Life With djbdns

Henning Brauer <lists-lwd@bsws.de>
11 August 2005

Table of Contents

1. Introduction

This document will probably never be as comprehensively helpful as its inspiration, Life With Qmail by Dave Sill, but then djbdns is nowhere nearly as complex as qmail (it does a much, much simpler job), so that's fitting.

djbdns is a simple software kit for serving and resolving DNS data. It is intended to be a replacement for BIND in many settings, although it does not yet include every feature implemented by BIND, and may never do so; djbdns includes features that can be demonstrated to be needed, and there are some features offered by BIND that do not make the cut. Like qmail, especially in its younger days, djbdns can require some redesign to deploy.

This document will attempt to give the Big Picture, which is most of what you need to get going; describe the components of djbdns and how they fit together; provide illustrations of typical installations; and hopefully in the process field some of the more Frequently Asked Questions; but it is not intended to be a replacement for any of the existing FAQs; this is an introductory paper, to be read in sequential order, not a reference.

"Life with djbdns" was started by Bennett Todd <bet@rahul.net> who has written big parts of it. Without his extraordinary amount of work and his skills this document would not be what it is today.

2. Translations

Spanish

3. Other resources

Official djbdns page by Dan Bernstein

www.tinydns.org by Russel Nelson

There's also a Mailinglist, dns@list.cr.yp.to. Subscribe by sending an empty mail to dns-subscribe@list.cr.yp.to, unsubscribing works similarly by sending an empty mail to dns-unsubscribe@list.cr.yp.to.

---

4. The Big Picture

Domain Name Service [DNS] is a distributed database, supporting delegation of authority for segments of the key space. It is primarily used for serving mappings from hostnames to IPv4 addresses; it also offers address to name mappings, info about hosts, special routing support for email, myriad of more exotic application-specific goodies. It uses some special internal record types to define its own internal hierarchical structure, the delegation of subdomains; and there are features to support IPv6.

Like any complex system that does a tricky job well, there are an assortment of concepts with special jargon used to refer to them. I'll try and introduce the important ones here.

DNS is made of a bunch of servers passing Resource Records (RRs) around. There are many types of RRs, and several different protocols for requesting them. DNS is carried on TCP and UDP both, on port 53. Most normal queries are over UDP. TCP is used when the total of all the RRs in a reply exceed 512 bytes, or for performing "zone transfers" (more about them shortly).

4.1. Resolvers and Nameservers: definitions

The authors of the DNS standards (RFCs 1034, 1035, and many successors) describe it based on a design model, which is followed by the Berkeley Internet Name Daemon (BIND); this model includes the notion that all servers are really basically the same kind of code, and it includes the notation for zone data files that BIND uses. Unfortunately, the design of BIND, as presented in the RFCs, has a fair amount of undesirable complexity, and has produced some security worries as well; the design of djbdns is based on a much simpler model which I'm going to describe. Also unfortunately, however, this has produced a bit of a jargon problem: there are distinct roles that a DNS server can perform, which aren't clearly distinguished in the traditional DNS jargon, because BIND performs them all; as they are separated in djbdns it becomes important to give very precise names. I'm going to call the library routine that a client program uses to do a lookup a "resolver library"; I'm going to call the helper daemon that it sends the request to a "recursive resolver", and I'm going to call the ultimate authorities that are the final sources of the information "authoritative nameservers". These are sadly verbose, but please bear with me; for all their prolix verbiage, they do two things. First, they precisely describe the functions that are being performed, and second they are reasonably consistent with traditional usage.

4.2. DNS Query types

There are two different sorts of DNS queries that can be sent, distinguished by a bit called Recursion Desired (RD). Recursive queries, where this bit is set, are normally sent by application programs, using the routines in system libraries like gethostbyname(3). These will typically be sent to a server whose address is found by checking /etc/resolv.conf for the nameserver to use. A recursive query asks the server to do whatever needs to be done to find the answer, including recursively inquiring of whatever other servers it must to track down the answer, hence the name. The answer to a recursive query should be the final answer to the question, or a firm statement that the answer couldn't be found.

The other sort of query is a non-recursive query, also called an iterative query; it is typically sent by a program that is acting as a recursive resolver; such a program would be listening on an address that a client finds in /etc/resolv.conf. So a client wanting to find an address for a hostname, or a hostname for an address, or wanting to ask some other question of the DNS system, would look up the address of the appropriate recursive resolver in the local /etc/resolv.conf and send its recursive request, with the RD bit set; this is the functionality found in gethostbyname(3) and gethostbyaddr(3). The recursive resolver would then begin the process of tracking down the actual requested information on behalf of the user, in a fairly clever (i.e. complex) way, using a bunch of iterative queries (with the RD bit cleared). These iterative queries do not ask the server to track down answers for them with further queries, they simply ask the server to answer the question, or tell who might be closer to knowing the answer.

So where a recursive query for www.example.com might return its IP address, or a firm statement that there's no such machine, an iterative query might return the identities of other nameservers to try in the hunt for the answer.

4.3. Structure of Domain Names

Domain names are labels for nodes in a tree structure. The the root of the domain name tree is ".", although that root is not commonly written in domain names. A domain name might normally be written "www.example.com", but the absolute version of that domain name is "www.example.com.".

Normal users don't need to write things this way, and you don't need the trailing dots when configuring djbdns either, but it can be helpful to know when testing DNS out; by explicitly using a trailing "." you can prevent your resolver libraries for trying to append the local domain name, as configured by the "search" keyword in /etc/resolv.conf. So if your domain is "foo.bar", and you ask for "www.example.com", your browser may (depending on details of your local library) try looking up "www.example.com.foo.bar" before it tries looking up "www.example.com"; if instead you ask for "www.example.com." that extra check will definitely not be tried.

The tree structure of the DNS namespace directly reflects a tree structure of authoritative nameservers. The root nameservers, whose addresses are configured into every recursive resolver to "prime the pump" and get them started, live at the root of the authority tree; links within the tree follow delegations of authority.

These root servers can authoritatively tell which nameservers have data for the various domains directly under ".", like "com", "net", "edu", "org", and the many two-letter country top-level domains.

4.3.1. Common Resour ce Record Types

DNS supports many, many resource record types. A half-dozen or so will be used by nearly everyone. They'll be described in more detail below. But without 3 record types we can't pursue the simplest illustration, so let's get them in here:

```
A   Host record, give the IP addr for a hostname

NS  delegation record, specifies the domainname of a
    Nameserver to use to find additional info on a
    domain

PTR pointer record, maps one domainname onto another,
    commonly used for reverse-lookups, to translate IP
    addresses back into hostnames
```

4.3.2. Example: find the IP addr ess of www.example.com

When a client wants to look up the address of www.example.com., it sends a recursive request for an A record (with the RD bit set) to the recursive resolver whose address it finds by looking in /etc/resolv.conf. If all goes well, it will [eventually] get an answer back.

