CODO: Firewall Traversal by Cooperative On-Demand Opening[*]

Sechang Son,[1] Bill Allcock,[2] and Miron Livny[1]
[1]*Computer Science Department, University of Wisconsin*
[2]*Mathematics and Computer Science Division, Argonne National Laboratory*
sschang@cs.wisc.edu, allcock@mcs.anl.gov, miron@cs.wisc.edu

Abstract

Firewalls and network address translators (NATs) cause significant connectivity problems along with benefits such as network protection and easy address planning. Connectivity problems make nodes separated by a firewall/NAT unable to communicate with each other. Due to the bidirectional and multi-organizational nature of grids, they are particularly susceptible to connectivity problems. These problems make collaboration difficult or impossible and cause resources to be wasted. This paper presents a system, called CODO, which provides applications end-to-end connectivity over firewalls/NATs in a secure way. CODO allows applications authorized through strong security mechanisms to traverse firewalls/NATs, while blocking unauthorized applications. This paper also formalizes the firewall/NAT traversal problem and clarifies how a traversal system fits in the overall security policy enforcement by a firewall/NAT.

1. Introduction

A network address translator (NAT) [1] provides easy address planning as well as a solution to the IPv4 address shortage problem. Firewalls play a vital role in protecting networks and are ready to play an even more important role as the security headquarters of integrated security systems, which generally include anti-virus checking, intrusion detection, logging, and content investigation [5]. Today, many firewalls/NATs are deployed in the Internet. However, these devices come at a price, notably *non-universal connectivity*. Two endpoints separated by one or more firewalls/NATs[1] cannot talk to each other in general. The Internet has also become asymmetric because most firewalls allow outbound (to the world) but block inbound (from the world) communications. Asymmetry is a special case of non-universal connectivity. However, it deserves attention because most client-server applications can get around it by placing servers in publicly accessible places.

The grid [2] may be one of the areas most damaged by the connectivity problem because it is generally bidirectional in communication, multi-organizational, huge in scale, and geographically distributed. In grids, the connectivity problem generally results in the waste of resources because researchers may not harness resources separated from their networks by firewalls. Computing jobs cannot be staged from the world into a firewalled network, and vice versa [3] [4]; data placement cannot be completed because data cannot move into or out of a firewalled network.

Middleware approaches are very attractive for dealing with the connectivity problem. They are easy to deploy because neither the Internet nor operating systems need be changed, and many applications can benefit from them.

This paper presents a middleware firewall traversal system called *CODO* (Cooperative On-Demand Opening). CODO dynamically configures a firewall so that authorized applications can communicate through it. In CODO, both firewalls and applications benefit through their cooperation. CODO-enabled firewalls can protect networks better because pinholes are made only for authorized applications, are narrow and exist only when required. Unauthorized applications cannot get through the firewalls. Also, better understanding of firewall parameters by authorized applications enables them to communicate without frustration. Unlike previous approaches, CODO supports the most restrictive settings in that both inbound and outbound communications are controlled. Since CODO provides

[*] This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

[1] Throughout, we use *firewall* to collectively refer firewall and NAT. The term *NAT* is used to specifically denote NAT.

the Berkeley socket API, applications can easily become CODO-enabled. With interposition mechanisms such as [6] and [7], applications can benefit from CODO even without re-linking.

This paper also discusses how a firewall traversal system can fit in the overall security enforcement of a network. We introduce firewall traversal mechanisms as components that complement firewall functions.

In §2, we discuss a packet flow model within a firewall and define the firewall traversal problem within that model. The architecture and connection procedure of CODO are presented in §3 and §4, respectively. §5 discusses the fault tolerance issue and §6 explains the implementation. §7 and §8 present performance data and related research, respectively.

2. Problem Definition

The firewall traversal problem has been around for many years, though it is vaguely defined, raising many questions such as "if a firewall is opened for an application, does it blindly pass packets to/from the application?" and "how does a traversal mechanism fit in the security policy the firewall tries to enforce?" To avoid confusion, we define the problem as follows.

Firewalls block malicious or unwanted traffic while allowing benign and desired traffic. What is malicious or unwanted (or equivalently benign and desired) is defined by firewall rules. To traverse a firewall, a packet must pass the tests defined by the firewall rules. If a packet fails a test, then it is rejected. Otherwise, it continues to traverse the chain of tests until it fails a test or passes all the tests.

