Peer-to-Peer Networking with Firewalls

LU YAN, KAISA SERE and XINRONG ZHOU
Turku Centre for Computer Science (TUCS) and
Department of Computer Science, Åbo Akademi University
FIN-20520 Turku
FINLAND

Abstract: A lot of networks today are behind firewalls. In peer-to-peer networking, firewall-protected peers may have to communicate with peers outside the firewall. This paper shows how to design peer-to-peer systems to work with different kinds of firewalls within the object-oriented action systems framework by combining formal and informal methods. We present our approach via a case study of extending a Gnutella-style peer-to-peer system to provide connectivity through firewalls.

Key-Words: Firewall, peer-to-peer, object-oriented, action systems, Gnutella

1 Introduction

The idea of peer-to-peer networking, in a sense that nodes on the network communicate directly with each other, is as old as Internet itself. Internet used to be a peer-to-peer network if we go back to those early days in the 70's when Internet was limited to researchers in a few selected laboratories. Nowadays Internet has developed into a non peer-to-peer network, in the sense that most exchanges rely on mediation through gateways and servers. Moreover, most networks today employ firewalls, for security reasons, which impede direct communication by filtering packets and limiting the port numbers open to bi-directional traffic.

The goal of peer-to-peer networking is to remove the distinction between client and server. Instead of running web browsers that can only request information from web server, users can run peer-to-peer applications to contribute contents or resources in addition to requesting them. As a vision of peer-to-peer networking, it is necessary for peer-to-peer applications to work in most environments, whether home, small business, or enterprise.

We have specified a Gnutella-like peer-to-peer system in [1]. When implementing such a system in Java, we realized that a lot of networks today are behind firewalls. In peer-to-peer networking, firewall-protected peers may have to communicate with peers outside the firewall. Thus a solution should be made to create communication schemes that overcome the obstacles placed by the firewalls to provide universal connectivity throughout the network. This motivates us to conduct a study of firewalls in peer-to-peer networking and achieve a way to traverse firewalls.

The rest of the paper is organized as follows. We start by defining our problem in Section 2. In Section 3 we give an overview of the Gnutella network. A solution to uni-directional firewalls is derived in Section 4. In Section 5 we provide a solution to another kind of firewalls that limit the open port numbers. We end in Section 6 with related work and concluding remarks.

2 Problem Definition

As firewalls have various topologies (single, double, nested, etc.) and various security policies (packet filtering, one-way-only, port limiting, etc.), our problem has multiple faces and applications have multitude requirements. A general solution that fits all situations seems to be infeasible in this case. Thus we define the problem as shown in Fig. 1: How to provide connectivity between private peers and public peers through a single firewall?

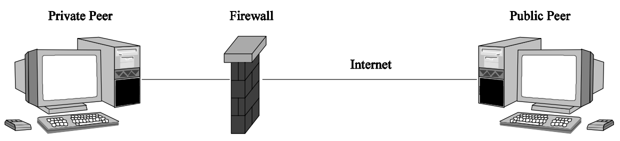

Fig. 1. Problem definition

We select the object-oriented action systems framework with UML diagrams as the foundation to work on. In this way, we can address our problem in a unified framework with benefits from both formal and informal methods.

Action systems is a state based formalism. It is derived from the guarded command language of

Dijkstra[2] and defined using *weakest precondition* predicate transformers. An *action*, or guarded command, is the basic building block in the formalism. An action system is an iterative composition of actions. The action systems framework is used as a specification language and for the correct development of distributed systems.

OO-action system is an extension to the action system framework with object-oriented support. An OO-action system consists of a finite set of classes, each class specifying the behavior of objects that are dynamically created and executed in parallel. The formal nature of OO-action systems makes it a good tool to built reliable and robust systems. Meanwhile, the object-oriented aspect of OO-action systems helps to build systems in an extendable way, which will generally ease and accelerate the design and implementation of new services or functionalities. Furthermore, the final set of classes in the OO-action system specification is easy to be implemented in popular OO-languages like Java, C++ or C#.

In this paper, however, we skip the details of semantics of action systems and its object-oriented extension, which can be found in [3] and [4].

3 Gnutella Network

Gnutella[5] is a decentralized peer-to-peer file-sharing model that enables file sharing without using servers. To share files using the Gnutella model, a user starts with a networked computer A with a Gnutella *servent*, which works both as a server and a client. Computer A will connect to another Gnutella-networked computer B and then announce that it is *alive* to computer B. B will in turn announce to all its neighbors C, D, E, and F that A is alive. Those computers will recursively continue this pattern and announce to their neighbors that computer A is alive. Once computer A has announced that it is alive to the rest of the members of the peer-to-peer network, it can then search the contents of the shared directories of the peer-to-peer network.

