Lecture 19: Proxy-Server Based Firewalls

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 24, 2016
10:03pm

©2016 Avinash Kak, Purdue University

Goals:

- The SOCKS protocol for anonymizing proxy servers

- Socksifying application clients

- The Dante SOCKS server

- **Perl and Python scripts for accessing an internet server through a SOCKS proxy**

- Squid for controlling access to web resources (and for web caching)

- The Harvest system for information gathering, indexing, and searching

- How to construct an SSH tunnel through a web proxy

CONTENTS

19.1: FIREWALLS IN GENERAL (AGAIN)

- To expand on what was mentioned at the beginning of Lecture 18, firewalls can be designed to operate at any of the following three layers in the TCP/IP protocol stack:

    - the Transport Layer   (example: packet filtering with iptables)

    - the Application Layer   (example: HTTP Proxy)

    - the layer between the Application Layer and the Transport Layer   (example: SOCKS proxy)

- Firewalls at the Transport Layer examine every packet, check its IP headers and its higher-level protocol headers (in order to figure out, say, whether it is a TCP packet, a UDP packet, an ICMP packet, etc.) to decide whether or not to let the packet through and to determine whether or not to change any of the header fields. (**See Lecture 18 on how to design a packet filtering firewall.**)

- A firewall at the Application Layer examines the requested session for whether they should be allowed or disallowed based on

where the session requests are coming from and the purpose of the requested sessions. Such firewalls are built with the help of what are known as **proxy servers**.

• For truly application layer firewalls, you'd need a separate firewall for each different type of service. For example, you'd need separate firewalls for HTTP, FTP, SMTP, etc. Such firewalls are basically access control declarations built into the applications themselves. As a network admin, you enter such declarations in the server config files of the applications.

• A more efficient alternative consists of using a protocol between the application layer and the transport layer – this is sometimes referred to as the **shim layer** – to trap the application-level calls from *intranet* clients for connection to the servers in the internet.   [The shim layer corresponds to the Session Layer in the 7-layer OSI model of the TCP/IP protocol stack. See Lecture 16 for the OSI model.]

• Using a shim layer protocol, a proxy server can monitor all session requests that are routed through it in an *application-independent manner* to check the requested sessions for their legitimacy. *In this manner, only the proxy server, serving as a firewall, would require direct connectivity to the internet and the local* intranet *can "hide" behind the proxy server.* The computers in the internet at large would not even know about the existence of your machine in the local intranet behind the firewall.

• When a proxy is used in the manner described above, it may also
be referred to as an **anonymizing proxy**.

• Some folks like to use anonymizing proxies for privacy reasons.
Let's say you want to visit a web site but you do not wish for that
site to know your IP address, you can route your access through
a third-party anonymizing proxy.

• There are free publicly available proxy servers that you can use
for such purpose. Check them out by entering a string like
"public proxy server" in a search engine window. You can also
use publicly available scanners to search for publicly available
proxy servers within a specific IP range. The website `http:`
`//publicproxyservers.com` claims to offer a marketing-pitch-free
listing of the public proxy servers.

• In addition to achieving firewall security, a proxy server operating
at the application layer or the shim layer can carry out data
caching (this is particularly true of HTTP proxy servers) that
can result in transfer speed improvement. If the gateway machine
contains a current copy of the resource requested, in general it
would be faster for a LAN client to download that copy instead
of the version sitting at the remote host.

• The SOCKS protocol (RFC 1928) is commonly used for designing
shim layer proxy servers.

• Note that a transport layer firewall based on packet filtering and an application layer firewall implemented with the help of proxy servers of the type presented in this lecture often coexists for enhanced security. [You may choose the former for low-level control over the traffic and then use proxies for additional high-level control within specific applications and to take advantage of centralized logging and caching made possible by proxy servers.]

19.2: SOCKS

- SOCKS is referred to as a generic proxy protocol for TCP/IP based network applications.

- SOCKS, an abbreviation of "SOCKetS", consists of two components: A SOCKS client and a SOCKS server.

- It is the socks client that is implemented between the application layer and the transport layer; the socks server is implemented at the application layer.

- The socks client wraps all the network-related system calls made by a host with its own socket calls so that the host's network calls get sent to the socks server at a designated port, usually 1080. This step is usually referred to as **socksifying** the client call.

- The socks server checks the session request made by the socksified LAN client for its legitimacy and then forwards the request to the server on the internet. Any response received back from the server is forwarded back to the LAN client.

- For an experimental scenario where we may use socks, imagine that one of your LAN machines has two ethernet interfaces (eth0 and eth1) and can therefore act as a gateway between the LAN and the internet. We will assume that the rest of the LAN is on the same network as the eth0 interface and that the eth1 interface talks directly the internet. A socks based proxy server installed on the gateway machine can accomplish the following:

  – The proxy server accepts session requests from clients in the LAN on a designated port. If a request does not violate any security policies programmed into the proxy server, the proxy server forwards the request to the internet. Otherwise the request is blocked. This property of a proxy server to receive its incoming LAN-side requests for different types of services on a single port and to then forward the requests onwards into the internet to specific ports on specific internet hosts is referred to as **port forwarding**. Port forwarding is also referred to as **tunneling**.

  – The proxy server replaces the source IP address in the connection requests coming from the LAN side with with its own IP address.   [So the servers on the internet side cannot see the actual IP addresses of the LAN hosts making the connection requests. In this manner, the hosts in the LAN can maintain complete anonymity with respect to the internet.] This ploy is frequently used by business organizations to hide the internal details of their intranets.

– Focusing specifically on the HTTP traffic, the above ploy would cause all of the HTTP traffic emanating from the intranet to get routed through the socks server where it would be subject to various firewall rules and where, if desired, one can provide logging facilities and caching of the web services.

19.2.1: SOCKS4 versus SOCKS5

- Version 4 (usually referred to as SOCKS4) lacks client-server authentication. On the other hand, version 5 (usually referred to as SOCKS5) includes built-in support for a variety of authentication methods.

- SOCKS5 also includes support for UDP. So a SOCKS5 server can also serve as a UDP proxy for a client in an intranet.

- Additionally, with SOCKS4, the clients are required to resolve directly the IP addresses of the remote hosts (meaning to carry out a DNS lookup for the remote hosts). SOCKS5 is able to move DNS name resolution to the proxy server that, if necessary, can access a remote DNS server.

19.2.2: Interaction Between a SOCKS Client and a SOCKS Server

- To see how a socks client (more precisely speaking, a socksified client) interacts with a socks server, let's say that the client wants to access an HTTP service on the internet.

- The first part of the interaction is similar to what happens between an SSH client and an SSH server — the server needs to authenticate the client. This interaction is described below.

- The socks client opens a TCP connection with the socks server on port 1080. The client sends a "Client Negotiation" packet suggesting a set of different authentication methods that the server could use vis-a-vis the client. This packet consists of the following fields:

```
Client Negotiation:        VER    NMETHOD    METHODS
                            1        1        1-255
```

with the one-byte VER devoted to the version number (SOCKS4 or SOCKS5), the one-byte NMETHOD devoted to the number of methods that will be listed subsequently for client-server authentication, and, finally, a listing of those methods by their ID numbers, with each ID number as a one-byte integer value. [The value 0x00 in METHODS field means no authentication needed, the value 0x01 means authentication according

to the GSSAPI (Generic Security Services Application Programming Interface), `0x02` means a username/password based authentication, a value between `0x03` and `0x7E` defines a method according to the IANA naming convention, and the `0x80` through `0xFE` values are reserved for private methods. (IANA stands for the Internet Assigned Numbers Authority) Note if the method number returned by the socks server is `0xFF`, that means that the server has refused the method offered by the client. Also note that GSSAPI (RFC 2743) is meant to make it easier to add client-server authentication to an application as the modern practice is to expect all security software vendors to provide this API in addition to any proprietary APIs. For example, if you wanted to use Kerberos for client-server authentication, you could write your authentication code to GSSAPI.]

- If the socks proxy server accepts the client packet, it responds back with a two-byte "Server Negotiation" packet:

```
Server Negotiation:     VER    METHOD
                         1        1
```

where the `METHOD` field is the authentication method that the server wishes to use. The socks server then proceeds to authenticate the LAN client using the specified method.

- After the authentication step, the socks client then sends the socks proxy server a request stating what service it wants at what address in the internet and at which port. This message, called the "Client Request" message consists of the following fields:

```
Client Request: VER   CMD   RSV   ATYP   DST.ADDR   DST.PORT
                 1     1     1      1     variable      2
```

where the 1-byte CMD field contains one of three possible values: `0x01` for "CONNECT", `0x02` for "BIND", `0x03` for "UDP As-

sociate". [The ATYP field stands for the "Address Type" field. It takes one of three possible

values: 0x01 for IPv4 address, 0x02 for domain name, and 0x03 for IPv6 address. As you'd expect,

the length of the target address that is stored in the DST.ADDR field depends on what address type

is stored in the ATYP field. An IPv4 address is 4 bytes long; on the other hand, an IPv6 address 8

bytes long. Finally, the DST.PORT fields stores the the port number at the destination address. The

RSV field means "Reserved for future use."]

- The client always sends a CONNECT (value of the 1-byte CMD field) request to the socks proxy server after the client-server authentication is complete. However, for services such as FTP, a CONNECT request is followed by a BIND request. [The BIND request

  means that the client expects the remote internet server to want to establish a separate connection

  with the client. Under ordinary circumstances for a direct FTP service, a client first makes what is

  known as a control connection with the remote FTP server and then expects the FTP server to make

  a separate data connection with the client for the actual transfer of the file requested by the client.

  When the client establishes the control connection with the FTP server, it informs the server as to

  which address and the port the client will be expecting to receive the data file on.]

- After receiving the "Client Request" packet, the proxy server evaluates the request taking into account the address of the client on the LAN side, the target of the remote host on the internet side and other access control rules typical of firewalls.

- If the client is not allowed the type of access it has requested, the proxy server drops the connection to the client. Otherwise, the proxy server sends one or two replies to the socks client. [The socks

server sends to the client two replies for BIND requests and one reply for CONNECT and UDP requests.] These replies, different in the value of the REP field (and possibly other fields depending on the success or failure of the connection with the remote server) are called the "Server Reply" are according to the following format:

```
Server Reply:  VER    REP    RSV    ATYP    BND.ADDR    BND.PORT
                1      1      1      1       variable       2
```

where the BND.ADDR is the internet-side IP address of the socks proxy server; it is this address that the remote server will communicate with. Similarly, BND.PORT is the port on the proxy server machine that the remote server sends the information to.

- The REP field can take one of the following ten different values:

```
0x00:    successful connection with the remote server
0x01:    SOCKS proxy error
0x02:    connection disallowed by the remote server
0x03:    network not accessible
0x04:    remote host not accessible
0x05:    connection request with remote host refused
0x06:    timeout (TTL expired)
0x07:    SOCKS command not supported
0x08:    address type not supported
0x09 through 0xFF:  not defined
```

- If the connection between the proxy server and the remote server is successful, the proxy server forwards all the data received from

the remote server to the socks client and vice versa for the duration of the session.

- About the security of the data communication between the socks server and the remote service provider, note that since socks works independently of the application-level protocols, **it can easily accommodate applications that use encryption to protect their traffic.** To state a case in point, **as far as the socks server is concerned, there is no difference between an HTTP session and an HTTPS session.** Because, after establishing a connection, a socks proxy server doesn't care about the nature of the data that shuttles back and forth between the LAN client and the remote host in the internet, such a proxy server is also referred to as a **circuit-level proxy**.

19.2.3: Socksifying a Client-Side Application

- Turning a client-side application (such as a web browser, an email client, and so on) into a socks client is referred to as **socksifying the client**.

- For the commonly used socks server these days, **Dante**, this is accomplished as simply as by calling

      socksify name_of_your_client_application

  provided you have installed the Dante client in the machine on which you are trying to execute the above command.   [If you are on a Ubuntu machine, you can install both the Dante server and the Dante client directly through your packet manager. Just search for the sting "dante" in the packet manager's search window.]

- Let's say you are unable to directly access an FTP server in the internet because of the packet-level firewall rules in the gateway machine, you might be allowed to route the call through the proxy server running on the same machine by

      socksify ftp url_to_the_ftp_resource

- For another example, to run your web browser (say, the Firefox browser) through a socks proxy server, you would invoke

```
socksify firefox
```

By the way, when you socksify Firefox in this manner, you must keep the browser's connection settings at the commonly used "Directly connect to internet" in the panel for Edit-Preferences-Advanced-Network-Settings. You do NOT have to be logged in as root to socksify a browser in this manner. [According to Michael Shuldman of Inferno Nettverk, you can get your Firefox browser to work through a socks server by just clicking on the "Manual Proxy Configuration" tab in the window that comes up for Edit-Preferences-Advanced-Network-Settings and entering the IP address and the port for the socks proxy server.]