That resolver then starts a long-sounding process to track down the answer. In practice it's usually reasonably quick, since resolvers cleverly cache the answers to questions, and so can often answer them without having to go through the whole search outlined here. But it's critically important to understand how this search works, so let's walk through one.

A recursive resolver is configured with the addresses of the root authoritative nameservers. It starts by sending an iterative request (RD bit clear) for the A record of www.example.com to one of the root authoritative nameservers. It may pick one at random; it may have collected data on how quickly they replied and try sending to the one that seems to be answering fastest; it may use them in rotation; that's up to the author of the recursive resolver.

Now www.example.com is probably not in the root nameserver's authoritative data; it doesn't know any www.example.com. But it _does_ know a list of servers which are authoritative for com.; in fact, it knows every server that is authoritative for every top level domain (TLD) --- that's the precise definition of a root nameserver.

So the authoritative root nameserver sends a reply back that says sorry, I don't know www.example.com, but these are the authoritative nameservers for "com", oh, and by the way, here are their addresses. Since the recursive resolver would immediately have to ask for the addresses anyway to get the answers, the A records mapping the hostnames of the ".com" top-level-domain nameservers are included in the reply, along with all the NS records giving the delegations of ".com" to those servers. These extra "A" records are called "glue records", remember them, they are important to understanding how to best create data files for djbdns.

So now the recursive resolver knows who to ask for questions that end in .com, and it knows their addresses. So it picks one (as discussed before), and sends the exact same query it sent before: an iterative (RD bit clear) request for the A record for www.example.com. Now the .com TLD nameserver probably don't know about the insides of example.com either, they have their hands full just telling people about example.com as a whole (and every other domain that ends in .com). More importantly, details about the insides of the example.com domain are administered by the owners of example.com, not by the global administrators of the TLD nameservers. So again the recursive resolver doesn't get the answer to its question, it gets the NS records giving the authoritative nameservers for example.com, along with _their_ glue records.

As an aside at this point, DNS still works if glue records are missing, but it's way slower, since the recursive resolvers have to turn around and go asking for A records for the nameservers --- and it's very very easy to get trapped where "you can't get there from here", since the nameservers for (e.g.) example.com will normally be _within_ example.com, so you can't find their addresses in the normal way. Glue records are important.

So now the recursive resolver knows that the authoritative nameservers for example.com are (e.g., and one typical convention) "a.ns.example.com" and "b.ns.example.com". Another common convention is "ns1.example.com" and "ns2.example.com". The recursive resolver will pick one of them and send its A query one more time, and this time it will be asking authoritative nameserver for example.com for the A record for www.example.com, and it should get its answer back. Now it returns it to the client. Of course when the web browser immediately comes back and asks for the address of pics.example.com to start filling in the pictures, the nameserver doesn't have to go back to the roots, nor does it have to pester the com TLD server again, it remembers who to ask for questions about example.com and sends its query there directly.

---

5. Components of djbdns and how they fit together

5.1. dnscache

The djbdns package was born from a couple of observations. First, that many security problems with BIND come from the way it makes decisions about what answers to trust, and what answers to discard; the "glue records" work because any answer can contain extra information in it, and what's more any answer can be forged. The easiest way to solve these problems was to create a new recursive resolver that applies strict security rules about how who it queries, and what parts of the answers it will use. Thus was born dnscache. It is _only_ a recursive resolver; unlike BIND it never returns authoritative data, and it never returns data that was not retrieved directly from an authoritative nameserver, whose authority it had proven by tracing the chain of NS delegations from its configured roots. To provide simple flexibility it can also be configured with distinct authoritative nameservers to use for specific domains, overriding the search from the roots; this makes it easy to set up "split-horizon" DNS (discussed below). dnscache has configurable options to control the cache size, and to control what interface it listens on.

5.2. tinydns and tinydns-data

tinydns is quite possibly the simplest possible full-function authoritative nameserver. It serves authoritative data only; any query that cannot be answered from its disk database is simply not answered. The disk database has provisions for telling the tinydns that it has comprehensive knowledge of a specific domain; only if that is set (indicated by the presence of a "." record in the data file) does tinydns return NXDOMAIN, the official statement that the requested domain does not exist.

tinydns runs off a very small and efficient database in cdb format. This database has the performance characteristic that when it's in reasonably active use (and so the header block at the beginning of the database remains lodged in cache) any query can be answered by two disk accesses. For the very highest-performance nameservers this is inadequate; however, cdb format is not too bulky to allow provisioning servers to keep the entire working database in memory, either by using an explicit ramdisk, or by simply ensuring that the server uses enough buffer memory to allow the whole database to get cached.

Tinydns's database is arranged such that the query is the key, and the reply is the answer; all the real knowledge of dns structures is in the (offline) program that builds the database, tinydns-data, translating the text file "data" to the binary database "data.cdb". This program reads a clean, lean format that makes it exceedingly simple to encode the commonly-used records; as time progresses more record types have been added to the support. For all records not currently special-cased, there's a "generic" record format, and a preprocessor can be used to produce whatever record is desired in that format. For documentation of this file format, see the tinydns-data documentation.

5.3. axfrdns

tinydns only serves data via UDP port 53. There are two cases where DNS needs service on TCP port 53 as well. One occurs when a reply to a query exceeds 512 bytes. In that case, the reply is truncated, a bit is set indicating the truncation, and if the listener wants the full answer they retry the query via TCP. This is naturally not terribly swift, and by and large it's worth avoiding. As long as you make sure you don't code your data file to produce too much redundant glue, you shouldn't run into this problem unless you're doing something fairly exotic, like a static DNS round-robin for a huge server farm or something. Note that the well-known examples of sites that used to do that, have stopped; giant DNS replies just don't work that well.

The other place where TCP is used in DNS is for zone transfers (AXFR). Zone transfers aren't a terribly secure mechanism for replicating DNS data; it's really advised that you use something like rsync over ssh instead. That really is the best approach to setting up reliable, robust, efficient, and secure replication between tinydns servers. However, when tinydns and other nameservers (like BIND) need to interchange full zones, the only meeting ground is AXFR. djbdns includes both server functionality (in axfrdns) and client AXFR (axfr-get, below). These can be used to arrange zone xfers for replication ("secondary" nameservers), as well as for converting between BIND's zone data format and tinydns-data format and back.

axfrdns is designed to run alongside tinydns, on TCP port 53, serving the same data from the same binary database (data.cdb) file. It runs under the tcpserver program, from the ucspi-tcp package (below).

There are two controls available for restricting what axfrdns will allow. First, tcpserver offers restrictions based on src IP addr, specified using tcprules. The action taken when an IP address matches can be to deny the connection, or to allow it; and if it's allowed, environment variables can be set. That allows it to hook into the second control, the AXFR environment variable, a list of domains for which zone xfers will be permitted, separated by slashes.

Suppose you wanted to allow anyone to connect to axfrdns for retrying queries with large (greater than 512-byte) replies, but you only wish to allow zone xfers from example.dom.

```
=example.dom:allow
:allow,AXFR=""
```

Note that since axfrdns exports tinydns data in zone xfer format, it's (among other things) a conversion tool into zone data format

- just use something like dig to pull the zone xfer into the other format.

