UDP shell scripts with inetd

Table of Contents

1. About the author 2
2. Introduction 2
3. UDP 2
4. shdns 11
5. Getting the code from this article 16

1. About the author

Michael has been working in the image processing field for several years, including a couple of years managing and developing large image databases for an Australian government department. He currently works for TOWER Software, who manufacture a world leading EDMS and Records Management package named TRIM. Michael is also the developer of Panda, an open source PDF generation API, as well as a bunch of other Open Source code.

Michael has a website at http://www.stillhq.com.

2. Introduction

This article is about two things. The main focus of the article is to discuss how to write useful UDP servers in a common scripting language such as bash. The other, more minor, focus of the article is to give a brief tutorial on the differences between disconnected and connected sockets.

In this article I assume that you have a working knowledge of both bash scripting and C programming. If you don't then hopefully you'll still get something out of the article, but you might have to skip bits which are too technical.

3. UDP

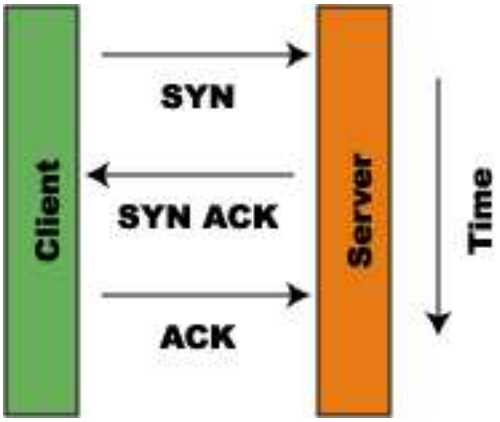
It strikes me as logical to start this article with an extremely brief introduction to UDP. At some points it makes sense to compare this with the alternative, TCP.

Both UDP and TCP sit on top of the Internet protocol, which handles all the plumbing of actually getting the data out of the back of the client machine to the server. This is where the similarity ends. UDP is the *User Datagram Protocol*, and is unreliable. On the other hand, TCP is the *Transmission Control Protocol*, and is reliable.

What is reliability? Well, TCP will hold your hand and ensure for you that every packet you send is recieved by the other machine. It will also ensure that the packets arrive in the right order. There are also some games played with the choice of initial sequence number to make it harder for man in the middle attacks to be successful.

UDP is unreliable, which means it does none of this for you. It is the programmer's responsibility to ensure that all the packets sent arrived, and were in the right order.

So why would you ever use UDP? Well, all this reliability in TCP comes at a price. That price is performance. Before a TCP connection is estabilished, the following protocol sequence occurs.



So before the two machines can even communicate, they've spent a round trip just setting up the connection (the ACK can be sent at the same time as the first packet, as we don't have to wait for an ACK ACK to come back).

UDP, on the other hand, does none of this. A single packet is sent, and it either arrives or it doesn't. Normally the client application will note that a reply was never received after a given timeout, and retransmit the packet.

A good example of a common network protocol that uses UDP is the Domain Name System (DNS). UDP is well suited here because the packets are short (they fit in a single datagram) and we therefore don't have to worry about our packets arriving out of order. We also want DNS to be as fast as possible.

TODO: Does the DNS RFC specify why UDP is used?

TODO: is there more processor load associated with all the extra processing required to make TCP reliable?

For more information on UDP, and IP in general, I recommend: TODO

3.1. Disconnected sockets

By default when a UDP socket is created, it is disconnected. What's a disconnected socket? The short answer is that it isn't associated with any given remote machine. This means that each time we fetch data from the socket we have to use the *recvfrom*(2) function call:

```
int recvfrom(int s, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

The *struct sockaddr* arguement in this function call is populated with enough information for the program to be able to determine where to send the reply packet. The reponse needs to be sent with the *sendto*(2) call:

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

The *sendto* function call takes another *struct sockaddr* arguement which specifies where the packet should be sent.

The code listing below is an example of how to use the *recvfrom* and *sendto* functions. This example is a simple UDP echo server. The program waits on a given port, and whenever it receives data, sends it straight back.

