

Lecture 19: Proxy-Server Based Firewalls

Lecture Notes on “Introduction to Computer Security”

by Avi Kak (kak@purdue.edu)

April 12, 2007

©2007 Avinash Kak, Purdue University

Goals:

- Packet filtering firewalls vs. proxy server firewalls
- The SOCKS protocol for anonymizing proxy servers
- Socksifying application clients
- The Dante SOCKS server
- Squid for controlling access to web resources (and for web caching)
- The very, very, very large Squid configuration file
- The Harvest system for information gathering, indexing, and searching through a web interface
- How to construct an SSH tunnel through a web proxy

Firewalls in General

- Firewalls can be designed to operate at any of the following three layers in the TCP/IP protocol stack:
 - Application Layer (example: HTTP Proxy)
 - Transport Layer (example: packet filtering with iptables)
 - the layer in-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 order to determine whether or not to change any of the header fields. (See **Lecture 18 slides/notes 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**.

- Transport Layer firewalls based on packet filtering and Application Layer firewalls implemented with the help of proxy servers often coexist 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.)
- 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.
- 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** between the two layers — to trap the application-level calls from intranet clients for connection to the servers on the internet. (**Note that this shim layer corresponds to the session layer in the OSI model. See Lecture 16 for the OSI model**)
- Using such a protocol, a proxy server can monitor all session re-

quests 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 rest of intranet can "hide" behind the proxy server.* The computers in the internet would not even know about the existence of your machine on 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 an 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.
- In addition to achieving firewall security, a proxy server operating

at the application layer or the layer between the application layer and the transport 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 such proxy servers.

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 the LAN client so that they 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 servers checks the session request made by the 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 to 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 then forwarding them 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 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 on the LAN can maintain complete anonymity with respect to the internet.

- *This would cause all of the HTTP traffic emanating from the intranet to get routed through the SOCKS server where it would be subject to your firewall rules and where, if desired, you could provide logging facilities and caching of the web services.*

SOCKS4 versus SOCKS5

- In all of the negotiation messages (that we showed on the previous few slides) between a SOCKS client and a SOCKS server, the first byte, VER, designates the version number. The value of this field is either **0x04** for version 4 or **0x05** for version 5 of the SOCKS protocol.
- Version 4 (usually referred to as SOCKS4) lacks the client-server authentication we have described above. 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 that we did not go into in our discussion above. 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 IP addresses of the remote hosts (meaning to carry out a DNS lookup). SOCKS5 is able to move DNS name resolution to the proxy server that, if necessary, can access a remote DNS server.

Interaction Between a SOCKS Client and a SOCKS Server

- To see how a SOCKS client interacts with a SOCKS server, let's say that a LAN client wants to access an HTTP service on the internet.
- The SOCKS client opens a TCP connection with the SOCKS server on port 1080. The client sends a "Client Negotiation" packet suggesting a few authentication methods. This packet consists of the following fields:

Client Negotiation:	VER	NMETHOD	METHODS
	1	1	1-255

with 1 byte **VER** devoted to the version number (SOCKS4 or SOCKS5), 1 byte 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. (The value 0x00 in a methods field means no authentication needed, the value 0x01 means authentication according to the GSSAPI (Generic Security Services Application Programming Interface), 0x02 means a user-name/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; it is this body that manages the RFC's, controls the top-level domain names such as '.edu', '.com', '.uk', '.museum', etc.) 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.

- If the authentication succeeds, the SOCKS client then sends the SOCKS proxy server a request stating what service it wants at what address 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 Associate". (The ATYP field stands for the "Address Type" field. It takes one of

the 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 field stores 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 **two replies** to the SOCKS client. 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

- First "Server Reply" is sent to the SOCKS client when the proxy server makes its connection request to the remote server and the second upon either the success or the failure of that connection request.
- 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. Note that the SOCKS protocol allows for a secure communication link (using, say, SSL/TLS) between a SOCKS client and a SOCKS server.
- 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. In other words, as far as the SOCKS server is concerned, there is no difference between an HTTP session and a 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 an intranet client and the remote host in the internet such a proxy server is also referred to as a *circuit-level proxy*.