5.4. axfr-get

axfr-get is the zone xfer client provided in djbdns. It runs under tcpclient, and writes a data file in tinydns-data format.

It can be used to set up a tinydns server as secondary to a BIND server. It can also be used (with axfrdns publishing) for master-slave relationships between tinydns servers, but it's not recommended for that use; rsync over ssh is preferred in every way, simplicity, performance, security, it wins all over. The output format written by tinydns has as its first and primary goal ensuring that every possible record format is accurately preserved; it falls back to the generic record format whenever it needs to. Creating data that is pleasing and appropriate to hand-maintain is a secondary goal in this program. There are some scripts available at the djbdns home page which can help clean up the data to make it more pleasing to hand-maintain.

5.5. DNS client programs

The djbdns package includes some utilities for performing some of the same jobs that programs like host(1), dig(1), and nslookup(8) are used for.

In the synopses, FQDN means that a fully qualified domain name must be specified there, as distinct from a hostname (i.e. the name specified won't have the local domainname appended to it). The notation is

```
cmdname arguments
    -> output line produced
    possible other comments
```

These are the scripting utilities; useful for testing, but tuned for use as building-blocks for other programs; their output is stripped of the sort of chatter than can assist debugging, it's pared down to make it as simple as possible to use as the input to other programs.

```
dnsip FQDN
    -> ipaddr
dnsipq hostname
    -> FQDN ipaddr
dnsname ipaddr
    -> FQDN
dnsmx FQDN
    -> preference FQDN
    If there's no MX record, dnsmx fakes one with a
    preference of zero --- even if there's no record of any
    sort. dnsmx no-such-host gives "0 no-such-host".
dnstxt FQDN
    -> text record if any associated with FQDN
    blank line if none found
```

Then there are the more pure diagnostic tools; their output is less targeted at program use and more for human analysis, in debugging. In these commands, "type" is a DNS query type, such as "ptr", "a", "txt", "mx", "ns", "soa", or "any".

```
dnsq type FQDN server
    -> debugging query output
    Sends an iterative request to the indicated server; this is
    the preferred tool for testing tinydns.
dnsqr type FQDN
    -> debugging query output
    This sends a recursive request, suitable for testing
    dnscache. If you need to override the default cache
    specified in /etc/resolv.conf, you can set the DNSCACHEIP
    environment variable:
    DNSCACHEIP=x.y.z.t dnsqr type FQDN
dnstrace type FQDN server
    -> debugging trace of a full search
    The server specified here should be a _root_ nameserver;
    dnstrace will launch a search for FQDN starting from that
    root, and will show all possible paths for answering the
    question; this will generally reveal a host of
    evils.
```

5.6. dnsfilter

This program is designed for bulk conversion of things like logfiles; it also makes a terrific stress-tester for a recursive resolver. It reads lines from stdin and writes to stdout. If the first whitespace-separated field on a line looks like an IP addr, dnsfilter tries doing a reverse lookup on it --- for an IP address x.y.z.t, it will try a PTR request for t.z.y.x.in-addr.arpa. If that succeeds, the domain info is appended to the IP addr in the output line. These queries are performed in parallel, cmdline options can let you control the details. See the documentation for more details.

5.7. specialty servers

In principle, the jobs these servers perform, could be done by tinydns, but they are sufficiently specialized that precise targeted code can do a better job of them.

```
pickdns
    As of 1.04, pickdns is no longer needed. The functionality is part of
    tinydns now.
    Load-balancing authoritative server, designed for using DNS
```

```
        to distribute client load over multiple servers; includes
        facilities for directing specific classes of clients to
        specific subsets of a distributed server farm (e.g. to try
        to route users to nearby servers)
    walldns
        serves purely generic authoritative DNS data, concealing
        everything about an area: for all IP addrs x.y.z.t, acts
        like (in tinydns-data notation):
                =t.z.y.x.in-addr.arpa:x.y.z.t
    rbldns
        For serving RBL-like data; replies to A, TXT, or * queries
        for t.z.y.x.$BASE if x.y.z.t is listed in its data file,
        which can specify individual addrs and subnets, as well as
        the replies to send.
```

5.8. daemontools

Daemontools is a companion package, prerequisite to djbdns. It provides some helper programs which assist in launching and managing daemons.

```
    svscan does a vaguely similar job to the System V init: it
        monitors jobs and ensures that they keep running. It's
        normally started with the directory /service, and each
        subdirectory of that directory defines a service; they are
        normally symlinks to directories under /etc, which are in
        turn initially created by various -conf programs
        (dnscache-conf, tinydns-conf, etc).
    svc provides an interactive user interface for controlling
        svscan; it takes an option like "-u" for up, "-d" for down,
        or "-t" to take down and up again, followed by a service
        name in the form of a path to a controlled directory.
        Commonest case might be
                svc -t /service/dnscache
        to restart the dnscache, the simplest way to force it to
        re-acquire data when you don't like what's in its cache.
    supervise is run by svscan to watch over a specific daemon, and
        restart it if it dies
    multilog reads log data from stdin, optionally filters it, and
        deposits the results into one or more logfiles, handling
        rotation automatically.
```

5.9. ucspi-tcp

UCSPI is the UNIX Client-Server Program Interface. It defines a command-line structure and environment variable specifications for inter-process communications helper programs; these make it easier to write clients and servers.

UCSPI-TCP is the specific variety of UCSPI for TCP applications; it specifies more details about specific environment variables and suchlike details.

ucspi-tcp is djb's package implementing UCSPI-TCP.

```
    tcpserver like inetd, only for a single service. An invocation
        of tcpserver will listen for connections on a port, when one
        arrives it will start a client program with args as
        specified on the tcpserver cmdline, and with envars as
        specified by UCSPI-TCP. tcpserver implements access-control
        rules.
    tcprules
        compiles the access control rules for tcpserver into a cdb
        database.
    tcpclient
        A client helper program; does the setup for writing network
        clients for TCP protocols, following the UCSPI-TCP
        specification.
```

6. Example Configurations

Now come an assortment of illustrative examples. These are drawn from individuals' working experience, and are intended more to reflect circumstances that various folks have found common and routine, than to cleanly and precisely illustrate specific techniques or principles.

6.1. Simple setup

So enough theory, let's start now. We won't explain downloading, compiling, installing and adding users here, you can find this information in the Appendix aside to OS specific notes. There are also some alternative Installation methods mentioned. For first, you need to decide which packages you need. Usually you need dnscache to be your resolver and tinydns to be your authoritative nameserver. If you need to support secondaries running BIND, you would also need axfrdns (this is not 100% truth, if you do some lousy tricks and single answers would become greater than 512 bytes, you need axfrdns, too, but this is not common). After installation, you need to configure and fire up the services. This is described below. This depends on using daemontools (and ucspi-tcp if you are using axfrdns) in the default locations.

6.1.1. setting up dnscache

You will need to create two users, call them dnscache and dnslog. Make sure they can't login for security reasons.

