The Design of the Dents DNS Server

Todd Lewis
MindSpring Enterprises
1430 West Peachtree Street NW
Atlanta, GA 30306
tlewis@mindspring.com

Abstract

Dents is a server implementation of the Internet's Domain name System. Dents main features are a modular driver architecture, a CORBA-based control facility, a replaceable tree system, a clean design and good karma. Dents is free software, licensed under version 2 of the GPL. In this paper, I describe the design of Dents, concentrating on the innovations and evolutions it embodies, and including the future directions in which we hope to take the server. I describe some of the problems we've had. Finally, I summarize some lessons about server design which Dents reflects.

1. What is Dents?

Dents is a server for the Domain Name System, the system whereby information concerning host names, including, most importantly, their IP address, is communicated through the Internet. DNS is a hierarchical caching directory keyed by name, with an extensible set of attributes which are associated with names.

Dents is free software, released under the terms of the GNU project's General Public License, version 2. It is coded in ANSI C, or as close thereto as we can come, and is oriented towards POSIX-conformant and POSIX-like systems. Dents uses POSIX threads, and while it is possible to compile the server without threads, it is not recommended, since several major features do not work without threads. Dents has a modular driver architecture, which permits various means to be used to look up names, and it includes a CORBA-based control facility, which allows administrators to control a running server. Dents should work on any modern unix-like system which supports shared libraries and threading; a win32 port is not out of the realm of the possible, although such is not presently planned. Primary development happens under the Linux operating system.

Dents as a project was started by Todd Lewis in early 1997. Johannes Erdfelt wrote the majority of the code, and following our first public release in late 1998, Greg Rumple joined the team and has contributed significantly since then.

Dents was inspired by some particularly unpleasant experiences of the author in dealing with the DNS systems built at MindSpring. Thanks to its virtual web hosting product, MindSpring was then and is now one of the largest registrars of domain names in the world, and dealing with serving a large number of zones has been and remains a trying problem.(*) Dents grew from frustration with how unfriendly existing DNS technology was to people trying to build higher-level management systems on top of the basic server. Better would be a server which you could reconfigure in a serious way without restarting, one which would allow the use of relational databases and other external systems to implement the underlying data management and retrieval functions, and the ability to update zones quickly and with a minimum of effort. Out of these desires grew some evolutionarily new ideas for how servers should be built. This paper describes some of these ideas, our experiences implementing them in Dents, and what we have learned from the experience.

2. The Driver Mechanism

One major feature of Dents is our modular driver mechanism. Conventional name servers simply read in all zone data at startup and store it in core memory. In cases where one has a large number of rarely-read records, this is a very sub-optimal use of resources. We wanted to be able to use a relational database as the underlying storage mechanism for DNS data, translating DNS queries into SQL queries and the reverse for answers. Further, we wanted for this system to be flexible enough that we could substitute _any_ underlying engine for answering DNS queries, and clean enough that these engines would remain useful through multiple revisions of Dents itself.

The conceptual model which we settled into was similar to the notion of file system drivers within unix-like operating systems. We decided that we wanted to "mount" zones at certain mount points within the hierarchical space, that each underlying zone would have a type and that it would be handled by a driver corresponding to that type. The driver serves to hide the actual details of the implementation and adapt them to a common interface, so that all instances have the same behaviour, regardless of the underlying technology.

It comes as no surprise, then, that Dents driver modules look fairly similar to file system drivers. There is a finite and well-defined set of functions which a driver must support. When a driver is loaded, these functions are used to populate a structure full of function pointers; this common interface allows all zones to behave identically, just as all file systems behave identically. Certain features are optional, and so if a driver does not support them, then the attempted use of them simply returns an error. (Symbolic links in the case of file systems, resource record addition and deletion in the case of Dents drivers.)

Dents uses the unix *dlopen*()/*dlclose*() mechanism for loading drivers. These drivers are identical to shared libraries, but instead of being linked in by the loader at start time, they are loaded by Dents when zones are added; they are demand loaded, so no explicit load is necessary, and they are reference counted and automatically unloaded when all zones using them are removed. The actual code was modeled after the GTK widget set's theming mechanism. To aid in robustness, one can specify at compile time for certain modules to be compiled statically into the server; the only difference internally is that statically-included drivers are never unloaded.

