Recovering Internet Symmetry in Distributed Computing

Sechang Son, Miron Livny

Computer Science Department
University of Wisconsin
{sschang, miron}@cs.wisc.edu

Abstract

This paper describes two systems to recover Internet connectivity impaired, especially in distributed computing, by private networks and firewalls. Private networks and firewalls brought Internet asymmetry and made peer-to-peer computing difficult or even impossible to work across private network boundaries or over firewalls. Condor is one of those that are severely impaired by the asymmetry. Compared to normal peer-to-peer computing applications, Condor has stricter requirements set, which, we believe, is representative to any grid computing. To make Condor seamlessly work across private networks and over firewalls, we designed and implemented Dynamic Port Forwarding (DPF) and Generic Connection Brokering (GCB). Both DPF and GCB satisfy the representative requirements. Furthermore DPF supports dedicated large cluster very well because it is simple, efficient, and highly scalable. On the other hand, GCB perfectly supports non-dedicated or virtual cluster because it is independent to private network or firewall technologies and does not require any administrative power to deploy it. In this paper, we describe the implementations of DPF and GCB and analyze them with respect to performance, deployability, scalability, and etc.

1. Introduction

Since private networks were introduced, many institutions have deployed them to solve IPv4 address shortage and to improve security. Also firewalls are usually deployed with Network Address Translator (NAT) [RFC1631] based private networks in order to hides internal machines and more importantly to provide a choke point where firewall policies can be applied.

Though private network was conceived as a temporary solution to the address shortage problem and the IPv6 project is a massive effort to solve the problem in a permanent way, many experts predict that it will persist even after the full deployment of IPv6 for its easy network manageability and economic reasons [FRNGUM]. We believe that grid computing gives one of the most convincing examples that support this argument. Grid is the infrastructure that enables coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [FSTKSS]. In grid computing, pools of hundreds or thousands machines are not rare. All or some of those machines are dedicated to grid computing and have much less reason to have world addressable IP addresses than those owned by individuals and used for general applications and daily use. Administrators of those pools would prefer private network configuration because they can easily manage their clusters and also reduce the cost by paying for only several public IP addresses for head nodes instead of hundreds or thousands ones.

Private network and firewall, however, damaged Internet connectivity and made it asymmetric. Internet was originally designed as being symmetric at least above the transport layer, i.e. if a process A can talk to B, then B is always able to talk to A. This symmetry, however, is no more guaranteed if A is inside a private network or behind a firewall, because NAT or firewall usually blocks all or some of inbound communications. Among others, Peer-to-Peer (P2P) computing may be the most damaged one by the asymmetry because, in P2P, any process needs be able to talk to any other. Condor system [LIVNY] [LTZLVN], in which virtually every machine must be able to communicate with each other, is a P2P application by nature and damaged by the asymmetry.

As a grid system, Condor has the following requirements for any solution to recover Internet connectivity, in addition to those required by regular P2P systems. We believe that all of the requirements listed below are common to any grid approach and, at our best knowledge, no single system so far satisfies all of them.

1. *The solution must be highly scalable*. Condor clusters with hundreds of nodes are very common and ones with thousands exist. Furthermore, *flocking* [EPMLEP] makes clusters even bigger by putting existing ones together. Hence we can't use an approach that assumes small number of machines inside private network or behind firewall.
2. *It must provide a way to communicate with (existing) regular sockets*. Many different versions of Condor clusters have already been installed and are running all over the world, and they need be able to communicate with new

clusters with private network and firewall support. Hence the solution must provide a way to communicate with existing sockets without any change to them.

3. *Changes to network components must be minimized and any change to kernel or having system-wide impact is not allowed*. Condor does not require any kernel change or even root privilege to run it, and this was turned out to be one of the most important features of Condor's success. We want to keep this advantage and would not take any approach that hurts easy deployment of Condor.

To bring symmetry back to Condor, we implemented two different approaches, DPF (Dynamic Port Forwarding) and GCB (Generic Connection Brokering), which have different characteristics in terms of clusters supported, security, performance, and etc. so that institutions may choose the better one depending on their policies and situations.

Firewall and NAT based private network are essentially the same from the perspective of impacts on grid computing. Also connectivity loss due to private network is considered more severe because connections blocked are side effects of private network. Hence the following discussions are made in the context of NAT based private network. In section 2, we briefly explain previous works. DPF and GCB are explained in section 3 and 4, respectively. Some experimental results are presented in section 5.

2. Previous Works

Many researches and developments have been done or being carried out to recover Internet connectivity. Some systems took local or fill-the-gap approaches, requiring changes to components within an institution's administration domain. Other systems took global approaches and require major changes to Internet or need agreement between various institutions. For example, TRIAD [CHRGRT, GRTCHR] and IP Next Layer (IPNL) [FRNGUM] use name-based and realm-to-realm routing to make inbound communications possible and propose changes to Internet protocol stack. Address Virtualization Enabling Service (AVES) [EUGSTO] uses proxy and packet rewriting technique and requires changes to DNS servers and NAT machines. Because global approaches will take years to be accepted by large community and because they fail to satisfy the last requirement in section 1, we will only consider local approaches in this section.

