide a consistent and unique name space in which objects are defined. A large network such as the ARPAnet has many hundreds of host machines, tens or hundreds of thousands of users, and several esoteric protocols and data formats. Providing a standard naming convention in this complicated and changeable environment is one major goal of a name server.

Just as important, however, is the need to distribute the work of modifying and maintaining the database among the appropriate administrative units. At present, the ARPA Network Information Center at SRI is solely responsible for updating and distributing copies of a global table of the hosts in the ARPA Internet. The size of this table and especially the frequency of updates are near the limit of manageability [Mock2 1983].

The SRI-NIC host table does not contain names of mailboxes or of other objects smaller than single machines. The handling of such vast amounts of data necessitates abandoning the centralized host table in favor of a distributed database in which local administrators have complete control over their part of the environment. Changes can more quickly and easily be made to these small, local subsets of the name space. Processes could even "rendezvous" over local networks or post messages on the name server in (perhaps) real time. It is the name server's job to provide access to the distributed data in the name space in such a way that users need never know they are

† This work was partly sponsored by the Defense Advanced Reasearch Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under Contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the US Government.

transcending administrative and physical boundaries.

The concept of the Internet Domain Name Space and the inspiration for the Berkeley Name Server Project (BINDS) come from P. Mockapetris *et al.* at USC-ISI [Mock1, Mock2, Mock3, Mock4 1983]. Their design is likely to become a standard on the ARPA and MILNET networks. They envision the Domain Name Space as a tree (see Fig. 1). Each node and leaf on the tree contain information called Resource Records (RRs). Nodes could represent hosts or mailboxes, for example, and resource records at those locations could be network addresses, phone numbers, mailing lists, or anything else. In the Berkeley implementation, which also supports user updates, "finger" information maintained by individual users could easily be stored in the name space.

Each node in the Domain Name Space has a unique name given by its path from the root or "null" domain name in the tree. For example, one name might be "ernie.berkeley.arpa." The name server's job, in essence, is to take any name in the Domain Name Space and return whatever information is associated with it. How it accomplishes this selfless task in the face of political and geophysical obstacles is more properly left to another paper [Zhou 1984].

Sample Domain Name Space

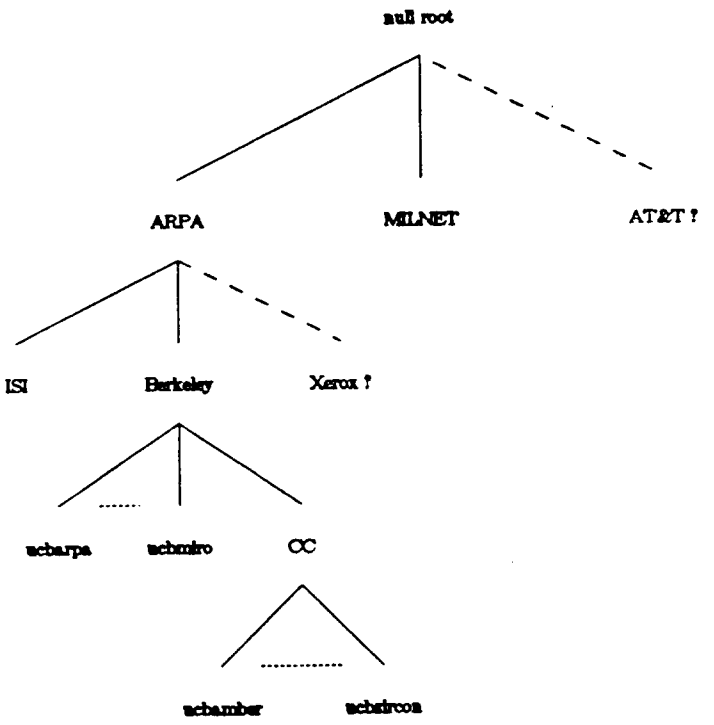

Figure 1.

3. The ISI Domain Name Space

I will now discuss peculiarities of the ISI name space and how they relate to the implementation of an individual name server and its private database.

The most striking feature of the ISI design is the ability to delegate authority to arbitrarily small portions of the Domain Name Space. In the extreme, each machine could be responsible for its own information and no other. Most likely a single name server would have detailed knowledge of a few dozen or so machines and a few thousand users. By database standards, individual name servers have very little information to store. For this reason and for efficiency, a name server's database could reside in core.

A name server may be responsible for more than one administrative partition of the name space (called a zone of authority). Its database therefore must be able to provide logical partitioning of the data into separately managed tree structures. No sharing of data between these zones is encouraged [Mock3 1983]. For refreshing and caching purposes, zones may be distributed wholly to other name servers upon request. One zone, where cached information from various sources is stored, is designated the "cache tree."

The database is initialized by reading domain names and their associated resource records from a "master file." Updates to the database are made by hand to this "master file." The proposed design is for a read only database much like a telephone book. No user updates are allowed. Crash recovery is simple enough: the database is reloaded from the "master file" when the system comes back up.