All zones in Dents are served using this mechanism; we rewrote both of our system components which serve zone data into modules. They are:

mod_stddb: RFC-1035, section 5, specifies the format for transferring zones between name servers. This format has historically served as the native storage medium for zone data with other name servers. This module makes Dents behave just as traditional name servers do, reading in zone files in this format and storing them in an in-memory structure, looking up records in this structure later to answer queries. Although this is not our preferred method for running servers, it is a popular choice, and so we feel compelled to offer it, at least for migration purposes.

mod_recursive: The DNS system relies heavily on local servers to perform DNS queries on a proxy basis for local clients, in turn caching the results and using that cache for future requests for the same name. This module fills that role, allowing Dents to serve as a recursive name server for the root (i.e., the entire space with the exception of any local authoritative zones) or for any other zone. Answers are stored in a tree, which is then periodically purged. We hope to turn mod_recursive into a very well-tuned cache; historically, DNS caches have simply never purged data, the system failing when system memory is exhausted. We aim to do better.

Further, we have several other modules in development by the Dents team:

mod_frl: One particular case where the behaviour of traditional name servers is clearly suboptimal is in the case of in-addr.arpa zones. Internet service providers have large banks of modems, each of which usually has an IP address associated with it; i.e., customers who dial into that modem and negotiate a PPP session will receive that IP address. These IP addresses must be associated with names through the in-addr.arpa zone, or else certain classes of internet services will not work; e.g., certain FTP servers will not allow users access unless their IP has both forward and reverse name resolution.

At MindSpring, none of us in the Engineering department volunteered to name our several tens of thousands of modems individually, and so we came up with another solution: we name them algorithmically. If you look up the IP address *247.2.192.209.IN-ADDR.ARPA*, you will get back the name *user-38s00nn.dialup.mindspring.com*. This is simply the base-36 encoding of the IP address; the reverse of this query, i.e., looking up this name and getting back the IP address, works as well.

This is all very nice, but with a conventional name server we must generate several tens of thousands of these name/IP pairs and load them into the fairly expensive error-correcting system memory on our name servers. This seems rather silly for names which are generated algorithmically.

mod_frl performs this algorithmic translation on the fly, allowing one to serve a very large number of zones with a comparatively small amount of memory. (The memory size of the module is constant; it doesn't even pay attention when you tell it that it is responsible for a new zone, simply answering any and all queries the server sends its way.) The code to perform this is 86 lines long, and the binary module takes a little over 4k of memory. We think that this is a particularly good example of how a modular approach to serving DNS data can result in a significant advantage in metrics which are important to administrators.

mod_bdb: Another fairly simple module is the Berkeley DB module, presently under development. The Berkeley database is a very simple and well-tested associative database, offering key=>value associations using either hash tables or btrees. We simply prepend the query type and a period to the record name to gain our key; for requests for all records for a given name (query type "*"), we store a record which contains pointers to all other records for that name. This is a god example of how a flexible mechanism allows you easily to leverage the efforts of others; Berkeley DB is very easy to program, and we get a very fast mechanism which is well tested and which even supports transactional updates. (More on transactional updates below.)

mod_covert: Cheswick and Bellovin suggest, in their book "Firewalls and Internet Security", that one could use DNS as a covert channel for tunneling data through supposedly secure environments. It would be an interesting exercise to write a module to accomplish this goal. Since many legacy DNS servers do not expire entries from their cache, this approach would have the unfortunate effect of crashing many of these servers, and so it would have to be used with care.

Finally, several outsiders are developing modules for Dents:

mysql: several people are working on implementing a module which uses mysql, a relational database management system, to store DNS data. When the module is passed a query, it communicates with the RDBMS via SQL to discover enough information to answer the query. It then formulates the answer.

The exciting thing about modules which use databases as their functional substrate is that it takes a class of tasks at which the DNS system is not very good and at which databases are good, and it transfers responsibility for these tasks from the DNS system to the database system. Specifically, large portions of the DNS standard deal with how zones are replicated across multiple servers. This replication happens wholesale and is unauthenticated. Major effort has been put into loading (I would call it overloading) the DNS system to allow incremental zone updates and for updates to be secure. Updates are still not transactional, and there are at present no plans to make them so. (More on transactional updates below.) There are a number of commercial database systems which have long since solved these problems to a degree of completeness and reliability that DNS will never approach. This is yet another reason I believe that most zone data is better off being stored in a database rather than in the way zone data is conventionally stored.