```
// Disconnected UDP socket example: this example simply reads from
// clients (there can be more than one), and returns what they
// said straight back to them. You'll note that we can't use read
// and write to get to the traffic, as this is not available for
// disconnected UDP sockets.

#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char *argv[]){
  int lfd;
  struct sockaddr_in servaddr;
  struct sockaddr clientaddr;
  char buf[1024];
  size_t len;
  socklen_t clen;

  // We will listen with this file descriptor
  if((lfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    fprintf(stderr, "Error whilst starting to listen\n");
    exit(42);
  }

  // Define what we are listening for
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(1234);

  // Bind to the address
  if(bind(lfd, (struct sockaddr *) &servaddr, sizeof(servaddr))
```

```
     < 0){
    perror("Couldn't bind");
    exit(42);
  }

  // Do stuff
  while(1){
    len = 1024;
    printf("Reading...\n");
    clen = sizeof(clientaddr);
    if((len = recvfrom(lfd, buf, len, 0,
              (struct sockaddr *) &clientaddr,
              &clen)) < 0){
      perror("Socket read error");
      exit(42);
    }
    if(len == 0) break;

    // The buffer is not null terminated
    buf[len] = '\0';
    printf("Read: %s\n", buf);

    // And send it straight back
    if(sendto(lfd, buf, len, 0, &clientaddr, clen) < 0){
      perror("Socket write error");
      exit(42);
    }
  }
}
```

Code: disconnected.c

Let's have a look at this program running. As we can see from the source code, the program listens on UDP port 1234 (very imaginative). We can use **netcat** to test the program. First, we need to start the server in a different terminal, this is as simple as running it.

netcat rocks

netcat rocks. It's a little application which lets you push data to arbitary port numbers using both UDP and TCP. It can also be used to create very simple servers, because **netcat** can do all the listening for you. For more information, and download details, checkout http://www.atstake.com/research/tools/network_utilities/.

A client interaction with the server looks like this:

```
[mikal@localhost article]# nc -u localhost 1234
hello
hello
out
out
there
there
 punt!
[mikal@localhost article]#
```

Note that the "punt!" is me hitting control c on **netcat**. The server output for this session is:

```
[root@localhost sockets]# ./disconnected
Reading...
Read: hello

Reading...
Read: out

Reading...
Read: there

Reading...
```

There are a couple of things worth noting here. Firstly, the UDP server can handle multiple clients at once in the single process. This is because each packet comes in, and is then responded to — there is no assumption made that packets come from the same machine. This model works well for the single packet communication paradigm. Secondly, because the server simply waits for a packet, responds, and then starts waiting again, the server survives disconnects from the client, it simply waits until a new packet comes along.

3.2. Disconnected sockets and scripting

This article is really about writing useful UDP servers using scripting languages — specifically *bash*. In this instance, the disconnected nature of these default UDP sockets causes great pain. This is because there is no trivial way in bash to call *recvfrom* and *sendto*. Shell scripts really want to be able to use the standard *read* and *write* calls, because these languages are really intended for file and pipe input output, as opposed to socket traffic.

We can demonstrate this by writing a very simple, and probably quite impractical **inetd** server.

What is inetd?

inetd, and it's fairly common **xinetd** alternative are super daemons. Their role in the networking food chain is as a way of running programs to process network traffic as demand requires. For example, many **rsync** servers operate from **inetd**. What this means is that **inetd** runs in the background as a daemon waiting for connections from clients to the **rsync** port. It then starts a new copy of **rsync** for each of these connections, and **rsync** processes the traffic (and perhaps responds).

Many developers write network servers which are intended to be run by **inetd** because it simplifies development of the application. **inetd** handles most of the plumbing for the application.

Please bear in mind that **inetd** and **xinetd** are very configurable, and this is only a general description. You should refer to the relevant documentation for more information.

This is a particularly useful example, because our shell scripts are eventually going to run from **inetd**, so having an understanding of how it works is very useful...