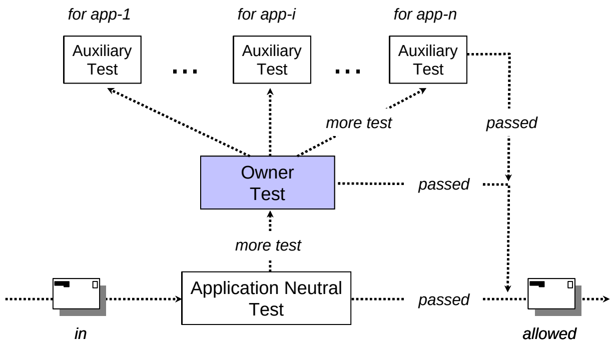

Figure 1: Packet flow model

Figure 1 shows a packet flow model in a firewall. When a packet enters a firewall, it undergoes one or more tests that we collectively call the *application-neutral test.* The test is specified with application-independent properties such as IP address, source routing flag, and ICMP message type. This test drops packets that are considered dangerous no matter what application sends or receives them. For example, overly fragmented packets are considered dangerous and may be dropped at this stage. If a packet passes this test, it may be allowed or sent to the *owner test*. The owner test allows traffic for authorized applications but blocks it for unauthorized or dangerous applications. For example, many firewalls allow SSH but block telnet and rlogin traffic. If a packet belongs to an authorized application, it may be allowed or sent to *auxiliary tests* that are specifically designed for individual applications. If an application is known to be vulnerable, say to a buffer overflow attack, an administrator may have an owner test to block the application. However, a better approach may be to allow the application traffic only if it does not contain an attack signature. The auxiliary tests can be used to block only malicious packets while allowing benign ones.

Depending on firewall implementations and configurations, packets may flow differently from the model: tests may be applied in a different order; multiple tests from different stages may be combined; some tests are not available in a firewall and may be performed by a third party product such as an IDS (Intrusion Detection System) [8]. However, we believe that this model is general and accurate enough for our discussion.

The connectivity problem may be defined as a situation where a benign application cannot traverse a firewall. We believe that the problem occurs mostly because benign applications fail the owner test (*false negative*), as firewalls are overzealous in blocking malicious applications. The owner test is also very important to network security because errors in this test may result in (1) malicious or undesirable applications passing firewalls (*false positive*) or (2) incorrect auxiliary tests being applied to packets, resulting in false negatives and false positives. For these reasons, this paper (and firewall traversal problems in general) focuses on the owner test. Our goal is to satisfy the following requirement:

> *Authorized applications' traffic must pass the owner test and unauthorized traffic must fail.*

Note that the above condition is crucial to not only an application's correct operation (grid perspective) but also network security (security perspective). Also note that we do not aim to reinvent firewalls to achieve our goal. Instead, we propose a scheme that dynamically configures existing firewalls to help them with the owner test.

We have challenges to achieving our goal. To perform the owner test, a firewall must know what application has sent or will receive the packet under scrutiny. However, packets generally do not convey the information about their source/destination applications. Almost every firewall uses port number to bind packets to an *owner application*. For example, packets with port 80 are considered be Web traffic. However, a port number is at most a hint to an application's identity because it is a shared resource used by any application with the appropriate privileges. It is extremely difficult to bind packets to owner applications without help from applications themselves. The second challenge is that applications are not aware of state changes in stateful firewalls they need to traverse. For instance, if a TCP connection is inactive for a while, it may become stale because a firewall flushes its state without any notification to the application. False negative errors occur in this case. Therefore, applications need notification from firewalls.

3. Architecture

To handle challenges and achieve our goal, CODO uses extensive cooperation between firewalls and applications. In CODO, connections into or out of a network are enabled through the cooperation of stateful firewalls, firewall agents (FAs), and client libraries (CLs) linked with the application. Figure 2 shows a typical CODO topology.

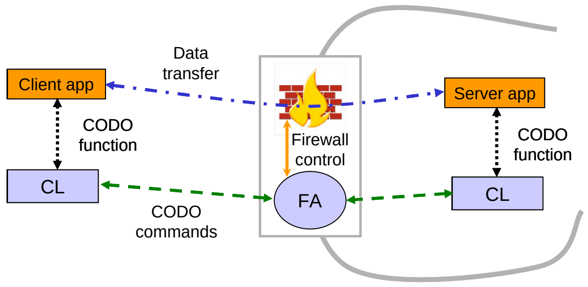

Figure 2: CODO topology

With CODO, a firewall can start with a configuration that allows no application to traverse. It also need not allow outbound communications. However, it may still have other rules for application-neutral or auxiliary tests. The FA running on the firewall machine dynamically adds and deletes rules for owner tests. During the initialization, it adds a few firewall rules to allow CODO commands to be delivered to it. The FA has a list of applications that can traverse the firewall. Since the list must be part of the firewall policy, we may think of the FA as a part of the firewall or an entity that enforces a part of the policy delegated from the firewall.