4. Definition of a Resource Record and Typical Queries

A Resource Record (RR) is the basic element of data storage in the ISI Domain Name Space. Network addresses, mailboxes, and so on, are all stored in RRs. There are five fields in an RR. They are Domain Name, TTL, Class, Type, and Data. The Class and Type fields are small integers describing the Internet class (e.g., CSnet or ARPAnet) and the type of the resource record (e.g. Mailbox, Name Server, and so on). The TTL (Time To Live) field is a time-out counter. Domain Names and Data are character strings of up to a maximum length (256 bytes).† A typical RR might be as follows:

Domain Name	TTL	Class	Type	Data
sailors.berkeley.arpa	0	CS	NS	20.0.0.4

Typical queries are of three kinds. Standard queries specify the name, class, and type of the desired RR. Either or both of class and type may be wildcards. The wildcard ("*") matches all patterns. Several RRs may be returned from the database as a result of a single query. Different RRs may even have the same name, class, and type fields but different data.

† The symbolic constants used in this paper to represent integer constants stored in the database do not necessarily correspond to real classes and types. For an accurate description of the facilities provided by the Berkeley Name Server and for more information on timeouts and "Data" see [Zhou 84] and [Painter 84].

Inverse queries are the second major kind. Given the data and possibly a restricting class and/or type, the database must return all RRs with that data which meet the constraints on class and type. Inverse queries are useful for finding machine names from addresses and could even be used to retrieve user names from office numbers, for example.

Completion queries are the third kind. A partial name may be given to the name server in a query. By climbing up the Domain Name Space Tree and adding suffixes to the partial name, a full path name may be constructed. The actual name completion algorithm requires rather detailed knowledge of the Domain Name Space Tree and the names actually stored in the database, and it therefore may be most appropriate for the name server to administer the searching of the database during completion queries.

5. The Database Implementation Proposed by ISI

The recommended database implementation vouchsafed by ISI is a tree structured design corresponding to the Domain Name Space Tree. Separate trees are constructed for each zone of authority, including the cache tree. RRs are attached to nodes and leaves in the trees. The ISI designers suggest tree traversal as the name look-up procedure and hierarchical name storage so that full path names need not be stored in each node [Mock3 1983].

These zone-trees must be built dynamically since zones exist merely to delimit administrative units and can change with time. A standard query would be answered by tree searching the appropriate zone-tree in the database. This zone-tree is chosen because its root is closest to the desired name in the Domain Name Space. A linear search of all zone-tree roots determines which zone is the appropriate one to look into. At each node of the tree there are pointers to descendant nodes representing subtrees in the name space. Each descendant's name must be compared to the remaining parts of the hierarchical name we are looking for. If there are many children per node, as is most likely (consider the number of users per machine or the number of machines per university), we could spend a lot of time comparing names before we found the one we were looking for.

After the desired domain name has been found by traversing the zone-tree, the node corresponding to that name must be searched for RRs matching the Class and Type fields in the query. No mention is made in [Mock3 1983] of the storage structure or search mechanism used within nodes, but the simplest would be a linked list of resource records containing Class, Type, TTL, and Data fields (of maximum length) for each node. These records could be easily re-used if they were the same size, but allocating the maximum size for each record is rather wasteful of space, especially when most data is about as long as "20.0.0.4." Also, a linear search of all the RRs within a node to find the one you want is not very time efficient, but that is about all you can do with a simple linked list.

Although the ISI database is read only in the sense that no update queries are allowed -- all changes must be made by hand to the "master file" which is periodically reloaded into the database -- it is not truly static. Some form of memory management must be used to recover old space when whole zones are reloaded. No provision for incremental zone refreshes is included in the ISI description of the name server.

Nevertheless, incremental updates are a central issue in the Berkeley name server project (see [Zhou 1984] and [Painter 1984]).

No crash recovery mechanism is required in the ISI database since a permanent copy of the data can always be found in the "master file."

The ISI design proposes concurrent access to the database so that the maintenance operations (refreshing zones) do not interfere with normal query processing. However, the overhead of concurrency control on some systems (namely UNIX†) could significantly slow down the database operations during normal queries. I will discuss this problem a little later. Another drawback of the ISI database implementation is probably its main virtue: the tree structured encoding of the name space. Interesting data such as user names will invariably be at the bottom of the tree, requiring a lengthy search. If the name space is broken up into many zones to try and reduce the height of the many search trees, the result is not necessarily better since the root name of each zone must be compared to the name being searched for at the beginning of the retrieval to find the appropriate zone in which to search for that name.

To some extent, then, performance depends on the administrative boundaries of the name space: how many zones there are, how many nodes per zone, and how many children per node.

6. Ideas on a Name Server Database

Despite the space cost of storing the full path name with each node in the Domain Name Space Tree, doing so essentially turns the hierarchical tree structure into a flat name space. Nodes can then be found in a single look-up (i.e. by hashing) instead of being found by a longer, although more natural, tree search. Within each node a considerable amount of data sharing can be accomplished with only a little effort. Every RR will have the same Domain Name field and can share the same name storage. Those of the same class can share that datum as well. Those of the same class and type would share all significant data. If many similar RRs (such as members of a mailing list) exist under a domain name, the storage savings due to shared data could be significant compared to a scheme that stored the fields of each RR independently.