Socksifying an Application

- Turning a usual application client (such as a web browser, an email client, etc.) 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
```

- 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, mozilla firefox) through a SOCKS proxy server, you would invoke

```
socksify firefox
```

By the way, 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. I had originally believed that all you would need to do to get a browser to work through the SOCKS server was to click on “Manual Proxy Configuration” in the window that comes up for Edit-Preferences-Advanced-Network-Settings and enter the IP address and the port for the SOCKS proxy server. But that did not work.

- For yet another example that we will discuss in greater detail later, suppose I have a custom Perl client script — let’s call it **SockExptClient.pl** — that can engage in an interactive session with a custom Perl server script 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:

```
SockExptClient.pl 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) on your local gateway machine, all you’d need to do is to call

```
socksify SockExptClient.pl 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 'README.usage' 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.

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 is **1.1.19**. A standard install of Dante gives you the following:

```
/usr/share/doc/dante-1.1.19/  ---  various useful README files

/usr/sbin/sockd               ---  the SOCKS5 server

/etc/init.d/sockd             ---  the script for starting/stopping/restarting
                                   the server

/var/log/sockd                ---  the server log file

/etc/sockd.conf               ---  the server configuration file

/etc/sock.conf                ---  the client configuration file

/lib/libdsocks.so             ---  the shared library for socksifying a client
```

- Although you can start the server by executing as root:

```
/etc/init.d/sockd start
```

initially you'd want to start up the server in the debug mode by executing as root:

```
/usr/sbin/sockd -d
```

- However, before you fire up the server, you'd want to edit the *server* configuration file `/etc/sockd.conf` and the *client* configuration file `/etc/socks.conf`. The next few slides address this issue.

Configuring the SOCKS 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 **ath0**. We will therefore not worry about client-server authentication here.
- The server config file, **/etc/sockd.conf**, consists of three sections:
 - Server Settings
 - Rules
 - Routes
- **Server settings, logoutput:** Where the log messages should be sent to.
- **Server settings, 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.

- **Server Settings, 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 ath0) 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, ath0, etc) can only work with a legal IP address that an upstream 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 understandd) you use

a SOCKS router behind a home 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.

- **Server Settings, 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.
- **Server Settings, 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 **sockd** if you wish. Ignore it for now since we will not be doing any client authentication.
- **Server Settings, user.unprivileged:** This specifies as to what read/write/execute the server should be set to. Set it to **nobody** which does not any privileges at all with regard to all other programs in your computer.
- **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 sockd.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 `sockd.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/sockd.conf` file for examples.

- The third and final section of the `/etc/sockd.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.

An Example of a /etc/sockd.conf SOCKS Server Config File

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

#####      ServerSettings      #####

# the server will log both via syslog, to stdout and to /var/log/sockd
logoutput: syslog stdout /var/log/sockd

internal: 127.0.0.1 port = 1080

# all outgoing connections from the server will use the IP address
# 195.168.1.1
external: ath0          # See my Slide 23 for what it means to
                        # to run a SOCKS server behind a home router

# list over acceptable methods, 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 that can require privilege, it will use the userid "sockd".
#user.privileged: sockd

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

# do you want to accept connections from addresses without dns info?
# what about addresses having a mismatch in dnsinfo?
srchost: nunknown nomismatch
```

RULES

There are two kinds and they work at different levels. (See Slides 24)

#

#===== rules checked first =====

Allow our clients, also provides an example of the port range command.

#client pass {

from: 10.0.0.0/8 port 1-65535 to: 0.0.0.0/0

method: rfc931 # match all idented users that also are in passwordfile

#}

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 =====

#block {

from: 0.0.0.0/0 to: 127.0.0.0/8

log: connect error

#}

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

}

block {

from: 0.0.0.0/0 to: 0.0.0.0/0

log: connect error

}

See /etc/sockd.conf of your installation for additional examples

of such rules.