Through a secure TCP connection established using a certificate given to an application, the CL interacts with the FA on behalf of the application. It informs the FA of application activities such as binding a socket to an address, closing a socket, and trying to connect to a server. Using this information, the FA adds and deletes firewall rules for the application. The FA also informs the CL of necessary information such as how often a connection state will be flushed by the firewall. The CL uses this information to help the application communicate over the firewall. The application uses CODO services by calling CODO socket functions that the CL provides.

CODO has several desirable characteristics:

- **Quality owner test**. Through the exchange of information about firewalls and applications, the FA and the CL have up-to-date and sufficient knowledge to avoid owner test errors. Since the FA knows what (IP, port) pairs an authorized application is using at any given moment, it knows the exact binding between an (IP, port) and the application using the endpoint address. Therefore, CODO can avoid owner test errors caused by the errors in binding from (IP, port) pairs to owner applications. Also, the CL's knowledge about the firewall enables it to refresh or recreate firewall's state appropriately. This helps avoid false negative owner tests caused when the firewall flushes states for connections inactive for a while.
- **Narrow and short opening**. The FA adds firewall rules with no wildcard. In other words, rules are specified with a specific (protocol, source IP, source port, destination IP, destination port). This means that (1) a rule is added to the firewall only when there is an authorized pair of client and server and (2) only the intended client and server can traverse the firewall using the rule. In addition, to limit the duration of firewall rules as much as possible, the FA deletes the rules it adds as soon as the stateful firewall creates the necessary states (i.e. stateful rules) to allow subsequent packets to traverse. Therefore, with CODO, firewall openings are as narrow and short as possible.
- **Flexible control**. CODO uses X.509 certificates to authenticate and authorize applications. This means that CODO is very flexible and can enforce various security policies. For example, CODO can differentiate versions or implementations of an application. If a vendor's implementation of an

application turns out to be vulnerable to a dangerous attack, then it can be given a different certificate from other implementations and disallowed from communicating with the world.

- **Inbound & outbound control**. CODO controls outbound communications as well as inbound. With CODO, only authorized clients and servers can communicate with the world.
- **Easy deployment**. CL interface is almost the same as the Berkeley socket API. In fact, CODO functions have the same arguments as their Berkeley socket counterparts. This allows for easy integration of applications with CODO.

4. Connection procedure

With CODO, applications call CODO functions. The call sequence is the same as with a Berkeley socket. For instance, a server creates a TCP socket, binds it to an address, makes it passive, and accepts connections from clients. A client creates a TCP socket, optionally binds it to an address, and connects it to a server. The server and client exchange data through the established connection. This section explains how CODO connections are established over firewalls as responses to CODO calls from applications.

4.1 Server binding

To be able to accept connections from the outside world, a server socket behind a firewall must be locally bound, registered to the FA of its network, and officially bound.

Local binding is nothing new. Just as with regular binding, a socket is bound to an address. Through the local binding, an (IP, port) pair, called the local address, is assigned to the socket.

To arrange connections to a server socket, the FA of the server network must have enough information about the server socket. The FA needs this information to avoid owner test errors as explained in §3. The information is collected via *registration*. After a server socket is bound to a local address, the server's CL sends a registration request with the local address and the type of the socket. After authentication and authorization and the official binding (explained shortly), the FA records the information sent by the CL and other information that it collects from the official binding process.

NAT translates private addresses into public ones, and vice versa, as packets pass through it. Because of this translation, we may think of a socket inside a private network as having two addresses, a private (IP, port), called the local address in this paper, and a public (IP, port) that the NAT of the private network assigns for address translation. We may view the public (IP, port) as the address that the socket leases from the NAT box. Since the Berkeley socket API allows only one address per socket, a NAT traversal system with the same API must choose one address to make visible to the application and hide the other inside the system. We define the address that is known to the application as the *official address* of a socket. Similar to previous systems [10] [11], CODO uses the address a socket leases from a NAT box as its official address. Note that this is a natural decision because the leased address is globally unique.

Official binding is the process of assigning the official address to a server socket. When an FA receives a registration request with a private local address, it finds a public address and rents the address to the server socket. This leased address becomes the official address of the socket and will be used to add NAT binding rules. Of course, if the local address is public, then it becomes the official address without address leasing. As a successful response to the registration request, the FA sends the official address to the CL of the server application.