Run dnscache-conf:

```
    dnscache-conf dnscache dnslog /etc/dnscache 1.2.3.4
```

where 1.2.3.4 is the IP dnscache should listen on. By default, logfiles live in /etc/dnscache/log/main. I don't like to have anything else than configuration files in /etc, if you want to have your logfiles in /var/log/dnscache, create this directory owned by dnslog and replace ./main in /etc/dnscache/log/run by /var/log/dnscache.

Start dnscache by telling svscan about it:

```
ln -s /etc/dnscache /service
```

dnscache should now fire up within 5 seconds.

If dnscache should be your local resolver only, you're finished here. By default, dnscache does not accept queries from other hosts. If you want to accept queries from 1.2.3.*, simply do

```
touch /etc/dnscache/root/ip/1.2.3
```

You can add or remove networks without telling the running dnscache about it.

6.1.2. Setting up tinydns

You need two users again, tinydns and dnslog. If you previously set up dnscache as described above, dnslog exists.

Run tinydns-conf:

```
tinydns-conf tinydns dnslog /etc/tinydns 1.2.3.5
```

where 1.2.3.5 is the IP tinydns should listen on. Note that you cannot use the same IP you used for dnscache! By default, logfiles live in /etc/tinydns/log/main. I don't like to have anything else than configuration files in /etc, if you want to have your logfiles in /var/log/tinydns, create this directory owned by dnslog and replace ./main in /etc/tinydns/log/run by /var/log/tinydns.

Start tinydns by telling svscan about it:

```
ln -s /etc/tinydns /service
```

tinydns should now fire up within 5 seconds.

You must tell tinydns the hosts it should resolve. Open /etc/tinydns/root/data in you favorite editor and type:

```
#define the authoritative nameserver
.example.com::ns1.example.com
#mail exchanger
@example.com::mail.example.com
#IP for machine1,2,3,4,5
=machine1.example.com:1.2.3.1
=machine2.example.com:1.2.3.2
=machine3.example.com:1.2.3.3
=machine4.example.com:1.2.3.4
=machine5.example.com:1.2.3.5
#machine5 is also known as ns1
+ns1.example.com:1.2.3.5
#machine1 is our mailserver
+mail.example.com:1.2.3.1
#and our webserver
+www.example.com:1.2.3.1
```

After editing, cd to /etc/tinydns/root and run "make". This compiles the data.cdb which tinydns reads.

There are also some scripts in /etc/tinydns/root which can prevent you from editing the data file. We don't think these are too useful because you could only add things, not remove or edit. See Appendix for examples on using them.

6.1.3. Setting up axfrdns

Same procedure as always, create the user axfrdns and do:

```
axfrdns-conf axfrdns dnslog /etc/axfrdns /etc/tinydns 1.2.3.5
```

where 1.2.3.5 is the IP axfrdns should listen on. Normally this must be the same IP tinydns runs on. You must define which servers (IP addresses) may ask you for a zone transfer. If you want to allow 9.8.7.6 zone transfers of example.com and example2.com, add this line to /etc/axfrdns/tcp:

```
9.8.7.6:allow,AXFR="example.com/example2.com"
```

6.2. split horizon

6.3. DNS secondaries

6.3.1. tinydns to tinydns

6.3.1.1. copy the data.cdb

You can just copy the generated data.cdb to the second machine (using rsync over ssh for example). The data.cdb becomes somewhat larger than the data file, so this option is only useful in an local network. Note: data.cdb is architecture independent, so it's even no problem if you are using different OS's. I don't prefer this method. Storing the data file on multiple machines gives me an extra measure of security in case my primary fails. Your mileage may vary...

If you do copy data.cdb, you don't need to do anything on the secondary end to make the new data become live, as tinydns works directly off the disk file and will notice the update immediately. However, you must make sure that the file isn't moved into place until it's completely copied, so either propagate it with rsync, which does the right thing automatically, or script something like

```
    scp data.cdb secondary:/etc/tinydns/root/data.cdb-new && \
    ssh secondary mv /etc/tinydns/root/data.cdb-new /etc/tinydns/root/data.cdb
```

6.3.1.2. using rsync over ssh

This is the method I prefer. It is even usable if your tinydns servers are connected through slow lines, for example at different locations, because only changes get transferred, compression is also done.

```
    rsync -e ssh -az /etc/tinydns/root/data $host:/etc/tinydns/root/data
    ssh $host "cd /etc/tinydns/root; make"
```

where $host is your second tinydns server. NB: both rsync and ssh share a problem, and that is that other programs want to execute them in various contexts --- e.g. rsync wants to execute ssh, and rsync wants to invoke itself via ssh on the remote system. Furthermore, both pkgs default to installing into /usr/local/ which built from sources with the default config, and /usr/local/bin isn't on the default path in many shells. The upshot of all this is that you may need to take the above command and rewrite it as something like

```
        /usr/local/bin/rsync -e /usr/local/bin/ssh \
                --rsync-path=/usr/local/bin/rsync ...
    instead of the simpler
        rsync -e ssh ...
```

For this reason, I generally try and ensure that the paths /usr/bin/rsync and /usr/bin/ssh will work, even if they aren't the default paths installed, and will add symlinks to my systems to make them work if needed.

6.3.2. BIND to tinydns zone transfer

(would be nice to have tinydns reacting on incoming BIND-style notifies with firing up an external script which could do authentication checking and calling axfr-get - anybody volunteering to write a patch?)

(UPDATE: I got a patch, but hadn't time to review and test it yet.)

6.3.3. tinydns to BIND zone transfer

If the secondary is not running tinydns and it is under your control, you should upgrade it to tinydns. If it is not under your control, consider using another secondary ;-)). Ok, enough fun: if you really have to support secondaries running another DNS server, you should implement BIND-style Notifies. These Notifies are telling BIND "hey, the domain xyz has changed, reload it!". Then BIND checks if the serial on your tinydns is higher then the one it has and if it is, BIND will issue an AXFR for this domain (which means that it fetches the data and reloads it). The Notify MUST have your tinydns's IP as source, which makes the task a bit complicated if tinydns's IP is not equal to your machine's primary IP. Note that you must have set up axfrdns to make this working.

The Notify script itself (developed by Jos Backus) is not too complicated:

```
#!/usr/bin/perl -w

# usage: dnsnotify zone slave [...]
# example: dnsnotify example.org 1.2.3.4 1.2.3.5
# requires Net::DNS >= 0.20

use Net::DNS;
use Data::Dumper;
use strict;

my $MY_IP = "a.b.c.d";  # your own IP here

my $zone   = shift;

die "usage: dnsnotify zone slave [...]\n"
  unless defined $zone and @ARGV;

my $res = new Net::DNS::Resolver;
$res->srcaddr($MY_IP);

for my $slave ( @ARGV ) {
  my $packet = new Net::DNS::Packet($zone, "SOA", "IN")
    or die "new Net::DNS::Packet failed\n";

  $packet->header->opcode("NS_NOTIFY_OP");
  $packet->header->aa(1);
  $packet->header->rd(0);

  #$packet->print;

  $res->nameservers($slave);
  print STDERR Dumper($packet);
  my $reply = $res->send($packet);
  if ( defined $reply ) {
    $reply->print;
  } else {
    warn "\n;; TIMED OUT\n";
  }
}

exit 0;
```

There is one change from the version published by Jos: The $res->srcaddr line. It sets the queries source IP so that the BIND on the other side will accept the Notify. With Net::DNS >= 0.20 there is no need to patch any more.