```
// Disconnected UDP socket example: this example simply waits for
// traffic, and the starts a process to deal with the results. One
// process per packet, one packet per process. This version wont
// work, because the socket is not connected. In fact, cat is
// smart enough to warn us about this:
//
//           cat: write error: Transport endpoint is not connected

#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/poll.h>
#include <netinet/in.h>

int main(int argc, char *argv[]){
  int lfd;
  struct sockaddr_in servaddr;
  struct pollfd pfd;

  // We will listen with this file descriptor
  if((lfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    fprintf(stderr, "Error whilst starting to listen\n");
    exit(42);
  }

  // Define what we are listening for
  bzero(&servaddr, sizeof(servaddr));
```

```
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(1234);

  // Bind to the address
  if(bind(lfd, (struct sockaddr *) &servaddr, sizeof(servaddr))
     < 0){
    perror("Couldn't bind");
    exit(42);
  }

  // Setup the list of file descriptors we want to wait for
  // events on
  pfd.fd = lfd;
  pfd.events = POLLIN | POLLPRI;

  // Do stuff
  while(1){
    if(poll(&pfd, 1, -1) < 0){
      perror("Waiting for new data failed");
      exit(42);
    }

    printf("Data arrived\n");

    // Spawn a child to handle this packet
    switch(fork()){
    case -1:
      perror("Couldn't spawn child to handle connection");
      exit(42);

    case 0:
      // Child process -- setup the file descriptors, and the run
      // the helper application
      dup2(lfd, 0);
      dup2(lfd, 1);

      execl("/bin/cat", "cat", NULL);
      perror("Exec failed");
      exit(42);
      break;

    default:
      // Parent process
      break;
    }
  }
}
```

Code: disconnexec.c

All this program does is wait on port 1234 until a connection comes in, forks, and the new child process starts executing the server program (in this case **cat**). This technique relies on the fact that the child process will share the environment of the parent process, including it's filehandles. These are duplicated to stdin and stdout before the server program starts executing, so that network input goes into stdin, and server output is sent back over the network.

This of course doesn't work because the socket is disconnected, so **cat**'s *read* call will work, but it's *write* call fails because the socket layer doesn't know where to send the response to.

3.3. Connected sockets

The answer to our need to use *read* and *write* for our shell scripts is a connected socket. A connected socket is simply a socket which has had the *connect*(2) function called upon it.

When the *connect* function is called, it simply records which machine the network packet came from, so that the socket layer knows where to send the reply packet when it is output with the *write* function call.

Here's a simple example of a connected socket version of the disconnected echo server above. Note that we now use *read* and *write* function calls for all the network input output.

```
// Connected UDP socket example: this example simply reads from
// clients (there can be more than one), and returns what they
// said straight back to them. You'll note that we can now use
// read and write to get to the traffic...

#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char *argv[]){
  int lfd;
  struct sockaddr_in servaddr;
  struct sockaddr clientaddr;
  char buf[1024];
  size_t len;
  socklen_t clen;

  // We will listen with this file descriptor
  if((lfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    fprintf(stderr, "Error whilst starting to listen\n");
    exit(42);
  }

  // Define what we are listening for
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(1234);

  // Bind to the address
  if(bind(lfd, (struct sockaddr *) &servaddr, sizeof(servaddr))
     < 0){
    perror("Couldn't bind");
    exit(42);
  }

  // Do stuff
  while(1){
    // We need to peek at the first part of the packet to
    // determine who to connect to
    len = 1;
    printf("Reading...\n");
    clen = sizeof(clientaddr);
    if((len = recvfrom(lfd, buf, len, MSG_PEEK,
               (struct sockaddr *) &clientaddr,
               &clen)) < 0){
      perror("Socket peek error");
      exit(42);
    }
    if(len == 0) break;

    // Connect
    if(connect(lfd, &clientaddr, clen) < 0){
      perror("Could not connect");
      exit(42);
    }

    // And now we can just use the normal read and write
    len = 1024;
    if((len = read(lfd, buf, len)) < 0){
      perror("Socket read error");
      exit(42);
    }
    if(write(lfd, buf, len) < 0){
      perror("Socket write error");
```