2.0 Global approaches (*Informational and optional*)

TRIAD [CHRGRT, GRTCHR] proposes to hide IP address from applications and use URL as the sole network end-to-end identity. It uses URL based routing to setup connections and semi source routing to deliver packets of the connection. In TRIAD, connection is established by cooperating name servers as part of name lookup operation and address path, a realm-to-realm source route, is returned as the result of the connection. Packets of the connection are relayed according to the address path, which is a part of packet header. To support NAT, NAT machines must act as the TRIAD name server and relay point. In addition to the changes of NAT machines, client machines must be changed too because TRIAD proposes the change to Internet protocol stack.

IP Next Layer (IPNL) [FRNGUM] views Internet as a hierarchical connection of realms, where each private network becomes a realm, possibly containing sub-realms, and public network is a middle realms connecting private realms at the highest hierarchy, and proposes to add IPNL layer, between (inter) networking and transport layer, that performs inter-realm routing and name-based routing. In IPNL, network endpoint is uniquely defined both by FQDN and IPNL address, which basically explains where the endpoint is located in the hierarchy. Like TRIAD, IPNL uses name based routing for connection and uses more efficient IPNL address to transfer packets once the connection is established. Basic idea to support private networks is to make NAT machines as IPNL router, which performs name-based routing and relays packets between realms that can't normally communicate with each other directly.

Unlike TRIAD and IPNL, Address Virtualization Enabling Service (AVES) [EUGSTO] does not require changes on Internet protocol stack, but still requires changes of major network components such as DNS server and NAT. In AVES, public IP address of the *waypoint* is leased to the private machine at the time of DNS query by the connector, through the cooperation of DNS server and the waypoint chosen by the server. DNS server answers the leased IP address to the connector. Inbound connection is made by waypoint's rewriting and then relaying of the packet, which is sent from the connector to the waypoint. NAT machine for the private machine is also need to change so that it may understand AVES protocol and pass packets from the waypoint to the real destination.

2.1 Application-specific connection brokering

Napster [NAPSTR] server acts as a connection broker for its clients. Normally it arranges that a downloading site make a connection to an uploading site. However, when the uploader is inside private network, it asks the uploader to push files to the downloader in public network. Gnutella [GNUTLL] also uses the same idea, but without any server. When an uploader is inside

private network, the downloader in public network asks the uploader to actively push a file. This approach is very simple and has little overhead. This can also be used with any private network technique and requires no change to network components. However, it has a few disadvantages, which make this approach fail to satisfy those requirements in section 1, including:

- It is an *application specific approach*. Any application that wants to apply this idea needs to implement its own version of connection brokering.
- It is *not interoperable with regular sockets*. Since every node, including clients and server, needs to follow an application-specific protocol of brokering, no regular socket that is ignorant of the protocol could be brokered.
- Without additional help such as relay or rendezvous service, *private-to-private connection is impossible.*

2.2 SOCKS

IETF took SOCKS [RFC1928] as a standard for performing network proxies at the transport layer. Basic idea is that the SOCKS server, which must be placed at the outskirts of a private network, plays as a relay point at transport layer between machines inside private network and those at public network. When a node A at public network wants to connect to B behind a SOCKS server, A sends connection request to the SOCKS server. Then the server establishes two transport connections: one with A and the other with B, and then relays packets between them. The initiation of UDP communication is handled in a similar manner.

SOCKS has several advantages. First of all, it can be viewed as an application independent approach because applications need not be rewritten to use SOCKS. SOCKS people call applications that become aware of SOCKS protocol *socksified*. Application can be socksified either by changing its codes, by relinking with SOCKS library, or by modifying system's dynamic library path so that SOCKS library is used instead of regular network library. Another advantage is that it is a mature system, because it has been used in many applications and institutes. Currently commercial products are available as well as research systems. Finally it is private network technology independent and can be used with or even without any NAT-like proxy. When it is used without a NAT-like proxy, private-to-public connections as well as public-to-private ones are relayed by the server.

It, however, has a few drawbacks, which makes it fail to satisfy our requirements:

- *It is not highly scalable*. Every socket whose packets are relayed by a SOCKS server needs to maintain a management TCP connection with the server during its lifetime. In every operating system the number of TCP connections opened at the same time is limited and thus the maximum number of sockets supported by a SOCKS server is limited by this number.
- *Regular socket on the public side cannot initiate communication to SOCKS client at the private side*. With SOCKS, clients inside private network need not be changed at all. Nodes at public side, however, must be aware of SOCKS protocol and this violates our important requirement.