6.4. Bigger sites needing separate files for each zone

6.5. using djbdns in an ISP Environment

6.5.1. Exporting tinydns-data fr om a Relational Database (RDBMS)

There is one main difference between djbdns in small environments and at an ISP: automation. As an ISP, you will probably build your dnsdata from a database. You will have your database to tinydns interface dealing directly with the datafile rather than using the add-ns, add-mx and so on.

The format of this datafile is documented at http://cr.yp.to/djbdns/tinydns-data.html. It looks a bit strange at first because it is not optimized to be readable by humans, but rather is optimized for parsing. This makes our "export from the database" job really easy.

Think of small simple tables, one for the domain data and one for the dns data itself. Let's call them "Domains" and "NSEntry".

Domains could look like this:

```
Domain          varchar         Domainname              example.com
nserver1        varchar         1st Nameserver          ns1.isp.com
nserver2        varchar         2nd Nameserver          ns2.isp.com
nserver3        varchar         3rd Nameserver          ns3.isp.com
nserver4        varchar         1st Nameserver          ns4.isp.com
nserver5        varchar         2nd Nameserver          ns5.isp.com
nserver6        varchar         3rd Nameserver          ns6.isp.com
mx1             varchar         1st Mailserver          mx1.isp.com
mx2             varchar         2nd Mailserver          mx2.isp.com
mx3             varchar         3rd Mailserver          mx3.isp.com
qmtp1           varchar         1st qmtp-Mailserver     qmtp1.isp.com
qmtp2           varchar         2nd qmtp-Mailserver     qmtp2.isp.com
serial          varchar         Serial No.              972244824       (set to time() on every change)
SOAmail         varchar         Mail-addr. SOA          hostmaster.isp.com (replace @ by . !!!)
NS              bit             1=we do dns, 0=no       1
changed         bit             1=changes were made     1               (reset to 0 after building data)
```

and NSEntry is very simple:

```
Domain          varchar         Domainname              example.com
Host            varchar         Hostname                www
Type            varchar         Entry-Type              A
Value           varchar         Entry-Value             1.2.3.4
```

so lets have a look how to get the data from the database to tinydns (using perl):

```
#!/usr/bin/perl

# Takes DNS-Info out of the database and creates tinydns' data-file.
# written Henning Brauer, Hostmaster BSWS, in July 2000
# License: BSD
# update Feb 8, 2001

use DBI;

#connect to your database
$dbh=DBI->connect("DBI:mysql:mydb:myhost", "login", "password") || die "Cannot connect to db server DBI::errstr,\n";;

#look for changes
$sth=$dbh->prepare("SELECT serial FROM Domains WHERE changed=1");
$rv=$sth->execute;
$sth->finish;

if ( $rv > 0 ) {
  print "Rewriting Nameserverdata\n";

  #These values are accepted by DENIC and CORE
  $refresh=10000;
  $retry=3600;
  $expire=604800;
  $min=86400;

  $i=0; $j=0;

  #Backup old File and use the template for the new one
  system("mv /etc/tinydns/root/data /etc/tinydns/root/data.old");
  system("cp /etc/tinydns/root/data.tl /etc/tinydns/root/data");
  open OF, ">> /etc/tinydns/root/data";

  #fetch domain-data
  $sth=$dbh->prepare("SELECT Domain, nserver1, nserver2, nserver3, nserver4, nserver5, nserver6, mx1, mx2, mx3, serial, SOAmail, NS, changed,
  $sth->execute;
  while ( ( $zone, $ns[1], $ns[2], $ns[3], $ns[4], $ns[5], $ns[6], $mx1, $mx2, $mx3, $serial, $mail, $ns, $changed, $qmtp1, $qmtp2, $mtype )
    #support for glue-entries (like ns.bsws.de 213.128.133.188)
    foreach $dns (@ns) {
      $dns=~s/^([^ ]+) .*/$1/;
    }

    #SOA
    print OF "Z$zone\:$ns[1]\:$mail\:$serial\:$refresh\:$retry\:$expire\:$min\:\:\n";
    #Nameserver
    foreach $dns (@ns) {
      if ( $dns ne "" ) { print OF "."; print OF "$zone\:\:$dns\:\:\n"; }
    }

    #MX - supporting MXPS (http://cr.yp.to/proto/mxps.txt) now
    if ( $mx1 ne "" ) { print OF "\@$zone\:\:$mx1\:12816\:\:\n"; }
    if ( $mx2 ne "" ) { print OF "\@$zone\:\:$mx2\:12832\:\:\n"; }
    if ( $mx3 ne "" ) { print OF "\@$zone\:\:$mx3\:12848\:\:\n"; }
    if ( $qmtp1 ne "" ) { print OF "\@$zone\:\:$qmtp1\:12801\:\:\n"; }
    if ( $qmtp2 ne "" ) { print OF "\@$zone\:\:$qmtp2\:12817\:\:\n"; }

    #fetch the entries itself
    $st2=$dbh->prepare("SELECT Host, Type, Value FROM NSentry WHERE Domain=\"$zone\" ORDER BY host");
    $st2->execute;
    while ( ( $host, $typ, $value ) = $st2->fetchrow_array ) {
      #A-Record - Name to IP
      if ( $typ eq "A" ) {
        print OF "+$host\.$zone\:$value\:\:\n";
      }
      #A-Record + PTR for reverse lookup
      if ( $typ eq "AR" ) {
        print OF "=$host\.$zone\:$value\:\:\n";
      }
      #CNAME - bad thing...
      if ( $typ eq "CNAME" ) {
        print OF "\C$host\.$zone\:$value\:\:\n";
      }
    }
    $st2->finish;

    #remember changes and nameservers to send notifies to
    if ( $changed eq 1 ) {
      foreach $dns (@ns) {
        #replace bsws.de you your nameserverdomain
        if ( $dns !~ /.*bsws.de$/ && $dns ne "" ) {
          $chng[$j]=$zone;
          $ntfy[$j++]=$dns;
        }
      }
    }
  }
  $sth->finish;
  close OF;
  chdir("/etc/tinydns/root");
  system("make");

  #note 1

  $dbh->do("UPDATE Domains SET changed=0 WHERE NOT(changed=0)");
}

$dbh->disconnect();
```

So here we are! If you set up tinydns and axfrdns as described earlier, you now have a name server running djb's great software and fetching its data out of a database. It's now easy to add a cronjob which regularly runs the above procedure. It also easy to build some kind of interface to let your customers enter the dns data themselves.