Configuring SOCKS Clients

- The *client* configuration file `/etc/socks.conf` regulates the behavior of a *socksified client*. (As stated earlier on Slide 15, you *socksify* a client application either by supplying the call to the application as an argument to the script `/usr/bin/socksify`, or by recompiling the client application and dynamically linking the executable with the `libsockd.so` library, or by a static recompilation of the client application as explained in the `README.usage` documentation that comes with Dante.)
- At the beginning of the client configuration file, `/etc/socks.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 whether you want the SOCKS client to carry out the DNS lookup directly or whether you want to leave the DNS lookup to the SOCKS server. In my client config file, this rule looks like

```
route {
```

```

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

```

This rule says that we want all “bind” commands (means the DNS lookup requests) to go to the local SOCKS server running on port 1080.

- See the official `/etc/socks.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  
}
```

An Example of a /etc/socks.conf SOCKS Client Config File

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

#debug: 1

# Send the dns lookups to the proxy server:
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
}
```

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 Slide 39.**)
- 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.

A Simple Example of Running an Application Client through a SOCKS Proxy Server on Your Machine

- To understand this example, please keep straight the meaning to be associated with each of the following:
 - an **application server**
 - an **application client**,
 - the **SOCKS** server, and
 - the **socksified client**
- Ordinarily, when SOCKS is not involved, you will run the client program on your machine and this program will talk to the application server on some remote 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.

- The application server, named `SockExptServer.pl` is shown below:

```
#!/usr/bin/perl -w
use strict;

### SockExptServer.pl

### This script is from the forthcoming ‘Scripting with Objects’ by
### Avinash Kak, John-Wiley, 2007

use IO::Socket;
use Net::hostent;

my $server_soc = IO::Socket::INET->new( LocalPort => 9000,
                                         Listen   => SOMAXCONN,
                                         Proto     => 'tcp',
                                         Reuse      => 1);

die "No Server Socket" unless $server_soc;

print "[Server $0 accepting clients]\n";
while (my $client_soc = $server_soc->accept()) {
    print $client_soc "Welcome to $0; type help for command list.\n";
    my $hostinfo = gethostbyaddr($client_soc->peeraddr);
    printf "\n[Connect from %s]\n",
           $hostinfo ? $hostinfo->name : $client_soc->peerhost;
    print $client_soc "Command? ";
    while ( <$client_soc> ) {
        next unless /\S/;
        printf "    client entered command: %s\n", $_;
        if (/quit|exit/i) { last; }
    elsif (/date|time/i) { printf $client_soc "%s\n",
                                   scalar localtime; }
    elsif (/ls/i )      { print $client_soc 'ls -al 2>&1'; }
    elsif (/pwd/i )     { print $client_soc 'pwd 2>&1'; }
    elsif (/user/i)     { print $client_soc 'whoami 2>&1'; }
    elsif (/rmtilde/i)  { system "rm *~"; }
    else {
        print $client_soc
            "Commands: quit exit date ls pwd user rmtilde\n";
    }
    } continue {
        print $client_soc "Command? ";
    }
}
```

```
    }  
    close $client_soc;  
}
```

- As you can see, the application server 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.