Search requests are transmitted over the Gnutella network in a decentralized manner. One computer sends a search request to its neighbors, which in turn pass that request along to their neighbors, and so on. Figure 2 illustrates this model. The search request from computer A will be transmitted to all members of the peer-to-peer network, starting with computer B, then to C, D, E, F, which will in turn send the request to their neighbors, and so forth. If one of the computers in the peer-to-peer network, for example, computer F, has a match, it transmits the file information (name, location, etc.) back through all the computers in the pathway towards A (via computer B in this case). Computer A will then be able to open a direct connection with computer F and will be able to download that file directly from computer F.

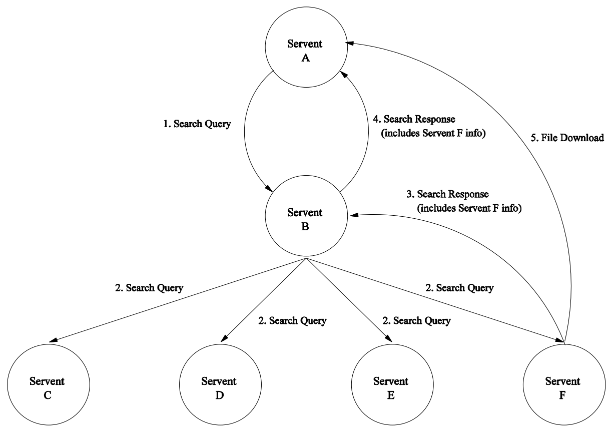

Fig. 2. Gnutella peer-to-peer model[6]

4 Uni-directional Firewalls

Most corporate networks today are configured to allow outbound connections (from the firewall protected network to Internet), but deny inbound connections (from Internet to the firewall protected network) as illustrated in Fig. 3.

These corporate firewalls examine the packets of information sent at the transport level to determine whether a particular packet should be blocked. Each packet is either forwarded or blocked based on a set of rules defined by the firewall administrator. With packet-filtering rules, firewalls can easily track the direction in which a TCP connection is initiated. The first packets of the TCP three-way handshake are uniquely identified by the flags they contain, and firewall rules can use this information to ensure that certain connections are initiate in only one direction. A common configuration for these firewalls is to allow all connections initiated by computers inside the firewall, and restrict all connections from computers outside the firewall. For example, firewall rules might specify that users can browse from their computers to a web server on Internet, but an outside user on Internet cannot browse to the protected user's computer.

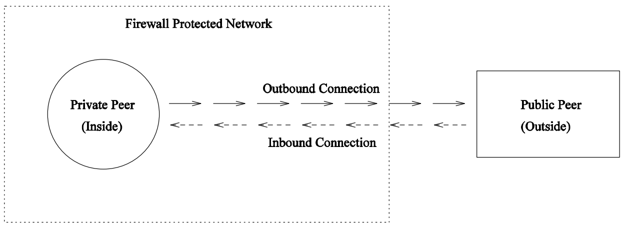

Fig. 3. Uni-directional firewall

In order to traverse this kind of firewalls, we introduce *Push* descriptor and routing rules for servents: Once a servent receives a QueryHit descriptor, it may initiate a direct download, but it is impossible to establish the direct connection if the servent is behind a firewall that does not permit incoming connections to its Gnutella port. If this direct connection cannot be established, the servent attempting the file download may request that the servent sharing the file *Push* the file instead. i.e. A servent may send a Push descriptor if it receives a QueryHit descriptor from a servent that doesn't support incoming connections.

Push descriptors are routed by *ServentID*, not by *DescriptorID*. Intuitively, Push descriptors may only be sent along the same path that carried the incoming QueryHit descriptors as illustrated in Fig. 4. A servent that receives a Push descriptor with *ServentID* = *n*, but has not seen a QueryHit descriptor with *ServentID* = *n* should remove the Push descriptor from the network. This ensures that only those servents that routed the QueryHit descriptors will see the Push descriptor.