We believe that the last constraint of SOCKS shows that it was originally invented for client-server model as hinted in [RFC1928] rather than P2P computing, because, in P2P, clients at public side are usually indefinite and it is usually impossible to make changes to every public peer application or node.

2.3 Realm Specific IP (RSIP)

Realm Specific IP (RSIP) [RFC3102, RFC3103, BRLMNT] has been proposed and adopted by IETF as a standard way to solve NAT problems, especially those related to IPSec [RFC2401] and inbound connection.

The client inside private network leases one or more public IP addresses and ports from RSIP server when the system boots up or dynamically when it needs them, and uses those leased address as network endpoint identities. The RSIP server maintains mapping between leased addresses and leaser address to handle inbound communications to the leaser. When the leaser needs to send a packet to a public peer, it prepares the packet as if it is from one of those leased addresses and then sends it through the tunnel to RSIP server. Upon receiving a packet through the tunnel, the server stripes off the tunnel header and forwards it to public network. Inbound communications, including replies from the public peer, are handled in the reverse way. The server wraps the received packet using the leaser's address from the mapping and sends it to the leaser. Then the leaser unwraps the tunnel header and passes the resulting packet to the application.

In addition to the support for inbound communications, RSIP solves NAT's incompatibility with IPSec [RFC3104]. Since RSIP server relays packets untouched, other than ripping off extra header for tunneling, end-to-end security at IP level required by IPSec can be easily achieved.

RSIP also supports nested private networks by cascading RSIP servers. [RFC3102] also recommends that RSIP server be compatible with NAT clients so that it may support private networks with the combination of RSIP enabled sockets and regular ones.

RSIP has many desirable characteristics as explained above. It is, however, still an ongoing effort and more importantly it was proposed as a substitute of NAT. Though RSIP can be implemented as an extension to NAT for some NAT implementations such as *iptables* in Linux 2.4, generally RSIP should replace well-tested NAT. We don't believe that, in a near future, it will be developed for every major platform and becomes prevalent so network administrators are willing to use RSIP instead of NAT.

3. Dynamic Port Forwarding (DPF)

For easy explanation, we introduce two notations below and use them throughout the paper. *A:B* represents a pair of IP address *A* and port number *B*. *[A:B > C:D]* represents a mapping or translation rule from *A:B* to *C:D*.

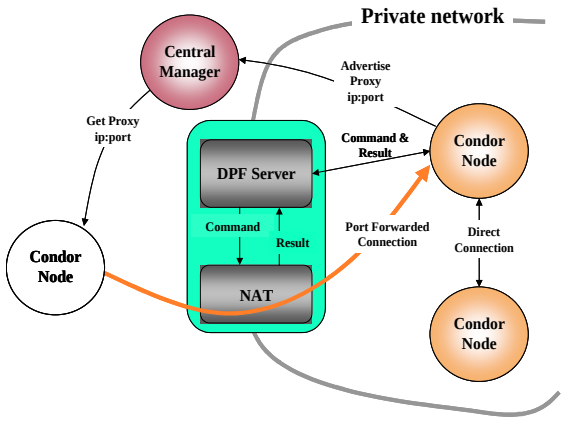
NAT port forwarding is a combination of packet rewriting and routing mechanism based on ports as well as IP addresses, and is the most popular way, if not the best nor the sole way, to make inbound communication possible in NAT. When an NAT gateway receives a packet destined to Nip:Nport and has a forwarding rule [Nip:Nport > ipX:portY], it rewrites the destination as ipX:portY and routes the rewritten packet toward IP address ipX. Hence machines inside private network can accept inbound communications by setting port forwarding rule at NAT gateway.

At our best knowledge, port forwarding must be set /unset by administrator in a static way and can be used when user knows both (the range of) port numbers the application running in private node uses and how long it uses them. On the contrary, DPF uses NAT port forwarding in dynamic way and does not require user's such knowledge.

3.1 Architecture

Fig-1 shows a Condor pool managed by a *central manager* and composed of machines inside private network as well as those in public network. You can also think the node in public network as a condor node that flocked to this pool. Condor nodes inside private network are DPF enabled, while those in public network need not be. We will call DPF-enabled Condor nodes *DPF clients*. Central manager can be placed anywhere in the administration domain: inside private, in public, or on the boundary of private network. *DPF server* is a process, running with root privilege on the NAT gateway.

DPF server manages a private network or part of it and acts as a proxy for clients within it. A private network can be partitioned and managed by multiple DPF servers, however a server can manage at most one network.



[Fig-1] Architecture of DPF

When a DPF client binds a sockets to a local ip:port, it sends to the DPF server forwarding requests with its local ip:port and optional desired public ip:port. DPF server sets port forwarding rule by calling NAT's API and replies failure with an appropriate error code, if it can't set the rule as requested. If succeed, it registers the client and replies success with public ip:port through which nodes in public network can connect to the client.