When an application calls `getsockname` asking for the address to which a CODO socket is bound, the CL returns with the official address instead of the local (real) address. Thus, the local address of a CODO socket is hidden inside the system.

4.2 Connection arrangement

This section explains how a TCP connection is made for the most complex client-server configuration. In this configuration, both the client and server networks allow neither inbound nor outbound connections (figure 3). Simpler configurations follow a similar process with the omission or modification of some steps. For example, if the client network allows

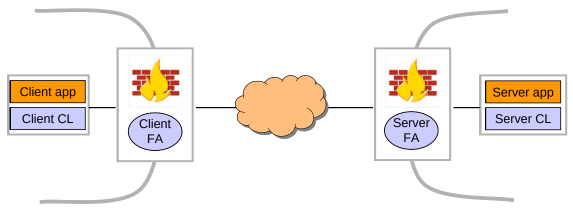

Figure 3: Firewall-to-firewall connection. Both networks allow neither inbound nor outbound communications. The client and server FA add firewall rules for outbound and inbound communications, respectively.

outbound connections, then we do not need steps to add rules to allow outbound connections. UDP connections will be explained in §4.3.

We assume that the server FA has the information about the server socket through the binding process described in §4.1. We also assume that the client knows the official address of the server socket. For example, Condor [4] components such as the job scheduler and machine manager advertise their addresses to a central manager that collects and maintains information about jobs and machines. To connect to a Condor component, its address is retrieved from a central manager. In this case, the official address of a component will be stored in and retrieved from the central manager because only official addresses are known to applications.

A connection from a client socket at address *C1* to a server socket whose official address is *S1* and local address *S2* is established through the following steps:

(1) The client CL makes a TCP connection to the client FA and asks for a connection to S1.

(2) The client FA reserves an address *C2* for an NAT binding, if C1 is a private (IP, port). The client FA makes a TCP connection to the server FA and asks for a connection from C2 (or C1 if it is public) to S1.

(3) The server FA adds a firewall rule to allow the inbound connection from C2 (or C1) to S1. If S1 $\neq$ S2, i.e. S2 is a private (IP, port), then a NAT binding rule [C2 (or C1)$\rightarrow$S1, C2 (or C1)$\rightarrow$S2] is added instead. By a NAT binding rule *[W$\rightarrow$X, Y$\rightarrow$Z]*, packets with source address *W* and destination address *X* are translated into packets with source address *Y* and destination address *Z*. The timeout value after which states in the server firewall are flushed is returned to the client FA.

(4) The client FA adds a firewall rule to allow the outbound connection from C1 to S1. If C2 was reserved, then a NAT binding rule [C1$\rightarrow$S1, C2$\rightarrow$S1] is added instead. The minimum of the timeout value of the client firewall and the value returned from the server FA is returned to the client CL. The client CL uses this value to periodically send heartbeats to refresh states in the firewalls.

(5) The client CL makes a connection to S1. At this point, necessary states are created at both firewalls and subsequent packets between the client and server can traverse those firewalls without the rules added in (3) and (4).

(6) The client and server FAs delete the firewall or NAT binding rules they added, respectively. How FAs detect that necessary states have been created at firewalls will be discussed in §6.

How the client CL knows that it should contact the client FA (step 1) and how the client FA knows that it must contact the server FA (step 2) need explanation. In the current implementation of CODO, each node has a manually configured table. Given an IP, the table tells if the node can directly connect to the IP. It also tells what FA must be contacted if a direct connection is impossible. This approach is not scalable and a better solution is under investigation. However, the table should not be very big because multiple IPs can be aggregated with a mask. If all IPs within a network can be aggregated with a mask as is true for most private networks, only a single entry is needed for the network. Furthermore, when we add a new network, we only need to change the tables at public nodes (including FAs). Nodes behind a firewall only need know about local nodes. For all the other nodes, they just need ask their local FA for connection arrangement.

Connection establishment within a private network also needs help from CODO. A client in the same private network as a server cannot make a direct connection with the server's official address. In this case, the client CL asks the server FA[2] for the server's local address and then makes a direct connection to it. No NAT bindings are made for intra-network connections.

4.3 UDP connections

Although there is no connection setup in UDP communications, we can loosely define a UDP session as a set of UDP messages that are allowed or rejected as a whole by stateful firewalls. More precisely, it is a series of UDP messages from a client to a server such that each message passes a firewall before a predefined timeout from the previous one.