- In Section 19.3.4, I will present an example of socksifying a user-created application program. There I'll show custom Perl and Python clients – `DemoExptClient.pl` and `DemoExptClient.py` – that can engage in an interactive session with custom Perl and Python servers running on a remote host in the internet. Ordinarily, the command-line invocation you'd make on the LAN machine would be something like this:

```
DemoExptClient.pl  moonshine.ecn.purdue.edu  9000

DemoExptClient.py  moonshine.ecn.purdue.edu  9000
```

assuming that the hostname of the remote machine is `moonshine.ecn.purdue.edu` and that port 9000 is assigned to the server script running on that machine. In order to route this call through the socks server (assuming you are running the Dante proxy server)

17

on your local gateway machine, all you'd need to do is to make one of the two calls shown below:

```
socksify DemoExptClient.pl  moonshine.ecn.purdue.edu  9000

socksify DemoExptClient.py  moonshine.ecn.purdue.edu  9000
```

- The call to `socksify` as shown above invokes a shell script of that name (that resides in `/usr/bin/` in a standard install of Dante). Basically, all it does is to set the `LD_PRELOAD` environment variable to the `libdsocks` library that resides in the `libdsocks.so` dynamically linkable file.

- By setting the `LD_PRELOAD` environment variable (assuming your platform allows it), 'socksify' saves you from the trouble of having to recompile your client application so as to redirect the system networking calls to the proxy server. [As explained in the 'README.usage' document that comes with the Dante install, this only works with non-setuid applications. The `LD_PRELOAD` environment variable is usually ignored by setuid applications. When a previously written client application can be compiled and linked to dynamically, you can socksify it by linking it with the `libdsocks` shared library by supplying the linking command with the '`-ldsocks`' option assuming that the file `libdsocks.so` is at the standard location (otherwise, you must provide the pathname to this location with the '`-L pathname`' option). If such dynamic linkage is not possible, you can always resort to static recompilation of your client application. See the file '`README.usage`' mentioned above for further information on how to do this.]

- All of the presentation so far has been from a Linux perspective. There is an implementation of the socks protocol, called SocksCAP, that enables Windows based TCP and UDP networking clients to traverse a socks firewall. Visit `http://www.socks.permeo.com/` for further information.

19.3: DANTE AS A SOCKS PROXY SERVER

• Dante, available from `http://www.inet.no/dante/`, is a popularly used implementation of the socks protocol. The current version of Dante (the version you download through your Synaptic Package Manager) is **1.1.19**. Visit `http://www.inet.no/dante/docs` for links to documentation pages for Dante. [As mentioned earlier, If you are on a Ubuntu machine, you can install both the Dante server and the Dante client directly through your packet manager. Just search for the sting "dante" in the packet manager's search window.]

• A standard install of Dante will give you the following configuration files:

```
/etc/danted.conf              the server configuration file

/etc/dante.conf               the client configuration file
```

• Start the server by executing as root:

```
sudo /etc/init.d/danted start
```

You can verify that the server is running by executing in a command line '`ps aux | grep dante`' that will return something like the following:

```
nobody    8455  0.0  0.0  24136   652 ?           Ss   01:51   0:00 /usr/sbin/danted -D
nobody    8456  0.0  0.0  24136   468 ?           S    01:51   0:00 /usr/sbin/danted -D
nobody    8457  0.0  0.0  24136   468 ?           S    01:51   0:00 /usr/sbin/danted -D
nobody    8458  0.0  0.0  24136   468 ?           S    01:51   0:00 /usr/sbin/danted -D
nobody    8459  0.0  0.0  24136   468 ?           S    01:51   0:00 /usr/sbin/danted -D
nobody    8460  0.0  0.0  24136   468 ?           S    01:51   0:00 /usr/sbin/danted -D
nobody    8461  0.0  0.0  24136   468 ?           S    01:51   0:00 /usr/sbin/danted -D
root      8466  0.0  0.0   9456   944 pts/4       S+   01:51   0:00 grep --color=auto dante
```

Although you can stop the server by executing in a command line '/etc/init.d/danted stop', should that not kill all the processes above, you can also invoke 'killall danted'. [According to Michael Shuldman of Inferno Nettverk, not killing all the child processes when you terminate the main server process is less disruptive to the socks clients. If you kill the main server process because, say, you want to upgrade your Dante server, the still-alive child server processes would continue to serve the socks clients that are already connected. Subsequently, after you restart the main server process, any new clients would be handled by the new server process and its children, whereas the old clients would continue to be served by the previously created child server processes. For further information, see http://www.inet.no/dante/doc/faq.html#processes_do_not_die.]

• Although you would normally start up the Dante server through the start/stop/restart script in /etc/init.d/ as indicated above, when you are first learning socks, you would be better off firing up the executable directly with the '-d' option so that it comes up in the debug mode. The command line for this in the standard Ubuntu install of Dante is

```
sudo /usr/sbin/danted -d
```

Note that the option is '-d' and NOT '-D'. (The former stands for "debug mode" and the latter for "detach mode" for running the

Dante server in the background. When you bring up the server
with the command string shown above, you can actually see the
server setting up the child processes for accepting requests from
the socks clients, the server reaching out to a DNS server for IP
lookups, and then finally accessing the services requested by the
client. See Section 19.12 for a small example.

- However, before you fire up the server in any manner at all, you'd
  want to edit the *server* configuration file `/etc/danted.conf` and
  the *client* configuration file `/etc/dante.conf`. The next couple of
  sections address this issue.

19.3.1: Configuring the Dante Proxy Server

• For our educational exercise, we will assume that our socks proxy server based firewall is protecting a `192.168.1.0/24` intranet and that the interface that connects the firewall machine with the internet is `eth0`. We will therefore not worry about client-server authentication here.

• The server config file, `/etc/danted.conf`, consists of three sections:

  – Server Settings
  – Rules
  – Routes

• With regard to the options in the "Server Settings" section of the config file:

logoutput: Where the log messages should be sent to.

internal: The IP address associated with the proxy server (I chose 127.0.0.1) and the port it will monitor (1080 by default). What is needed is the IP address of the host on which the proxy server is running. Since my proxy clients will be on the same machine as the proxy server, it makes sense to use the loopback address for the proxy server.

external: The IP address that all outgoing connections from the server should use:

– This will ordinarily be the IP address of the interface on which the proxy server will be communicating with rest of the internet.

– You can also directly name the interface (such as eth0) that the proxy server will use for all outgoing connections, which is what I have done. It will now automatically use the IP address associated with that interface. This is convenient for DHCP assigned IP addresses.

– About using a fictitious IP address for all outgoing connections from the server, it probably won't work since – at least ordinarily — your outgoing interface (eth0, eth1, wlan0, etc) can only work with a legal IP address that an upstream router can understand. **[It appears that the only way to take advantage of the anonymity offered by a socks server is if you route your personal outgoing traffic through a socks server run by a third party. Now the recipients of your traffic will see the IP address of that party.]**

– If for some reason (that is difficult to understand) you use a socks proxy behind a home or a small-business router, you won't gain any anonymity from the outgoing IP address used by the SOCKS server since the router will translate the outgoing (the source) IP address into what is assigned

to router by the ISP anyway.

method: Methods are for authenticating the proxy clients. Remember that a socks server and a socks client do not have to be on the same machine or even on the same local network.

user.privileged: If client authentication requires that some other programs be run, the system would need to run them with certain specified privileges. For that purpose, you can create a user named `proxy` if you wish and set this option accordingly. Ignore it for now since we will not be doing any client authentication. [According to Michael Shuldman of Inferno Nettverk, when the server is used in a production setting, it would need to run "at least temporarily" with an effective ID of 0 (that is, as root) in order to read the system password file (which would be the `/etc/shadow` for Linux) so that it can later verify the passwords provided by the socks clients. This becomes particularly necessary if you chose 'method: username' for the previous option.] [To elaborate on the "at least temporarily" phrase, let's say that `user.privileged` is set to root and `user.notprivileged` is set to `nobody`, the server will run with the default privileges of `nobody` all the time except when the server needs to, for example, authenticate a client on the basis of the passwords in, say, `/etc/shadow`. At that moment, the server would elevate its privileges to the root level, extract the needed information from system password file, and then revert back to the default privilege level of `nobody`.]

user.notprivileged: This specifies as to what read/write/execute privileges the server should be set to when running in the default non-privileged mode. Set it to `nobody` which means that

the server would have no permissions at all with respect all
the other files in the system.

• **Rules**: There are two kinds of rules:

– **Rules, first kind**: There are rules that control as to which
socks clients are allowed to talk to the proxy server. These
are referred to as *client rules*. All such rules have the `client`
prefix as in

```
client pass {
    from: 127.0.0.0/24 port 1-65535 to: 0.0.0.0/0
}
client pass {
    from: 192.168.1.0/24 port 1-65535 to: 0.0.0.0/0
}
client block {
    from: 0.0.0.0/0 to: 0.0.0.0/0
    log: connect error
}
```

These rules say to allow all local socks clients on the same
machine and all socks clients on the local LAN to talk to the
SOCK proxy server on this machine. The third rule says to
deny access to all other socks clients. Note that "to:" in these
rules is the address on which the socks server will accept a
connection request from a socks client. And, of course, as
you'd expect, "from:" is the source IP address of the client.

– **Rules, the second kind**: These are rules that control as
to what remote services the proxy server can be asked to talk

to (in the rest of the internet) by a socks client. These rules do NOT carry the `client` prefix. **Be careful here since how you set up these rules may dictate whether or not the proxy server can successfully carry out DNS lookups.** The comment statements in the `danted.conf` file recommend that you include the first of the four rules shown below for this section. But if you do, your proxy server will **not** be able talk to the local DNS server. In my `danted.conf` file, these rules look like:

```
# Comment out the next rule since otherwise local DNS will not work
#block {
#    from: 0.0.0.0/0 to: 127.0.0.0/8
#    log: connect error
#}
pass {
    from: 127.0.0.0/24 to: 0.0.0.0/0
    protocol: tcp udp
}
pass {
    from: 192.168.1.0/24 to: 0.0.0.0/0
    protocol: tcp udp
}
block {
    from: 0.0.0.0/0 to: 0.0.0.0/0
    log: connect error
}
```

The second rule says that any local socks client will be able to call on any service anywhere for a TCP or UDP service. The third rule does the same for any socks client in the local LAN. The fourth rule blocks all other socks client requested services. Note that "to:" in these rules is the *final destination* of the request from a socks client. And "from:" carries the same meaning as before — it is the source address of a socks client.

• In the second set of rules shown above (the ones without the `client` prefix), it is possible to allow and deny specific services with regard to specific client source addresses and client final destination addresses. See the official `/etc/danted.conf` file for examples.

• The third and final section of the `/etc/danted.conf` file deals with the route to be taken if proxy server chaining is desired. The route specifies the name of the next upstream socks server.

• The **internal** and **external** option settings mentioned earlier in this section are for the "normal" mode of operation of a proxy server — the mode in which the clients access the services in the rest of the internet through a proxy server. However, there is another mode in which such proxy servers can be used — the **reverse proxy** mode. In the reverse mode, you may offer, say, an HTTP server in a private network but with the traffic to your HTTP server directed through a Dante proxy server. You could, for example, use a SOCKS server front-end to control access to the private server. [You might ask: Why not use HTTPD's access control settings directly? While that may be true for an HTTP server, what if I wanted to control access to the server described in Section 19.3.4? Instead of having to write all the additional authentication and access-control code myself for that server, I could use a Dante server as a reverse proxy and achieve the same results with very little additional effort.] When a Dante server is used as a reverse proxy, the meanings of **internal** and **external** options become reversed, as you'd expect. [That the Dante server can be used as a reverse proxy was

brought to my attention by Michael Shuldman of Inferno Nettverk.]