The client uses the public ip:port instead of the local ip:port as its endpoint identity at the application layer. That is, the client uses the ip:port whenever it needs to notify its communication endpoint to its peer or to information server such as the central manager. However, unlike RSIP, the client can still use regular socket calls to send packets because NAT will automatically modify packets when they traverse through it.

Now Condor nodes in public network can connect to DPF clients by sending packets to the public ip:port, which they can obtain from the central manager or through another connection to the clients that was established before.

For efficient communication within a private network, client inside private network sends to the server the query with peer's ip:port. Server answers the query with the local ip:port of the peer if the peer is registered to the server, i.e. the peer is in the same private network, otherwise it answers NAK. If the server answers success, the client connects to the peer using the local ip:port instead of that known to public.

3.2 Implementation (*Optional*)

3.2.1 DPF client

Each socket of DPF client is coupled to a management socket, a passive TCP socket, and shares its lifetime with the management socket. That is, it is created,

duplicated, inherited, and closed together with its peer: the management socket. Because it is guaranteed that the client socket is in use as long as the peer is open, DPF server is able to know that the socket is no longer in use when it cannot connect to its peer. This approach is a little inefficient than SOCKS, where each client socket has its management socket connected to SOCKS server during its lifetime, because DPF server must go through TCP connection setup process each time it wants to know whether a client socket is still in use. However, the server becomes much more scalable because it need not have as many auxiliary sockets open as client sockets in use. Also notice that DPF clients need not pay attention to the management sockets, because the kernel will accept connections to it as long as it open.

Passive DPF client sockets, i.e. UDP and listening TCP sockets, have port forwarding rules at the server and are publicly represented by public ip:port returned by the server. To make sure that the correct ip:port be advertised to the client's peer or central manager, DPF client asks the server to set a port forwarding rule right after it binds a socket to local ip:port. To maintain only necessary rules at the server, however, the client asks the server to delete the rule for the socket when the application makes it active, for example application calls 'connect' in Unix.

3.2.2 DPF server

DPF server is implemented as a daemon process running with root privilege on a machine with Linux 2.2 or higher and NAT enabled. Also the server must be placed where it can directly communicate with clients.

To handle client's request efficiently, the server maintains the mapping table, which contains port forwarding rules and client information that owns the rule. It also has the mirror file of the mapping table to make DPF run gracefully in the face of server failure and/or NAT machine reboot. As a result, we have three representations of port forwarding on server machine: the mapping table, the mirror file, and the kernel table of forwarding rule. The mapping table and the mirror file contain the same information almost all the time, which the kernel table has less information on each rule but may have more rules because rules may be set by methods other than DPF such as manually set for *ssh* server by administrator. DPF server is deliberately implemented so that the consistency between those three representations is maintained and the appropriate representation is used each time as explained below.

Queries from clients are answered based on the mapping table.

The addition of a rule is recoded in the order or the mapping table, the mirror file, and the kernel table. This order was carefully chosen to make system fault-tolerant and efficient. Because our design allows reuse of forwarding rules, the server stops modifying representation as soon as possible. If the same rule in the mapping table as the one being added is found, it reuses the rule and modifies nothing. On the other hand, if a rule with the same public ip:port as the new one but different client information is found, the server reuses the rule with modification only to the mapping table and the mirror file. If no match is found, it will add a new rule and modify all representations.

The deletion of rules is applied in the order of the mapping table, the kernel table, and the mirror file. Unlike the case of the addition, all the representations are modified.

The server replies to clients after modifying all the necessary representations to maintain the consistency with them. The addition and deletion orders above also guarantee the consistency among three representations in the server as well, because the mapping table and the kernel table are synchronized to the mirror file as explained below. Note that garbage rules may remain in the server if the server fails or system crashes after the mirror file is updated but answer is sent to the client. This causes no problem because the server periodically probes client sockets by making TCP connections to their peers and deletes those not in use by clients.

3.2.2.1 synchronization

The DPF server synchronizes three representations right after the periodic probe of client sockets. The synchronization is performed based on the contents of the mirror file.

For performance reason, new rules are always appended to the mirror file. As a result of this, rules overwritten by new rules due to the reuse of rules explained above may still remain in the mirror file. Hence the first step of synchronization is to delete all deprecated rules from the file. After that, the mapping table and the kernel table are synchronized to the mirror file. Note that the rules that are in the kernel table but not in the mirror file should not be deleted since they might be added by other methods.

4. Generic Connection Brokering (GCB)

Generic Connection Brokering (GCB) uses the idea similar to that of Napster, that is, the connection broker arranges which party should initiate a communication based on network situation of each party. To solve the interoperability problem of application-specific connection brokering system such as Napster and Gnutellar, GCB uses the idea of layer injection. By introducing GCB layer between application and system-call library, GCB makes connection

establishment at the application layer orthogonal to real connection setup at the kernel level. In other words, application program can call *connect* or *accept*, depending on the semantics of the application layer, without worrying about whether it can reach to or can be reached from its peer. GCB layer determines whether it should make a connection to or accept connection from the peer.