Space requirements, one of the motivating reasons for breaking up the SRI-NIC host table in the first place [Mock2 1983], should not be forgotten in the database implementation. Allocating resource records of the maximum size to hold data which are only a few bytes long is a luxury that only a few computers and even fewer personal workstations can afford. Complete re-use of free storage with as little overhead as possible would be very desirable in a system which is supposed to stay up for a very long time and handle many updates or refreshes. Running out of core because of poor memory management or a placebo free() routine is not acceptable.

ISI proposes that the database should be able to handle concurrent access and modification (for refreshes). A system of two locks is suggested to implement exclusive and shared access. Unfortunately, in 4.2. BSD UNIX there is no shared data memory.

† UNIX is a trademark of Bell Laboratories.

A database program providing concurrent access and modification would either have to be entirely disk based, or else it would have to act like a "database server." In a disk based system, shared file access can be used as a substitute for shared memory access. Given shared *something,* concurrency can be achieved with semaphore locks on various portions of the database, allowing different database processes to read and write simultaneously. The database would become just a file, and many database programs, started up by the name server to handle individual queries, would use this file at the same time.

With a "database server," queries would be sent as messages to a single database program, which would have to handle its own multi-programming, by perhaps forking and piping, and responses would be sent back as messages. The only advantages to this scheme are that the complexity of the name server is reduced (each one does not need a database program) and the database need not be on disk. However, the message overhead needed to implement concurrency is a terrible drain on performance. It seems we cannot win either way, and the immense overhead puts concurrent access out of the question, in my opinion. A name server using a database without concurrent access would have to answer all queries and maintenance operations serially.

Along the same lines, it is apparent that a name server cannot afford to use a general purpose database system for retrieving queries. First of all, it would be overkill. The name server requires only a few kinds of retrieval and update operations. Mechanisms for joins, selects, and even elaborate crash recovery are superfluous. Secondly, large database systems tend to be huge, heavily CPU bound, and terribly slow (try running INGRES in Berkeley UNIX). The trend toward smaller workstations in a distributed environment precludes the use of these dinosaurs. Especially with respect to the CPU, distributed programs need to have a "social conscience." That is, they must be made a little less ambitious and a little more economical. Considering the network and IPC delays among distributed systems, database access times are probably not all that critical. Frugal CPU usage, however, will always be critical.

7. Implementation of the Berkeley Name Server Database

Two of my goals for the database were simplicity and speed. Almost the first design decision I made was to keep the database entirely in core. Luckily the amount of data needing to be stored is small enough to make an in-core database feasible (see section 13, "Overall Performance," for some examples). If page faults in a virtual memory environment are considered normal disk accesses, a low level disk based system with caching might conceivably locate data with fewer disk accesses than my scheme, but the CPU requirements and the added complexity would not be worth the questionable time savings.

An in-core (or virtual-core) database which is expected to run for a very long time with many updates and deletions needs to have an efficient way to reclaim storage. The pitfalls of memory fragmentation are a very real danger since data can be quite variable in length and may be freed in any order. Allocating maximum-size buffers is too wasteful of space, and memory compaction schemes to coalesce empty regions are too expensive. The solution I chose is to have only one size of data buffer

in the whole database. Everything must fit into these buffers, making complete re-use of storage very easy, i/o and memory operations uniform, and debugging easier. Data that is longer than one full buffer can be stored in several buffers which are then linked together. A hierarchical structure can be built from the data buffers by means of two pointers provided in each.

Buffers are allocated in blocks of a few hundred or thousand buffers. Within these blocks the individual buffers are referenced like records in an array. Blocks are dynamically allocated and linked together as the need for more buffers arises. Unused buffers are garbage collected periodically. Garbage collection is particularly easy with this scheme because the buffers can be readily found in memory, and they are all identical.

In summary, this scheme of single sized buffers provides simple storage management, 100% re-use of space, and arbitrarily long data strings. The cost of this flexibility is a small amount of internal fragmentation (how much depends on the average length of the data). If the buffer size is longer than most data, then memory is being wasted. Also, the pointer overhead per buffer is significant. Currently 23 out of 32 bytes of a buffer are used for data. In other words, 28% of a buffer's space is overhead. However, all but one byte of the overhead is used in representing the hierarchical structure of a resource record and achieving data sharing. In this light, the space overhead does not seem too outlandish.

Another problem arises if a linked list is made from buffers widely separated in memory. Traversing this list could result in many page faults. Fortunately, this problem should not occur very often. Initially, buffers are allocated sequentially so that lists built from fresh buffers will be close together in memory. As discussed under Garbage Collection, buffers are grouped on the free list as closely as they can be in memory without resorting to compaction. Therefore, lists built from used buffers will also tend to be very close together.

Alternative schemes besides my one-size-fits-all data buffers are possible. Any variable sized buffer implementation would have to face external fragmentation and either deal with memory compaction or suffer a very sparsely populated address space. Many page faults are likely in the second case. As noted before, a maximum sized buffer scheme is flagrantly wasteful of space, although such a scheme would not have to worry about extra pointer traversals to retrieve data that is longer than one of my buffers.

7.1. The Hash Table and Hierarchical Buffer Structure

I use a hash table to map full domain names (a flat name space) into a hierarchical class and type structure built from the single sized data buffers. Hash table collisions are resolved by chaining. Associated with each domain name is a zone name and a list of classes. Each class has a list of types. Each type has a list of data. In this way storage is shared by many RRs. For example, if two RRs have the same name, class, and type fields, only the TTL and data fields of the second RR need to be stored (see Fig. 2).

