Recovering Internet Symmetry in Distributed Computing

Sechang Son and Miron Livny
Computer Science Department, University of Wisconsin
{sschang, miron}@cs.wisc.edu

Abstract

This paper describes two systems to recover the Internet connectivity impaired by private networks and firewalls. These devices cause asymmetry in the Internet, making peer-to-peer computing difficult or even impossible. The Condor system is one of those that are severely impaired by the asymmetry. Compared to normal peer-to-peer computing applications, Condor has stricter requirements, which are 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 clusters very well because it is simple, efficient, and highly scalable. On the other hand, GCB perfectly supports non-dedicated or personal clusters 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, security, and scalability.

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) [13] in order to hide internal machines and more importantly provide a choke point where firewall policies can be applied.

Though private networks were 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 [5]. We believe that grid computing gives one of the most convincing examples that support this argument. The grid is the infrastructure that enables coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [6]. 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. 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 networks and firewalls, however, damaged Internet connectivity and made it asymmetric. The 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 not 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. The Condor system [9, 10], 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 [3] makes clusters even bigger by putting existing ones together. Hence we can't use an approach that assumes small number of machines inside a private network or behind a 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, and performance so that institutions may choose the better one depending on their policies and situations.

Firewalls and NAT-based private networks are essentially the same from the perspective of impacts on grid computing. Also connectivity loss due to private networks is considered severer because blocked connections in private networks are side effects. 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. In section 6, we explain how DPF and GCB satisfy grid requirements and compare two systems. Section 7 addresses security issues and section 8 concludes.

2. Previous Work

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 the Internet or need agreement between various institutions. For example, TRIAD [2, 8] and IP Next Layer (IPNL) [5] 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) [4] 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.

2.1. Application-specific connection brokering

Napster [11] 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 the private network, it asks the uploader to push files to the downloader in the public network. Gnutella [7] also uses the same idea, but without any server. When an uploader is inside the private network, the downloader in the 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 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 [14] as a standard for performing network proxies at the transport layer. The 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 the private network and those at the public network. When a node A at the public network wants to connect to B behind a SOCKS server, A sends a 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. It can be viewed as an application-independent approach because applications need not be rewritten to use SOCKS. Another advantage is that it is independent to private network technology and can be used with or even without any NAT-like proxy. It, however, has a few drawbacks, which makes it fail to satisfy our requirements:

- *It is not highly scalable*. Every socket served 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 sockets at public side cannot initiate to the private side*. With SOCKS, clients inside private

networks need not be changed at all. Nodes at public side, however, must be aware of SOCKS protocol and this violates our important requirement.

The last constraint of SOCKS shows that it was originally invented for client-server model as hinted in [14] 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) [1, 17, 18] has been proposed and adopted by IETF as a standard way to solve NAT problems, especially those related to IPSec [16] and inbound connection.

The client inside a private network leases public addresses—IP and port pair—from its RSIP server and uses those address as network endpoint identities. 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 to RSIP server, through a tunnel to the server. Upon receiving a packet through the tunnel, the server stripes off the tunnel header and forwards it to the public network. Inbound communications, including replies from the public peer, are handled in the reverse way. When the server receives a packet delivered to one of leased addresses, it forwards the packet to the leaser, through the tunnel to the leaser.

In addition to the support for inbound communications, RSIP solves NAT's incompatibility with IPSec [19]. 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.

RSIP, however, is 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 platforms, generally it should replace well-tested NAT. We don't believe that, in the 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)

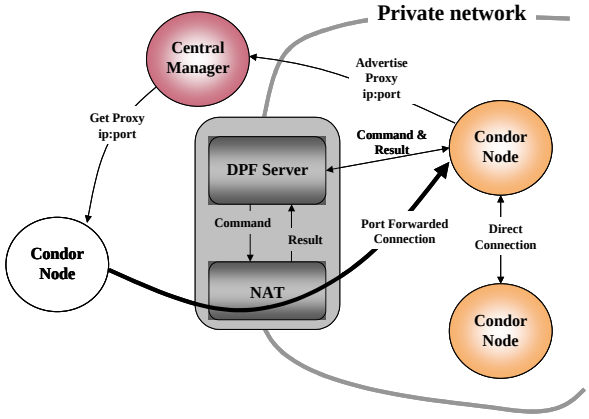
For easy explanation, we introduce two notations below and use them throughout the paper. *A:B* represents a pair of IP address *A* and port number *B*. *[A:B > C:D]* represents a mapping or translation rule from *A:B* to *C:D*.

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, to make inbound communication possible in NAT. When an NAT gateway receives a packet destined to ipA:portB and finds a forwarding rule [ipA:portB > ipX:portY], it rewrites the destination of the packet as ipX:portY and routes the rewritten packet toward IP address ipX. Hence machines inside a private network can accept inbound communications by setting port forwarding rules at NAT gateway.

At our best knowledge, port forwarding must be manually set /unset by the administrator. So maintaining the right set of forwarding rules for the right period is almost impossible because users do not know which ports and how long they will be used by applications. On the contrary, DPF maintains as many rules as required only for as long as necessary by setting and deleting rules dynamically.