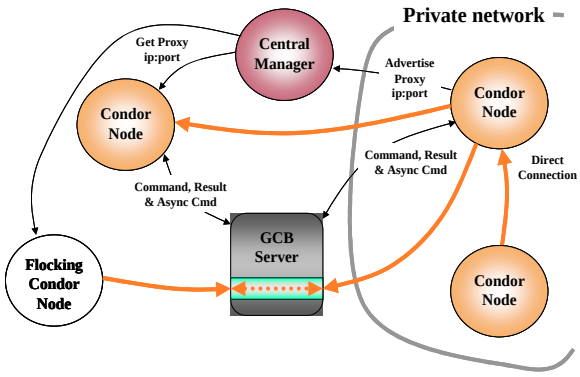
4.1 Architecture

Fig-2 shows a typical Condor pool using GCB. The pool has three Condor nodes, two inside private network and one in public, and is managed by a central manager node on the top. Fig-2 also shows a node on lower left that flocks from another pool. All nodes except the flocking node are GCB enabled and brokered by the GCB server. You may view the flocking node as another type of public node of the pool that is not aware of GCB protocol. We will call those nodes, which know GCB protocol and are managed by the GCB server, *GCB clients*.

GCB server generally manages clients within an administrative domain and arranges connections to those clients either from a process within the domain or outside the domain by arbitrating who should actively connect. Unlike DPF server, GCB server is a normal user process and can be placed either in public network or on the boundary between private and public network.

For easy explanation, let us call the processes willing to accept connections *listeners* and those trying to connect to one of listeners *connectors*.

GCB enabled listener registers the passive socket at its GCB server by sending a *register* request to the server. Upon receiving the register request, the server creates a proxy socket of the same type as the client socket, binds it, make it passive, and returns the address the proxy socket is bound to. From now on, the listener uses the proxy address as its network identity.



[Fig-2] GCB architecture

In other words, whenever it needs to inform other processes of its address, it sends the proxy address instead of its real address.

When another GCB client, a connector, wants to connect to the listener, it asks the listener's GCB server to broker the connection by sending *connect* request. The listener's GCB server can be contacted using the same IP address as the listener's proxy IP and the predefined port. The server decides, based on network situation of the connector and the listener, who should actively connect and arranges accordingly. If either cannot connect to the other because, for example, both are inside private networks, it lets both parties connect to the server and relays packets between them.

Since normal connectors do not know GCB protocol and think the proxy address of the listener as the real address, they will directly connect to the proxy socket that the server created when the listener registered its socket. Upon accepting a direct connection to the proxy socket, the server will ask the corresponding client to connect to the server and then will relay the packets between two connections.

Connection between GCB enabled connector and normal listener is established in a little ugly way. Since the connector thinks the listener's address as proxy one, it will try but fail to contact the listener's GCB server. When there is no process using the supposed-to-be GCB server's address, it will take one round trip time (RTT) for the connector to detect that the listener is not a GCB client. However, if any process happens to use the address, the connector needs a little more time to detect that the server does not understand *Reliable UDP* (RUDP) protocol that we implemented for exchanging GCB commands.

We understand drawbacks of our approach. First, network addresses are wasted because the real address of the listener in public network is unnecessarily hidden by proxy address. Second, connections to listeners in public network need not be brokered. Lastly, connections to normal listeners are slow due to the unnecessary RTT waste. If we used the combination of the real address and the proxy address as the network identity of GCB client, we would not have had these problems. However, we took this inefficient approach because this scheme allows us to use legacy socket address and gives us a great chance to extend GCB to be used any application. Furthermore, the uniform indirection gives sockets mobility. Suppose a process moving around machines. This type of process is very common in grid computing. Since the process can use the same address regardless of its location and connections to it will be appropriately brokered based on its current location, other processes can always make connections to it.

4.2 Implementation (*Optional*)

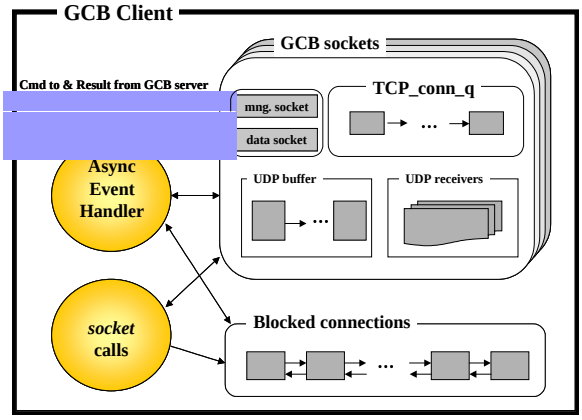
4.2.1Client Implementation

As shown in Fig-3, a GCB client has a pool of records of listening sockets and a list of blocked passive connections. Each listening socket is tightly coupled with a management socket and has the exactly same fate as its management socket, that is, they are created, duplicated, inherited, and closed together. The management socket, a UDP socket that uses reliable protocol we developed, is used for GCB command exchange with GCB server. It is also used to periodically send heartbeats to the server so that the server may be able to send asynchronous commands to the client and may also know that the listening socket is open and in use. Note that GCB server may be placed where it cannot directly talk to clients and, in this case, it can only send asynchronous commands to clients through UDP holes clients have made by initiating communication.