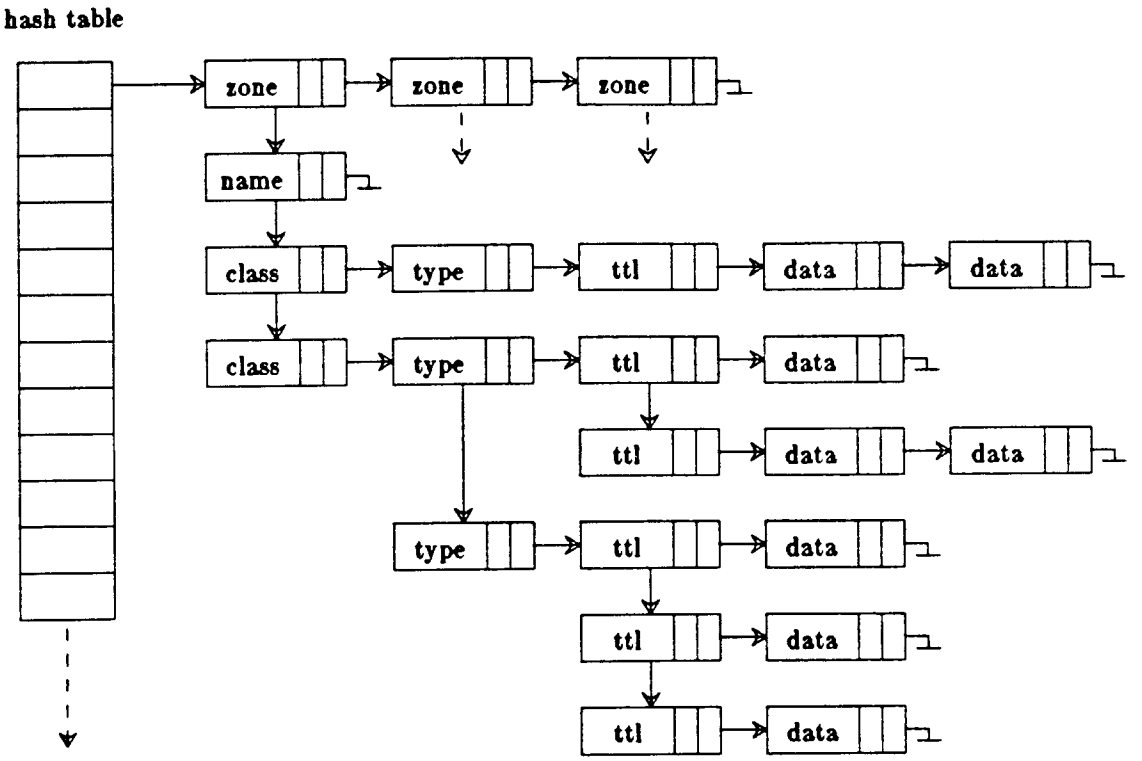

Figure 2.

Both the domain name and the data fields can be arbitrarily long. For efficiency's sake and because there was no need for more (a 23 digit integer is a *very* large "small integer"), zone names, classes, types, and TTLs are limited in size to one data buffer. A standard query would be answered as follows: the domain name is hashed and the collision chain examined until a match for both domain name and zone is found; the desired class is found by a linear search through the class list. From that class the search continues down its type list for the desired type. After the type is located, the data fields below it are returned to the name server.

The interface between the name server and the database is really quite simple. The name server specifies the zone, name, class, and type fields, and the database returns the RRs found in an array of structures. Several of the database primitives used by the name server are given below:

```
/* returns a count of the RRs found.  Zone, class, and type may be "*" */

getdata(zone, name, class, type, result)
char zone[], name[], class[], type[];
struct RR result[];

/* adds new data to the database.  It is an error if it already exists */

adddata(zone, name, class, type, ttl, data)
char zone[], name[], class[], type[], ttl[], data[];

/* changes olddata to newdata in one operation.  It is an error if olddata
does not exist or newdata already exists */

changedata(zone, name, class, type, olddata, newttl, newdata)
char zone[], name[], class[], type[], olddata[], newttl[], newdata[];

/* delete data.  It is an error if the data does not exist */

deletedata(zone, name, class, type, data)
char zone[], name[], class[], type[], data[];

/* the name server's view of a resource record.  DATASIZE = 23, DATAMAX = 256 */

struct RR {
        char zone[DATASIZE];
        char name[DATAMAX];
        char class[DATASIZE];
        char type[DATASIZE];
        char ttl[DATASIZE];
        char data[DATAMAX];
}
```

7.2. Parallel Disk Version of the Database

While an in-core database is simpler to program and probably runs faster than a disk based system where no data are cached in memory, it has one major drawback for a name server which allows dynamic updates to the database: everything is volatile. Modifications not saved to disk will be lost when the system crashes. Updates could be written to the "master file" in human readable form, but that would be inefficient. The whole file might have to be read and parsed to make just one deletion.