```
      exit(42);
    }
  }
}
```

Code: connected.c

We do cheat a little in this example, there is one *recvfrom* call. This is used to "peek" at the data which is waiting on the socket to determine the address to connect to. The *MSG_PEEK* means that the data is not actually removed from the queue of data to be processed by this call.

I wont include an example of what this program looks like when it runs, because it looks exactly the same as the disconnected echo server above.

3.4. Back to our simple inetd server

You'll recall that a couple of examples ago I showed you the code for a simple **inetd** server. This example below expands that so that now it uses a connected socket. This example will behave just like the hard coded echo server in the first and third examples in this article (except that the server debugging output doesn't happen any more).

```
// Connected UDP socket example: this example this example simply
//waits for traffic, and the starts a process to deal with the
// results. One process per packet, one packet per process. You'll
// note that we can now use read and write to get to the traffic,
// and that this all works...

#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char *argv[]){
  int lfd;
  struct sockaddr_in servaddr;
  struct sockaddr clientaddr;
  char buf[1024];
  size_t len;
  socklen_t clen;

  // We will listen with this file descriptor
  if((lfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    fprintf(stderr, "Error whilst starting to listen\n");
    exit(42);
  }

  // Define what we are listening for
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(1234);

  // Bind to the address
  if(bind(lfd, (struct sockaddr *) &servaddr, sizeof(servaddr))
     < 0){
    perror("Couldn't bind");
    exit(42);
  }

  // Do stuff
  while(1){
    // We need to peek at the first part of the packet to
    // determine who to connect to
    len = 1;
    printf("Reading...\n");
    clen = sizeof(clientaddr);
```

```
    if((len = recvfrom(lfd, buf, len, MSG_PEEK,
               (struct sockaddr *) &clientaddr,
               &clen)) < 0){
      perror("Socket peek error");
      exit(42);
    }
    if(len == 0) break;

    // Connect
    if(connect(lfd, &clientaddr, clen) < 0){
      perror("Could not connect");
      exit(42);
    }

    printf("Data arrived\n");

    // Spawn a child to handle this packet
    switch(fork()){
    case -1:
      perror("Couldn't spawn child to handle connection");
      exit(42);

    case 0:
      // Child process -- setup the file descriptors, and the run
      // the helper application
      dup2(lfd, 0);
      dup2(lfd, 1);

      execl("/bin/cat", "cat", NULL);
      perror("Exec failed");
      exit(42);
      break;

    default:
      // Parent process
      break;
    }
  }
}
```

Code: connexec.c

3.5. inetd and xinetd patches

The two standard inetd implementations that most people use are called **inetd** and **xinetd**. Neither of these implements a connected socket for the server before starting it, which means that it is effectively impossible to write a standard shell script which processes UDP network traffic.

It therefore seems logical to me to include some patches to make **inetd** and **xinetd** behave in the manner that I would like them to. Afterall, surely if your goal is as noble as implementing a DNS server in shell script, then surely the operating system should be modified to accomodate that?

3.5.1. inetd patch

The package which contains **inetd** as run on Debian is called *netkit*. The following is a patch to use connected sockets for UDP and TCP servers.

```
diff -ur netkit-base-0.16/inetd/inetd.c netkit-base-0.16-hacked/inetd/inetd.c
--- netkit-base-0.16/inetd/inetd.c    Wed Nov 24 06:31:53 1999
+++ netkit-base-0.16-hacked/inetd/inetd.c    Thu Jan 23 13:42:59 2003
@@ -470,7 +470,30 @@
             return;
         }
         if (pid==0) {
+                char buf[2];
+            size_t len = 1;
```