An Example of the /etc/danted.conf Server Config File

```
# A sample danted.conf that I use for demonstrating SOCKS
#
# See the actual file /etc/danted.conf in your own installation of
# Dante for further details.

######################    ServerSettings    ##########################
# server will log both via syslog, to stdout and to /var/log/lotsoflogs
logoutput: syslog stdout /var/log/lotsoflogs


internal: 127.0.0.1 port = 1080

# All outgoing connections from the server will use the IP address
# 195.168.1.1
external: eth0                # See page 23 for what it means to run
                             # a SOCKS server behind a home router

# List acceptable methods for authentication in the order of
# preference.  A method not set here will never be selected.
# If the method field is not set in a rule, the global method is
# filled in for that rule.  Client authentication method:
method: username none

# The following is unnecessary if not doing authentication.  When
# doing something requiring privilege, it will use the userid "proxy".
user.privileged: proxy

# When running as usual, it will use the unprivileged userid of:
user.notprivileged: nobody

# Do you want to accept connections from addresses without dns info?
# What about addresses having a mismatch in dnsinfo?
srchost: nounknown nomismatch
```

```
############################  RULES ############################
# There are two kinds and they work at different levels.
#
#====================  rules checked first ====================


# Allow our clients, also shows an example of the port range.
client pass {
from: 192.168.1.0/24 port 1-65535 to: 0.0.0.0/0
}
client pass {
from: 127.0.0.0/8 port 1-65535 to: 0.0.0.0/0
}
client block {
from: 0.0.0.0/0 to: 0.0.0.0/0
        log: connect error
}


#==================  the rules checked next  ==================
pass {
from: 192.168.1.0/24 to: 0.0.0.0/0
protocol: tcp udp
}
pass {
from: 127.0.0.0/8 to: 0.0.0.0/0
protocol: tcp udp
}
pass {
        from: 0.0.0.0/0 to: 127.0.0.0/8
        protocol: tcp udp
}
block {
from: 0.0.0.0/0 to: 0.0.0.0/0
log: connect error
}


#  See /etc/danted.conf of your installation for additional
#  examples of such rules.
```

19.3.2: Configuring SOCKS Clients

- The *client* configuration file `/etc/dante.conf` regulates the behavior of a *socksified client.*

- At the beginning of the client configuration file, `/etc/dante.conf`, you are asked if you want to run the socksified client with the debug option turned on.

- All the other significant rules in the client config file are **route** rules, that is rules that carry the **route** prefix.

- The first of these **route** rules lets you specify that you want to allow for "bind" connections coming in from outside. The "bind" command allows incoming connections for protocols like FTP in which the local client first makes a control connection with a remote server and the remote server then makes a separate connection with the client for data transfer:

```
route {
    from: 0.0.0.0/0 to: 0.0.0.0/0 via: 127.0.0.1 port = 1080
    command: bind
}
```

• See the official `/etc/dante.conf` file in your own installation of Dante for other examples of the `route` rules that allow a client to directly carry out the DNS lookup on the localhost or by directly reaching out to a remote DNS server.

• Whereas the previous `route` rule for the "bind" command, the next `route` rule tells the client where the SOCKS proxy server is located and what port the server will be monitoring. This rule also tells the client that the server supports TCP and UDP services, both SOCKS4 and SOCKS5 protocols, and that the server does not need any authentication:

```
route {
    from: 0.0.0.0/0   to: 0.0.0.0/0   via: 127.0.0.1 port = 1080
    protocol: tcp udp                 # server supports tcp and udp.
    proxyprotocol: socks_v4 socks_v5 # server supports socks v4 and v5.
    method: none #username            # we are willing to authenticate via
                                      # method ''none'', not ''username''.
}
```

• The "from:" and "to:" in the previous rule are the IP address ranges for the client source addresses and the client *final* destination addresses for the remote services requested through the proxy server. In order to allow for the final destination addresses to be expressed as symbolic hostnames, we now include the next `route` rule:

```
route {
    from: 0.0.0.0/0   to: .   via: 127.0.0.1 port = 1080
    protocol: tcp udp
    proxyprotocol: socks_v4 socks_v5
```

```
        method: none #username
    }
```

- **Shown below is an example of the** `/etc/dante.conf` **SOCKS Client Config File:**

```
# A sample dante.conf that I use for demonstrating SOCKS clients
#
# See the actual file /etc/dante.conf in your own installation of
# Dante for further details.

#debug: 1

# Allow for "bind" for a connection initiated by a remote server
# in response to a connection by a local client:
route {
from: 0.0.0.0/0 to: 0.0.0.0/0 via: 127.0.0.1 port = 1080
command: bind
}

# Send client requests to the proxy server at the address shown:
route {
from: 0.0.0.0/0   to: 0.0.0.0/0   via: 127.0.0.1 port = 1080
protocol: tcp udp               # server supports tcp and udp.
proxyprotocol: socks_v4 socks_v5 # server supports socks v4 and v5.
method: none #username  # we are willing to authenticate via
 # method "none", not "username".
}

# Same as above except that the remote services may now be named
# by symbolic hostnames:
route {
from: 0.0.0.0/0   to: .   via: 127.0.0.1 port = 1080
protocol: tcp udp
proxyprotocol: socks_v4 socks_v5
method: none #username
}
```

19.3.3: Anonymity Check

- How can you be certain that when you go through a proxy server, your IP address will not be visible to the remote host?

- A common way to check for your anonymity is to visit a web site (of course, through the proxy server) that displays your IP address in the browser window. (An example of such a web site would be `http://hostip.info`.)

- This is usually sufficient check of anonymity for SOCKS proxy servers, but not for HTTP proxy servers. (**HTTP Proxy Servers are presented starting with Section 19.4.**)

- Even when an HTTP proxy server does not send the `HTTP_X_FORWARDED_FOR` field to the remote server, it may still send the `HTTP_VIA` and `HTTP_PROXY_CONNECTION` fields that may compromise your privacy.

- When an HTTP proxy server does not send any of these fields to the remote server, it is usually called an **elite** or a **high-anonymity** proxy server.

19.3.4: Perl and Python Scripts for Accessing an Internet Server Through the `danted` Proxy

- To understand the scripts shown in this section, please keep straight the meaning to be associated with each of the following:

  – an **internet server**, means a server running somewhere in the internet;

  – a **client** that wants to interact with the internet server;

  – the **socks proxy server** (`dantd`, naturally); and

  – a **socksified client**, which comes into existence when the network calls made by an otherwise ordinary client are routed through a socks proxy.

- Ordinarily, when socks in not involved, you will run the client program on your machine and this program will talk to the internet server on some remote machine.

- For the demonstration in this section, we will assume that both the Dante socks client and the Dante socks server are running on

the same machine — we will refer to that machine as the client machine. With the Dante socks server running on the client machine, we want to route all of the client's communication with the remote application server through the socks server on the client machine.

- With regard to the internet server that I'll use for the demonstration in this section, its purpose will be to display a set of server-side commands to the client, and have the client choose one of the commands. The internet server will then execute the command on its side and send the output back to the client.

- In what follows, I'll first show the Perl version of the internet server used in this demonstration. That will be followed by the Python version of the same. Both these programs are taken from Chapter 15 of my book *"Scripting with Objects."*

- In the Perl server shown below, Lines (C) through (F) of the script create a server socket on port 9000. The special symbol `SOMAXCONN` in line (D), defined in one of the low-level socket libraries used by the high-level module `IO::Socket`, stands for the system-dictated maximum number of client connections that the server socket can wait on at any given time. [The value of this special constant was 128 for the Linux machine on which I executed the server script shown.] The call to the constructor in lines (C) through (F) also sets `Reuse` option to 1. This is useful during debugging since it allows immediate

reuse of the port that is supposed to be monitored by the server
after the server process is killed and then started again in quick
succession. If you don't set the `Reuse` option as shown, a restart
of the server process will not succeed as long as the various buffers
assigned to the server process during its previous run are not
cleared out.

- With regard to what actually is accomplished by the server script
  shown below, on account of the call to `accept()` in line (I), it
  waits patiently for client requests for connections with the server.
  A client request causes `accept()` to spit out a socket handle that
  becomes the value of the variable `$client_soc`. The server can
  now read the client messages through this socket handle and send
  information to the client through the same socket handle. The
  first thing the server does is to send the client a welcome message
  in line (J) where the variable `$0` will be bound to the name of
  the server script.

- The rest of the server code shown below is to figure out which
  command was selected by the client, to execute the command on
  the server side, and to then send the output back to the client.
  This is done in lines (N) through (a) of the script.

```
#!/usr/bin/env perl

##  DemoExptServer.pl

##  This code from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak
```

```
use strict;
use warnings;
use IO::Socket;                                                      #(A)
use Net::hostent;                                                    #(B)

my $server_soc = IO::Socket::INET->new( LocalPort => 9000,           #(C)
                                        Listen    => SOMAXCONN,      #(D)
                                        Proto     => 'tcp',          #(E)
                                        Reuse     => 1);             #(F)
die "No Server Socket" unless $server_soc;                           #(G)

print "[Server $0 accepting clients]\n";                             #(H)
while (my $client_soc = $server_soc->accept()) {                     #(I)
    print $client_soc "Welcome to $0; type help for command list.\n";#(J)
    my $hostinfo = gethostbyaddr($client_soc->peeraddr);            #(K)
    my $clientport = gethostbyaddr($client_soc->peerport);
    printf "\n[Connect from %s]\n",
            $hostinfo ? $hostinfo->name : $client_soc->peerhost;    #(L)
    printf "[Client used the port %s]\n\n",
            $clientport ? $clientport : $client_soc->peerport;
    print $client_soc "Command? ";                                   #(M)
    while ( <$client_soc> ) {                                        #(N)
        next unless /\S/;                                            #(O)
        printf "   client entered command: %s\n", $_;
        if    (/quit|exit/i) { last; }                              #(P)
elsif (/date|time/i) { printf $client_soc "%s\n",
                                          scalar localtime;}         #(Q)
        elsif (/ls/i )      { print  $client_soc 'ls -al 2>&1'; }   #(R)
        elsif (/pwd/i )     { print  $client_soc 'pwd 2>&1';}       #(S)
        elsif (/user/i)     { print  $client_soc 'whoami 2>&1'; }   #(T)
        elsif (/rmtilde/i)  { system "rm *~"; }                     #(U)
        else {                                                       #(V)
          print $client_soc
            "Commands: quit exit date ls pwd user rmtilde\n";        #(W)
        }
    } continue {                                                     #(X)
        print $client_soc "Command? ";                               #(Y)
    }                                                                #(Z)
    close $client_soc;                                               #(a)
}
```

• As you can see, the internet server shown above monitors port
  9000. When a client checks in, the server first welcomes the client
  and then, in an infinite loop, asks the client to enter one of the
  following commands: `quit`, `exit`, `date`, `time`, `ls`, `pwd`, `user`,
  and `rmtilde`. Except for the last, these are system functions

that are ordinary invoked on the command line in Unix and Linux system. The last, `rmtilde` calls the system function `rm` to remove all files in the directory in which the server is running whose names end in a tilde.