3.1. Architecture

Fig-1 shows a Condor pool managed by a *central manager* and composed of machines inside a private network as well as those in the public network. The node in the public network can be viewed as a Condor node that flocked to this pool. Condor nodes inside the private network should be DPF-enabled, while those in the public network need not be. We will call DPF-enabled Condor nodes *DPF clients*. Central manager can also become DPF-enabled and if that is the case it can be moved into the private network. *DPF server* is a process, running on the NAT gateway.



[Fig-1] Architecture of DPF

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

When a DPF client binds a socket to a local address ipX:portY, it sends to the DPF server *forwarding* requests with ipX:portY and optional public address ipA:portB.

DPF server sets port forwarding rule [ipA:portB > ipX:portY] by using NAT API. The server 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 the public address ipA:portB through which nodes in the public network can connect to the client.

The client uses ipA:portB instead of its real address ipX:portY as its endpoint identity at the application layer. That is, the client uses ipA:portB whenever it needs to notify its address to peers or to information servers such as the central manager. However, unlike RSIP, the client still uses ipX:portY as the source address of packets it sends because NAT will automatically modify packets when they traverse through it.

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

For efficient communication within a private network, the client sends to its server a query with peer's address. If the peer is registered at the server too, i.e. the peer is in the same private network, the server answers the query with the local address of the peer, but it answers NAK otherwise. If the server answers success, the client connects to the peer directly at the local address instead of to the address known to the public.

3.2. Implementation

3.2.1. Client implementation. DPF client is implemented as a part of Condor communication layer, which provides some of presentation layer functionalities, various security mechanisms, and convenient network functions.

DPF command session, which consists of one or more DPF command exchanges, is always activated by clients. To start a session, the client makes a TCP connection to the server and then exchanges commands through it. This design makes command exchange a little slow but makes DPF server scalable because the server needs not maintain a TCP connection per client socket.

To handle the cases that client sockets are closed without reporting to the server, each client socket is coupled with a management socket, a listening TCP socket, and share its fate with the management socket. That is, it is created, duplicated, inherited, and closed together with its management socket. Because the management socket is open as long as the client socket is in use, DPF server can safely delete client's record and forwarding rules when it cannot connect to a management socket.

3.2.2. Server implementation. DPF server is implemented as a daemon process running with root privilege on NAT-enabled Linux 2.2 or 2.4 machines. Also the server must be placed where it can directly communicate with its clients.

To handle client's request efficiently, the server maintains a mapping table, which contains port forwarding rules and client information. It also has the mirror file of the mapping table to make DPF run gracefully over server restart and/or machine reboot. As a result, the server has three representations of port forwarding: the mapping table in memory, the mirror file in disk, and the kernel table of forwarding rules. The mapping table and the mirror file contain the same information but in different format almost all the time. The kernel table, which the kernel actually uses to rewrite and relay packets, has less information per rule, but may have extra rules set by other methods such as manual setting. DPF server is deliberately implemented so that the consistency between those three representations is maintained and the appropriate representation is used.

4. Generic Connection Brokering (GCB)

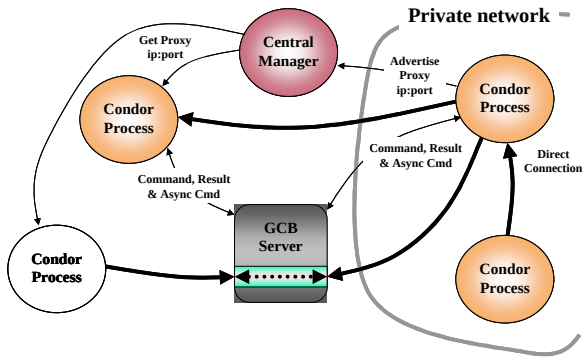
Generic Connection Brokering (GCB) uses the idea similar to that of Napster, that is, the connection broker arranges which party should initiate a communication based on network situation of each party. To solve the interoperability problem of application-specific connection brokering systems such as 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, the application program calls *connect* or *accept*, following application semantics 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 a private network and one in the 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 that know GCB protocol and are managed by the GCB server as *GCB clients*.

GCB server generally manages clients within an administrative domain and arranges connections to them

by arbitrating who should actively connect to whom. Unlike DPF server, GCB server is a normal user process and can be placed either in the public network or on the boundary of private networks.



[Fig-2] GCB architecture

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

GCB-enabled listener registers passive sockets at its GCB server by sending *register* requests to the server. Upon receiving a 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. 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 the 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 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 a process happens to use the address, it needs a little more time to detect that the process does not understand the reliable UDP protocol that we implemented for exchanging GCB commands or the GCB protocol.

Though GCB supports any combinations of connector and listener as explained above, connections for some combinations are made inefficiently. For example, GCB listeners in the public network need not be brokered, and GCB-to-normal connections waste at least an RTT. To handle this, GCB clients can be configured so as not to register sockets to the server and can also have a routing table that tells whether a site can be connected directly or using GCB.