mysql is popular in certain circles, but I would very much like to see other modules in this vein written: one for PostgreSQL, which is the preeminent free relational database, and some for the various commercial relational databases now available under Linux and other unices, including Oracle, DB2, Informix, Sybase, and Interbase.

dhcp integration: One very popular feature of Microsoft's DNS server for Windows NT is its integration with the Microsoft DHCP server. One developer is working on a driver module to integrate Dents with the ISC DHCP server. This module will take updates that happen in the DHCP server when a new DHCP lease is issued or an old one expires, and populate the DNS space with corresponding information on the machines dropping into or out of view.

(As an aside, Johannes Erdfelt is convinced that Dents can serve as a fairly generic server for associative data, serving such protocols as DHCP and LDAP in addition to DNS. This is why we segregate DNS-specific code from other server code in the code base. This is so remote a prospect at this point that I will mention it no further.)

other modules: One user is investigating using Dents to export a database of HAM radio operators via DNS. Another set of users is looking to use Dents as part of a project to adapt their DNS setup to use their own underlying database to deal with bandwidth-constrained links between their DNS servers. Others have expressed interest in Dents as part of various dynamic-IP naming projects. Our goal was to produce a very flexible system amenable to whatever tortuous uses people wish to apply the Domain Name System, and we seem to have succeeded with this.

3. The Tree System

After the driver module system, a second major architectural piece of Dents is our tree system. This component actually snuck up on us; we did not plan for it to be a major part of the Dents story.

Initially, Dents used a home-grown implementation of red-black trees to store its hierarchical data. (Which data is a lot for a hierarchical directory service.) We also used it for several internal data structures. The initial implementation dated from very early in the project, and we had several problems with it: for one, it was buggy, and for another, the caller managed all of the memory for record structures, which was a big mistake. Greg Rumple's big contribution to the project has been in working on this issue.

We realized that our initial code had problems and, rather than simply rewrite it, we decided to segregate it from the remainder of the server and make the red-black tree engine replaceable. We decided also to attempt this separation for several reasons.

First, we wanted completely to isolate the internals of the tree system from the rest of the code, to enforce good layering between server code and the tree code. Isolating the routines would allow us, e.g., to *mprotect()* internal memory upon leaving the tree code an *munprotect()* it when reentering the tree code, allowing very easy determination if any external code was touching the internals of our tree system. Good coding practice would accomplish this goal, but good programming practice can sometimes be in short supply, and mandatory discipline makes a fair substitute.

Second was the issue of memory usage. Unlike our major competitor, we had no problems growing past 64k zones as of version 0.0.1 and subsequently. Further, our implementation is plenty fast in the rough tests we have run on it. However, it consumes a lot of memory, almost twice as much as our competitor for an identical configuration. While this might be acceptable for certain installations, some others may be more memory sensitive. Rather than forcing one decision down our users throats, we thought that having a pluggable tree system would allow them to chose the tree engine which best suits their needs.

Third and most important was the issue of performance. The only real performance barrier in our main competitor is its use of a single hash table to store all DNS information. We believe that this is a design mistake and a competitive vulnerability. If you were to construct a histogram of DNS names and their hit counts, you would see a very tall spike at one end, representing a few names which receive a very, very large number of queries, rapidly flattening out to a very long, flat tail which represents the remainder of the DNS space, records which are queried infrequently if ever. Using a hash table to store data of this profile guarantees that you will not gain the benefit of modern computer hardware architectures, with their large caches connected to (relatively) slow main memory. One thing which we hope to accomplish once we can plug in new tree engines is to implement a heavily cached engine, which will use layered caches to utilize the underlying hardware in the best possible way, allowing frequently-asked-about names to sit very high in the machine's cache hierarchy. We believe that the speed gain of this move will in the aggregate more than offset the additional cost of maintaining the cache, which can be accomplished mostly during idle time on the server. This comprises the core of our strategy for being the fastest name server.

Finally, almost as an afterthought, driver modules often have a need for the sort of functionality which our tree engine exports. As a courtesy, we would like to export this functionality to them, both to maximize code reuse, but more importantly also to minimize instruction cache invalidations as the server jumps from server code to driver module code and back to server code.