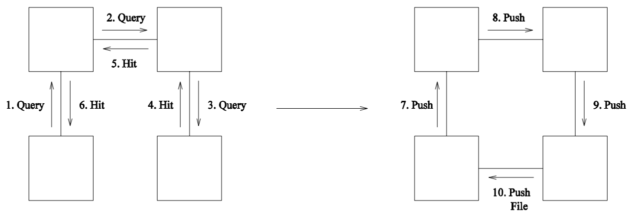

Fig. 4. Push routing[6]

We extend the original system specification in [1] to adopt uni-directional firewalls by adding a Push router *Rf*, which is a new action system modeling Push routing rules as shown in Table 1. We compose it with the previous two action systems in [1] *Rc* modeling Ping-Pong routing rules and *Rl* modeling Query-QueryHit routing rules together, to derive a new specification of router

$$R = |[\ Rc\ //\ Rl\ //\ Rf\ ]|\ ,$$

where on the higher level, we have components of a new router

{<*Router*, *R*>, <*PingPongRouter*, *Rc*>, <*QueryRouter*, *Rl*>, <*PushRouter*, *Rf*>}.

Table 1. Specification of Push router

```
Rf = |[ attr serventDB := null; cKeyword := null;
          filename := null; target := null;
          pushTarget := null
        obj receivedMsg : Msg; newMsg : Msg;
          f : FileRepository
        meth SendPush( ) =
              (newMsg := new(Msg(Push));
               newMsg.info.requestIP := ThisIP;
               newMsg.info.filename :=
                 receivedMsg.info.filename;
               newMsg.info.destinationIP :=
                 receivedMsg.info.IP;
               OutgoingMessage := newMsg);
             ReceiveMsg( ) = receivedMsg :=
               IncomingMessage;
             ForwardMsg(m) = (m.TTL > 0 →
               m.Transmit( );
               OutgoingMessage := m)
        do
           true →
             ReceiveMsg( );
             if receivedMsg.type = QueryHit →
                 serventDB := serventDB U
                 receivedMsg.serventID;
               if receivedMsg.info.keyword =
                 cKeyword →
                   target := receivedMsg.info.filename
                   @receivedMsg.info.IP;
                   if f.firewall →
                       SendPush( )
                   fi
                   cKeyword := null
               [] receivedMsg.info.keyword ≠
                  cKeyword ^
                  receivedMsg.descriptorID є
                  descriptorDB →
                    ForwardMsg(receivedMsg)
               fi
             [] receivedMsg.type = Push →
                 if receivedMsg.info.destinationIP =
                   ThisIP →
                     PushTarget :=
                       receivedMsg.info.requestIP®
                       receivedMsg.info.filename@
                       receivedMsg.info.destinationIP
                 [] receivedMsg.info.destinationIP ≠
                   ThisIP ^
                   receivedMsg.serventID є
                   serventDB →
```

```
                ForwardMsg(receivedMsg)
            fi
        fi
    od
]|
```

A servent can request a file push by routing a Push request back to the servent that sent the QueryHit descriptor describing the target file. The servent that is the target of the Push request should, upon receipt of the Push descriptor, attempt to establish a new TCP/IP connection to the requesting servent. As specified in the refined file repository in Table 2, when the direct connection is established, the firewalled servent should immediately send a HTTP GIV request with *requestIP*, *filename* and *destinationIP* information, where requestIP and destinationIP are IP address information of the firewalled servent and the target servent for the Push request, and filename is the requested file information. In this way, the initial TCP/IP connection becomes an outbound one, which is allowed by uni-directional firewalls. Receiving the HTTP GIV request, the target servent should extract the requestIP and filename information, and then construct an HTTP GET request with the above information. After that, the file download process is identical to the normal file download process without firewalls. We summarize the sequence of a Push session in Fig. 5.

Table 2. Specification of file repository

```
F = |[ attr firewall* := false; fileDB := FileDB;
        cFileDB; filename := null; target := null;
        pushTarget := null
    meth SetTarget(t) = (target := t);
         PushTarget(t) = (pushTarget := t);
         Has(key) = ({key} $\in$ dom(fileDB));
         Find(key) = (filename := file ^
             {file} $\in$ ran({key} ◄ fileDB))
    do
        target $\neq$ null →
            cFileDB := fileDB;
            HTTP_GET(target);
            target := null;
            Refresh(fileDB);
            if fileDB = cFileDB →
                firewall := true
            [] fileDB $\neq$ cFileDB →
                firewall := false
            fi
        [] pushTarget $\neq$ null →
            HTTP_GIV(pushTarget);
            pushTarget := null;
            Refresh(fileDB)
    od
]|
```