Instead I decided to maintain a parallel version of the database on disk. It closely resembles the internal structure of my in-core database. Part of the disk file is a picture of the hash table, and the rest is divided into fixed sized buffers. Each disk buffer has a number which corresponds to a data buffer's address in memory. Every time a data buffer is modified, its disk buffer is similarly modified. Fflush() is called

before the modifying routine returns, but no more assurance of the data actually being written to disk is provided.

Since updates are always performed by switching a single pointer from the old data buffers to the new data buffers, the database should be in a consistent state when the system comes back up after a crash. If not, it is up to the consistency checker to reconstruct the database. Providing more crash recovery mechanisms -- such as forcing updates all the way to disk, writing in parallel to two disk drives, and logging changes on tape -- would reduce the chance of losing information, but the performance penalty would be astronomical (see INGRES again). Elaborate crash recovery is provided in commercial databases, but for our purposes, a small measure of crash worthiness should be sufficient. Data stored in a name server should not be so critical that no update can ever be lost.

The disk file is used to boot the database when it first comes up. For initializations I have a load program to read the "master file" and insert the RRs into the database.

The cost of making updates to the database is greater than the cost of making queries. No disk access (other than page faulting) is needed to answer a query. In addition to modifying the disk buffers when an update occurs, the inverse query table must be modified if new data has been added. Nevertheless, updates are fast enough to encourage use of the name server as a rendezvous point for cooperating processes. Such use would require frequent updates to the database.

7.3. Performance and Analysis

For the average query, the only computations which have to be performed are the hash value calculation and a few string comparisons. Following pointers which cause page faults will no doubt be the greatest source of delay. The hash table collision chains are a large contributor of page faults. Since the names which collide are not added to the database in any particular order, moving along the collision chain will be like jumping randomly from one part of memory to another. Minimizing the hash table load factor is therefore very important.

For a given name in the database, most of the buffers connected to it are allocated at the same time. Nevertheless, if RRs for the same domain name are added to the database at widely separated intervals, the hierarchical structure built will be all over memory, and many page faults are possible for a single lookup. Hopefully, if such a name happens to be in a popular query, the operating system will be able to keep the necessary pages in memory to eliminate excessive page faulting. In any case, do not despair; for there is a way of completely rejuvenating the database by re-clustering the data buffers to produce a minimal number of page faults (see below).

It is interesting to note that performance is independent of the size of zones, the number of zones, and the administrative partitioning of the Domain Name Space.

7.4. Database Maintenance

Several configuration constants affect the performance of the database. The most important is the hash table size. It is currently the prime number 1129. It should be adjusted to keep the load factor low (the hash table size should be comparable to the number of distinct domain names in the database). The next most important constant

is the number of data bytes per buffer. It is currently 23. This constant should be comparable to the average name and data length. Changing either of these constants would require dumping the database to an intermediate file (currently the human readable "master file"). The changes could then be made to the program, the database recompiled, and the data reloaded.

To facilitate such modifications, two routines are provided. They are "dumpdb()" and "loaddb()." The first creates a "master file" for each zone in the database. The second takes a number of "master files" and loads each one into its appropriate zone. If the database is to be reloaded from scratch, it is necessary to zero the database's disk file before reloading it (done with "cp /dev/null disk_file" in UNIX).

Even if no modification is made to the database program, dumping and reloading the database have the interesting side effect of grouping all RRs of the same name together and hence clustering them as close as possible in memory, reducing page faults. I do not anticipate the database ever getting "out of tune" as time goes by, but the dump and reload routines seem fast enough to be run as part of an initialization script if it were thought necessary.

8. Garbage Collection

A major goal of the Berkeley Name Server Project was to provide updating facilities to the name server database. Deleting and modifying resource records necessarily leaves some data buffers unused and unreachable. How to identify and collect these data buffers so they can be re-used is a problem of no small importance. We would like the method to incur as little overhead as possible. Because of the unique structure of this database, the job is particularly suited to a garbage collector. But first, before I describe my implementation, I will give a brief description of the various ways by which it would be possible to collect and re-use memory in this database.

8.1. Explicitly Inserting Buffers on The Free List

The simplest way to re-use data buffers in the heap is explicitly to link them together on a free list at the time of deletion. This housekeeping takes a small amount of time during each update. Explicitly recovering discarded buffers is undoubtedly the least costly way of re-using space since no time needs to be spent to locate the unused buffers. There are some major disadvantages, however. Since the free list is a unique structure (i.e. it could not be recreated from available information if the system were to crash), it must be stored somehow in the parallel disk version of the database. Updates to the free list (buffer allocation or de-allocation) would have to be written to the disk. What is worse, an untimely crash could leave some buffers in limbo (neither on the free list nor in use), and they would be lost forever.

Another disadvantage is that buffers are inserted on the free list in essentially random order. A linked list built from buffers on the free list could be scattered all over memory. A traversal of this list could be very slow because of the many page faults encountered. Sorting the free list would be a costly exercise.

Thus, the simplest scheme is not the safest nor the best, and it requires too much disk i/o.