```
+            struct sockaddr clientaddr;
+            socklen_t clen;
+
             /* child */
+            len = sizeof(clientaddr);
+            if ((len = recvfrom(ctrl, buf, len, MSG_PEEK,
+                       (struct sockaddr *) &clientaddr,
+                       &clen)) < 0) {
+                  syslog(LOG_WARNING, "failed to peek for (for %s): %m",
+                     sep->se_service);
+            }
+            if (len==0) {
+                  syslog(LOG_WARNING, "no data (for %s): %m",
+                     sep->se_service);
+            }
+
+            // Connect
+            if (connect(ctrl, &clientaddr, clen) < 0) {
+                  syslog(LOG_WARNING, "connect failed (for %s): %m",
+                     sep->se_service);
+            }
+
             dup2(ctrl, 0);
             close(ctrl);
             dup2(0, 1);
```

Code: netkit.patch

A simple example of how to configure **inetd** to handle our UDP echo server is shown below — the configuration file this line is added to is /etc/inetd.conf. It should be noted that *myecho* is the name of a service I had to add to /etc/services because **inetd** insists on all services being listed there...

```
# An example UDP echo server
myecho    dgram   udp     nowait    root    /bin/cat cat -
```

This service has the following characteristics:

- The service name is *myecho*. The listing in /etc/services is:

  ```
  myecho 1234/udp # Simple echo example for LJ article
  ```

  Why not echo?

  Note that we can't use the name *echo* for our service because of a couple of reasons: it is already defined in /etc/services; it is an internal service which **inetd** already offers using the normal port number; and we wanted to define a different port number for the service, without b reaking things which might use the old /etc/services entry.

- The socket type is *dgram*, which is short for datagram, the type of packet that the UDP protocol uses.
- The protocol is UDP.
- *nowait* is used to indicate that one server should be started per packet, instead of waiting for the previous server to exit. This arguement should always be nowait for TCP servers.
- The server runs as root.
- The server is located at /bin/cat
- The server should be started with these arguements. Note that the name of the executable is always the first arguement. Checkout *execve*(2) for more details.

3.5.2. xinetd patch

xinetd is the **inetd** implementation shipped by Red Hat. It is much more complex than that implemented by *netkit*.

TODO PATCH

4. shdns

TODO

4.1. The code

TODO

```
#!/bin/bash

logger "shdns $$ Started listening"
/home/mikal/opensource-unstable/shdns/shdns /tmp/shdns-$$ &
cat - > /tmp/shdns-$$
logger "shdns $$ Waiting for processor to end"
wait
logger "shdns $$ Stopped listening"
```

Code: shdns-server

TODO

```
#!/bin/bash

# shdns: take a query and build a response

################################################################################
# Utility functions
################################################################################

# The state of a given bit in the byte: (byte, poweroftwo)
dumpbit(){
  local temp
  temp=$1

  if [ $1 -gt $(( $2 - 1 )) ]
  then
    echo -n "1"
    temp=$(( $1 - $2 ))
  else
    echo -n "0"
  fi

  return $temp
}

# The state of the whole byte: (byte)
dumpbyte(){
  dumpbit $1 128
  dumpbit $? 64
  dumpbit $? 32
  dumpbit $? 16
  dumpbit $? 8
  dumpbit $? 4
  dumpbit $? 2
  dumpbit $? 1
}

# Is a given bit on? (byte, poweroftwo)
testbit(){
  return ‘dumpbit $1 $2‘
```

```
}

# Turn on a given bit in the byte: (initial byte state, poweroftwo, desiredstate)
# Returns a decimal version of the byte
twiddlebit(){
  local temp
  temp=$1

  testbit $1 $2
  if [ $? = 1 ]
  then
    if [ $3 = 0 ]
    then
      temp=$(( $temp - $2 ))
    fi
  else
    if [ $3 = 1 ]
    then
      temp=$(( $temp + $2 ))
    fi
  fi

  return $temp
}

# Spin until a byte exists: (filename, bytenumber)
spinfor(){
  local len

  len=‘cat $1 | wc -c | tr -d " "‘
  while [ $len -lt $2 ]
  do
    logger "shdns $$ Spin on byte $2"
    usleep $3
    len=‘cat $1 | wc -c | tr -d " "‘
  done
}

# Read a single byte from a file: (filename, bytenumber)
readbyte(){
  spinfor $1 $2 10
  cat $1 | cut -b $2
}

# Read a range of bytes from a file: (filename, startbyte, length)
readstring(){
  spinfor $1 $(( $2 + $3 - 1 )) 10
  logger "shdns $$ Getting byte range $1:$2-"$(( $2 + $3 - 1 ))":$3"
  cat $1 | cut -b $2-$(( $2 + $3 - 1 ))
}