In order to support UDP sessions over firewalls, a CL maintains a mapping table. Each entry *(X, Y)* in the table maps the official address *X* of a peer to the peer's address *Y*, meaning that UDP messages addressed to *X* by the application must be sent to *Y*. When the application calls `sendto` with the receiver's address *X*, the CL searches the mapping table for the address. If the entry *(X, Y)* is found, then it refreshes the timeout value of the entry and sends the packet to *Y*. Otherwise,

[2] The server FA is also the client FA in this case because the client and the server are in the same network.

a procedure similar to the TCP cases explained in §4.2 is performed. If the procedure succeeds, then the sender's CL creates a new entry with the address that the receiver's FA has returned and then sends the UDP message using the entry. Entries not referred to for a while are deleted.

5. Fault tolerance

Successful connection depends on the reliability of FAs. Nevertheless, applications should continue to work with a limited ability in the event of FA failure. During FA downtime, CLs operate as if the FAs did not exist. For example, if a server CL cannot contact its FA—the server FA at binding time, it downgrades the socket to a regular Berkeley socket, which bypasses all CODO mechanisms. In this case, the server can accept connections only from clients that do not need help from the server FA. If a client CL cannot contact a server FA, it attempts a direct connection to the server.

If an FA recovers from its failure, servers affected by the failure should upgrade their sockets to support CODO mechanisms[3]. To achieve this goal, we design FAs to maintain soft state so that they can recover by receiving socket information from server CLs. Therefore, the CL periodically tries to contact the failed FA. If successful, it upgrades sockets by doing whatever it would have done if the FA had not failed, and the FA recovers its state during this upgrade process.

If a firewall fails before step (6) in the connection process, unnecessary rule may still exist in the firewall when it recovers. The firewall must delete these unnecessary rules to maintain a high level of security. If a firewall supports timeouts on rules, then garbage collection would be able to clean up the unnecessary rules. Unfortunately, the firewalls we targeted do not support timeouts. Instead of garbage collection, each FA records a snapshot of rules it created in a persistent file. During startup, it deletes all the rules recorded in that file. This blind flush will certainly delete necessary rules as well. However, the necessary rules are recreated as a part of the (soft) state recovery explained above.

6. Implementation

The CL is implemented as a C/C++ library and as a layer between the application and the kernel, as depicted in Figure 4. Applications use CODO socket calls to create a CODO socket, bind it to an address, connect to a server, accept a connection from a client, and so forth. The CL provides some file system calls so that applications may duplicate socket descriptors, make a socket non-blocking, and multiplex multiple file descriptors, including CODO sockets. The CL also has a few functions for process control, such as *CODO_fork* and *CODO_execve*. These are mainly for inheriting open sockets to child processes. All CODO calls have the same APIs as their regular counterparts.

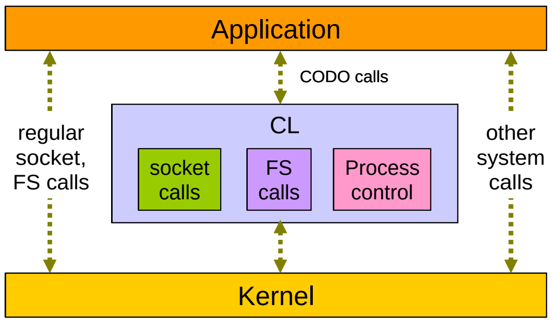

Figure 4: CL implementation

The FA is implemented as a daemon running on the firewall machine. It uses the Linux Netfilter [9] API to add, delete, and list rules. Although CODO currently supports only firewalls based on Netfilter, it interacts with firewalls through an abstraction layer that defines necessary firewall functions to dynamically control it. Therefore, any firewall with those functions can be easily supported.

In §3 and §4.2, we claimed that the FA deletes firewall rules it added when they become unnecessary after the stateful firewall creates enough state information to allow subsequent packets to traverse. Here, we explain how the FA interacts with the stateful firewall. To detect that the rule becomes unnecessary, CODO uses Netfilter's user space packet-processing mechanism. Netfilter allows user processes to specify various conditions and to handle packets satisfying those conditions. To allow a connection from a client to a server, the FA adds a Netfilter rule that allows *initial* packets[4] (first SYN packets, for example) from the client to the server. In addition, it also adds other rules to catch *non-initial* packets that are sent from the

[3] The upgrade process should not change the official address of a socket. Therefore, sockets with private (IP, port) may not be upgraded. A private official address is assigned to a socket behind a NAT box when its FA is down at binding time.