8.2. Reference Counting

The next scheme is reference counting. Each data buffer would keep a count of the number of pointers pointing to it. When the count reached zero, the buffer would consider itself free. As in the first scheme, the overhead is spread out over every deletion and insertion. The obvious trouble with simple reference counting is its inability to detect cycles of unused buffers. Although my database is not supposed to have any cycles in it, who knows what the structure might look like after a bizarre crash? If the system crashed as the reference counts were being decremented, some of the data buffers could be lost forever. This scheme also suffers from the liability of recovering the buffers in a random order. Also, the reference counts cannot be recreated from other available information, and so disk i/o is necessary for every update to change the reference counts stored on disk.

8.3. Garbage Collecting

By far the best solution for recovering unused space under our circumstances is to use a garbage collector. Deleted buffers are just forgotten, and atomic updates are accomplished by allocating new space and then simply switching pointers. After a sizable number of buffers has been deleted, it is worth our while to collect them. Since the database resides in core, the smaller the working set required in memory the better. No housekeeping overhead is incurred during routine updates and deletions. Every once in a while, however, the database must stop its work completely and run the garbage collector. It is this periodic drain on performance which antagonists decry. Nevertheless, I claim the price is very small and the best bargain we have seen so far.

Since the free list can be rebuilt by the garbage collector from the available information (we know which buffers are reachable from the hash table and which are not), no information about the free list needs to be written out to disk. Free buffers can be linked together and managed in the memory alone -- a remarkable result. What is more, after the sweep operation of the garbage collector, the free buffers are in sorted order. A string of re-allocated buffers will be as close together in memory as possible without running an expensive compaction algorithm. If the system were to crash at any time, the garbage collector could reconstruct the free list easily enough. No data buffers can be permanently lost.

If I were to implement a memory compaction scheme in the database during garbage collection, it would be necessary to update to disk all the data buffers moved around in memory. This extra overhead would reduce to a snail's pace the current lightning speed of the garbage collector. I opt for simplicity and speed as usual.

8.4. The Consistency Checker

By incorporating a few checks into the mark phase of the garbage collector, we get a very useful consistency checker almost for free. The hierarchical structure of the database is rather intricate, and any wrench thrown into the works by a system crash could be fatal. One of my design goals was to make the database robust enough to recover from any amount of internal damage. The simplicity of the internal structure (i.e., only one type of data: the buffer) brought this goal within reach. The only error deadly enough to crash the database is an out-of-range pointer. The consistency

checker detects and breaks cycles, it sets to NULL any out-of-range pointers, and it deletes headers to nonexistent data. Garbage collection provides the perfect opportunity to perform these checks since it is done at boot time as well as occasionally throughout the execution of the program.

I have tested the consistency checker on numerous occasions with the aid of a small program which lets me write over the database's disk file. I have even booted the database from random object files, which I tell the database are its disk files. In every case the consistency checker is able to reconstruct a legal database, albeit one with much information lost.

The garbage collector again earned its salt when I implemented inverse queries. I did not want to store the inverse mapping tables on disk because updates would require too much disk i/o. I put a few function calls in my garbage collector, and now, when the database first comes up, the inverse query tables are built very economically in memory.

The garbage collector has proven a boon because of its low disk overhead, its guarantee to find and re-order every unused buffer, and its easy modification to perform new tasks quite different from its original charter. I cannot imagine a more cost effective piece of code.

9. Zones and Zone Transfer

Zones are useful administrative subdivisions of the Domain Name Space. They can be dumped, loaded, or transferred to other name servers independently. Whole zones are only transferred to other name servers at initialization time in the Berkeley design because incremental updates are used to acquaint the other name servers of dynamic changes. Zones are implemented in my database as tags to the domain names. No hierarchical relationship between zones and names is maintained. A zone transfer requires a linear search of the whole database to be performed. I will attempt to justify my implementation.

In the first place, zone transfers should be done very infrequently (only at initialization time) although it is possible for name servers from other parts of the world to request zone transfers as often as they want. Secondly, since zones separate different authorities, it is unlikely that there will be very many zones per database. Hence a single zone will comprise a sizable fraction of the database. On the average, then, a linear search will only be a few times slower than direct access.

If it turns out that zone transfers are putting a strain on the normal operation of the name server, I still have one card up my sleeve. When a request for a zone transfer comes in, the name server could fork off another process to handle the transfer. Since the database is in core, a fork would give the child process its own copy of the database to use for the transfer, freeing the name server to continue answering queries undisturbed.

Alternative implementations of zones include physically separate databases for each zone (similar to the ISI design) and many different types of logically separate databases. Physically separate databases within the same process would be rather hard for my database implementation to handle since the number of separate databases could change dynamically. A different hash table for each database would be needed as well as different disk files for each. The complexity and overhead grow

with the number of zones.

Logically separate databases seem to be a better choice overall. A secondary index structure similar to the inverse query table could be used to link zones to names. Some amount of runtime overhead is then required to update the secondary index as names are added and deleted. Some kind of mapping is also needed from names to zones so name look-ups can verify that the name found is in the proper zone of authority.

When implementing zones in my database, I decided to favor simplicity. Considering that my design requires no update overhead or secondary tables, it is not clear that I made such a great sacrifice of speed.