# Read a single binary byte as decimal from a file: (filename, bytenumber)
readbytebinary(){
  local temp

  spinfor $1 $2 1000
  temp=‘cat $1 | cut -b $2 | od -Ad -c | head -1 | tr -s " " | cut -f 2 -d " "‘
  if [ ‘echo $temp | cut -b 1‘ = "\\" ]
  then
    case ‘echo $temp | cut -b 2‘ in
    0 ) temp="0";;
    a ) temp="7";;
    b ) temp="8";;
    t ) temp="9";;
    n ) temp="10";;
    v ) temp="11";;
    f ) temp="12";;
    \\ ) temp="92";;
    * ) echo "Error: Unknown escape binary sequence"; exit;;
    esac
  fi
```

```
  return $temp
}

# Output the bit for this value: (inputvalue, byteoffset)
writebinarybit(){
  if [ $1 -gt $(( $2 - 1 )) ]
  then
    echo -n "1"
    return $(( $1 - $2 ))
  else
    echo -n "0"
    return $1
  fi
}

# Turn a number into a binary byte: (inputvalue)
writebinarybyte(){
  writebinarybit $1 128
  writebinarybit $? 64
  writebinarybit $? 32
  writebinarybit $? 16
  writebinarybit $? 8
  writebinarybit $? 4
  writebinarybit $? 2
  writebinarybit $? 1
}

# Output the byte which is represented by a decimal number
tobyte(){
  # Echo only takes octal numbers, so we convert
  echo -n -e \\$1
}

################################################################################

process(){
  logger "shdns $$ Started parsing $1 at $2"
  inset=$2

  # Identification: 2 bytes
  id=‘readstring $1 $inset 2‘; inset=$(( $inset + 2 ))
  logger "shdns $$ Packet id: $id"

  # Flags: 2 bytes
  temp=‘cat $1 | cut -b $inset |  od -Ad -c | head -1 | cut -f 2 -d " "‘; inset=$(( $inset +
  testbit $temp 128; qr=$?
  testbit $temp 8; op=$?
  testbit $temp 4; aa=$?
  testbit $temp 2; trun=$?
  testbit $temp 1; rd=$?

  logger "shdns $$ Query / response: $qr"
  logger "shdns $$ Opcode: $op"
  logger "shdns $$ Authoritative answer: $aa"
  logger "shdns $$ Packet truncated: $trun"
  logger "shdns $$ Recursion desired: $rd"

  readbytebinary "$1" $inset; temp=$?; inset=$(( $inset + 1 ))
  testbit $temp 128; ra=$?

  logger "shdns $$ Recursion available: $ra"

  # The number of questions is the next two bytes
  readbytebinary "$1" $inset; topbyte=$?; inset=$(( $inset + 1 ))
  readbytebinary "$1" $inset; botbyte=$?; inset=$(( $inset + 1 ))
  qcount=$(( ($topbyte * 128) + $botbyte ))
  logger "shdns $$ Number of questions: $qcount"

  # The number of answers is the next two bytes
  readbytebinary "$1" $inset; topbyte=$?; inset=$(( $inset + 1 ))
```

```
readbytebinary "$1" $inset; botbyte=$?; inset=$(( $inset + 1 ))
acount=$(( ($topbyte * 128) + $botbyte ))
logger "shdns $$ Number of answers: $acount"

# The number of authority RRs is the next two bytes
readbytebinary "$1" $inset; topbyte=$?; inset=$(( $inset + 1 ))
readbytebinary "$1" $inset; botbyte=$?; inset=$(( $inset + 1 ))
authcount=$(( ($topbyte * 128) + $botbyte ))
logger "shdns $$ Number of authorities: $authcount"