• Shown next is the Python version of the server. the module `sys` imported in line (A) is needed for terminating the script with a call to `sys.exit(1)` in line (N) should something go wrong while trying to create a server socket, and for gaining access to the name of the server by calling `sys.argv[0]` in the messages composed in lines (O) and (R). The module `socket` imported in line (B) is needed for constructing a server socket in line (G). The modules `time`, `os`, and `commands` imported in lines (C), (D), and (E) are required for the execution of the various commands made available by the server to a remote client. As shown in line (F), the server script monitors the same port as the previous Perl script, that is, port 9000.

```
#!/usr/bin/env python

##  DemoExptServer.py

##  This code is from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak


import sys                                                        #(A)
import socket                                                     #(B)
import time                                                       #(C)
import os                                                         #(D)
import commands                                                   #(E)

port = 9000                                                       #(F)

try:
```

```
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  #(G)
    server_sock.bind( ('', port) )                                   #(H)
    server_sock.listen(5)                                            #(I)
except socket.error, (value, message):                               #(J)
    if server_sock:                                                  #(K)
        server_sock.close()                                          #(L)
    print "Could not establish server socket: " + message            #(M)
    sys.exit(1)                                                      #(N)

print "[Server %s accepting clients]" % sys.argv[0]                  #(O)

while 1:                                                             #(P)
    (client_sock, address) = server_sock.accept()                   #(Q)
    client_sock.send( "Welcome to %s; type help for command list." \
                      % sys.argv[0] )                                #(R)
    client_name, client_port = client_sock.getpeername()            #(S)
    print "Client %s connected using port %s " % \
                                (client_name, client_port)           #(T)
    client_sock.send( "\nCommand? " )                               #(U)
    while 1:                                                         #(V)
        client_line = ''                                            #(W)
        while 1:                                                    #(X)
            client_byte = client_sock.recv(1)                       #(Y)
            if client_byte == '\n' or client_byte == '\r':          #(Z)
                break                                               #(a)
            else:                                                   #(b)
                client_line += client_byte                          #(c)
        if client_line.isspace() or client_line == '':              #(d)
            client_sock.send( '\nCommand? ' )                       #(e)
        elif  client_line == 'quit' or client_line == 'exit':       #(f)
            break                                                   #(g)
        elif client_line == 'date' or client_line == 'time':        #(h)
            client_sock.send( time.ctime() )                        #(i)
            client_sock.send( '\nCommand? ' )                       #(j)
        elif 'ls' in client_line:                                   #(k)
#            client_sock.send( commands.getoutput( "ls -al" ) )
            client_sock.send(commands.getstatusoutput("ls -al")[1]) #(l)
            client_sock.send('\nCommand? ')
        elif 'pwd' in client_line:                                  #(m)
            client_sock.send( commands.getoutput( "pwd 2>&1" ) )    #(n)
            client_sock.send('\nCommand? ')
        elif 'user' in client_line:                                 #(n)
            client_sock.send( commands.getoutput( "whoami 2>&1" ) ) #(o)
            client_sock.send('\nCommand? ')
        elif 'rmtilde' in client_line:                              #(p)
            os.system( "rm *~" )                                    #(q)
            client_sock.send('\nCommand? ')
        else:                                                       #(r)
            client_sock.send(                                       #(s)
                "Commands: quit exit date ls pwd user rmtilde" )    #(t)
            client_sock.send("\nCommand? ")
    client_sock.close()                                             #(u)
```

• I'll next present the client scripts, one for Perl and the other for Python, both taken from my book *"Scripting with Objects."*

• Shown below is the Perl version of the client script. A client must not use a line-input-based operator for reading the messages received from the server and, if the server messages are meant to be displayed on the client terminal as soon as they are received, the client must also override the default flushing behavior of the output buffer associated with `STDOUT`. By default, the output buffer is flushed only when a line terminator is received. The statement in line (M) of the script shown next would cause each transmission received from the server to be displayed on the client's terminal immediately. This client interacts with either of the two servers shown previously in an interactive session. The server prompts the client to enter one of the permissible commands and the server then executes that command.

• Note also that the client script shown below uses two separate processes for reading from the server and for writing to the server. While the parent process takes care of writing to the server the information entered on the keyboard of the client, a child process takes care of reading the server's messages and displaying those on the client's terminal. The statement in line (J) of the script creates a child process. The call to `fork()` returns in the parent process the process ID (PID) of the child process if the child process was created successfully; the value returned by the above call in the child process is 0. Otherwise, it returns `undef`.

```perl
#!/usr/bin/env perl

##  DemoExptClient.pl

##  This code from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak

use strict;
use warnings
use IO::Socket;                                                  #(A)

die "usage: $0 host port" unless @ARGV == 2;                     #(B)

my ($host, $port) = @ARGV;                                       #(C)

my $socket = IO::Socket::INET->new(PeerAddr  => $host,           #(D)
                                   PeerPort  => $port,           #(E)
                                   Proto     => "tcp",           #(F)
                                  )
    or die "can't connect to port $port on $host: $!";           #(G)

$SIG{INT} = sub { $socket->close; exit 0; };                     #(H)

print STDERR "[Connected to $host:$port]\n";                     #(I)

# spawn a child process
my $pid =  fork();                                               #(J)
die "can't fork: $!" unless defined $pid;                        #(K)

# Parent process: receive information from the remote site:
if ($pid) {                                                      #(L)
    STDOUT->autoflush(1);                                        #(M)
    my $byte;                                                    #(N)
    while ( sysread($socket, $byte, 1) == 1 ) {                  #(O)
        print STDOUT $byte;                                      #(P)
    }
    kill("TERM", $pid);                                          #(Q)
} else {                                                         #(R)
    # Child process: send information to the remote site:
    my $line;                                                    #(S)
    while (defined ($line = <STDIN>)) {                          #(T)
print $socket $line;                             #(U)
    }
}
```

- That brings us to the last script in this section: a Python version of the client script shown above. As in the Perl script, we fork off

a child process that takes care of the sending part of the commu-
nication link, while the parent process takes care of the receiving
part. In Python you create a child process by calling `os.fork()`.
The script makes this call in line (S). Since the client-side script
will always be on (provided the server has not shut down its side
of the connection), we need to be able to take down the client by a
keyboard-generated interrupt (as generated by pressing Ctrl-C).
This can be done by associating an appropriate signal handler
with the `SIGINT` signal. Signal handlers in Python are specified
by the `signal()` method of the `signal` module. Line (R) of
the script associates the signal handler of lines (N) through (Q)
with the `SIGINT` signal.

```
#!/usr/bin/env python

##  DemoExptClient.py

##  This code from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak

import sys                                                     #(A)
import socket                                                  #(B)
import os                                                      #(C)
import signal                                                  #(D)

if len(sys.argv) < 3:                                          #(E)
    sys.exit( "Need at least two command line arguments, the " +
              "first naming the host and the second the port" )

host, port = sys.argv[1], int(sys.argv[2])                     #(F)

try:
    sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM ) #(G)
    sock.connect( (host, port) )                               #(H)
except socket.error, (value, message):                         #(I)
    if sock:                                                   #(J)
        sock.close()                                           #(K)
    print "Could not establish a client socket: " + message   #(L)
    sys.exit(1)                                                #(M)

def sock_close( signum, frame ):                               #(N)
    global sock                                                #(O)
```

```
    sock.close                                                  #(P)
    sys.exit(0)                                                 #(Q)

signal.signal( signal.SIGINT, sock_close )                      #(R)

# spawn a child process
child_pid =  os.fork();                                         #(S)

if child_pid == 0:                                              #(T)
    # Child process: send information to the remote site:
    while 1:                                                    #(U)
        line = sys.stdin.readline()                             #(V)
sock.send( line )                                   #(W)
else:                                                           #(X)
    # Parent process: receive information from the remote site:
    while 1:                                                    #(Y)
        byte = sock.recv(1)                                     #(Z)
        if byte == '': break                                    #(a)
        sys.stdout.write( byte )                                #(b)
        sys.stdout.flush()                                      #(c)
    os.kill( child_pid, signal.SIGKILL )                        #(d)
```

• Before proceeding further with the demonstration described in
this section, download either the Perl versions or the Python
versions of the scripts from the lecture notes website and play
with them on your own machines. Fire up the server on your
own or a friend's machine and the client on another machine.
If the server is running on a machine with the internet address
xxx.yyy.www.zzz, the client can interact with it with a command
like 'DemoExptClient.py xxx.yyy.www.zzz 9000'. If the server machine
has a symbolic hostname, you can also use that name in place of
xxx.yyy.www.zzz in the command line on the client side.

• Assuming you have played with the server and client scripts as
described in the previous bullet, we are ready for the demon-
stration that shows the client interacting with the internet server

through a socks proxy server.

- For the demonstration, I will run the server on `moonshine.ecn.purdue.edu` by invoking one of the following two commands:

        DemoExptServer.pl

        DemoExptServer.py

- Now we **socksify** the client by using one of the following command lines:

        socksify DemoExptClient.pl moonshine.ecn.purdue.edu 9000

        socksify DemoExptClient.py moonshine.ecn.purdue.edu 9000

- The above call will work the same as when you tried the client script without socksification. As a user on the client side, you should notice no difference between the socksified call and the unsocksified call.

- Of course, before you make the above invocation to `socksify` you must fire up the `danted` server on the client machine. As mentioned in Section 19.3, to easily see the client requests going through the proxy server, start up the socks server with the command line:

```
sudo /usr/sbin/danted -d
```

When you bring up the socks server in this manner, you can actually see it making DNS queries and eventually talking to the internet server on behalf of the socks client. Of course, as previously mentioned, for "production" purposes you'd fire up the proxy server by

```
sudo /etc/init.d/danted start
```

and stop it by

```
sudo /etc/init.d/danted stop
```

19.4: SQUID

- If all you want to do is to control access to the HTTP and FTP resources on the web, the very popular Squid is an attractive alternative to SOCKS. **As with SOCKS, Squid can also be used as an anonymizing proxy server.**

- Although very easy to use for access control, Squid is also widely deployed by ISP's for web caching.

- You can install Squid on your own Linux laptop for personal web caching for an even faster response than an ISP can provide.

- Web caching means that if you make repeated requests to the same web page and there exists a web proxy server between you and the source of the web page, the proxy server will send a quick request to the source to find out if the web page was changed since it was last cached. If not, the proxy server will send out the cached page. [This can result in considerable speedup in web services especially when it comes to downloads of popular web pages. A popular web site is likely to be accessed by a large number of customers frequently or more or less constantly.]

- Squid supports ICP (Internet Cache Protocol, RFC2186, 2187).
  You can link up the Squid proxy serves running at different places
  a network through **parent-child** and **sibling** relationships. If
  a child cache cannot find an object, it passes on the request to
  the parent cache. If the parent cache itself does not have the
  object, it fetches and caches the object and then passes it to on
  to the child cache that made the original request. Sibling caches
  are useful for load distribution. Before a query goes to the parent
  cache, the query is sent to adjacent sibling caches.

- Squid also speeds up DNS lookup since it caches the DNS infor-
  mation also.

- Since Squid is a caching proxy server, it must avoid returning to
  the clients objects that are out of date. So it automatically expires
  such objects. You can set the refresh time in the configuration
  file to control how quickly objects are expired.

- **Squid was originally derived from the Harvest project.
  More on that in Section 19.5.**

- The home page for Squid:

      http://www.squid-cache.org/

• Windows has its own version of web proxy for caching internet
  objects and for performance acceleration of web services. It is
  called the Microsoft Internet Security and Acceleration Server
  (ISA Server).

19.4.1: Starting and Stopping the Squid Proxy Server

- If you installed version 3 of Squid (squid3) on your Ubuntu machine through the Synaptic Packet Manager, you will find the Squid configuration file at the following pathname:

    `/etc/squid3/squid.conf`

  and you will find the rest of the goodies in the `/usr/lib/squid3/` directory. As you would expect, the start/stop/restart script is invoked by (as root)

    ```
    /etc/init.d/squid3 start
                       stop
                       restart
    ```

  and the executable in

    `/usr/sbin/squid3`

  Note that version 3 is a major rewrite of Squid in C++ and it includes several new features.

- If Squid is already running in your computer (you can check that by executing '`ps ax | grep squid`'), this would be a good time to stop it as indicated above and to then re-start it as root using the following command line:

    `/usr/sbin/squid3 -N -d 1`

which bring up the proxy server in the debug mode to actually see
what it is doing as you first become familiar with it. In the command line above, the option '`-N`' means to run the server in the
foreground and the option '`-d 1`' means to run the server at debug level 1. An additional option to consider is '`-D`' is to suppress
DNS lookups by the server. If the server has a need to do DNS
lookups and it can't, the server may die without warning. The
directory where the objects are cached in a default installation of
Squid is

```
/var/spool/squid3/
```

You must uncomment the line

```
cache_dir ufs /var/spool/squid3 100 16 256
```

in the `squid3.conf` file in order for caching to take place. If you do
not uncomment this or a similar such line, your Squid proxy will
only act as a firewall through its access control lists.

- Apart from the above mentioned changes, the default installation
  of Squid should prove good enough for practically all your needs
  if you are running it as personal caching proxy server on your
  own machine.

- The default port monitored by the proxy server is 3128.

• After you have brought up the proxy server, it is useful to look at the following log, especially after you have made at least one client request through the proxy server:

```
/var/log/squid3/cache.log
```

This log shows you as to what host/port squid is monitoring for incoming requests for service, what port for ICP messages, how much cache memory it is using, how many buckets to organize the fast-memory entries for the cache objects, etc.

• The other very useful log at the same pathname as above is

```
access.log
```

What makes this log file particularly useful is that it shows whether an object was doled out from the cache or obtained from the origin server. The `access.log` file uses the following format for its entries

```
timestamp  elapsed  client  action/code  size  method  URI ident ...
```

• Here is a line entry from `access.log` if you make an SSH connection through Squid:

```
1170571769.664  96591 127.0.0.1 TCP_MISS/200 4403 \
      CONNECT rvl4.ecn.purdue.edu:22 - DIRECT/128.46.144.10 -
```

where the timestamp is a *unix time* — it is the number of seconds from Jan 1, 1970. The action TCP_MISS means that the internet

object requested was NOT in the cache, which makes sense in this case because we are not trying to retrieve an object; we are trying to make a connection with the remote machine (rvl4). By the way, when you see `TCP_HIT` for action, that means that a valid copy of the object was found in the cache and retrieved from it. Similarly `TCP_REFRESH_HIT` means that an expired copy of the object was found in the cache. When that happens, Squid makes an `If-Modified-Since` request to the origin server. If the response from the origin server is `Not-Modified`, the cached object is returned to the client.