[4] The initial packet may be sent multiple times because of retry mechanism of reliable protocols such as TCP.

client to the server, or vice versa, that would otherwise be allowed by the firewall. Note that those non-initial packets will be denied by the firewall and not caught by the FA until the necessary state has been created at the firewall because the first rule only allows initial packets. When such a packet is caught, FA deletes the rule that allows initial packets and those that catch non-initial packets.

7. Performance measurement

To measure the performance, we set up two private networks. Each network has two private nodes behind a Linux NAT box (headnode) with two network interfaces. Nodes within each private network are connected via 100Mbps Ethernet. The two networks are connected via a department network (100Mbps). Neither inbound nor outbound connections are allowed through the NATs. Every machine has two 2.4 GHz CPUs with 512K cache and 2G RAM with about 1.7G free space.

Using a test suite that we wrote, we measured connection setup and data transfer times. In our test suite, a client makes a connection to a server and then sends 100 messages of 10K bytes long back-to-back. The server echoes back to the client. When every message is echoed, the client tears down the connection. We inserted random delay between connections. Actual delay was determined using a Poisson process with a mean (λ) of 3 seconds.

Table 1: TCP connection and transfer. Numbers are microseconds. Those in parenthesis are standard deviation.

	Inter		Intra	
	Conn	Data	Conn	Data
CODO	27320 (1330)	279945 (6921)	4958 (142)	141853 (5370)
Reg.	543 (77)	278187 (7022)	221 (58)	141494 (5366)

Table 1 shows the average time to make a connection and the average (total) time that 100 messages are echoed. In order to indicate the overhead of CODO, the table also has numbers for regular sockets with NATs manually configured to allow traffic between two networks. For private-private measurements ('Inter' column), we used a client in one network and a server in the other. For intra-network communication ('Intra' column), we used a client and a server both in the same private network. We used X.509 (RSA) public key for authentication and session keys establishment. SHA-1 and 3DES were used for integrity and encryption of CODO commands, respectively.

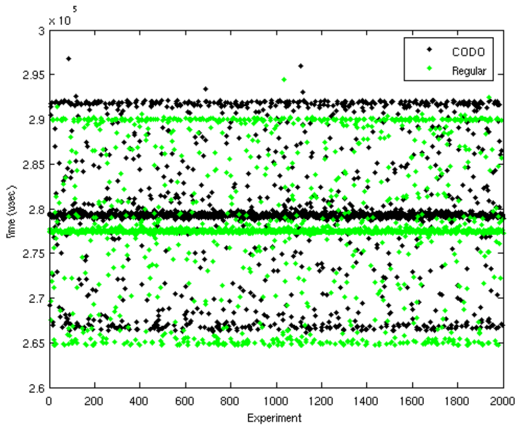

Figure 5: Inter-network data transfer

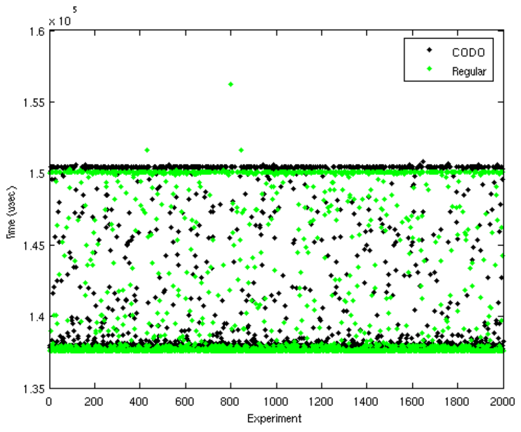

Figure 6: Intra-network data transfer

The tables show that CODO overhead is large for connection setup, increasing the latency of communications. Considering the security mechanisms used and the number of interactions between client and CODO agents, the overhead is not surprising. As explained in §6, CODO uses Netfilter's user space packet-processing to detect when necessary states are created. Our profile showed that CODO consumes 10 ~ 15msec per connection for processing packets at the user level. If firewalls were to support one-time rules that are automatically deleted after allowing a certain number of connections, we would be able to dramatically reduce CODO connection time. Once a connection is made, minimal overhead is observed for data communication. Figure 5 and 6 show the scatter plot of inter-network and intra-network data transfer,