The record also has a connection queue, UDP message buffer, and UDP receiver list. The connection queue is only valid for TCP sockets while UDP message buffer and receiver list for UDP sockets. Notice that there are passive and active connections in GCB. Passive connections are maintained by the kernel and taken from kernel's queue by socket call *accept* in Unix while active ones, which are established in the reverse way by listener connecting to the connector, must be established and maintained by GCB module. Regardless of their types, connections should be returned to the application in the order that they were established. GCB client stores both active and passive connections in the connection queue in the order that they are established, and returns the one from the head of the queue to the application.

Since DUMMY messages can be received at a UDP socket anytime, GCB client must examine UDP packets received and acknowledge DUMMY packets or stash data packet somewhere to pass them to the application when requested. This is what UDP buffer is for.

UDP receiver list is a collection of records of peers who have recently communicated with the owner of it. Among other things, the record contains mappings between peer's public and local address. When the application wants to send UDP data to a receiver, GCB client searches UDP receiver list to find the record of the recipient. If it finds the record, it sends the packet to the local address of the record. If it can't find, it performs GCB connection setup procedure explained



[Fig-3] GCB client architecture

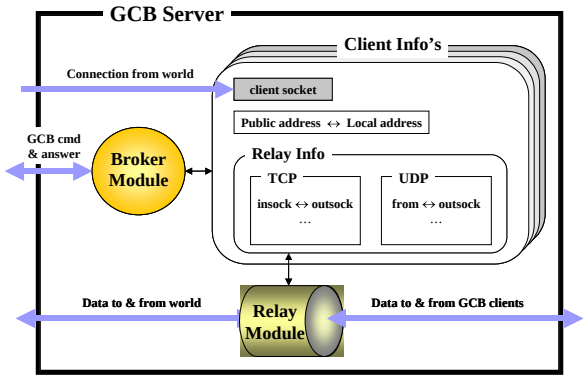
in section 4.2, adds a new record for the recipient to the receiver list, and then sends data through the channel established. Because GCB client does not know when a peer's record becomes invalid, it deletes the record when it encounters a network error with sending or receiving using the record. It also periodically flushes the list, for the same reason, and adds new records later when it needs to send something to deleted peers.

In GCB, commands from the server, DUMMY, DUMMY_ACK, new connection, normal UDP data packets, etc. can arrive anytime. This fact requires GCB client to be always prepared to handle any of those asynchronous events and makes its implementation very complex. Asynchronous Event Handler (AEH) is the routine that handles those events and is called once or more by every GCB client routines.

As explained earlier, GCB client is implemented in a generic way so that any application can use it to communicate across private network boundary. To achieve this goal, we overloaded socket calls so that they follow GCB protocol. We also changed blocking socket calls such as connect, select, recv, etc. so that they periodically call AEH and also send periodic heartbeats to the server.

4.2.2 Server Implementation

GCB server is composed of client socket records, each of which is created per client's listening sockets, and two functional modules: broker and relay module. Broker module handles all GCB commands from clients and brokers TCP connections and UDP communications as explained in section 4.2.



[Fig-4] GCB server architecture

Relay module handles TCP connections and initial UDP packets to the proxy socket, which is created for the listening socket at GCB client when it is registered, and creates relay rules. TCP relay rule is defined as a mapping between two TCP sockets. One is connected to the connector, who makes connection to the proxy socket, and the other to the client socket. Data packets arrived at one socket are relayed by relay module to the other socket and one connection is torn down when the other one is closed. UDP relay rule, on the other hand, is defined a mapping between a UDP socket and an address. When a UDP packet is received from an address for the first time[1] at UDP proxy socket, the relay module creates a UDP socket and connects it to the client socket and then adds a relay rule between the socket and the address the UDP packet came from. Packets received at the socket are sent to the address through the proxy socket, and those sent from the address and received at the proxy socket are passed to the corresponding client through the socket connected to it. To keep only valid relay rules, the server deletes relay rules when no packet is relayed using the rule for a long time. When clients resume communication after a long pause and hence the relay rule for them is deleted, the server creates a new rule.