- The critical hardware disk parameter for a cache is `random seek time`. If the random seek time is, say, 1 ms, that means you could at most do 1000 separate disk accesses per second.

- From Squid On-Line Users Manual: "Squid is not generally CPU intensive. It may use a lot of CPU at startup when it tries to figure out what is in the cache and a slow CPU can slow down access to the cache for the first few minutes after startup."

- Also from the on-line manual: Squid keeps in the RAM a table of all the objects in the cache. Each objects needs about 75 bytes in the table. Since the average size of an internet object is 10 KBytes, if your cache is of size 1 Gbyte, you would be able to store 100, 000 objects. That means that you'd need about 7.5 MBytes of RAM to hold the object index.

- **Now let's get the browser on your machine to reach out to the internet though the Squid proxy.**

- You will have to tell your web browser that it should NOT connect directly with the internet and, instead, it should route its calls through the Squid proxy. For example, for the **firefox** browser, the following sequence of button-clicks (either on menu items or in the dialog windows that pop up) will take you to the point where you'd need to enter the web proxy related information:

```
Firefox:
  -- Edit
     -- Preferences
        -- Advanced
           -- Network
              -- Settings
                 -- Manual Proxy Configuration
                    -- HTTP_Proxy 127.0.0.1   Port 3128
```

and then check the box for "Use this proxy for all protocols".

19.4.2: The Squid Cache Manager

- The cache manager is a neat utility. It consists of a CGI script located at '`/usr/lib/cgi-bin/cachemgr.cgi`'. The script will be automatically placed at this location when you install the `squid-cgi` package with the Synaptic package manager. This package contains the Squid cache manager CGI script. This script can provide statistics about the various objects in the cache. It is also a convenient tool for managing the cache.

- When you install the cache manager package as indicated above, it will also place a config file called `cachemgr.conf` in the `/etc/squid/` directory. However, for the experiments described here you would not need to change anything in this directory.

- To have the most fun with Squid's Cache Manager utility, you have to have the Apache web server installed on your Linux machine. With the web server running on your own machine, you can interact with the cache manager through a web browser on any host, including the same host that contains the cache.

  – I'd recommend that you install the Apache web server with the Synaptic Package Manager. If you install the `apache2` package, the package manager will automatically install four other related packages. In addition to needing it here for demonstrating what the Squid cache manager can do, we will also

need the Apache server when we discuss the Harvest system for information gathering and indexing later in this lecture.

- **Listed below are some things to watch out for if you do install the Apache web server on your Ubuntu machine.**

- First note that even when I casually refer to the web server as `httpd`, its official name is `apache2`. Even when you launch the web server daemon, `apache2`, as root, the child-server `httpd` processes that are created for handling individual connections with the clients will most likely be setuid to the user '`www-data`'. You can check this for yourself by executing '`ps aux | grep apache`' on your machine. As you should know by this time, this is for ensuring security since the user '`www-data`' has virtually no permissions with regard to the files on your system.

- With a standard install, your Apache HTTPD directory will be installed at the location `/etc/apache2/`. For convenience, in the `.bashrc` file of the root account, sets the environment variable `APACHEHOME` to point to this directory.

- The behavior of the Apache httpd server is orchestrated by the configuration files and subdirectories in the `/etc/apache2/` directory. The main config file is `apache2.conf` that in turn pulls in the contents of the site-specific config files in the `sites-enabled` and `mods-enabled` directories. See the HTTP server installation notes in Section 27.1 of Lecture 27 for additional comments related to the contents of the `mods-enabled` and `sites-enabled` directories. Suffice it to say that for me to enable Apache to serve out my web page in my `public-web` directory, the file `kak.conf` in the `sites-enabled` directory contains the following entries:

```
<VirtualHost *:80>
 ServerAdmin webmaster@localhost
    # The following directive names the file the server
    # will serve out when the 'kak' directory is requested
    # through '~kak':
    DirectoryIndex Index.html index.html

    # In the following. AllowOverride controls what directive
    # may be placed in .htaccess file. For example, it can be
```

```
    # All, None, etc.  The Indexes option allows a client to
    # see a listing of the directory if the client's request
    # is for a directory and if the DirectoryIndex has not
    # been set for that directory.
    <Directory "/home/kak/public-web/">
        Options Indexes FollowSymLinks MultiViews
        AllowOverride None
        Require all granted
    </Directory>

    # If I want cgi scripts to be served out of my own web
    # directory:
    ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
    <Directory "/usr/lib/cgi-bin">
        AllowOverride None
        Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
        Require all granted
    </Directory>

    ErrorLog /var/log/apache2/error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog /var/log/apache2/access.log combined
</VirtualHost>
```

– And I had to insert the following block of directives in
the /etc/apache2/apache2.conf configuration file:

```
UserDir enabled kak
UserDir public-web public_html

## For seeing the Squid cachemgr web page:
ScriptAlias /Squid/cachemgr /usr/lib/cgi-bin/cachemgr.cgi
<Location "/usr/lib/cgi-bin/cachemgr.cgi">
    allow from localhost
    deny from all
    <Limit GET>
    </Limit>
```

```
        require user kak
    </Location>
```

The first two lines tell Apache that it will be asked to dole out the `public-web` pages for the `kak` account on the machine. And the rest of the above directive allows the Squid cache manager to display its handiwork in the browser on my laptop. Note that the `ScriptAlias` directive tells Apache that the URL extension `/Squid/cachemgr` points to the location `/usr/lib/cgi-bin/cachemgr.cgi` and that the resource at this location is a cgi script that Apache must be executed before doling it out. The same directive for mapping a URL to a directory or a filename is just `Alias` if you do not want Apache to execute the contents before delivery.

– I did not change any other config files for the demos in this lecture.

– After your httpd server is up and running, you can read all the help files by pointing your browser to `http://localhost/ manual`.

– To start and stop the Apache HTTPD server, login as root and enter in the command line

```
    sudo /etc/init.d/apache2 start
                            stop
                            restart
```

Ordinarily, as you are experimenting with the config files, you can reload them into Apache by executing `/etc/init.d/apache2 reload` each time you make a change and you want to see its effects.

– If you run into any problems with the server, it can be extremely useful to look at `/var/log/apache2/error.log` for any error messages.

– For the httpd server daemon to serve out the web pages in the `public-web` subdirectory of my home directory, this subdirectory must carry the permission 755. Note that on Purdue's computers, the permission of a `public-web` directory in a user's account is 750. But that will not work for your personal Linux machine because, as mentioned already, the httpd server runs as the user '`www-data`'. Since the ownership/group of the `public-web` directory does not include '**www-data**', it is the permission bits that are meant for "other" that would determine whether or not '**www-data**' can access your `public-web` directory. This problem can be particularly vexing if you use **rsync** to download the updates for the public-web directory from your Purdue account. **rsync** will reset the permission bits to what they are in your Purdue account.

– If in addition to using the web server locally, you want to be able to access it from other machines, make sure that you have modified your packet filtering firewall accordingly (See Lecture 18).

• Next you need to make sure that the Squid configuration file `/etc/squid3/squid.conf` has the following definitions in it:

```
acl localhost src 127.0.0.1/32
acl our_networks src 192.168.1.0/24 127.0.0.1
acl all src 0.0.0.0/0
```

where **manager** stands for the cache manager.

• Using the above access control lists (acl), now make sure that the Squid configuration file `/etc/squid3/squid.conf` has the following permissions declared:

```
http_access allow manager localhost
http_access deny manager
http_access allow our_networks
http_access deny all
```

This says that the cache manager is allowed access only from the localhost. Any calls to the cache manager cgi script from any other host will be denied. We also allow access from any one in **our_networks**. Finally, we deny all other requests. In my case, the above settings were already in the `squid.conf` file as installed by the package manager.

• After you have set up the Apache web server and the Squid cache manager on your laptop, point your browser to

    `http://localhost/Squid/cachemgr`

You will first see a authorization page asking for the Cache Manager's login name and password. These must be as specified in the config file that is shown in the next section.

• To see the 25 biggest objects in the cache, execute the following in the `/var/log/squid3/` directory:

```
 sort -r -n +4 -5 access.log | awk '{print $5, $7}' | head -25
```

• Finally, note that there is a config file for the cache manager also that you can normally forget about if you are using the standard port for the Squid proxy. If not, you may need to make an entry in the cache manager config file at

     `/etc/squid/cachemgr.conf`

19.4.3: Configuring the Squid Proxy Server

- The configuration file `/etc/squid3/squid.conf` defines an incredibly large number of parameters for orchestrating and finetuning the performance of the web proxy.

- Fortunately, the default values for most of the parameters are good enough for simple applications of Squid – as, for example, for using it as web proxy on your own Linux machine. For my demonstrations of the Squid proxy, I only make the following three changes to the configuration file:

      cache_dir ufs /var/spool/squid3 100 16 256

      cache_mgr kak@localhost

      cachemgr_passwd none all

  If you search for the strings `cache_dir`, `cache_mgr` and `cachemgr_passwd`, you would know where to make these changes. The first entry above turns on web caching on the local disk. The second entry above designates where to send messages in case of problems, such as the proxy shutting down inadvertently, and the third declares that no password is needed for any of the actions made through the cache manager viewer in your browser. [The passwords can be set selectively for a large number of different actions vis-a-vis the cache manager. For example, if you wanted to subject the "shutdown" action to password based

authentication, you would replace the second declaration above by "`cachemgr_passwd`
`xxxx shutdown`" where "xxxx" is the password that must be entered for the shutdown
action. When you set some of the actions to password based authentication in this man-
ner, when you display the cache manager in your browser window, you will be shown
as to which actions require authentication.]

- For more general changes to the config file, note that each pa-
  rameter in the configuration file is referred to as a "**tag**" in a
  commented-out line. The default for each tag is shown below
  the commented-out section for a tag. If you are happy with the
  default, you can move onto to the next parameter.

- The very few parameters (tags) that you'd need to set for a simple
  one-machine application of Squid deal with:

  – The IP address of the interface though which the clients will
    be accessing the web proxy.

  – The IP addresses of the DNS nameservers (the '`dns_nameservers`'
    tag). (I recommend that for the application at hand, you leave
    it commented out. That will force the Squid daemon to look
    into the file '`/etc/resolve.conf/`' for the IP addresses of the
    nameservers. A manually specified entry for `dns_nameservers`
    in `quid.conf` overrides `/etc/resolv.conf` lookup.)

- Location of the local hostname/IP database file. For a Linux machine, this is typically `/etc/hosts`. This file is checked at startup and upon configuration.

- Definitions for *Access Classes*, abbreviated 'acl'. See the sample 'acl' definitions in the portion of the config shown later in this section.

- `http_access` declarations for the different 'acl' access classes. These declare as to who is allowed to access the web proxy for what services.

- Defining the effective user ID and group ID for the Squid processes that will be spawned for the incoming connections. (This is an important security issue.)

- Telling Squid whether or not you want the `forwarded_for` tag to be turned off to make the proxy anonymous. The default for this tag is 'on'. So, by default, the web proxy will forward a client's IP address to the remote web server.

- Specifying a password for the Cache Manager.