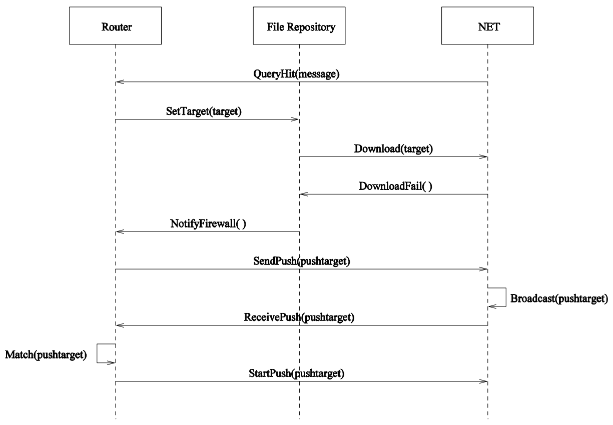

Fig. 5. Sequence diagram of a Push session

5 Port-blocking Firewalls

In corporate networks, another kind of common firewalls are port-blocking firewalls, which usually do not grant long-time and trusted privileges to ports and protocols other than port 80 and HTTP/HTTPS. For example, port 21 (standard FTP access) and port 23 (standard Telnet access) are usually blocked and applications are denied network traffic through these ports. In this case, HTTP (port 80) has become the only entry mechanism to the corporate network. Using HTTP protocol, for a servent to communicate with another servent through port-blocking firewalls, the servent has to *pretend* that it is an HTTP server, serving WWW documents. In other words, it is going to mimick an *httpd* program.

When it is impossible to establish an IP connection through a firewall, two servents that need to talk directly to each other, solve this problem by having SOCKS support built into them, and having SOCKS proxy running on both sides. As illustrated in Fig. 6, it builds an HTTP-tunnel between the two servents.

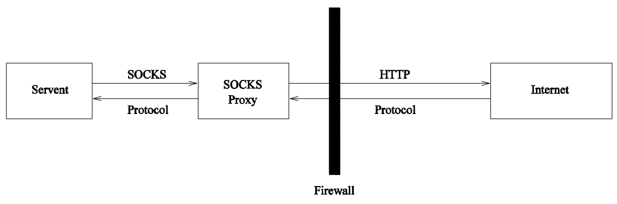

Fig. 6. Firewall architecture and extendable socket

After initializtion, the SOCKS proxy creates a *ProxySocket* and starts accepting connections on the Gnutella port. All the information to be sent by the attempting servent is formatted as a URL message (using the GET method of HTTP) and a *URLConnection* via HTTP protocol (port 80) is made. On the other side, the target servent accepts the request and a connection is establish with the attempting servent (actually with the SOCKS proxy in the target servent). The SOCKS proxy in the target servent can read the information sent by the attempting servent and write back to it. In this way, transactions between two servents are enabled.

We extend the original system specification in [1] to adopt port-blocking firewalls by adding a new layer to the architecture of servent in Fig. 7. This layer will act as a tunnel between servent and Internet.

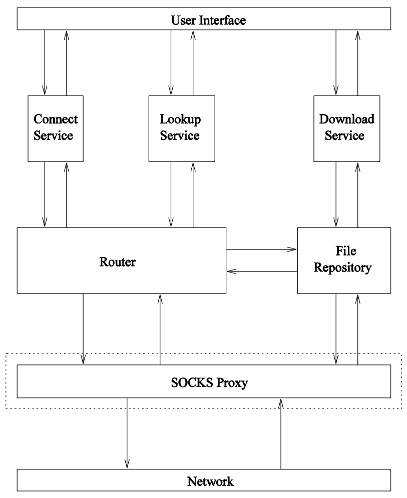

Fig. 7. Refined architecture of servent

As specified in Table 3, after receiving messages from the attempting servent and encoding them into HTTP format, the SOCKS proxy sends the messages to Internet via port 80. In the reverse way, the SOCKS proxy keeps receiving messages from HTTP port and decoding them into original format. With this additional layer, our system can traverse port-blocking firewalls without any changes in its core parts. We summarize the sequence of a SOCKS proxy session in Fig. 8.

Table 3. Specification of SOCKS proxy

```
S = |[ attr listenPort := GnutellaPort;
            DestinationPort := 80
       obj ProxySocket : Socket;
            HTTPSocket : Socket;
            imsg : Msg; omsg : Msg
       init ProxySocket = new(Socket(listenPort));
            HTTPSocket =
               new(Socket(destinationPort))
       do
            IncomingRequest ≠ null →
               imsg := EncodeSOCK(DecodeHTTP
                  (HTTPSocket.Read( )));
               IncomingMessage :=
                  ProxySocket.Write(imsg)
          [] OutgoingRequest ≠ null →
               omsg := EncodeHTTP(DecodeSOCK
                  (ProxySocket.Read( )));
               OutgoingMessage :=
                  HTTPSocket.Write(omsg)
       od
     ]|
```