Unlike DPF server, GCB has no persistent file to recover client information after the server or machine failure and hence it needs not synchronize multiple representations of client information. Once a connection is brokered by the server, clients can establish the connection and communicate with each other at the server failure. Broken TCP relay channel due to the server failure will be reliably detected by both parties being relayed. Also UDP relay rules will be resumed when the server is brought up again and it creates new rules upon receiving packets at proxy sockets. Hence the only thing that the server has to do for fault tolerance is to recover client's information at the level that new connections may be established to clients that registered themselves before server failure and are still valid after the server recovers from the failure. When the server receives a heartbeat from a client that is not registered, it sends *reregister_request* to the client and registers the client again upon receiving register_request from the client.

[1] This is not always true because of the periodic flush of UDP relay rules explained below

5. Performance

This section presents experimental results. We set two NAT-based private networks and collected data using a test suite. The test suite comprised of client and echo server and was written to use Cedar, the communication library of Condor, to establish connections and transfer data. Time was recorded at the client side. To minimize the effect of network fluctuation, we collected data for relatively short period of time but multiple times and averaged them.

The data were collected for three communication patterns: communication from private to public, from public to private, and from private to different private network. Private to public communication means that the client of the test suite runs inside private network and the echo server runs at public network. Other patterns are interpreted similarly. For each pattern, we compared regular, DPF, and GCB, and for each of these TCP and UDP communication data were collected.

For regular communication, we set static port forwarding at the head nodes so that every inbound connection is passed to the nodes the echo servers were running. For DPF testing, we placed DPF servers on the head nodes so that each server managed one private network. We used DPF clients and regular clients at the private and public network, respectively. For GCB, we placed GCB servers on the head nodes and used GCB clients at private networks as DPF case. However, at public network we tested both cases of client being GCB enabled and not enabled.

Table-1, 2, and 3 show data for each communication pattern. The first row shows the average connection times with their standard deviation in parentheses and the second row shows the times, also with standard deviations, for the data being echoed back to the client. The connection time actually includes all the time from socket creation to connection establishment. Since DPF and GCB make UDP holes through NAT or firewall when the first packet is sent, we included the time for the first UDP send to be echoed to the connection time. The numbers are shown in microseconds.

We must note that we just included UDP cases for informational purpose because it is almost impossible

to draw conclusion from UDP measurement due to its unreliable nature.

As the tables show, DPF is very fast both in connection setup and data transfer. Connection setup time of DPF is just a little slower than that of regular communication. For data transfer, DPF is as fast as regular communication as expected.

As expected, GCB connection is slower than DPF and data transfer is comparable to DPF and regular communication, even though we expected GCB to be a little slower because of the GCB layer introduced between application and system library and extra data copies between layers. From this data, we cannot draw a conclusion on the case that we ran regular client in the public side versus the case that we did GCB client at the public.

<Table-1> private-to-public communication

	Regular		DPF		GCB			
					Reg. Public		GCB Public	
	tcp	udp	tcp	udp	Tcp	udp	tcp	udp
Conn	1656 (258)	10167 (2032)	1703 (552)	12086 (303)	31428 (2720)	22868 (5193)	33934 (9259)	18692 (2255)
Data	22952 (3800)	2010 (912)	24863 (2121)	693 (260)	21051 (1045)	745 (136)	27629 (7388)	1650 (463)

<Table-2> public-to-private communication

	Regular		DPF		GCB			
					Reg. Public		GCB Public	
	tcp	udp	tcp	udp	Tcp	udp	tcp	udp
Conn	2007 (620)	12456 (206)	2074 (458)	10894 (351)	2624 (753)	12038 (410)	33530 (2902)	25408 (5184)
Data	21229 (933)	340 (32)	20842 (954)	1004 (150)	36620 (4367)	608 (105)	19455 (1664)	673 (25)

<Table-3> private-to-private communication

	Regular		DPF		GCB	
	tcp	udp	tcp	Udp	tcp	udp
Conn	922 (37)	788 (5)	1101 (40)	2204 (161)	6887 (182)	6590 (490)
Data	103726 (727)	592 (4)	102905 (720)	653 (1)	108293 (1736)	2959 (207)

6. Analysis

In this section, we explain how DPF and GCB satisfy our requirements. We also compare two approaches so that institutions can choose one of them depending their concerns and situations. We claim that both DPF and GCB satisfy all the requirements given in section 1.

DPF server is highly scalable. Since it does not maintain any TCP connections with its clients as SOCKS, the limiting factor of its scalability is the number of proxy addresses, i.e. ip:port pairs that can be leased to clients. Furthermore DPF server supports hosts with multiple public IP addresses, making the number of addresses that can be leased logically infinite. Hence its scalability is only limited by its processing and network speed.

We expect that GCB is less scalable than DPF. GCB server maintains a proxy socket per GCB client socket and uses two TCP connections for each TCP relay and one UDP socket for each UDP relay. Hence the number of listening GCB client sockets plus that of UDP communications being relayed are limited by the maximum number of file descriptor a process can open, and the number of TCP communications being relayed are limited by the half of the maximum TCP connections a process can have. However, GCB server uses only one UDP socket for management purpose and is scalable enough to support most clusters. Furthermore a cluster can be easily partitioned by changing environment variables so that it may be brokered by multiple GCB servers.