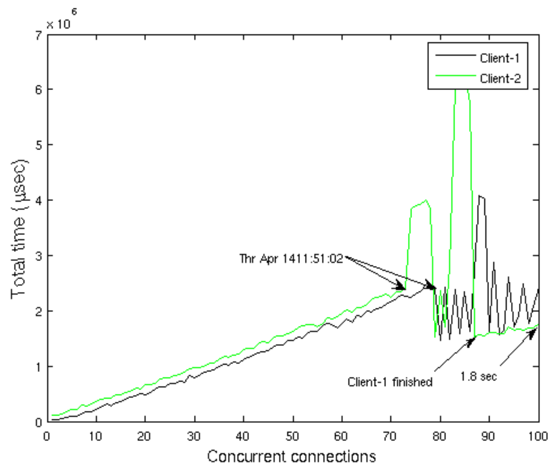

Figure 7: Concurrent connections. The X-axis shows the number of concurrent connections issued by each client. The Y-axis shows the total time to set up multiple connections.

respectively, for the first 2,000 experiments. X-axis represents experiments and y-axis shows the data transfer time for each experiment. For some reason, times were measured around 2 (figure 6) or 3 (figure 5) popular values forming bands. Those figures show that the overhead of CODO data transfer is very small and clearly within the range of network and measurement fluctuation. Intra-network data transfer occurs directly between the client and the server without any involvement of NAT or CODO FA. Therefore, figure 6 shows a slight overhead of CL. Figure 5 shows the overhead at headnodes (of the client and server networks) plus CL overhead.

To see how well CODO scales, we also tested concurrent connection setup. Figure 7 shows the time to establish multiple connections in parallel. In this test, two clients running on each host in a private network issued concurrent connections (i.e. non-blocking connections) to a single server running in another private network. Each client issued up to 100 non-blocking connections simultaneously. Each client issued one connection to measure the time of single connection setup, and then issued two connections in non-blocking fashion to measure the time to setup two connections, and so on. Two clients started almost at the same time. Figure 7 shows the total connection time for each concurrency level observed by each client. For example, it took about 2.4 seconds for client-1 to finish 79 connections. At that moment, client-2 was issuing about 70 parallel connections. Therefore, we should read the figure as 79 parallel connections established within 2.4 seconds while a total of 150 parallel connections are being made to the server. Packet loss was observed when the total concurrency level was about 150 or higher, resulting in delay in connection setup. In the test, client-1 finished when client-2 had 88 concurrent connections. Therefore, the figure shows that 100 parallel connections without competing clients were established within 1.8 seconds, which is about 63 times (instead of 100 times or more) slower than a single connection setup. The result shows that the connection overhead is amortized as we have multiple connections occurring simultaneously. CODO achieves this speed up by interleaving multiple connection establishments.

8. Related research

Many firewall traversal systems have been proposed or developed. Unlike CODO, previous research mainly focused on how to enable applications to traverse firewalls, with no or little attention to the security of the network. No previous system allows strong control on both inbound and outbound communications.

GCB [10], STUN [11], and TURN [12] use the fact that most firewalls allow outbound connections. Since these systems do not interact with firewalls, they are relatively easy to deploy. However, these systems exploit the common configuration of firewalls to a degree that most network administrators may not intend. For this reason, they are sometimes considered to deceive firewalls and be harmful to network security. Similar to CODO, DPF [10], RSIP (Realm Specific IP) [20] [21], UPnP (Universal Plug-and-Play) [18], some personal firewalls, and port knocking [19] dynamically controls firewalls for applications. DPF, RSIP, and UPnP open a firewall whenever there is a server behind the firewall so that any client can reach the server through the firewall. Therefore, these systems open firewalls wider and longer than CODO. Personal firewalls can reliably and securely control traffic based on sender/receiver applications so that only authorized applications can communicate with others. However, they can only be used for the host protection but not for the network protection. In port knocking, users can open a firewall through a sequence of unsuccessful connection attempts. It may not work for applications using many dynamic ports because it may need too many unsuccessful connections to code arbitrary port numbers that a firewall must open. Port knocking also has a scalability problem because predefined ports must be reserved for each user or application. SOCKS

[14] enables communications through a firewall by a proxy relaying connections. Like CODO, it uses a strong security mechanism and therefore can enforce various security policy using certificates. However, unlike CODO and other systems, it uses the local address (§4.1) as the official address and is not able to support private networks because multiple server sockets in different private networks may have the same official address. Overlay networks [22] can be used to traverse firewalls. However, they are rather area or application specific and are not adequate for general purpose use. CODO can be used to facilitate communications between overlay nodes (i.e. overlay routers and end nodes). VPNs also provide a secure mechanism to traverse firewalls. However, these are mainly for extending corporate networks across insecure public networks. In VPN, therefore, traffic is controlled based upon sending/receiving networks or hosts instead of applications.