As I write this, we have just released version 0.0.3 of Dents, which embodies our preparation for performing this modification to the server. We are now embarking in earnest on implementing this code separation in the server. It will be reflected in our next release, and at that point work will begin on building alternative tree engines.

4. The Control Facility

One feature which we hoped to have from the first day of the project was an administrative interface whereby we could interact with a running server, query it in detail concerning its state, and instruct it to perform various actions. Specifically, we wanted to depart from the stale convention of using configuration files read at server start as the exclusive means of determining the server's configuration. This convention causes downtime for the sake of configuration changes that do not themselves require downtime, and when your servers are very large, i.e., they serve a large number of zones, the cost of restarting the server is far from negligible. Even with small servers, such unnecessary outages offend the sysadmin aesthetic; this is another case where Dents reflects, we hope, the sensibilities of a sysadmin instead of those of a developer. We do not believe that restarting the server should be required at all to make changes to the server's configuration, and we designed Dents with the goal of making this the case. For this reason, we resolved to include in Dents a control facility to serve this purpose.

4.1. Inspiration

This feature is far from an original one; many other systems have allowed administrators to query and control them while they are running. Of course, the all time champions in this are commercial relational database management systems; these usually let you tweak any knob which is not inherently untweakable in a running system. The database community may have certain failings, but their attitude towards downtime is sufficiently paranoid, and is therefore laudable.

Somewhat closer to the traditional backyard of Usenix, two particular systems stand out as having inspired our approach in Dents. The first is the XNTP suite from the University of Deleware. Their xntpdc facility does a very nice job of allowing an administrator to configure a running xntpd daemon. However, it does this by (ab)using the NTP protocol. We did not want to use the DNS protocol to control Dents for several reasons, which I discuss below.

The second inspiration for the Dents control facility was the kadmin interface of MIT's Kerberos distribution. kadmin is really the closest thing to our control facility of anything out there. kadmin uses ONC-RPC to talk to the server, rather than overloading the Kerberos protocol. Because of this, it gains several advantages. First, it benefits from the IPC infrastructure, and the IPC infrastructure benefits from it. Partially thanks to kadmin, Kerberos-protected ONC-RPC exists and is available for others to use; if this work had been done in a kadmin-specific way, then no one else would have benefited from it. Second, because kadmin uses ONC-RPC, they do not have to worry about handling the underlying wire protocol; all they see is an API, with the RPC package taking care of the details. Third, an RPC environment allows for much more expressive administrative interfaces. Hand-rolled protocols, because of the amount of work involved, inherently limit the expressiveness which they allow; the more expressive you are, the more work you make for yourself to handle the protocol in your code. RPC does not really suffer from this; the programming cost you see is the cost inherent in your interface design and nothing more. These considerations weighed heavily when we designed the Dents control facility.

4.2. Why administrative protocols should be divorced from regular protocols

Before talking further about the Dents control facility, I would like to explain why I think that shoehorning these interfaces onto regular protocols is usually a mistake. First, overloading the regular protocol can damage the protocol. DNS has the interesting property that the addition of new resource record types is dangerous; because of the scheme used for compressing data in DNS packets, new RR types can render packets including that RR type undecipherable to older agents which do not understand that type. Even worse, if support for server control is included when the protocol is designed rather than added as an afterthought, (the later usually being the case), then the protocol is almost necessarily made more complex, introducing cost on all users of the protocol for the benefit of a few. Such additions can even be for the benefit of none, if the needs of the administrative community outstrip the ability of the protocols in question to handle them. Administrative needs can change significantly over time; one need merely compare sendmail to qmail to understand the variations possible in configuration needs among implementations of the same protocol. However, the protocols themselves should be timeless, or at least very slow moving, so as to get the maximum benefit of the programming effort expended in implementing them.

Second, regular protocols should not be overloaded to transform them into control protocols because they often make really bad control protocols. A prime example of this is updates in the Domain Name System. DNS is a stateless protocol, and as such it is virtually impossible, without butchering the protocol, to introduce transactional semantics. However, certain groups of domain updates (such as removing one machine from a DNS rotor and introducing a new machine to that rotor) _really_ need to happen atomically. There is no middle ground here: either you perform all sorts of gross changes to the fundamental nature of the protocol, or you settle for an inferior administrative interface. Finally, the security semantics of the functional and the administrative interfaces are usually radically different from each other; services offered publicly or semi-publicly usually need very little if any authentication, whereas administrative interfaces usually need very stringent authentication and often privacy. This is another case where an impedance mismatch between the two divergent needs means that no system can truly satisfy both.