- Shown below is a very small section of the official configuration file `/etc/squid3/squid.conf`:

```
# This is the default Squid configuration file. You may wish
# to look at the Squid home page (http://www.squid-cache.org/)
# for the FAQ and other documentation.
#       .......
# NETWORK OPTIONS
# -----------------------------------------------------------------------------
#  TAG: http_port
#       Usage:   port
#                hostname:port
#                1.2.3.4:port
#       The socket addresses where Squid will listen for HTTP client
#       requests.
#       The default port number is 3128.
http_port 127.0.0.1:3128


#  TAG: https_port
#       .....
#  TAG: ssl_unclean_shutdown
#       .....
#  TAG: icp_port
#       .....


# OPTIONS WHICH AFFECT THE NEIGHBOR SELECTION ALGORITHM
# -----------------------------------------------------------------------------
#  TAG: cache_peer
#       .....
#  TAG: cache_peer_domain
#       .....
#  TAG: icp_query_timeout (msec)
#       ......
#  TAG: no_cache
#       A list of ACL elements which, if matched, cause the request to
#       not be satisfied from the cache and the reply to not be cached.
#       In other words, use this to force certain objects to never be cached.
#
#       You must use the word 'DENY' to indicate the ACL names which should
#       NOT be cached.
#
#We recommend you to use the following two lines.
acl QUERY urlpath_regex cgi-bin \?
no_cache deny QUERY


# OPTIONS WHICH AFFECT THE CACHE SIZE
# -----------------------------------------------------------------------------
#  TAG: cache_mem (bytes)
#       .....


# LOGFILE PATHNAMES AND CACHE DIRECTORIES
# -----------------------------------------------------------------------------
#  TAG: cache_dir
#       ......


# OPTIONS FOR EXTERNAL SUPPORT PROGRAMS
# -----------------------------------------------------------------------------
#  TAG: ftp_user
#       .....
#  TAG: cache_dns_program
#       .....
#Default:
# cache_dns_program /usr/local/squid/libexec/dnsserver


#  TAG: dns_children
# Note: This option is only available if Squid is rebuilt with the
#       --disable-internal-dns option
```

```
#
#          The number of processes spawn to service DNS name lookups.
# .....
#  TAG: dns_retransmit_interval
#          Initial retransmit interval for DNS queries. The interval is
#          doubled each time all configured DNS servers have been tried.
#
#Default:
# dns_retransmit_interval 5 seconds

#  TAG: dns_timeout
#          DNS Query timeout. If no response is received to a DNS query
#          within this time then all DNS servers for the queried domain
#          is assumed to be unavailable.
#Default:
# dns_timeout 2 minutes

#  TAG: dns_defnames on|off
# Note: This option is only available if Squid is rebuilt with the
#          --disable-internal-dns option
#          ......
#Default:
# dns_defnames off

#  TAG: dns_nameservers
#          Use this if you want to specify a list of DNS name servers
#          (IP addresses) to use instead of those given in your
#          /etc/resolv.conf file.
#          .......
#Default:
# none

#  TAG: hosts_file
#          Location of the host-local IP name-address associations
#          database.  Most Operating Systems have such a file: under
#          Un*X it's by default in /etc/hosts MS-Windows NT/2000 places
#          that in %SystemRoot%(by default
#          c:\winnt)\system32\drivers\etc\hosts, while Windows 9x/ME
#          places that in %windir%(usually c:\windows)\hosts
#          ......
#Default:
hosts_file /etc/hosts

#  TAG: diskd_program
#          Specify the location of the diskd executable.
#          .....
#  TAG: external_acl_type
#          This option defines external acl classes using a helper program to
#          look up the status
# .....

# OPTIONS FOR TUNING THE CACHE
# -----------------------------------------------------------------------------

#  TAG: wais_relay_host
#          ....
#  TAG: positive_dns_ttl time-units
#          Upper limit on how long Squid will cache positive DNS responses.
#          Default is 6 hours (360 minutes). This directive must be set
#          larger than negative_dns_ttl.
#
#Default:
# positive_dns_ttl 6 hours
```

66

```
# TIMEOUTS
# -----------------------------------------------------------------------------
#  TAG: forward_timeout time-units
#       This parameter specifies how long Squid should at most attempt in
#       finding a forwarding path for the request before giving up.
#
#Default:
# forward_timeout 4 minutes

#  TAG: connect_timeout time-units
#       This parameter specifies how long to wait for the TCP connect to
#       the requested server or peer to complete before Squid should
#       attempt to find another path where to forward the request.
#
#Default:
# connect_timeout 1 minute

#  TAG: peer_connect_timeout time-units
#       This parameter specifies how long to wait for a pending TCP
#       connection to a peer cache.  The default is 30 seconds.   You
#       may also set different timeout values for individual neighbors
#       with the 'connect-timeout' option on a 'cache_peer' line.
#
#Default:
# peer_connect_timeout 30 seconds

#  TAG: read_timeout time-units
#       The read_timeout is applied on server-side connections.  After
#       each successful read(), the timeout will be extended by this
#       .....
#Default:
# read_timeout 15 minutes

#  TAG: request_timeout
#       How long to wait for an HTTP request after initial
#       connection establishment.
#Default:
# request_timeout 5 minutes

#  TAG: persistent_request_timeout
#       How long to wait for the next HTTP request on a persistent
#       connection after the previous request completes.
#
#Default:
# persistent_request_timeout 1 minute

#  TAG: client_lifetime time-units
#       The maximum amount of time that a client (browser) is allowed to
#       .....

# ACCESS CONTROLS
# -----------------------------------------------------------------------------

#  TAG: acl
# Defining an Access List
#Recommended minimum configuration:
acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl to_localhost dst 127.0.0.0/8
acl SSL_ports port 443 563
acl SSH_port port 22 # ssh
acl Safe_ports port 80 # http
acl Safe_ports port 21 # ftp
```

```
acl Safe_ports port 443 563 # https, snews
acl Safe_ports port 70 # gopher
acl Safe_ports port 210 # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280 # http-mgmt
acl Safe_ports port 488 # gss-http
acl Safe_ports port 591 # filemaker
acl Safe_ports port 777 # multiling http
acl CONNECT method CONNECT


#  TAG: http_access
# Allowing or Denying access based on defined access lists
#        .....
#Default:
# http_access deny all

#Recommended minimum configuration:
#
# Only allow cachemgr access from localhost
http_access allow manager localhost
# The following line will deny cache manager access from any other host:
http_access deny manager
# Deny requests to unknown ports
# http_access deny !Safe_ports
# Deny CONNECT to other than SSL ports
# http_access deny CONNECT !SSL_ports
# The following needed by the corkscrew tunnel (SSH_port was previously
# defined to be access class consisting of port 22 that is assigned to
# the SSH Remote Login Protocol:
http_access allow CONNECT SSH_port
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports


# We strongly recommend to uncomment the following to protect innocent
# web applications running on the proxy server who think that the only
# one who can access services on "localhost" is a local user
#http_access deny to_localhost

# INSERT YOUR OWN RULE(S) HERE TO ALLOW ACCESS FROM YOUR CLIENTS:

# Example rule allowing access from your local networks. Adapt
# to list your (internal) IP networks from where browsing should
# be allowed
acl our_networks src 192.168.1.0/24 127.0.0.1
http_access allow our_networks

#  Note that 'src' above means 'source of request' as opposed to
# 'dest' for 'destination of request'.

# And finally deny all other access to this proxy
http_access deny all

#  TAG: http_reply_access
#        Allow replies to client requests. This is complementary
#        to http_access.
#
#        http_reply_access allow|deny [!] aclname ...
#
#        NOTE: if there are no access lines present, the default is to allow
# all replies
#
#        If none of the access lines cause a match, then the opposite of the
#        last line will apply. Thus it is good practice to end the rules
```

```
#          with an "allow all" or "deny all" entry.
#
#Default:
# http_reply_access allow all
#
#Recommended minimum configuration:
#
# Insert your own rules here.
# and finally allow by default
http_reply_access allow all


#  TAG: icp_access
#       Allowing or Denying access to the ICP port based on defined
#       access lists
#       .....
#Default:
# none


#  TAG: ident_lookup_access
#       A list of ACL elements which, if matched, cause an ident
#       (RFC 931) lookup to be performed for this request.  For
#       example, you might choose to always perform ident lookups
#       .....
#Default:
# ident_lookup_access deny all


#  TAG: tcp_outgoing_tos
#       Allows you to select a TOS/Diffserv value to mark outgoing
#       ......

# ADMINISTRATIVE PARAMETERS
# -----------------------------------------------------------------------------


#  TAG: cache_mgr
# Email-address of local cache manager who will receive
# mail if the cache dies.  The default is "webmaster."
#
#Default:
# cache_mgr webmaster
cache_mgr kak@localhost


#  TAG: cache_effective_user
#  TAG: cache_effective_group
#       If you start Squid as root, it will change its effective/real
#       UID/GID to the UID/GID specified below.  The default is to
#       .....
#       If Squid is not started as root, the cache_effective_user
#       value is ignored and the GID value is unchanged by default.
#       However, you can make Squid change its GID to another group
#       ......
#Default:
# cache_effective_user nobody
cache_effective_user squid
cache_effective_group squid
#  The above change is necessary if you want to start
#  squid to monitor port 3128 for incoming connections
#  Otherwise, squid will start as user 'root' and
#  then changeover to user 'nobody'. According to the
#  user's guide, as 'nobody', squid will not be able
#  to monitor a high numbered port such as 3128.


#  TAG: visible_hostname
#       If you want to present a special hostname in error messages, etc,
#       .....
```

```
# OPTIONS FOR THE CACHE REGISTRATION SERVICE
# -----------------------------------------------------------------------------
#        This section contains parameters for the (optional) cache
#        ......

# HTTPD-ACCELERATOR OPTIONS
# -----------------------------------------------------------------------------
#  TAG: httpd_accel_host
#  TAG: httpd_accel_port
#        If you want to run Squid as an httpd accelerator, define the
#        host name and port number where the real HTTP server is.
#        ......

# MISCELLANEOUS
# -----------------------------------------------------------------------------
#  TAG: dns_testnames
#        The DNS tests exit as soon as the first site is successfully looked up
#        ....
#  TAG: logfile_rotate
#        Specifies the number of logfile rotations to make when you
#        type 'squid -k rotate'.  The default is 10, which will rotate
#        .....
#  TAG: forwarded_for on|off
#        If set, Squid will include your system's IP address or name
#Default:
forwarded_for on
# The following option for the above tag makes the proxy anonymous
# to the web servers receiving the requests from this proxy's clients:
#forwarded_for off

#  TAG: header_replace
#        Usage:   header_replace header_name message
#        Example: header_replace User-Agent Nutscrape/1.0 (CP/M; 8-bit)
#
#        This option allows you to change the contents of headers
#        denied with header_access above, by replacing them with
#        some fixed string. This replaces the old fake_user_agent
#        option.
#
#        By default, headers are removed if denied.
#
#Default:
# none

#  TAG: cachemgr_passwd
#        Specify passwords for cachemgr operations.
#
#        Usage: cachemgr_passwd password action action ...
#
#        Some valid actions are (see cache manager menu for a full list):
#                5min
#                60min
#                asndb
#                authenticator
#                cbdata
#                client_list
#                comm_incoming
#                .....
#                .....
#                .....
#        * Indicates actions which will not be performed without a
#          valid password, others can be performed if not listed here.
#
```

```
#         To disable an action, set the password to "disable".
#         To allow performing an action without a password, set the
#         password to "none".
#
#         Use the keyword "all" to set the same password for all actions.
#
#Example:
# cachemgr_passwd secret shutdown
cachemgr_passwd xxxxxx all
# cachemgr_passwd lesssssssecret info stats/objects
# cachemgr_passwd disable all


#    and much much more
```

19.5: HARVEST: A SYSTEM FOR INFORMATION GATHERING AND INDEXING

- Since Squid was borne out of the Harvest project and since the Harvest project has played an influential role in the design of web-based search engines, I believe you need to know about Harvest.

- You can download Harvest from **http://sourceforge.net**. Download the source tarball in any directory (on my Linux laptop, this directory is named **harvest**). Unzip and untar the archive. Installation is very easy and, as in most cases, involves only the following three steps **as root**:

```
./configure
make
make install
```

  By default, this will install the configuration files and the executables in a directory called **/usr/local/harvest**. Set the environment variable **HARVEST_HOME** to point to this directory. So if you say '**echo $HARVEST_HOME**', you should get

```
/usr/local/harvest
```

19.5.1: What Does Harvest Really Do?

- Harvest gathers information from designated sources that may be reside on your own hard disk (it could be all of your local disk or just certain designated directories and/or files) or specified sources on the web in terms of their root URL's.

- Harvest then creates an efficiently searchable index for the gathered information. **(Ordinarily, an index is something you see at the end of a textbook. It is the keywords and key-phrases arranged alphabetically with pointers to where one would find them in the text book. An electronic index does the same thing — it is an efficiently searchable database of keywords and key-phrases along with pointers to the documents that contains them. More formally, an index is an associative table of key-value pairs where the keys are the words and the values the pointers to documents that contain those words.)**

- Eventually, Harvest serves out the index through an **index server**. A user interacts with the **index server** through a web interface.

- The index server in Harvest is called a **broker**. **(Strictly speaking, a Harvest broker first constructs the index and then serves it out through a web interface.)**

• Just as you can download the Google tool for setting up a search
facility for all of the information you have stored on the hard disk
of a Windows machine, you can do the same on a Linux machine
with Harvest.