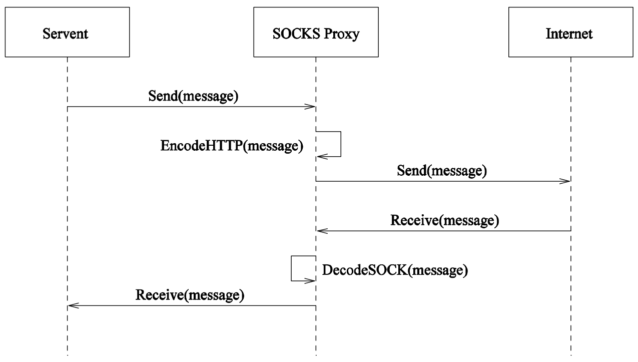

Fig. 8. Sequence diagram of a SOCKS proxy session

6 Related Work and Conclusions

The corporate firewall is a double-edged sword. It helps prevent unauthorized access to the corporate Web, but may disable access for legitimate peer-to-peer applications. There has been protocols such as PPTP (Point-to-Point Tunneling Protocol)[7], UPNP (Universal Plug and Play)[8], RSIP (Realm Specific IP)[9] and Middlebox protocol[10] to address the firewall problems in peer-to-peer networking. A recent protocol, JXTA[11] from Sun has provided an alternative solution to the firewall problem by adding a publicly addressable node, called “rendezvous server”, which firewalled peer can already talk to. The scheme is that peers interact mostly with their neighbors who are on the same side of the firewall as they are and one or a small number of designated peers can bridge between peers on the different sides of the firewall. But the problem posed by firewalls still remains when configuring the firewalls to allow traffic through these bridge peers.

As the previous work in [1], we have specified a Gnutella-like peer-to-peer system within the OO-action systems framework by combining UML diagrams. In this paper, we have presented our solution to traverse firewalls for peer-to-peer systems. We have extended a Gnutella-style peer-to-peer system to adopt uni-directional firewalls and port-blocking firewalls using OO-action systems. During the extending work, our experiences show that the object-oriented aspect of OO-action systems helps to build systems with a reusable, composable and extendable architecture. The modular architecture of our system makes it easy to incorporate new services and functionalities without great changes to its original design.

Peer-to-peer computing is currently attracting lots of attention, spurred by the surprisingly rapid deployment of some peer-to-peer applications like Napster, Gnutella and Kazaa. Firewalls have become a great challenge to peer-to-peer computing on Internet. A new technology, SOAP[12], has been developed to provide safe and reliable access through firewall protection. In the future work, we plan to explore this new standard and incorporate it into the development of peer-to-peer systems in firewall environments.

References:

[1] L. Yan and K. Sere, *Stepwise Development of Peer-to-Peer Systems*, Proceedings of the 6th International Workshop in Formal Methods (IWFM'03), Dublin, Ireland, July 2003. Electronic Workshops in Computing (eWiC), British Computer Society (BCS).

[2] E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall International, 1976.

[3] R. J. R. Back and K. Sere, *From Action Systems to Modular Systems*, Software Concepts and Tools. (1996) 17: 26-39.

[4] M. Bonsangue, J.N. Kok and K. Sere, *An approach to object-orientation in action systems,* Proceedings of Mathematics of Program Construction (MPC'98), Marstrand, Sweden, June 1998. Lecture Notes in Computer Science 1422. Springer Verlag.

[5] Clip2 DSS, *Gnutella Protocol Specification*, http://www.clip2.com/GnutellaProtocol04.pdf.

[6] I. Ivkovic, *Improving Gnutella Protocol: Protocol Analysis and Research Proposals*. Technical report, LimeWire LLC, 2001.

[7] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn, *Point-to-Point Tunneling Protocol (PPTP)*, RFC 2637, July 1999.

[8] *Understanding Universal Plug and Play,* Microsoft Corporation, WhitePaper, 2000.

[9] M. Borella and G. Montenegro, *RSIP: Address Sharing with End-to-End Security*, Proceedings of the Special Workshop on Intelligence at the Network Edge, California, USA, March 2000.

[10] B. Reynolds and D. Ghosal, *STEM: Secure Telephony Enabled Middlebox,* IEEE Communications Special Issue on Security in Telecommunication Networks, October 2002.

[11] L. Gong, *JXTA: A network programming environment*, IEEE Internet Computing, 5(3): 88-95, May/June 2001.

[12] W3C, *Simple Object Access Protocol (SOAP),* http://www.w3c.org.