- We will run this server on **moonshine.ecn.purdue.edu** by invoking

SockExptServer.pl

- Shown below is a client script, **SockExptClient.pl**, which we will run in a home LAN machine with the intranet using the Class C network address range **192.168.1.0/24**. This client interacts with the above server in an interactive session. The server prompts the client to enter one of the permissible commands and the server then executes that command.

```

#!/usr/bin/perl -w
use strict;

### SockExptClient.pl

### This script is from the forthcoming ‘Scripting with Objects’ by
### Avinash Kak, John-Wiley, 2007

use IO::Socket;

die "usage: $0 host port" unless @ARGV == 2;

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

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

$SIG{INT} = sub { $socket->close; exit 0; };

print STDERR "[Connected to $host:$port]\n";

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

# Parent process: receive information from the remote site:
if ($pid) {
    STDOUT->autoflush(1);
    my $byte;
    while ( sysread($socket, $byte, 1) == 1 ) {
        print STDOUT $byte;
    }
    kill("TERM", $pid);
} else {
    # Child process: send information to the remote site:
    my $line;
    while (defined ($line = <STDIN>)) {
        print $socket $line;
    }
}

```

- We now **socksify** the client by using the command line

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

- The above call will work the same as before. 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 SOCKS server on the client machine. To easily see the client requests going through the SOCKS proxy server, start up the server with (*while you are logged in as root on your Linux machine*):

```
/usr/sbin/sockd -d
```

Of course, as previously mentioned, for “production” purposes you’d fire up the SOCKS server by

```
/etc/init.d/sockd start
```

and stop it by

```
/etc/init.d/sockd stop
```

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 servers 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 information also.
- Squid was originally derived from the Harvest project. More on that later.
- 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).

Starting and Stopping the Squid Proxy Server

- For a standard Linux install of Squid, the pathname/name of the configuration file is:

```
/usr/local/squid/etc/squid.conf
```

- For a standard Linux install of Squid, the directory structure of the installation looks like

```
/usr/local/squid/  
    /bin/  
    /etc/  
    /libexec/  
    /man/  
    /sbin/  
    /share/  
    /var/  
        /cache/  
        /logs/
```

- The default port monitored by the proxy server is 3128.
- Start up the Squid process by invoking as root:

```
/usr/local/squid/sbin/squid &
```

I have placed this command in `/etc/rc.local` so that Squid would start up at boot time.

- To start Squid with an **empty cache**, invoke

```
/usr/local/squid/sbin/squid -z &
```

- After you have brought up the proxy server, it is useful to look at the following log

```
/usr/local/squid/var/logs/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 useful logs at the same pathname as above are

```
access.log  and  store.log
```

The former especially shows you 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.

To Get Squid to Work on Your Linux Laptop

- Assume that **SQUID_HOME** environment variable in your root account is set to main installation directory for Squid, which is **/usr/local/squid/**. So, if you say as root

```
echo $SQUID_HOME
```

the system should come back with **/usr/local/squid**.

- The two directories in **\$SQUID_HOME/etc/var**, these being **cache** and **logs**, have to be owned by the account **squid**. The group ownership must also be set to **squid**. If the directories **cache** and **logs** are owned by, say, **root**, Squid will start but then die after about 10 seconds. That is because after Squid fires up, the Squid process is run under the ownership of 'nobody' which does not have permissions to create the needed swap subdirectories in **cache** and the needed log files in the subdirectory **logs**.
- After you have set the ownership of **cache** and **logs** directories by executing

```
cd $SQUID_HOME/var/  
chown squid *  
chgrp squid *
```

it is best to execute the following BEFORE you fire up the main Squid process:

```
/usr/local/squid/sbin/squid -z
```

This will create the **swap** subdirectories in the **cache** directory. The need for this step will become evident if you examine the error messages in `/var/logs/messages`. As you'll recall, this is where Linux deposits its message concerning the starting up of new child processes and the death of old processes.

- 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".

The Squid Cache Manager

- The cache manager is a neat utility. It is a cgi script in Squid's `libexec` directory. It can provide statistics about the various objects in the cache. It is also a convenient tool for managing the cache.
- To have most fun with Squid's Cache Manager utility, you have to have the Apache web server installed on your Linux machine.
 - I'd recommend that you install the Apache web server (from <http://httpd.apache.org/>) on your Linux machine if you have not already done so. You will also need it when we discuss the Harvest system for information gathering and indexing.
 - Listed below are some things to watch out for if you do install the web server.
 - Even when you launch the web server daemon, `httpd`, 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 '`nobody`'. As you should know by this time, this is for ensuring security since the user '`nobody`' has virtually no permissions with regard to the files on your system.
 - With a standard install, your Apache HTTPD directory will be installed in `/usr/local/apache2/`. For convenience, in the `.bashrc` of the root account, set the environment variable `APACHEHOME` to point to this directory.

- The behavior of the Apache httpd server is orchestrated by the `httpd.conf` file in the `$APACHEHOME/conf` directory.
- 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