19.5.2: Harvest: Gatherer

• Briefly speaking, a Gatherer's job is to scan and summarize the documents.

• Each document summary produced by a Gatherer is a SOIF object. SOIF stands for **Summary Object Interchange Format**. Here is a very partial list of the SOIF document attributes: `Abstract, Author, Description, File-Size, Full-Text, Gatherer-Host, Gatherer-Name, Gatherer-Port, Gatherer-Version, Update-Time, Keywords, Last-Modification-Time, MD5, Refresh Rate, Time-to-Live, Title, Type,` ......

• Before a Gatherer scans a document, it determines its type and makes sure that the type is not in a **stoplist**. Files named **stoplist.cf** and **allowlist.cf** play important roles in the functioning of a Gatherer. You would obviously **not** want audio, video, bitmap, object code, etc., files to be summarized, at least not in the same manner as you'd want files containing ASCII characters to be summarized.

• Gatherer sends the document to be summarized to the **Essence** sub-system. It is Essence that has the competence to determine the type of the document. If the type is acceptable for sum-

marization, it then applies a *type-specific* summary extraction
algorithm to the document. The executables that contain such
algorithms are called *summarizers*; these filenames end in the
suffix *.sum*.

• The Essence system recognizes a document type in three ways:
1) by URL naming heuristics; 2) by file naming heuristics; and,
finally, by locating identifying data within a file, as done by the
Unix `file` command. These three type recognition strategies are
applied to a document in the order listed here.

• A Gatherer makes its SOIF objects available through the

```
gatherd
```

daemon server on a port whose default value is 8500.

• When you construct a Gatherer, it is in the form of a directory
that contains two scripts

```
RunGatherer
RunGatherd
```

The first script, `RunGatherer`, starts the process of gathering
the information whose root nodes are declared in the Gatherer
configuration file. If you are trying to create an index for your
entire home directory (that runs into, say, several gigabytes), it
could take a couple of hours for the `RunGatherer` to do its job.

- When the first script, `RunGatherer`, is done, it automatically starts the **gatherd** server daemon. For a database collected by a previous run of `RunGatherer`, you'd need to start the server daemon **gatherd** manually by running the script `RunGatherd`.

19.5.3: Harvest: Broker

• As mentioned previously, a Broker first constructs an index from the SOIF objects made available by the `gatherd` server daemon and serves out the index on a port whose default value is 8501.

• By default, Harvest uses Glimpse as its indexer. The programs that are actually used for indexing are

> `/usr/local/harvest/lib/broker/glimpse`
> `/usr/local/harvest/lib/broker/glimpseindex`

Note that `/usr/local/harvest/` is the default installation directory for the Harvest code.,

• When `glimpse` is the indexer, the broker script `RunBroker` calls on the following server program

> `/usr/local/harvest/lib/broker/glimpseserver`

to serve out the index on port 8501.

• See the User's Manual for how to use other indexers with Harvest. Examples of other indexers would be WAIS (both freeWAIS and commercial WAIS) and SWISH. The User's Manual is located at

> `DownLoadDirectory/doc/pdf/manual.pdf`
> `DownLoadDirectory/doc/html/manual.html`

19.5.4: How to Create a Gatherer?

- Let's say I want to create a gatherer for my home directory on my Linux laptop. This directory occupies about 3 gigabytes of space. The steps for doing so are described below.

- We will call this gatherer `KAK_HOME_GATHERER`.

- To create this gatherer, I'll log in as root and do the following:

```
cd $HARVEST_HOME              ( this is /usr/local/harvest )

cd gatherers

mkdir KAK_HOME_GATHERER       ( As already noted, this will also be the
                                name of the new gatherer )
cd KAK_HOME_GATHERER

mkdir lib                     ('lib' will contain the configuration files
                               used by the gatherer.  See explanation
                               below.)

mkdir bin                     ('bin' will contain any new summarizers
                               you may care to define for new document
                               types.)
cd lib

cp $HARVEST_HOME/lib/gatherers/*.cf .
cp $HARVEST_HOME/lib/gatherers/magic .
```

- The last two steps listed above will deposit the following files in the `lib` directory of the gatherer directory:

```
bycontent.cf
byname.cf
byurl.cf

magic

quick-sum.cf

stoplist.cf

allowlist.cf
```

- About the first three files listed above, these three files are to help the Essence system to figure out the type of a document. The `bycontent.cf` file contains the content parsing heuristics for type recognition by Essence. Similarly, the file `byname.cf` contains the file naming heuristics for type recognition; and the file `byurl.cf` contains the URL naming heuristics for type recognition. Essence uses the above three files for type recognition in the following order: `byurl.cf`, `byname.cf`, and `bycontent.cf`. Note that the second column in the `bycontent.cf` is the regex that must match what would be returned by calling the Unix command 'file' on a document.

- About the file `magic`, the numbers shown at the left in this file are used by the Unix 'file' command to determine the type of a file. The 'file' command must presumable find a particular string at the byte location given by the magic number in order

to recognize a file type. The bytes that are found starting at the magic location must correspond to the entry in the third column of this file.

- About the file `quick-sum.cf`, this file contains some regexes that can be used for determining the values for some of the attributes needed for the SOIF summarization produced by some of the summarizers.

- About the file `stoplist.cf`, it contains a list of file object types that are rejected by Essence. So there will be no SOIF representations produced for these object types.

- For my install of Harvest, I found it easier to use an `allowlist.cf` file to direct Essence to accept only those document types that are placed in `allowlist.cf`. However, now you must now supply Essence with the '`-allowlist`' flag. This flag is supplied by including the line

      Essence-Options: -allowlist

  in the header section of the `KAK_HOME_GATHERER.cf` config file to be described below.

- Now do the following:

      cd ..   (this puts you back in KAK_HOME_GATHERER directory)

  For now, ignore the `bin` sub-directory in the gatherer directory. The `bin` directory is for any new summarizers you may create.

• Now copy over the configuration file from one of the "example" gatherers that come with the installation:

```
cp ../example-4/example-4.cf   KAK_HOME_GATHERER.cf
```

In my case, I then edited the KAK_HOME_GATHERER.cf file so that it had the functionality that I needed for scanning my home directory on the laptop. My KAK_HOME_GATHERER.cf looks like

```
#
#  KAK_HOME_GATHERER.cf - configuration file for a Harvest Gatherer
#

#  It is possible list 23 options below before you designate RootNodes
#  and LeafNodes.  See page 38 of the User's Manual for a list of these
#  options.

#  Note that the default for TTL is one month and for Refresh-Rate
#  is one week.  One week equals 604800 seconds.  I have set TTL
#  to three years and the Refresh-Rate to one month.

#  Post-Summarising did not work for me.  When I run RunGatherer
#  I get the error message in log.errors that essence cannot parse
#  the rules file listed against this option below.

Gatherer-Name:  Avi Kak's Gatherer for All Home Files
Gatherer-Port: 8500
Access-Delay:           0
Top-Directory:          /usr/local/harvest/gatherers/KAK_HOME_GATHERER
Debug-Options:          -D40,1 -D64,1
Lib-Directory:          ./lib
Essence-Options:        --allowlist ./lib/allowlist.cf
Time-To-Live:           100000000
Refresh-Rate:           2592000
#Post-Summarizing:       ./lib/myrules


# Note that Depth=0 means unlimited depth of search.
# Also note that the content of the RootNodes element needs to be
# in a single line:
<RootNodes>
file:///home/kak/ Search=Breadth Depth=0 Access=FILE \
    URL=100000,mydomain-url-filter HOST=10,mydomain-host-filter
```

```
</RootNodes>
```

• Similarly, copy over the scripts **RunGatherer** and **RunGatherd** from one of the **example** gatherers into the **KAK_HOME_GATHERER** directory. *You would need to edit at least two lines in Run-Gatherer so that the current directory is pointed to. You'd also need to edit the last line of RunGatherd for the same reason.* My **RunGatherer** script looks like

```
#!/bin/sh

HARVEST_HOME=/usr/local/harvest; export HARVEST_HOME

# The following sets the local disk cache for the gatherer to 500 Mbytes.
HARVEST_MAX_LOCAL_CACHE=500; export HARVEST_MAX_LOCAL_CACHE

# The path string added at the beginning is needed by essence to
# to locate the new summarizer ScriptFile.sum
PATH=${HARVEST_HOME}/gatherers/KAK_HOME_GATHERER/bin:\
${HARVEST_HOME}/bin:${HARVEST_HOME}/lib/gatherer:${HARVEST_HOME}/lib:$PATH

export PATH

NNTPSERVER=localhost; export NNTPSERVER

cd /usr/local/harvest/gatherers/KAK_HOME_GATHERER
sleep 1
'rm -rf data tmp log.*'
sleep 1
exec Gatherer "KAK_HOME_GATHERER.cf"
```

and my **RunGatherd** script looks like

```
#!/bin/sh
```

```
#
#  RunGatherd - Exports the KAK_HOME_GATHERER Gatherer's database
#
HARVEST_HOME=/usr/local/harvest; export HARVEST_HOME
PATH=${HARVEST_HOME}/lib/gatherer:${HARVEST_HOME}/bin:$PATH; export PATH
exec gatherd -d /usr/local/harvest/gatherers/KAK_HOME_GATHERER/data 8500
```

Note that I have included the command 'rm -rf tmp data log.*
in the RunGatherer script for cleanup before a new gathering ac-
tion.

- Similarly, copy over the filter files

    mydomain-url-filter
    mydomain-host-filter

  from the example-5 gatherer into the KAK_HOME_GATHERER di-
  rectory. Both of these files are mentioned against the RootNode
  in the gatherer configuration file KAK_HOME_GATHERER.cf. My
  mydomain-url-filter file looks like

```
# URL Filter file for 'mydomain'
#
# Here 'URL' really means the pathname part of a URL.  Hosts and ports
# dont belong in this file.
#
# Format is
#
#    Allow regex
#    Deny regex
#
# Lines are evaulated in order; the first line to match is applied.
#

# The files names that are denied below will not even be seen by the
# essence system.  It is more efficient to stop files BEFORE the
```

```
# gatherer extracts information from them.  Compared to this action by
# mydomain-url-filter, when files are stopped by the entries in
# byname.cf, bycontent.cf, and byurl.cf, that happens AFTER the
# information is extracted from those files by the gatherer.

Deny \.gif$ # don't retrieve GIF images
Deny \.GIF$ # #
Deny \.jpg$ # #
Deny \.JPG$ # #
Deny /\..+                           # don't index dot files
Deny \.pl\.                          # don't index OLD perl code
Deny \.py\.                          # don't index OLD python code
Deny    /home/kak/tmp                   # don't index files in my tmp
Deny    ~$                              # don't index tilde files
Deny    /,                              # don't index comma files
Allow .* # allow everything else.
```

and my **mydomain-host-filter** file looks like

```
# Host Filter file for 'mydomain'
#
# Format is
#
#    Allow regex
#    Deny regex
#
# Lines are evaulated in order; the first line to match is applied.
#
# 'regex' can be a pattern for a domainname, or IP addresses.
#
Allow .*\.purdue\.edu         # allow hosts in Purdue domain
#Allow ^10\.128\. # allow hosts in IP net 10.128.0.0
Allow ^144\.46\. # allow hosts in IP net 144.46.0.0
Allow ^192\.168\. # allow hosts in IP net 192.168.0.0
Deny    .* # deny all others
```

- Apart from the fact that you may wish to create your own sum-

marizers (these would go into the `bin` directory of your gatherer, you are now ready to run the **RunGatherer**.

• You can check the output of the `gatherd` daemon that is automatically started by the **RunGatherer** script after it has done its job by

```
$HARVEST_HOME/bin/gather localhost 8500 | more
```

assuming that the database collected is small enough. You can also try

```
cd data
$HARVEST_HOME/lib/gatherer/gdbmutil stats PRODUCTION.gdbm
```

This will return the number of SOIF objects collected by the gatherer.

• As already mentioned, if you create a new summarizers in the `bin` directory of the gatherer, you also need a pathname to the this `bin` directory in the **RunGatherer** script.

• Finally, in my case, the `KAK_HOME_GATHERER` had trouble gathering up Perl and Python scripts for some reason. I got around this problem by defining an object type **ScriptFile** in the `bycontent.cf` configuration file in the `lib` directory of the gatherer. I also defined an object type called **Oldfile** in the `byname.cf` configuration file of the same directory. Since I did not include the type **OldFile** in my `allowlist.cf`, essence did not summarize any files that were of type **OldFile**. However, I

did include the type `ScriptFile` in `allowlist.cf`. So I had to provide a summarizer for it in the `bin` directory of the gatherer. The name of this summarizer had to be `ScriptFile.sum`.

