A TCP-layer name service for TCP ports

Sérgio Freire
PT Inovação / IEETA / Univ. of Aveiro

André Zúquete
IEETA / IT / Univ. of Aveiro

Abstract

This paper presents a simple name service for TCP ports, allowing services to be reached by name instead of number. Names are arbitrary byte arrays that are bound to listening ports. Name resolutions take place during the TCP three-way handshake, not requiring extra message exchanges. The new TCP handshake conforms with the standard and is fully compatible with existing TCP implementations. A prototype implementation was developed in Linux, paying special attention to backward compatibility with legacy systems (kernels and applications). Among the many opportunities created by the name service, it allows services with unusual names, known only by small communities, to remain undetected by port scanners (though not by network sniffers).

1 Introduction

Name systems are useful for translating user-friendly, readable strings into numerical identifiers. A popular name system is DNS [7], used for translating hierarchical, typed names into IP addresses, among other identifiers. Other name services frequently used by applications are RPC name services, such as `rpcbind` for Sun RPC and Microsoft Locator for Microsoft RPC. But up to now there is no widely used, generic name service for transport endpoints, such as TCP or UDP port names.

Historically, some TCP port names are statically bound to well known services or servers [11]. Examples are `ftp` for port 21, `telnet` for port 23, `http` for port 80, etc. This static mapping between names and ports was initially supported by local services using local data (e.g. file `/etc/services` in Unix systems). Currently there is a database service with all these static mappings [10]. However, these mappings are not mandatory; they just reflect a common use.

In this paper we propose a name service for TCP ports which enables clients and servers to resolve arbitrary names (byte arrays) to TCP ports. The advantages of using this name service are twofold: (i) users may discriminate servers using names instead of numbers and (ii) TCP port scanners, such as `nmap`, should not be capable of discovering servers bound to unusual names.

Using names for referring ports provides a more intuitive way to refer services, instead of numbers. Service names that formerly were bound to static well-known port numbers may continue to exist but do not need any more to be bound to the same ports. For instance, we can bind the names `http` to port 8080 and `http1` to port 80. Clients access either port specifying their name, `http` or `http1`, instead of numbers 8080 and 80. Port names are also useful for uniform and uniquely tagging ports used by the servers of overlay networks.

Using arbitrary byte arrays to name TCP ports also prevents port scanning tools to discover listening ports. In fact, the success of port scanners in discovering listening ports bound to servers is due to the current small domain of port numbers — $\left[1, 2^{16} - 1\right]$. With arbitrary port names, we are able to deploy services with unusual, possibly long port names which cannot be easily found by port scanners. Services with unusual, confidential port names may be useful in many circumstances requiring restricted access profiles, namely:

- Experimental server deployment in pre-production environments;
- Private service deployment, such as personal content providers (file or web servers, mail servers);
- Restricted overlay networks.

The screening of listening TCP ports by mapping them to unusual port names is somewhat similar to the Port Knocking mechanisms [1, 4]. However, Port Knocking is an access control mechanism that requires a per-host or per-network access key. Instead, we simply require the knowledge of port names and not any key-based access control mechanisms; the knowledge of a port name is the key to access the service that uses the port.

2 Related Work

In this paper we describe a way of addressing a peer port by a name formed by a set of bytes, which acts as a weak access control mechanism – one has to know the name to access the port. Thus, in this section we briefly describe other contributions concerning these two subjects: binding of names to transport ports and access control mechanisms to transport ports.

DNS SRV records: RFC 2782 [3] defined a new type of DNS records, SRV RR, for resolving port names to a set of $\langle$*host DNS name,port number*$\rangle$ pairs. By creating SRV RR entries in the DNS, domain administrators are able to specify a set of locations (hosts) of a given service described by a friendly name, for the domain. For example, when the name `_foobar._tcp` is resolved in a DNS domain, it returns a set of $\langle$*host DNS name,port number*$\rangle$ pairs where the `foobar` service over TCP sits.

These records are useful for locating public services with well-known names in a domain, but not for dealing with arbitrary port names used in ad-hoc client-server connections, due to three reasons. First, the client application must initially learn the DNS domain of the target host before resolving the port name. For instance, to connect to port `foobar` at host 192.168.1.1, the client must first discover the DNS domain of 192.168.1.1 (e.g. `example.org`) and then to resolve the name `_foobar._tcp.example.org`. Second, the server host must have a DNS name so that clients could match target IP addresses with host names returned by SRV RR resolutions. Finally, it requires frequent DNS updates, and the inherent administrative privileges to do so, in order to dynamically create SRV RR entries whenever servers bind names to port numbers. This is a major blocker for dynamic port name assignment and for normal end-user usage of port names.

TCPMUX: This is a service using TCP port 1 which allows a host to provide a port name handoff service for itself [5]. A client host opens a connection to port 1 on a server host and transmits the desired port name in the data stream; the server replies with a positive or negative name resolution by means of a reply character in the data stream. If the named service is available, the connection is transferred to the desired service.