6.5.2. Running additional tinydns servers

For most TLDs, you have to run at least two DNS servers. Running a second tinydns is no problem, you will have two primaries then, no classic primary-secondary scheme. That's good if your primary fails. I would suggest using rsync for transferring the data to the second tinydns server, so just add the following to lines to the above script at '#note 1':

```
    system("/usr/local/bin/rsync -e /usr/bin/ssh -az /etc/tinydns/root/data $host:/etc/tinydns/root/data");
    system("ssh $host \"cd /etc/tinydns/root; make\"");
```

where $host is your second tinydns server. Of course, you could also simply run the complete script on your second tinydns server.

If you need to support secondaries running BIND, use the dnsnotify-script presented earlier above and call it when data changes. For that, just enter the line

```
    for ( $i=0; $i<$j; $i++ ) { print "Notify: $chng[$i] -> $ntfy[$i]\n"; system("/path/to/dns_notify.pl $chng[$i] $ntfy[$i]"); }
```

at our db2tinydns script at '#note 1'.

6.5.3. Conclusion

You have at least one tinydns server fetching its data out of an mysql database, notifying secondary BIND servers and/or syncing its data with additional tinydns servers and everything is done automatically. So now you will have the time to develop a nice Web interface to your database ;-))

The complete script is always available from lwd.bsws.de.

6.6. Private root nameserver

If you have isolated networks (that means networks without Internet connection) and have services running that require a working DNS, then you have to fake the infrastructure found on the Internet. The main point here is that you have to provide the "root servers" that are the starting point for every DNS resolving process. At least you have to provide one server that is authoritative for the root of the DNS namespace. Fortunately this is easy with tinydns. Simply define the nameserver for "." in your data file.

```
  ./add-ns . 1.2.3.1
```

You may specify additional servers if you run multiple private root servers:

```
  ./add-ns . 1.2.3.1
  ./add-ns . 1.2.3.2
  ./add-ns . 1.2.3.3
```

For all the sub zones you define you don't have to specify nameservers if they are served from the same set of nameservers. You may also delegate sub zones to other nameservers if you run a larger organization network that mirrors the distributed nature of DNS. Let's say you use the domain "local" internally and the sales department runs its own nameserver at 1.3.1.1. Additionally you define your central mail hub and the intranet server. Then the above configuration would change to:

```
  ./add-ns . 1.2.3.1
  ./add-ns . 1.2.3.2
  ./add-ns . 1.2.3.3
  ./add-childns sales.local 1.3.1.1
  ./add-mx local 1.2.3.5
  ./add-host intra.local 1.2.3.4
```

To enable resolution through the private root servers you have to tell dnscache about them. This is done by replacing the contents of root/servers/@ in your dnscache directory and /etc/dnsroots.global with the IP addresses of your private root servers - line by line.

6.7. Alternate root nameservers

Eventually you want to be able to resolve alternate toplevel domain names that have not been confirmed by ICANN. This is not a problem - all you have to do is to establish a new list of root servers. You may have a look at the technical hints at The Open Root Server Confederation

6.8. Serving the same data over multiple interfaces

There exists a patch that lets bind tinydns to multiple interfaces. There are also recommendations to set the "IP" environment variable for tinydns to 0.0.0.0. Although these ways may work for you they are not recommended.

The recommended way is to run multiple instances of tinydns, one for each interface that should be served. The "ROOT" environment of all instances is then set to the same value.

This sounds wasteful? Think again. The data.cdb file is readonly memory mapped by tinydns thus giving the operating system the opportunity to share all associated buffers between the tinydns processes. Also the program itself can (and will) be shared by the operating system. So the overhead comes down to some more processes (supervise/tinydns) and the associated memory that doesn't count in todays environments.

The advantage is that you don't have to bind to all interfaces but only to the specified. You get also the ability to control your DNS servers at the different interfaces separately.

Here is an example configuration for three interfaces:

```
 tinydns-conf tinydns dnslog /etc/tinydns1 1.2.1.1
 tinydns-conf tinydns dnslog /etc/tinydns2 1.2.2.1
 tinydns-conf tinydns dnslog /etc/tinydns3 1.2.3.1
 echo "/etc/tinydns1/root" > /etc/tinydns2/env/ROOT
 echo "/etc/tinydns1/root" > /etc/tinydns3/env/ROOT
 ln -s /etc/tinydns[1-3] /service
```

7. Maintenance

7.1. tinydns

The maintenance of tinydns typically reduces to more or less frequent updates to your data file. You may have to optimize your setup over a longer period, for example when using the location feature.

For changes to records tinydns provides a nice feature that enables smooth transitions without manual intervention: timestamps. Have a look at the tinydns-data documentation to see how the ttl/timestamp combination may be used for this. There is one problem with the format of the timestamps: nobody may enter them without computing. There exists a conversion utility that converts ISO 8601 formatted timestamps into the required external TAI64 format. It has to be linked against Dan's libtai library:

```
*
// ---------------------------------------------------
// isotai64.c
// Frank Tegtmeyer <fte@pobox.com>, 2000-02-25
// see http://www.lightwerk.com/djbdns/isotai64.html
//
// input format:  YYYY-MM-DD HH:MM:SS zzzzz
// zzzzz is the timezone (+0200, +0000, -0400, ...)
//
// ---------------------------------------------------
*/

#include <stdio.h>
#include <time.h>
#include "tai.h"
#include "leapsecs.h"
#include "caltime.h"

char line[100];

char x[TAI_PACK];

main()
  {
  struct tai t;
  struct caltime ct;
  int i;

  if (leapsecs_init() == -1)
    printf("unable to init leapsecs\n");

  while (fgets(line,sizeof line,stdin))
    {
    if (!caltime_scan(line,&ct))
      printf("unable to parse\n");
    else
      {
      caltime_tai(&ct,&t);
      tai_pack(x,&t);
      for (i = 0;i < TAI_PACK;++i)
        printf("%2.2x",(unsigned long) (unsigned char) x[i]);
      }
    printf("\n");
    }
  exit(0);
}
```

7.2. dnscache

To get the most benefit from your dnscache you should regularly check if its size is appropriate for your organization. If your cache is too large it's probably not a big problem if you are able to devote the used memory to this task. However if your cache is too small things will get worse than necessary. First your machine running dnscache will become more loaded because dnscache has to request records more frequently. The same is true for the content servers that your dnscache will query. A too small cache also increases bandwidth use on your Internet connection. All this means too that your users have to wait longer for the results of their queries.

How to adjust the cache size is described in the FAQ. See also the section "dnscache memory use" in this document.