Since private-to-private and normal-to-GCB communications are relayed by GCB server, end-to-end reliability issue can be raised. Certainly errors injected to packets by malicious or erroneous GCB server may not be detected by applications. However, relay mechanism has been accepted by Internet community and has been being widely used by applications such as SSH [20, 21] and SOCKS. Furthermore, TCP checksum is not perfect and we have seen many cases that error packets pass TCP checksum without being detected. Hence we claim that application level reliability or integrity mechanisms should be used if strong end-to-end reliability is required.

4.2. Implementation

4.2.1. Client implementation. GCB client is implemented as a layer between application and system calls. Since GCB layer provides the same interfaces and almost the same semantics of socket calls, applications that follow a few programming guidelines can be linked with GCB without modification.

To provide applications the same view for connections established opposite way as those established normal way, GCB client is implemented in a way that it seems like a user level file system running on top of the kernel file system. GCB client serves application's socket calls using the real system calls and referring internal data structures.

As Unix file system has a file descriptor table and file table, GCB client has FD table, each entry of which contains information specific to the socket descriptor, and socket table, which contains GCB socket information and may be shared among multiple FD entries. Each socket table has a connection queue, a UDP peer list, and a UDP receiving buffer. Connection queue maintains connections accepted normally or in the opposite way. Connections are returned from this queue to the application when it calls *accept*. UDP peer list has mappings of proxy to real address for UDP peers and is used to send UDP packets.

Since GCB command message should not be passed to the application, GCB client checks every UDP message received. If the message is a GCB command, it is handled appropriately, or it is queued into the UDP receiving buffer so that it is passed to the application later.

4.2.2. Server implementation. GCB server is implemented as a daemon that can be run with the least privilege. Also GCB server does not assume that it can directly talk to its clients and hence can be placed anywhere both clients and public nodes can talk to it.

When GCB server receives a register request from a client, it creates a proxy socket and a record for the client. Connection requests to the client are brokered by referring the record. When a connection is accepted to a proxy socket, the server creates a relay record and uses it to relay packets.

GCB server, unlike DPF server, needs not maintain information persistent over server restart or machine reboot. Instead it just does fresh start when it restarts. To keep the correct set of client records, GCB server asks the client from which a heartbeat is received but whose record is not in the server to register again.

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 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: “private-to-public”, “public-to-private”, and “private-to-the other private network”. 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 the 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.

	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-1> Private-to-public 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-2> Public-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)

<Table-3> Private-to-private communication

We must note that we just included UDP cases for informational purpose because it is almost impossible to draw conclusion from UDP measurements 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.

GCB connection is slower than DPF as expected 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. We cannot draw a conclusion on regular client versus GCB client at the public network from this data.

6. Analysis

In this section, we explain how DPF and GCB satisfy our requirements and compare two systems. *DPF server is*

highly scalable. 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 processing and network speed.

GCB is also scalable, though not as much as 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 passive sockets that a server can support is the maximum number of file descriptors that a process can open. Also the number of concurrent TCP relays is limited by the half of the maximum number of TCP connections that a process can have. However, GCB server consumes TCP connections only for active communications from a private to another private network and regular socket to GCB clients communications. Furthermore, a cluster can be easily partitioned without any privilege, unlike DPF.

Both DPF and GCB satisfy the interoperability requirement. Regular sockets in the public network can communicate with DPF or GCB clients inside the private network without any change. In DPF, process in the public network does not have any reason to be DPF enabled. In GCB, the process in the public network needs to be GCB enabled for incoming connections to be brokered. However non-GCB-enabled processes still can talk to the private side through the relay service.

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. 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. Security Issues

Because our system was developed to recover connectivity damaged partly because of security reasons, we need to address security issues.

Obviously DPF opens holes at the head node, which malicious outsiders can exploit to attack the network. DPF, however, can be deployed to only when network administrators allow running DPF server with root privilege on the head node. Hence administrators would try to understand security implication of DPF and allow to run DPF server only if DPF confirms to institution's security policy.

GCB case is subtle. Without intervention of network administrators, it may be deployed to the most restrictive institutions, which allow only outbound connections, yet it uses the security policy as further as possible and maybe even to the extent that some policymakers never thought about. Thus GCB may generate more chances that the policy is abused. Though we strongly believe that security must be provided in an orthogonal way to connectivity, we admit that GCB needs to be able to return secure connections for institutions that use reduced connectivity as a security mechanism.

8. 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.

9. References

[1] 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.

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

[3] 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

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

[11] "Napster Protocol Specification", http://opennap.sourceforge.net/napster.txt

[12] P2P WG, "Bidirectional Peer-to-Peer Communication with Interposing Firewalls and NATs", http://www.p2pwg.org/tech/nat/Docs/NATWhitePaperv09.5.pdf.

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

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

[15] Ralph Droms, "Dynamic Host Configuration Protocol" *IETF RFC 2131*, Mar. 1997

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

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

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

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

[20] Ylonen, T., "SSH Protocol Architecture", I-D draft-ietf-architecture-13.txt, Sept. 2002.

[21] Ylonen, T., "SSH Connection Protocol", I-D draft-ietf-connect-16.txt, Sept. 2002.