```
/usr/local/apache2/bin/apachectl start
/usr/local/apache2/bin/apachectl stop
```

 I have aliased this command to 'starthttpdserver' in the `.bashrc` file for the root account.
- If you run into any problems with the server, it can be extremely useful to look at


```
/usr/local/apache2/logs/error_log
```

 for error messages.
- For httpd to serve out the web pages in your `~kak/public-web/` directory, make sure that this directory in your home account has 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 'nobody'. Since the ownership/group of the `public-web` directory does not include 'nobody', it is the permission bits that are meant for "other" that would determine whether or not 'nobody' 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).

- To get the cache manager to display its handiwork in a browser, make the following entries in the **httpd.conf** configuration file for the HTTPD server:

```
ScriptAlias /Squid/libexec/ "/usr/local/squid/libexec/"
Alias /Squid/ "/usr/local/squid/"
<Location "/usr/local/squid/libexec/cachemgr.cgi">
    order allow,deny
    allow from localhost
    deny from all
    <Limit GET>
    </Limit>
    require user kak
</Location>
```

- Now get the httpd server to read the configuration file again. This you can do by sending SIGHUP to the parent httpd process. To find out what the pid of this parent process is, do

```
ps ax | grep httpd
```

The first httpd process will be the parent process. Now send it the SIGHUP signal by

```
kill -1 pid_of_httpd_process
```

- Next you need to make sure that the Squid configuration file **\$SQUID_HOME/etc/squid.conf** has the following definitions in it:

```
acl manager proto cache_object
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. Note that the cache manager uses a very, very, very special protocol called **cache_object** (as opposed to, say, the **http** or the **file** protocol).

- Using the above access control lists (acl), now make sure that the Squid configuration file `$SQUID_HOME/etc/squid.conf` has the following permission 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.

- To see the operation of the Cache Manager from the command line, you can do the following (as yourself, that is without being root):

```
telnet localhost 3128
GET cache_object://localhost/info HTTP/1.0
```

followed by two `␣CR␣` for terminating the **GET** request. This above will show the stats on the objects currently in the cache. Note the '**cache_object**' protocol being used in the **GET** request.

- To see your cache stats inside the web browser, point your browser to

```
http://localhost/Squid/libexec/cachemgr.cgi
```

- In order for the Cache Manager to work through the web server, I also did the following in the `$SQUID_HOME/libexec/` directory that contains the cgi scripts:

```
chown squid:squid *
```

However, before you do the above you must configure the config file `$SQUID_HOME/etc/squid.conf`.

- To see the 25 biggest objects in the cache, execute the following in the **logs** directory:

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

Configuring the Squid Proxy Server

- The configuration file `$SQUID_HOME/etc/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.
- In the configuration file, each parameter 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 **squid.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 on Slide 59.
- **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 `$SQUID_HOME/etc/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

# 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 Nutscape/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

# and much much more

```

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
```

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 speak-**

ing, 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.

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 summarization, 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**.

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
```

Harvest: 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 presumably 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 **RunGatherer** 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 action.

- Similarly, copy over the filter files

mydomain-url-filter

mydomain-host-filter

from the **example-5** gatherer into the `KAK_HOME_GATHERER` directory. 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 evaluated 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 summarizers (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**.

Harvest: How to Create a Broker?

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

```
/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 created 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.

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 company 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.