4.3. Implementation

Initially, we implemented the Dents control facility using a line-oriented protocol similar to POP or NNTP. However, we soon decided to switch to some flavor of RPC for the facility. We examined ONC-RPC, DCE-RPC and CORBA, finally settling on CORBA.

Our decision to embrace CORBA for this function had several reasons behind it. Most of the reasons apply to any form of RPC.

- The details of the protocol are hidden from you, along with all of the work required to handle them.

- When one considers the human factors involved, from inaccurate documentation to imperfect implementations, they are much less error-prone than hand-rolled protocols.

- They come with lots of free add-on services, such as security and transactions, that are difficult and usually unpleasant to implement yourself for a single project; because this effort is shared across multiple projects, the resulting code is usually much better.

- Finally, RPCs are much more expressive than hand-rolled protocols, precisely because you are shielded from the programming which has to occur to support a complex client-server interaction.

We finally chose CORBA over the other forms of RPC because:

- it supported exceptions, rather than the more restrictive, conventional C style of signaling error conditions;

- its associated services were much broader, offering more value to us than ONC RPC, which sometimes has security associated with it and nothing else;

- there are free CORBA implementations which are being very actively developed, unlike wither ONC or DCE RPC;

- finally, CORBA itself is undergoing active development, from the core protocol to the CORBA services, whereas DCE RPC is, so far as I can tell, dead, and ONC RPC is developing very slowly.

With our recent public release of version 0.0.3 of Dents, we included this new CORBA-based control facility. During the upcoming release cycle, we hope dramatically to increase the range of functionality available through the control facility, now that the infrastructure is in place and working.

4.4.What a control facility makes possible

The control facility made an initial project feasible: SNMP support. RFCs 1611 and 1612 specify the DNS server and client MIBs. For the past few years, no DNS server to my knowledge has supported these MIBs. Because of the control facility, adding support for these MIBs to the University of California, Davis SNMP server was a very easy task. I simply added a new module to the server, as the UCD docs describe how to do, and had that module make a connection to Dents through the control facility. When requests for statistics come in, the SNMP server simply makes control facility calls and returns the results to the SNMP clients. I am an inexperienced C programmer, I had this scheme working and answering queries in an afternoon; this is a good indication of how powerful CORBA is as an easy way for doing IPC. This compares favorably in many respects with alternate schemes for IPC between SNMP servers and other system servers.

The second goal we have for the control facility is that users (i.e., administrators) be able to control their servers through automated scripts. A good example of this is, again, MindSpring's web hosting environment. When web hosting accounts are set up, often times there is no human intervention. Customers sign up on a web page; the billing system validates their credit card; the billing system then informs the web hosting system to create an account; the web hosting system creates both the WWW part and the DNS part of the account, and then informs the user via email that his account is ready to be used. It would be nice if these scripts could contact a running server and tell it to add a new zone, immediately, without having to restart it. The time to restart the DNS servers is, right now, the single biggest source of delay in creating web hosting accounts at MindSpring, by far. Similarly, when customers do not pay or exceed their bandwidth limits, it should be possible to turn them off immediately, rather than having to wait for the next scheduled server restart. These operations should be strongly authenticated.

The final goal for the control facility is to enable graphical administrative clients. Ironically, automated interfaces and graphical interfaces have the same requirements, and textfile-based interfaces are hostile to both. (For some reason, many people think that textfile-interfaces are friendly to automated environments; of course, nothing could be farther from the truth.) It should be possible through a graphical client, using the control facility, to set up a new server, to add or delete zones, to populate those zones, to examine the change logs for a zone, to set up primary and secondary servers, and all of the other things that admins need to do to

5. Future Directions

Transactional updates: One particular area where the approach of using an out-of-band control facility really shines is in the area of updates. Many users have the need for several related updates to DNS to be batched together and processed atomically; they need transactional semantics for updates. The DNS protocol itself, being stateless, is singularly unsuited for providing transactional semantics. We plan on offering users atomic updates to DNS data using the control facility and CORBA Transaction Service; this is another case, in addition to security, where having a rich suite of services available within the context of CORBA really pays off.