10. Inverse Queries

Inverse queries take classes, types, and data and return resource records having those fields. For example, an inverse query could ask for all RRs with data of "10.0.0.6." Another could ask for all RRs of type "Mailing List" with data "doctor@miro" -- that is, all mailing lists that "doctor@miro" belongs to. Inverse queries are considered rather rare, and it is optional for name servers following the ISI specifications to implement them.

Speed is not too important in answering inverse queries, but something better than a linear search is desirable. Secondary tables can be used to map the inverse keys to RRs in the database. To reduce overhead on updates, I do not keep a permanent copy of the inverse query table. Instead, like the free list, it is recreated at boot time. Additions to the database are noted in the inverse query table, but no disk i/o is used to save the changes. Deletions are ignored for simplicity's sake because there is a many-to-many mapping between inverse keys and domain names. Failed searches are discovered quickly enough.

My solution is to have one hash table to map data to domain names. Instead of storing the actual data and domain names, which would just about double the size of the database, I store only the hash values of domain names. I do not store the inverse keys at all. I hash an inverse key to a hash bucket in which are stored the hash values of all domain names under which the inverse key occurs. If several inverse keys map to the same hash bucket, a few inappropriate domain names will be examined for their inverse queries. Extra inverse query hash table buckets are allocated if overflow occurs. Only integers are stored in the inverse mapping tables so very little space is needed to implement inverse queries.

Selection on class and type is done as the RRs matching the desired data are found. An array of RRs meeting all the requirements is returned to the name server.

11. Completion Queries

Completion queries are queries which specify a portion of the hierarchical domain name of a resource record and desire the name server to discover the rest. An example of a partial domain name might be "ernie," which the name server would have to complete to "ernie.berkeley.arpa," say, before it could find any resource records to return. A knowledge of the hierarchical structure of the Domain Name Space and a description of the zones which the name server has authority over are needed to complete partial queries.

In general, the name server itself seems to be the appropriate place to implement completion queries. It has a detailed knowledge of the Domain Name Space and can apply "higher level" algorithms to find a completion. The database's view of the world is a bit too "low level" to know what zones of authority should be searched and what kind of pattern matching (first fit, best fit?) is appropriate for completing a partial domain name. This kind of decision making and domain name manipulation is better suited to the name server.

Lo and behold, my database uses a "flattened" version of the tree structured Domain Name Space, and it would be rather difficult to implement completion queries there regardless of the aesthetics. On the other hand, all the name server has to do to implement them is to concatenate a few domain names from whatever zone is appropriate onto the end of the partial domain name given in the completion query and then to try looking up the results in the database. This trial and error method should be as fast as anything the database could do by itself without encumbering it with more secondary indexes.

12. Overall Advantages of my Design

Speed and Spartan simplicity were two ideals I worked toward. My database has only one data type, making memory management and data manipulation easier. It resides in core, making data access as fast and easy as referencing a pointer. Locking mechanisms and concurrency control are not needed, freeing the one process that uses the database to run much faster. Moreover, since it is a special purpose database, we do not have to pay the cost of elaborate query processing, tape backups and other expensive crash recovery techniques, and intense CPU usage inherent in general purpose database systems.

My implementation of the Berkeley name server database is a potpourri of possible schemes. It incorporates the flavors and ingredients of many different designs. I tried to blend the best qualities of each to create a superbly seasoned stew. I use a hash table for quick access to a domain name. Yet each name has a hierarchical structure associated with it for data clustering, implicit data sharing, and navigational inspection of the data. The database resides in core for speed and simplicity. But a parallel disk version is kept so that we have a permanent record of updates. Mixed together, these dissimilar strategies should produce a database more palatable and satisfying than any one based on a single scheme.

13. Overall Performance

As described before, all the database has to do for standard and inverse queries is compute a hash value, do some string comparisons, and follow a few pointers. For update queries a little i/o is involved. The database is never CPU bound. A microcomputer with a winchester disk drive could run this database as fast as (if not faster than) a main frame computer. If virtual memory is not available, however, only a limited amount of data can be stored. As a rough estimate, given 64K bytes of data space we could store about 200-300 different domain names and resource records without modifying the program. If program and data have to fit in the same 64K, then without elbowing out the name server program which accesses the database via subroutine calls, we still might be able to store 50+ resource records, but it would be

necessary to allocate buffers in smaller blocks, one block of 1024 buffers of the current size probably being larger than all available memory in such a system.

My database program was written in C on a VAX 11/780 running Berkeley UNIX 4.2 BSD. To measure performance, I ran profiles on two test programs. The first made only standard queries, i.e. only data retrievals. The average response time was 1.67 msec per query, with about 40% of that time being spent in copying the data found into a return buffer, 40% doing string comparisons, and the rest in menial work. The second test program repeatedly inserted a long piece of data and then deleted it. The average response time for this i/o bound activity was 22.4 msec per query. About 50% of this time was spent writing to disk. The rest was used up in calling fseek(), saving the new data into buffers, and looking up names in the database. The garbage collector, which was run 40 times, accounted for only 0.7% of the total execution time.

The test databases were too small to provide a realistic picture of performance, but the results give a rough indication of the database's speed. The load average on the machine was moderate, ranging from 5.00 to 10.00 during my tests. Since zones are implemented as tags to domain names, neither the size of the zones nor the number of zones in the database affects performance. The sheer size of the database (in bytes) only affects performance by possibly increasing the likelihood of page faults. The lion's share of queries in a large database probably occurs in only a small portion of it. Page faults would be minimal in that case. On the other hand, if the database is large and uniformly accessed, then page faults will be inevitable, and their frequency will increase with the size of the database.