More fundamental approaches to solve connectivity problems have also been proposed. TRIAD [15] and IPNL (a NAT-extended Internet architecture) [16] propose a new layer between TCP/UDP and IP. They provide elegant solutions to NAT traversal, but they cannot be used for firewall traversal.

To solve various problems of ALG (Application Level Gateway), the IETF MIDCOM group defines a decoupled architecture [13]. The main idea of the architecture is to move the functionality of ALG that is currently embedded in firewalls to a separate entity called MIDCOM agent. Since the application awareness is moved out of the firewall, firewalls need not be changed when a new application is added to the support list. Since it starts from ALG, MIDCOM still shares with ALG approaches in how application's communication activities are understood. Like ALGs, MIDCOM agents try to understand application's communication activities by looking at packet payloads and alter them if necessary. This contrasts with CODO's approach in which the library linked with the application explicitly reports to the firewall agent. MIDCOM approach provides applications transparency. However, each application requires a specialized agent that may understand its communication semantics. Furthermore, not every application can be understood by looking at packets passing a firewall.

IPv6 [17] is beginning to be widely deployed. It provides enough address space and enables easy network management. Thus, it solves most problems that NATs try to solve. However, it is still questionable whether IPv6 can replace NATs completely. Furthermore, firewalls will certainly exist after the full deployment of IPv6.

9. Conclusion

This paper defined the firewall traversal problem within a framework of network security and discussed a firewall traversal mechanism as (1) a way to enable applications to traverse firewalls and (2) a component that helps an important firewall function—owner test. Within this context, this paper introduced a firewall traversal system, called CODO. CODO enables benign and authorized applications to communicate over firewalls and helps firewalls to block malicious or unwanted applications. Therefore, security managers as well as application developers, end users, and service providers may benefit from CODO. This is contrary to the general thought that firewall traversals are harmful to network security.

10. Acknowledgements

We thank Matthew Farrellee for technical and grammatical review of the paper.

References

[1] K. Egevang, P. Francis, "The IP Network Address Translator (NAT)," *IETF RFC1631* May 1994.

[2] I. Foster, C Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Intl. Journal of Supercomputing* Applications 2001.

[3] Globus web site, http://www.globus.org

[4] Condor web site, http://www.cs.wisc.edu/condor

[5] Checkpoint web site, http://www.checkpoint.com

[6] Douglas Thain and Miron Livny, "Multiple Bypass: Interposition Agents for Distributed Computing", *The Journal of Cluster Computing*, Volume 4, 2001, pp 39-47.

[7] Douglas Thain and Miron Livny, "Parrot: Transparent User-Level Middleware for Data-Intensive Computing", *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, September 2003.

[8] V. Paxson, Bro: a system for detecting network intruders in real-time. Computer Networks, 31(23/24), Dec. 1999.

[9] Netfilter web site, http://www.netfilter.org

[10] S. Son, M. Livny, "Recovering Internet Symmetry in Distributed Computing." *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid,* Tokyo, Japan, May 2003.

[11] J. Rosenberg, J. Weinberger, C. Huitema, R. Mahy, "STUN – Simple Traversal of User Data Gram (UDP) Through Network Address Translators (NATs)," *IETF RFC 3489*, March 2003.

[12] J. Rosenberg, R. Mahy, C. Huitema, "Traversal Using Relay NAT (TURN)," *Internet-Draft*, July 2004.

[13] P. Srisuresh et al., "Middlebox Communication Architecture and Framework," *IETF RFC 3303*, Aug. 2002.

[14] M. Leech, M.Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones, "SOCKS Protocol Version 5," *IETF RFC 1928*, March 1996.

[15] D. R. Cheriton, M. Gritter, "TRIAD: A New Next Generation Internet Architecture," March 2000. http://www-dsg.stanford.edu/triad/triad.ps.gz.

[16] P. Francis, R. Gummadi, "IPNL: A NAT-Extended Internet Architecture," *SIGCOMM'01*, Aug. 27, 2001.

[17] S. Deering, R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *IETF RFC 2460*, Dec. 1988.

[18] UPnP website, http://www.upnp.org

[19] Port Knocking website, http://www.portknocking.org

[20] M. S. Borella, G. E. Montenegro, "RSIP: Address Sharing with End-to-End Security", *Special Workshop on Intelligence* at the Network Edge, San Francisco, 2000.

[21] M. Borella, J. Lo, D. Grabelsky, G. Montenegro, "Realm Specific IP: Framework", *IETF RFC 3102*, July 2000.

[22] D. Anderson, et el. "Resilient Overlay Networks," *18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.