More Drivers: We will be working hard during the coming months to assist people other than the core developers in writing modules. We hope to use the driver system to allow developers to develop systems to meet their own needs with a minimum of development effort. We have talked about embedding interpreted languages as modules, so that less experienced programmers can write scripts to formulate answers instead of having to program in a compiled language which is compatible with our binary interface.

Tree Code: The separation of our tree code from the rest of the server and the ability to plug replacement tree engines into the server will allow a good deal of experimentation with different performance profiles for the server. Specifically, it would be good to have both a low-memory engine and a high-performance engine available for users to select as their needs dictate.

5. Problems we've encountered

We use the GNU project's automation tools for managing our build environment, specifically automake, autoconf, and libtool. We have decided to use the latest versions of these tools; this has caused a good deal of difficulty for non-experts trying to compile the system. As project lead, I feel torn between the developers who say that the tools really help them and the users who say that the tools really hurt them. As we get closer to a final rollout of v1.0 of the code, we will lean more towards helping users and away from favoring the developers.

We encountered a problem where certain of our system structs, which were included in modules, had certain portions *#ifdef*'d out if the server was not compiled with POSIX threads enabled. If the modules and the server were not compiled with the same setting for POSIX threads, then they were binarily incompatible. We solved this problem by declaring that POSIX threads need to be enabled as the standard, and that non-POSIX-thread-enabled builds were only for debugging and were aberrant.

Finally, we have simply taken too long to get Dents out the door. Sometimes we had good reasons and sometimes we didn't. Once we did the public release, then things really started moving with the development effort. We should have made a public release much sooner.

5. Lessons Learned

Divorce functional from administrative interfaces

Mixing functional and administrative interfaces damages your functional interface and cripples your administrative interface. Server designers should have the courage to divorce the two, and use the resulting freedom to deliver full-featured interfaces which allow admins control over their servers. Kerberos and relational databases are a good model to follow here.

Use CORBA

The difference between rolling your own protocol and using CORBA is like the difference between writing programs in assembly and writing them in a higher-level language. Just like the later case, surprises lurk when it comes to performance. Just as compiled programs are often more efficient than hand-programmed assembly, CORBA, by virtue of its pass-by-reference nature, can often achieve network efficiencies that hand-rolled network protocols can not. The goal the designer is trying to accomplish is usually something very close to a remote procedure call anyway and can be adapted to the RPC paradigm without problem. CORBA makes many great things possible.

Servers as machines

Here I speak speculatively and very explicitly for myself and not for my fellow Dents developers, some of whom might not agree with me on this issue. Increasingly, I come to the conclusion that a good model for servers is that they be machines, in the formal sense. Servers should not be responsible for configuring themselves, just as operating systems are traditionally not responsible for configuring themselves. Rather, like operating systems, servers should simply export functionality and bootstrap themselves only as much as it takes to allow outsiders to utilize their functionality, and they should allow their state to be read completely. In this way, the start and stop of the system are simply temporal holes in the provision of service, not configuration points. Under this rule, downtime need not be unnecessarily spent on configuration changes. Today, functional requirements about configuration needs, which should properly be embedded in the source code as assertions regarding the state of the system's configuration, are instead "enforced" by opportunistic order of configuration routines in the bootstrap section of the server. This is a horrible area of unfixed and unfixable bugs, as often these requirements are never documented. These requirements should be properly reflected in the code, ala: "ERROR: can not load new zone until system module path set; can't find modules!"

The conservatism of the unix community in the design of system servers has led to outright stagnation; new ideas are very far between these days in how servers are designed, and this is a real shame. Our real hope with Dents is not so much to take over the DNS space as it is to introduce and promote some new ideas about how servers should be written. It is our hope that they will be successful for us, and that our success will inspire others, not only to emulate our techniques, but to emulate our innovation.

Acknowledgements

Thanks, of course, to Johannes Erdfelt and to Greg Rumple for doing all of the hard work on Dents. Throughout the life of this project, MindSpring Enterprises has employed each of the three major authors of this project. While MindSpring has never officially supported the project, they have made project resources available to us and been very understanding of those few times when Dents has come first and work second. Additionally, our coworkers at MindSpring have been a great, unlauded help to us as we struggle to learn how to build servers. I am grateful for their support. Finally, I'd like to thank Paul Vixie and all of the other contributors to the DNS standardization process for creating such a fun protocol to work on; the entire Internet owes them a debt for such a functional name system.