Within the hierarchical structure of a resource record, the more classes and types that exist, the slower an individual RR look-up will be, since class and type fields are found by a linear search. My test databases were too small to analyze this particular kind of degradation.

14. Disadvantages

There are tradeoffs to be made in any design. Advantages must be weighed against disadvantages. To be fair, I should reiterate some of the sore points of my design.

First of all, there is no concurrent reading or writing of the database. User queries to the name server might have to wait a long time if the name server is engaged in lengthy maintenance operations, such as transferring zones and distributing incremental updates. However, as was also mentioned before, forking could be used to ease this problem, allowing the maintenance work to go on in parallel with user queries (forking is rather time consuming at present in Berkeley UNIX, so the decision to implement it is not a simple one). In a sense, I have designed a concurrent database with only one lock: i.e., the whole database. Write access is thus serialized, but so is read access, unless forking is used to make duplicate copies of the whole database. On the other hand, real concurrent databases can lock smaller units than the whole database, at a substantial cost, to allow both kinds of access to be carried out in parallel (as much as possible).

Secondly, all data are strings. Numeric data must be converted to ASCII strings before they can be stored. In the Domain Name Space, these data are classes, types, and TTLs. Thus we have to pay a small tax to have all of our data fit into the single

type of data buffers. As an aside, having only one data type made the program very easy to modify. I built and rebuilt the hierarchical RR structure many times, and to do so I had only to change the look-up and update procedures. All the buffer handling routines remained unchanged.

Thirdly, although the one-sized data buffers are very flexible, they do waste memory through internal fragmentation. If most data are only a few bytes long, my database will waste about ten times more in storing it. Buffer size could be readjusted at compile time if need be. For classes, types, and TTLs, the buffer size is definitely too large, but a compromise must be struck between these small pieces of data and the larger domain name and data fields. I must confess that frugal memory usage was not one of my design goals. Efficient re-use was. Compared to a static database design, mine does appear gluttonous. Having more than one size of data buffer, in the hopes of saving a little space here and there, would destroy the simplicity of my design and would make garbage collection and fragmentation control much more difficult to perform.

15. Conclusion

Considering the goals I wanted to fulfill, I would say that my database implementation is a success. It is fast, it has a clean and elegant simplicity about it, it re-uses memory absolutely, and it is very hard to crash unrecoverably.

As fast as name server databases may be, it is interesting to note that communication between hosts on the ARPAnet will unavoidably be slower in the future than it is today. A quick table look-up is all that is required now. With name servers installed and the ARPAnet broken up into a tree, it will take at least two message delays just to find out the network address of a particular host. After finding the desired address, the original message can be sent. In the worst case, it could take three times as long to send a piece of mail from Berkeley to MIT, say. Name servers will have to employ a substantial amount of caching to improve performance. In the absence of such caching, it is a rather academic question to ask how fast the name server's database is! Network delays are the limiting factor in name server performance.

Other data besides ARPAnet host addresses can and should be stored in name servers. Such an eyesore as "/usr/lib/aliases" better be the first to go. Its format could easily be converted to RRs. An alias name would be a domain name in our name space. Members of the alias would be stored beneath it as data. There would be no length limit on names or data in my database if this mapping were used. Other information could be stored beneath the same alias (in a separate class or type), such as the home addresses of the members, their phone numbers, or even what the alias is used for (e.g the "Doctor Who mailing list at Berkeley").

The "finger" information of each user would be the next logical mass of data to put into the name server. Some form of security protocol needs to be incorporated into the name server before any more sensitive data is stored, however. If the datagram communication between the name server and other processes could be verified, then a simple access list for each domain name would be a sufficient security measure in the database. The access list could be stored as easily as an additional class or type under a given domain name. The names of as many authorized users as

desired could be kept as pieces of data on this access list. An empty access list could be considered as having "everybody" on it. Thus, public bulletin boards could be easily established.

The Berkeley Name Server Project was developed in parallel with a design team at USC-ISI working with a different operating system. These two name servers have successfully communicated with each other over the ARPAnet in test demonstrations. Together they will occupy a very important position in the future of the ARPAnet community.

I hope this paper has given the reader some insight into the design and implementation issues unique and not so unique to name servers and their databases. Whenever I could, I tried to make my solutions to the many problems as novel as possible.

16. Acknowledgments

I would like to thank my fellow researchers in the Berkeley Name Server Project for their cooperation and support during the evolution of my database. They are Doug Terry, Songnian Zhou, and Mark Painter. Special thanks go to Songnian for his insight and always helpful comments on my design. Credit for organizing our group and instructing us with the utmost care goes to Doug Terry. Professor Domenico Ferrari was our faculty advisor, and he was always extremely interested and helpful. Professor Cabrera helped me with a last minute proofreading, and I am grateful for his help. Thanks also to my many friends who urged me to hurry up and find a research project last year. Now they can help me look for a job. Warmest affection to Chris Guthrie, Perry Caro, and Edward Wang.