19.5.5: How to Create a Broker?

• Log in a root and start up the httpd server by

    sudo /usr/local/apache2/bin/apachectl start

Actually, the httpd server starts up automatically in my case when I boot up the laptop since the above command is in my /etc/rc.local file.

• Now do the following:

    cd $HARVEST_HOME/bin

    CreateBroker

This program will prompt for various items of information related to the new broker you want to create. The first it would ask for is the name you want to use for the new broker. For brokering out my home directory on the Linux laptop, I called the broker KAK_HOME_BROKER. This then becomes the name of the directory under $HARVEST_HOME/brokers for the new broker. **If you previously created a broker with the same name, you'd need to delete that broker directory in the $HARVEST_HOME/brokers directory. You would also need to delete a subdirectory of that name in the $HARVEST_HOME/tmp directory.**

• Another prompt you get from the CreateBroker program is "*Enter the name of the attribute that will be displayed to the*

user as one-line object description in search results [description]:". The 'description' here refers to the SOIF attribute that will be displayed in the first line when query retrieval is displayed in the browser.

• Toward to the end of the broker creation procedure, say 'yes' to the prompt *"Would you like to add a collection point to the Broker now?"*. This will connect the `gatherd` daemon process running on port 8500 with the broker process.

• You will be prompted one more time with the same question as listed above. Now say "no".

• CreateBroker deposits the following executable shell file

```
RunBroker              (Make sure you kill off any previously
                        running broker processes before you
                        do this.)
```

in the new broker directory.

• Now fire up the broker by

```
        RunBroker -nocol
```

in the broker directory. The option '-nocol' is to make certain that the gatherer does not start collecting again when you invoke the RunBroker command. We are obviously assuming that you have established a gatherer separately and that it is already up and running. **If you have gathered up the information**

but the server 'gatherd' is not running to serve out the SOIF objects, execute the `RunGatherd` script in the gatherer directory. The `RunBroker` command starts up the `glimpseindex` daemon server.

• When you ran **CreateBroker**, that should also have spit out a URL to an HTML file that you can bring up in the browser to see the new searchable database. Or, in the broker directory, you can just say

```
        cd $HARVEST_HOME/brokers/KAK_HOME_BROKER

        firefox query.html
    or
        firefox index.html
    or
        firefox stats.html
```

• Whether or not you can see the query form page may depend on whether you use the URL returned by the **CreateBroker** command or whether you make a direct call with '**firefox query.html**'. The former uses the HTTP protocol and therefore goes through the Apache HTTPD server, whereas the latter would use the FILE protocol and would be handled directly by the firefox web browser.

• Assuming you use the http protocol for seeing the query form, let's say you get the error number 500 (in the **error_log** file in the **$APACHEHOME/logs** directory). This means that

$APACHEHOME/conf/httpd.conf is misconfigured. In particular, you need the following directive in the httpd.conf file:

```
ScriptAlias /Harvest/cgi-bin/ "/usr/local/harvest/cgi-bin/"
Alias /Harvest/ "/usr/local/harvest/"
<Directory "/usr/local/harvest">
    Options FollowSymLinks
</Directory>
```

for the HTTPD server to be able to find the search.cgi that is in the $HARVEST_HOME/cgi-bin/ directory.

- Finally, for the case of constructing an index for your own home directory (such as my /home/kak/), you may be able to see the search results, but clicking on an item may not return that item in the browser. That is because of the security setting in firefox browsers; this setting keeps the browser from displaying anything in response to the FILE protocol (as opposed to the HTTP protocol). You may to change the settings in the file .mozilla/firefox/qwjvm1oo.default/user.js of your home account for firefox to be able to show local files.

- After you have crated a new broker for a gatherer that previously collected its database, make sure you execute the following scripts:

```
RunGatherd                (in the gatherer directory)

RunBroker                 (in the broker directory)
```

The former runs the gatherd daemon server to serve out the

SOIF objects on port 8500 and the latter first constructs the index for the database and then run the `glimpserver` daemon to serve out the index on port 8501.

• After you have started **RunBroker**, watch the cpu meter. For the entire home directory, it may take a long time (up to 20 minutes) for the broker to create the index from the SOIF records made available by the `gatherd` daemon. *It is only after the* **RunBroker** *command has finished creating an index for the database that you can carry out any search in the browser.*

• If your scripts `RunGatherd` and `RunBroker` scripts are running in the background, if you want to search for something that is being doled out by Harvest, you can point your browser to

    http://localhost/Harvest/brokers/KAK_HOME_BROKER/admin/admin.html

    http://pixie.ecn.purdue.edu/Harvest/brokers/KAK_HOME_BROKER/query.html

• I have placed the command strings

    /usr/local/harvest/gatherers/KAK_HOME_GATHERER/RunGatherd

    /usr/local/harvest/brokers/KAK_HOME_BROKER/RunBroker

in **/etc/rc.local** so that the SOIF object server **gatherd** and the index server **glimpseserver** will always be on when the machine boots up.

19.6: CONSTRUCTING AN SSH TUNNEL THROUGH AN HTTP PROXY

- SSH tunneling through HTTP proxies is typically carried out by sending an HTTP request with the method `CONNECT` to the proxy. The HTTP/1.1 specification reserves the method `CONNECT` to enable a proxy to dynamically switch to being a tunnel, such as an SSH tunnel (for SSH login) or an SSL tunnel (for the HTTPS protocol). [**Here are all the HTTP/1.1 methods: GET, POST, OPTIONS, HEAD, PUT, DELETE, TRACE, and CONNECT.**]

- The two very commonly used programs that send a `CONNECT` request to an HTTP proxy are **corkscrew** and **connect**.

- The first of these, **corkscrew**, comes as a tar ball with config, make, and install files. You install it by calling, '`./config`', '`make`', and '`make install`'. My advice would be to **not** go for '`make install`'. Instead, place the `corkscrew` executable in the `.ssh` directory of your home account.

- The second of these, **connect**, comes in the form of a C program, `connect.c`, that is compiled easily by a direct call to gcc. Again

place the executable, `connect`, in your .ssh directory.

• The most convenient way to use either the `corkscrew` executable or the `connect` executable is by creating a 'config' file in your .ssh directory and making 'ProxyCommand' calls to these executables in the 'config' file. Here is my `~kak/.ssh/config` file

```
Host=*
#  The '-d' flag in the following ProxyCommand is for debugging:
#   ProxyCommand ~/.ssh/connect -d -H localhost:3128 %h %p
#  ProxyCommand ~/.ssh/connect -H localhost:3128 %h %p
   ProxyCommand ~/.ssh/corkscrew localhost 3128 %h %p
```

where the `Host=*` line means that the shown "ProxyCommand" can be used to make an SSH connection with all hosts. A regex can be used in place of the wildcard '*' if you want to place restrictions on the remote hostnames to which the proxycommand applies. What you see following the keyword "ProxyCommand" is what will get invoked when you call something like '`ssh moonshine.ecn.purdue.edu`'. For the uncommented line that is shown, this means that the `corkscrew` program will be called to tunnel through Squid by connecting with it on its port 3128. **(See the manpage for `ssh_config`)** If you want to use `connect` instead of `corkscrew`, comment out and uncomment the lines in the above file as needed.

• But note that when your .ssh directory contains a 'config' file, all invocations of SSH, even by other programs like 'rsync' and 'fetchmail', will be mediated by the content of the config file in the .ssh directory.

• To get around the difficulty that may be caused by the above, you can use the shell script 'ssh-proxy' (made available by Eric Engstrom) in your .ssh directory.

• You can construct an SSH tunnel through an HTTP proxy server only if the proxy server wants you to. Let's say that SQUID running on your own machine is your HTTP proxy server. Most sites running the SQUID proxy server restrict CONNECT to a limited number of whitelisted hosts and ports. In a majority of cases, the proxy server will allow CONNECT outgoing requests to go only to port 443. (This port is monitored by HTTPS servers, such as the Purdue web servers, for secure web communication with a browser. When you make an HTTP request to Purdue, it goes to port 80 at the Purdue server. However, when you make an HTTPS request, it goes to port 443 of the server.)

• An HTTP proxy, such as SQUID, must allow the CONNECT method to be sent out to the remote server since that is what is needed to establish a secure communication link. I had to place the following lines in the `squid.conf` file for my SQUID proxy server to allow for an SSH tunnel:

```
acl SSH_port port 22              # ssh
http_access allow CONNECT SSH_port
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
```

• What makes getting the corkscrew/connect based tunnels through

the SQUID proxy server to work very frustrating was that even
when you completely kill the squid process by sending it the
'`kill -9 pid`' command, and then when you try to make an
ssh login, you get the following sort of an error message

```
ssh_exchange_identification: Connection closed by remote host
```

This message holds no clue at all to the effect that the proxy
server, SQUID, has been shut down. I believe the message is
produced by the SSH client program. I suppose that from the
perspective of the client program, the proxy server is no different
from a remote server.

- To see you have made an SSH connection through the SQUID
  proxy, check the latest entry in the log file
  `$SQUID_HOME/var/logs/access.log`.

- So what is one supposed to do when the HTTP proxy server
  won't forward a CONNECT request to the remote SSH server
  on, say, port 22 (the standard port that the SSH server on the
  remote machine will be monitoring)?

- If the highly restrictive proxy server on your company's premises
  would not send out CONNECT requests to the SSHD standard
  port 22 on the remote machine, you could try the following ploy:
  *You could ask the SSHD server (running on a machine like
  moonshine.ecn.purdue.edu) to monitor a non-standard port
  (***in addition** *to monitoring the standard port) by:*

```
/usr/local/sbin/sshd -p 563
```

where the port 563 is typically used by NNTPS. [The assumption
is that the highly restrictive HTTP proxy server that your com-
pany might be using would allow outbound proxy connections for
ports 563 (NNTPS) and 443 (HTTPS). If 563 does not work, try
443.]

• Now, on the client side, you can place the following line in the
`~/.ssh/config` file:

```
Host moonshine.ecn.purdue.edu
ProxyCommand corkscrew localhost 3128 moonshine.ecn.purdue.edu 563
```

• Another approach is to use Robert MaKay's GET/POST based
"tunnel" that uses Perl scripts at both ends of a SSH connection.
There is only one disadvantage to this method: you have to run
a server script also in addition to the client script. But the main
advantage of this method is that it does NOT care about the
CONNECT restrictions in the web proxy that your outbound
http traffic is forced to go through.

19.7: HOMEWORK PROBLEMS

1. What do we mean by "shim layer" in the TCP/IP protocol stack?

2. What is an anonymizing proxy in a network? In which layer of the TCP/IP protocol stack does an anonymizing proxy server belong?

3. Let's say you are installing a SOCKS proxy for a LAN that you are the admin for. This proxy requires that you install a SOCKS server on a designated machine that is directly connected to the internet and that you install the SOCKS client software on all of the machines in the LAN. Why do you think you need both a server and a client for the proxy to work?

4. What is the standard port assigned to the SOCKS server?

5. What are the main differences between the SOCKS4 and the SOCKS5 implementations of the SOCKS protocol?

6. What are the essential elements of the negotiation between a SOCKS client and a SOCKS server before the latter agrees to

forward the client's request? How does the server tell the client that the latter's request cannot be granted?

7. Why is a SOCKS proxy also referred to as a "circuit level proxy?"

8. What is meant by socksifying an application?

9. What is meant by jargon phrases such as "port forwarding" and "tunneling"?

10. How can you make sure that when you go through an anonymizing proxy, your IP address is not visible to the remote server?

11. What is web caching? How is an HTTP proxy used for web caching?

12. What is the average size of an internet object — according to folks who compile such stats? If an ISP allocates, say, 4 Gbytes of memory to a web caching server like Squid, what is the maximum number of internet objects that could be stored in such a cache? Additionally, how much RAM would you need to hold the object index for all the objects stored in the cache?

13. If you run a web caching proxy such as Squid on your own laptop, how would you tell your browser that it needs to route all its

requests through the proxy?

14. What is the role of a cache manager vis-a-vis a proxy server such as Squid?

15. The option '-D' given to a SOCKS server when you first bring it up means something that is completely different from what the same option means for a Squid server. What is the difference?

16. What historical role has the Harvest information gathering and indexing system played in the evolution of the modern internet search engines?

17. What does a `broker` do in Harvest? Also, what is the function of a `gatherer`?