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, we believe, 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 the 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, 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 manageable 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 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 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 the 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 the 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.

Firewall and NAT based private network are essentially the same from the perspective of impacts on grid computing. Also connectivity loss due to the private network is considered more severe because connections blocked are side effects of the 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 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 private network technology independent 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 socket on the public side cannot initiate to the private side.** With SOCKS, clients inside the 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 [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 the private network leases public addresses—IP and port pair—from its RSIP server and uses those address as network endpoint identities. When the lessee 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 strips 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 lessee, through the tunnel to the lessee.
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 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 $B$. $[A:B \rightarrow C:D]$ 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 $[Nip:Nport \rightarrow ipX:portY]$, it rewrites the destination as ipX:portY and routes the rewritten packet toward IP address ipX. Hence machines inside the 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 the 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 the private network as well as those in the public network. You can also think the node in the public network as a Condor node that flocked to this pool. Condor nodes inside the private network are DPF enabled, while those in the public network need not be. We will call DPF-enabled Condor nodes DPF clients. Central manager is just another client and can be placed anywhere in the administration domain. 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.

When a DPF client binds a socket 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 the public network can connect to the client.
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 the 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, which know GCB protocol and are managed by the GCB server, GCB clients.

![Fig-2 GCB architecture](image)

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 the public network or on the boundary between the 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, makes 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 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 the reliable UDP protocol that we implemented for exchanging GCB commands or the GCB protocol.

We understand drawbacks of our approach. First, network addresses are wasted because the real address of the listener in the public network is unnecessarily hidden by proxy address. Second, connections to listeners in the 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. This mobility also opens the possibility to have Condor pools with mobile laptop computers or computers that obtain IP using DHCP [15].

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.

### Table 1: Private-to-public communication

<table>
<thead>
<tr>
<th></th>
<th>Regular</th>
<th>DPF</th>
<th>GCB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tcp</td>
<td>udp</td>
<td>tcp</td>
</tr>
<tr>
<td>Conn</td>
<td>1656</td>
<td>10167</td>
<td>1703</td>
</tr>
<tr>
<td></td>
<td>(258)</td>
<td>(2032)</td>
<td>(552)</td>
</tr>
<tr>
<td>Data</td>
<td>22952</td>
<td>2010</td>
<td>24863</td>
</tr>
<tr>
<td></td>
<td>(3800)</td>
<td>(912)</td>
<td>(2121)</td>
</tr>
</tbody>
</table>

### Table 2: Public-to-private communication

<table>
<thead>
<tr>
<th></th>
<th>Regular</th>
<th>DPF</th>
<th>GCB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tcp</td>
<td>udp</td>
<td>tcp</td>
</tr>
<tr>
<td>Conn</td>
<td>2007</td>
<td>12456</td>
<td>2074</td>
</tr>
<tr>
<td></td>
<td>(620)</td>
<td>(206)</td>
<td>(458)</td>
</tr>
<tr>
<td>Data</td>
<td>21229</td>
<td>340</td>
<td>20842</td>
</tr>
<tr>
<td></td>
<td>(933)</td>
<td>(32)</td>
<td>(954)</td>
</tr>
</tbody>
</table>

### Table 3: Private-to-private communication

<table>
<thead>
<tr>
<th></th>
<th>Regular</th>
<th>DPF</th>
<th>GCB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tcp</td>
<td>udp</td>
<td>tcp</td>
</tr>
<tr>
<td>Conn</td>
<td>922</td>
<td>788</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td>(37)</td>
<td>(5)</td>
<td>(40)</td>
</tr>
<tr>
<td>Data</td>
<td>103726</td>
<td>592</td>
<td>102905</td>
</tr>
<tr>
<td></td>
<td>(727)</td>
<td>(4)</td>
<td>(720)</td>
</tr>
</tbody>
</table>

We must note that we just included UDP data 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 the 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 and uses two TCP connections for each TCP relay and one UDP socket for each UDP relay. Hence the maximum 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 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 communications 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 expected. 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