# The number of additional RRs is the next two bytes
readbytebinary "$1" $inset; topbyte=$?; inset=$(( $inset + 1 ))
readbytebinary "$1" $inset; botbyte=$?; inset=$(( $inset + 1 ))
addcount=$(( ($topbyte * 128) + $botbyte ))
logger "shdns $$ Number of additionals: $addcount"

######################
# For each question
######################

len=42
questioncount=0

while [ $questioncount -lt $qcount ]
do
  questionstart=$inset
  logger "shdns $$ Question"
  name=""

  namelength=0
  readbytebinary "$1" $inset; len=$?
  while [ $len -gt 0 ]
  do
    inset=$(( $inset + 1 ))
    name="$name"‘readstring "$1" $inset $len‘"."
    inset=$(( $inset + $len ))

    namelength=$(( $namelength + $len + 1 ))
    readbytebinary "$1" $inset; len=$?
  done
  inset=$(( $inset + 2 ))
  logger "shdns $$ Lookup: $name"

  # Type of question
  readbytebinary "$1" $inset; type=$?
  error="none"
  temp="shdns $$ Determine the query type"
  case $type in
  1 ) temp="$temp A";;
  2 ) temp="$temp NS";;
  5 ) temp="$temp CNAME";;
  12 ) temp="$temp PTR";;
  13 ) temp="$temp HINFO";;
  15 ) temp="$temp MX";;
  * ) temp="Error: Unknown query type"; error="yes";;
  esac

  logger "$temp ($error)"
  inset=$(( $inset + 1 ))

  # The class should always be 1
  readbytebinary "$1" $inset; class=$?; inset=$(( $inset + 1 ))
  logger "shdns $$ Query class: $class"

  if [ "%$error%" = "%none%" ]
  then
    # Dodgy bug fix
    name=‘echo $name | sed ’s/\.$//’‘

    # Lookup the name in the db file
    result=‘grep "$name" /home/mikal/opensource-unstable/shdns/lookup | tr -s "\t" | cut -f
```

```
      logger "shdns $$ Result: $result"

      ############################################################
      # Now we need to build a response to the query

      # The id number we were handed gets handed straight back
      echo -n "$id"

      # Flag this packet as being a reply (we currently never claim to be authoritive)
      twiddlebit 0 128 1
      tobyte $?
#      tobyte 0

      # Number of questions (we have to return the question we are answering)
      tobyte 0
      tobyte 1

      # Number of answers
      tobyte 0
      tobyte 1

      # Number of authorities
      tobyte 0
      tobyte 0

      # Number of additionals
      tobyte 0
      tobyte 0

      # It's easy to return the question, we just copy it...
      echo -n `readstring $1 $questionstart $(( $inset - $questionstart - 2 ))`
      tobyte 0

      # The type is the same as in the question
      tobyte 0
      tobyte $type

      # The class is one
      tobyte 0
      tobyte 1

      # The time to live is always low, because we are dodgy
      tobyte 4
      tobyte 4
      tobyte 4
      tobyte 4

      # The length of the returned data is always an IP (32 bits)
      tobyte 0
      tobyte 4

      # And now the answer as a number
      # For the result in the answer, we are going to need this in a binary form of decimal
      while [ "%$result%" != "%%" ]
      do
        temp=`echo $result | cut -f 1 -d "."`
        result=`echo $result | sed 's/^[0-9]*\.*//'`
        tobyte $temp
        logger "shdns $$ Processing result segment: $temp ($result)"
      done

      # We need to return an authority as well...

      logger "shdns $$ Finished extracting result"
    else
      # Work out the erroneous type
      logger "shdns $$ Erroneous type was $type ("`dumpbyte $type`")"
    fi

    logger "shdns $$ Finished processing question"
    questioncount=$(( $questioncount + 1 ))
```

```
  done

  return $inset‘twiddlebit 0 128 1‘
}

################################################################################

offset=1
#while [ 1 -ne 0 ]
#do
  spinfor $1 $offset
  process $1 $offset
  offset=$?
  logger "shdns $$ Ended at $offset"
#  killall -9 cat
#done

exit
```

Code: shdns

4.2. Does it work?

TODO

5. Getting the code from this article

TODO