Both DPF and GCB satisfy the interoperability requirement. Regular sockets in public network can communicate with DPF or GCB clients inside private network without any change. In DPF, process in public network does not have any reason to be DPF enabled. In GCB, the process in public network needs to be GCB enabled for its incoming and outgoing connections to be brokered. However GCB server provides relay service for regular sockets.

As for the last requirement, neither DPF nor GCB requires any change to network component such as router or name server. GCB server is a user level daemon running with a normal privilege and is orthogonal to network configuration. DPF server is also a user level daemon but requires root privilege to call NAT library, and needs to be placed on the NAT head node of its clients.

Even though both DPF and GCB satisfy all the requirements in section 1, two systems have different characteristics in terms of scalability, performance, deployability, and etc. DPF is very efficient and scalable as we explained above. Also its implementation is relatively simple. It, however, is tightly coupled with NAT and supports only specific implementations of NAT: currently NAT on Linux 2.2 and 2.4. The fact that DPF server needs root privilege and should be placed on the head node, a very important and sensitive network element, can be a drawback. We believe that DPF fits very well to dedicated clusters, where cluster manager usually has the same administrative responsibility as network manager and high scalability and performance are essential because the clusters are usually big.

GCB has almost opposite characteristics to DPF. It is independent to network topology and private or firewall technology. Hence it supports almost every institution that allows outbound connections, supports nested private network, and works with NAT's non-promiscuous mode that is much stricter than its default promiscuous mode. GCB server can also runs with the least privilege. It, however, is less scalable and slower than DPF. As a consequence, we believe that GCB fits perfect to non-dedicated, small, or virtual clusters, where cluster managers usually cannot assume any administrative power over network or even cluster machines except several that belong to her.

7. Conclusion

In this paper, we presented two systems to recover the Internet connectivity in the Condor system. While satisfying representative requirements of grid computing, DPF and GCB have different characteristics in terms of performance, scalability, deployability, and security, thus allowing institutions to choose the better one depending on their policies and concerns.

References

[BRLMNT] 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.

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

[EPMLEP] D. H. J Epema, Miron Livny, R. van Dantzig, X. Evers, and Jim Pruyne, "A Worldwide Flock of Condors: Load Sharing among Workstation Clusters" *Journal on Future Generations of Computer Systems*, Volume 12, 1996

[EUGSTO] T. S. Eugene Ng, Ion Stoica, Hui Zhang, A Waypoint Service Approach to Connect Heterogeneous Internet Address Spaces, http://www-2.cs.cmu.edu/~eugeneng/papers/aves-paper.pdf.

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

[FSTKSS] I. Foster, C Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling scalable virtual organizations", Intl. Journal of Supercomputing Applications 2001.

[GNUTLL] "The Gnutella Protocol Specification v0.4 Document Revision 1.2", http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[GRTCHR] M. Gritter, D. R. Cheriton, "An Architecture for Content Routing Support in the Internet", Usenix Symposium on Internet Technologies and Systems, March 2001.

[LIVNY] Livny, M., "High-Throughput Resource Management", Foster, I. and Kesselman, C. eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, pp. 311-337.

[LTZLVN] Litzkow, M., Livny, M., and Mutka, M., "Condor - A Hunter of Idle Workstations", *Proc. 8th Intl Conf. on Distributed Computing Systems*, 1988, pp. 104-111.

[NAPSTR] "Napster Protocol Specification", http://opennap.sourceforge.net/napster.txt [P2PDOC] P2P WG, "Bidirectional Peer-to-Peer Communication with Interposing Firewalls and NATs", http://www.p2pwg.org/tech/nat/Docs/NATWhitePaperv09.5.pdf.

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

[RFC1928] M. Leech, M.Ganis, Y. Lee, R. Kuris, d. Koblas, L. Jones, "SOCKS Protocol Version 5", IETF RFC 1928 March 1996

[RFC2401] S. Kent, P. Atkinson, "Security Architecture for the Internet Protocol", IETF RFC 2401, Nov. 1998.

[RFC3102] M. Borella, J. Lo, D. Grabelsky, G. Montenegro, "Realm Specific IP: Framework", IETF RFC 3102, July 2000.

[RFC3103] M. Borella, D. Grabelsky, J. Lo, K. Taniguchi, "Realm Specific IP: Protocol Specification", IETF RFC 3103, Oct. 2001.

[RFC3104] G. Montenegro, M. Borella, "RSIP Support for End-to-End IPSEC", IETF RFC 3104, Oct. 2001.

[UPNPMS] "Understanding Universal Plug and Play", Microsoft Corporation, White Paper, 2000.

[UPNPURL] Universal Plug and Play Forum web site, http://www.UpnP.org.