---

8. Comparison with BIND, migration issues

8.1. Migration

In short:

```
  -install djbdns
  -run axfr-get against your BIND
  -stop bind
  -start tinydns
  -go have a beer
```

8.1.1. Conversion of your zone files

8.1.2. If you are primary server for a BIND server

8.1.2.1. Adjusting serials

If you don't want to use Z records in your tinydns configuration because it's much easier to use the defaults provided by tinydns-data you should have a look at your serials before switching to tinydns. Secondary nameservers running BIND (and probably others too) rely on the AXFR mechanism to keep their content in sync with your primary. The decision if data has to be updated relies on the so called "serial number" inside the SOA record. Zone transfers are only done if the serial of the primary server is higher than the one of the secondary. Because the values of the serials are represented by a positive integer value with 32 bits its value range is 0 to 2^32-1 (4294967295). To avoid problems when reaching the upper limit a comparison of serials is defined within "sequence space arithmetic" that is explained in RFC1982, and more clearly in How the AXFR protocol works.

Tinydns uses the modification time of the data file and sets the serial to the (decimal) number of seconds since the Unix epoch. This leads to values with nine numbers at this time.

One of the common formats for serials on BIND systems is to use the four digits of the year, two for the month, two for the day and one or two for an incremented number of the last change to the zone. For the fifth change on 2001-02-14 you would get "200102144" or "2001021404". The nine digits format is no problem because the serial that tinydns-data creates is higher than the one formed by this format. Therefor zone transfers will continue after a switch to tinydns.

Problematic is the ten digits format. Serials with this format are higher than the ones tinydns uses. To let the zone transfers continue after a switch to tinydns you have to adjust the serials before that.

To make the transition, you need to force all slave servers to wrap. You do this by temporarily forcing the serial to a value that is both greater than BIND's YYYYMMDD##, and less than tinydns-data's time_t, according to the rules of SOA serial arithmetic. There is a range of such values. The center of the range can be computed very simply:

(YYYYMMDDD## + time_t) / 2 + 2**31

which works until the average of YYYYMMDD## and time_t exceeds 2**31 (when you need to take the result modulo 2**32). But that doesn't happen until after time_t exceeds 2**31, in 2038, and time_t will pass YYYYMMDD## and make this problem disappear before then, in 2034. So, in short, you can compute the new serial with

echo `date +%Y%m%d00\ %s` + 2 / 2 31 '^' + p | dc

Finally we get the following steps:

```
   1. edit data to use a Z record, with an explicit serial
      as computed above; publish the results.
   2. Wait for the zone transfers by the secondaries (refresh in the
      SOA, which tinydns-data defaults to c. 4.5 hours)
   3. Check (all) the secondaries with one of the following commands
      dnsq soa <domain> <ip of the secondary>
      dig @<ip of the secondary> <domain> soa
      nslookup -type=soa <domain> <ip of the secondary>
   4. Edit the explicit serial off the Z line, or switch to using
      the simpler "." line; no need to publish this edit, it can
      wait for the next substantive change of content in the zone
      data.
```

You may get additional information about serials.

8.1.2.2. IPs per Hostname returned

In djbdns-1.04 and above, tinydns never returns more than 8 address records for a given hostname. If you need compatability with the traditional behaviour then you'll need to run djbdns-1.03, or to patch tdlookup.c. (Both of which void the warranty.)

---

9. Testing DNS configs, diagnosing problems

(note: comment on why dnsq and dnsqr are preferable to nslookup)

---

10. Performance

10.1. dnscache memory use

dnscache allocates its needed cache memory at startup. You should not see the dnscache process growing. You may set the CACHESIZE parameter to any value you want - the official FAQ explains how the measure its effectiveness. For anything that goes beyond "normal" use of one single workstation thinking about increasing CACHESIZE might be a good idea.

The DATALIMIT environment variable used in the standard run script is a security mechanism. It's used to keep the process from running wild in the improbable case that there may be any logical or technical bugs that let dnscache start growing. The FAQ gives an example how to increase the CACHESIZE to 100MB. If you want to set another value (2MB my be enough for a small company with mail and web access through a 64k line) you have to decide how to set DATALIMIT.

Although nobody has come up with a formula for computing the value here are three tips from the mailinglist:

"The smallest value that works. dnscache will allocate space for its cache at startup. If DATALIMIT is too small, the allocation will fail. After the initial allocation, you don't want dnscache to grow any further; DATALIMIT prevents it from doing so. Such growth would happen only as the result of a bug, so it's unlikely; DATALIMIT is an extra, just-in-case protective measure. (Paul Jarc in this message)

Dan says: "If you want to experiment with a larger cache, make sure to keep the -d value a couple of megabytes above the cache size."

Uwe Ohse: "the "-d" value is "somewhat" larger than the CACHESIZE simply because dnscache needs "some" amount of memory beside the cache."

To adjust the cache size you have to monitor the "stats" lines that dnscache puts into its logfile. Markus Stumpf creates a separate file for this using the following multilog invention:

```
exec setuidgid dnslog multilog t s250000 n20 ./main '-*' +'* stats * * *' =./dnsstatus
```

A. Installing packages

In general it is a good idea to install djbdns from the source instead of using packages. Once you've done it and fully understand djbdns, packages may make your life easier.

A.1. Manual source install

A.1.1. daemontools

Get daemontools from http://cr.yp.to/daemontools/install.html. Installation is really easy:

```
    mkdir -p /package
    chmod 1755 /package
    cd /package
    gunzip /path/to/daemontools-0.76.tar.gz
    tar -xf /path/to/daemontools-0.76.tar
    rm /path/to/daemontools-0.76.tar
    cd admin/daemontools-0.76
    package/install
```

A.1.1.1. Linux

A.1.1.2. *BSD

Note: This is not needed for daemontools >= 0.75

On *BSD, you would add these lines to your /etc/rc.local:

```
  echo -n "daemontools "
  PATH=$PATH:/usr/local/bin
  svscan /service &
```

A.1.1.3. Solaris

A.1.2. ucspi-tcp

If you want to run axfrdns or axfr-get, you need ucspi-tcp from http://cr.yp.to/ucspi-tcp/install.html. Installation is once more really simple:

```
    gunzip ucspi-tcp-0.88.tar.gz
    tar -xf ucspi-tcp-0.88.tar
    cd ucspi-0.88
    make
```

Become root once more for the installation:

```
    make setup check
```

That's all. No configuration needed.

A.1.3. Install djbdns

Get djbdns from http://cr.yp.to/djbdns/install.html. The installation process is once more so easy:

```
    gunzip djbdns-1.05.tar.gz
    tar -xf djbdns-1.05.tar
    cd djbdns-1.05
    make
```

Wonder, wonder, become root for installation:

```
    make setup check
```

A.2. Using RPMs

This one is by Bennett Todd; I work with Red Hat Linux most of the time, and Solaris occasionally, so I'll illustrate the installation procedure using the tools I favour; they include init scripts and rpm packaging which are in rpm-packaging.tar.gz. The contained spec files will build rpms from the sources (URLs for downloading the sources from djb's site included in the spec files). They also include documentation, which I construct by mirroring his website with ftpcopy, then packaging them up, with commands something like

```
        for pkg in djbdns daemontools ucspi-tcp;do
                tar cf - --exclude=\*.gz $pkg* | bzip2 \
                    > /usr/src/redhat/SOURCES/${pkg}-docs.tar.bz2
        done
```

So, for most machines, I run dnscache to work as a local resolver; that's straightforward:

```
        rpm -i daemontools-0.76.i386.rpm
        rpm -i djbdns-1.05.i386.rpm
        useradd -d /no/home -s /no/shell dnscache
        useradd -d /no/home -s /no/shell dnslog
        dnscache-conf dnscache dnslog /etc/dnscache
        ln -s /etc/dnscache /service
        $EDITOR /etc/resolv.conf # set nameserver to 127.0.0.1
```

A.3. Using ports on OpenBSD

All software from Dan Bernstein has been removed from the OpenBSD ports due to conflicts with his license (i.e. there is no license). But you still can install djbdns quite easily by following the original documentation. You only need to make sure to changed the softlimit in /etc/tinydns/run from

- d30000 to -d40000, otherwise tinydns won't start and you'll get logentries like this:

```
        /usr/libexec/ld.so: tinydns: libc.so.29.0: Cannot allocate memory
```

A.4. Using Ports on FreeBSD

The Procedure on FreeBSD is basically the same as on OpenBSD, assuming you have first installed the ports tree.

```
  su
  cd /usr/ports/sysutils/ucspi-tcp
  make install
  cd /usr/ports/sysutils/daemontools
  make install
  cd /usr/ports/net/djbdns
  make install
```

This will download, compile and install djbdns, daemontools and ucspi-tcp for you. There is a package created in /usr/ports/packages/ you can use to install djbdns on other machines using pkg_add.

A.5. Using ? on Solaris

A.6. SysV-style init script

Note: For use with daemontools >= 0.75 you'll need to adjust pathes

On SysV systems, you would like to have a SysV-style init script perhaps. Here is one:

```
  #!/bin/sh
  #
  # svscan        Start and stop the svscan daemon.
  #
  # description: svscan is a general-purpose service manager, handling \
  #              execution of daemons, and ensuring their operation.
  # processname: svscan

  # Source function library.
  . /etc/rc.d/init.d/functions

  SERVICESDIR='/usr/local/service'
  BINDIR='/usr/local/bin'
  LOCKDIR='/var/lock/subsys'
  PATH=$PATH:$BINDIR

  [ -f $BINDIR/svscan ] || exit 0
  [ -d $SERVICESDIR ] || exit 0

  RETVAL=0

  # See how we were called.
  case "$1" in
    start)
          # Start svscan.
          action 'Starting svscan:' "/bin/sh -c '( cd $SERVICESDIR && exec $BINDIR/svscan ) &'"
          RETVAL=$?
          [ $RETVAL -eq 0 ] && touch $LOCKDIR/svscan
          ;;
    stop)
          # Stop svscan.
          echo -n "Shutting down svscan:"
```

```
        killproc svscan
        RETVAL=$?
        echo
        [ $RETVAL -eq 0 ] && rm -f $LOCKDIR/svscan

        # Stop supervised services. Do this last, so that
        # svscan doesn't restart them.
        x="$allow_null_glob_expansion"
        allow_null_glob_expansion=1
        for service in $SERVICESDIR/*
        do
              $BINDIR/svok $service || continue
              action "Ending supervision of service `basename $service`:" "$BINDIR/svc -x $service"
              [ -k $service ] && action "Ending supervision of service log `basename $service`:" "$BINDIR/svc -x $service/log"
              action "Stopping service `basename $service`:" "$BINDIR/svc -d $service"
              [ -k $service ] && action "Stopping service log `basename $service`:" "$BINDIR/svc -d $service/log"
        done
        allow_null_glob_expansion="$x"
        ;;
  status)
        status svscan
        RETVAL=$?
        x="$allow_null_glob_expansion"
        allow_null_glob_expansion=1
        for service in $SERVICESDIR/*
        do
                $BINDIR/svstat $service
        done
        allow_null_glob_expansion="$x"
        ;;
  restart|reload)
        $0 stop
        $0 start
        RETVAL=$?
        ;;
  *)
        echo "Usage: svscan {start|stop|restart|reload|status}"
        exit 1
esac

exit $RETVAL
```

B. using the add-* scripts to create tinydns data file

```
cd /etc/tinydns/root
./add-host machine1.example.com 1.2.3.1
./add-host machine2.example.com 1.2.3.2
./add-host machine3.example.com 1.2.3.3
```

If example.com is delegated to your tinydns server correct, dnsip machine1.example.com should show you 1.2.3.1 now. Note that example.com must be delegated to a.ns.example.com with your IP to make this working. If you want machine1 to be reachable as www.example.com too, simply do

```
cd /etc/tinydns/root
./add-alias www.example.com 1.2.3.1
```

So let's define machine2 being our mailserver for example.com:

```
cd /etc/tinydns/root
./add-mx example.com 1.2.3.1
```

Tinydns will name the mailserver a.mx.example.com. You could also ad additional nameservers:

```
cd /etc/tinydns/root
./add-ns example.com 1.2.3.3
./add-ns 3.2.1.in-addr.arpa 1.2.3.3
```

Tinydns will name the second server b.ns.example.com.

C. useful Links

Setting up djbdns for use on dial-up connections by Matthias Andree <matthias.andree@gmx.de> Using the dumpcache-patch you can save your dnscache cache entries to disk and reload them. Michael Bacarella has a small script called dnspkt checking your server for answers exceeding 512 bytes and thus needing axfrdns running.

D. Acknowledgements

Bennett Todd <bet@rahul.net> for starting this great document, his amount of work for it and his great view for straight-through structuring. Thank you, Bennett. It was real fun to work with you.

Dave Sill for support and encouragement, and for letting me call this Life With djbdns in frank homage to his Life With qmail.

Joost van Baal <joostvb@xs4all.nl> for improving the description of an NS record.

Henning Brauer <hostmaster@bsws.de> for the section on ISP Design (exporting from RDBMS to tinydns-data, and doing Notify for BIND secondaries), the installation instructions and parts of appendix a.

Florent Guillaume <Florent.Guillaume@mail.com> for two changes to the dnsnotify script to let the notify be RFC 1996 conformant.

Jos Backus <josb@cncdsl.com> for an always up-to-date dnsnotify.pl script.

Clemens <rabat@web.de> for the FreeBSD installation notes.

Richard Lucassen <spam1@lucasson.org> for the SysV-style init script.

David Pick <D.M.Pick@qmw.ac.uk> for additions to the FreeBSD installation notes.

Frank Tegtmeyer <fte@fte.to> for the section about private root servers, upgrading from bind and various other stuff I can't count anymore ;-))

Bob <rd@thrush.com> for correcting various spelling errors.

Scott Schwartz <schwartz@bio.cse.psu.edu> for the IPs per hostname hint.

Bernhard Graf <fisch@augensalat.de> for an updated Net::DNS::Resolver patch and an updated dnsnotify script.

Fridtjof Busse <fridtjof@fbunet.de> for updated OpenBSD instructions

$Id: lwd.sdf,v 1.6 2001/09/26 07:42:47 brahe Exp $