In Linux systems the TCPMUX service is provided for named services handled by the *inetd* daemon. Thus, arbitrary servers cannot provide name-number bindings to TCPMUX; only servers listed in *inetd* configuration files can have named ports.

The use of TCPMUX is not transparent to clients, as they must use in a different way the connection to a server: first they must contact TCPMUX and provide the port name, then interpret the TCPMUX reply and only afterwards, in case of success, proceed with the intended client-server interaction. Since most TCP client-server applications use a different approach, they contact directly a target server using its port number, our goal was just to improve this semantic by allowing them to use names instead of numbers to identify servers.

Port Knocking: Also known as Spread-Spectrum TCP [1, 4], Port Knocking (PK) is a mechanism for restricting access to services by allowing only authenticated requests to reach servers. It is a passive authentication scheme for TCP connections, acting as an auxiliary and external mechanism, independent of the kernel and the applications. The so called *knock* or *authentication* is typically a sequence of connection attempts to closed ports, which are intercepted and validated by a PK daemon at the destination peer. After receiving a correct *knock* from a client, the daemon allows it to connect to the wanted service port. The sequence of ports contacted in a *knock* can have an encoded meaning, like the origin IP, the remote port number and a checksum, that allows further control of the connection establishment.

Port Knocking, though a simple concept, requires: (i) a PK daemon between the client and the server; (ii) a firewall close to the server with an interaction mechanism with the PK daemon; (iii) a client application, or library functions, to carry on the *knock* before starting a connection to a protected port; and (iv) a set of closed ports for sending knock sequences.

Alternative PK implementations, using a single knock datagram (Tailgate TCP) or either an IP or TCP option (Option-Key TCP), have also been discussed in [1]. Nevertheless, the requirements are mainly the same.

Proposed Internet Draft: A discontinued IETF Internet Draft [12] proposed an extension to support TCP port names. The main goal of this proposal was to increase the number of concurrent connections for existing services by decoupling them from fixed IANA reserved port numbers. In this proposal, named server ports are in fact resolved by clients, i.e., clients proposed a resolution that is accepted by the server host if the name exists. Thus, a SYN with a server named port contains also a proposed server port name, and the returned SYN+ACK returns a name-number acceptance reply.

Port names are UTF-8 strings exchanged in a TCP header option, which strongly limits their maximum length. As a TCP header is limited to a maximum of 60 bytes, 20 of them mandatory, TCP options can only occupy 40 bytes. Moreover, part of these 40 bytes may be occupied by other TCP options, which further reduces the possible lengths of port names. This was one of the major concerns with this proposal.

Our name service is somewhat similar to this proposal but takes some different approaches. First, name to number resolutions are given by servers, and not proposed by clients, which allows us to get the same port-service decoupling on the server side if required. Second, names have arbitrary contents, they are simply byte arrays, and not just UTF-8 strings. Third, names are transmitted in the payload of TCP segments to overcome the space limitations of TCP headers. And fourth, name-number resolutions are provided to client TCP stacks, which allows future enhancements to perform semantic-aware validations (e.g. signed name resolutions, which can be validated by client applications). Therefore, our proposal is a superset of this one, in the sense that it includes all its benefits while being more flexible and powerful.

3 Proposed Name System

DNS and RPC name systems are implemented by autonomous servers, using well-known port numbers, which receive resolution requests from the application layer. Their goal is to provide a mapping from a name to a number that could be used as a parameter for lower protocol layers, namely the transport layer. Our proposal for a TCP name system goes in the opposite direction, which is to integrate it in the existing mechanism used for TCP connection establishment: the three-way handshake. Consequently, we do not use any new or existing name server, which is preferable for fault tolerance, and the name service is implemented by the TCP layer.

Besides fault tolerance, we designed the new TCP name service with the following goals in mind. The first one was to maintain compatibility with existing TCP/IP stacks. The second one was to add a new name service, and not to replace the current addressing mechanism used by TCP segments, with port numbers, by names, as numbers are more efficient to handle than names. The third one was to allow TCP clients and servers to use arbitrary name formats, in order not to restrict future uses by upper protocol layers. We also foreseen another goal, which is not covered in this paper, which is to add arbitrary semantics to name resolution (e.g. versions). In this paper we only have one simple, default semantic: strict byte equality between names.

3.1 Name binding

Our TCP name service allows TCP servers to bind names to listening ports and clients to use port names when requesting a TCP connection. The port name is associated with the service/application during the socket binding procedure at the server side. Clients refer the port name when they specify the TCP address of the server.

As we just want to provide a TCP name service, and not to fully replace port numbers by port names, port names must always be associated to port numbers. Therefore, the modified TCP layer on the server side will keep a port number to each local port name. When a application binds a name to a TCP endpoint (socket), it immediately gets a number as well. For backward compatibility, applications may specify the port number; if not specified, the TCP layer allocates a free port number.

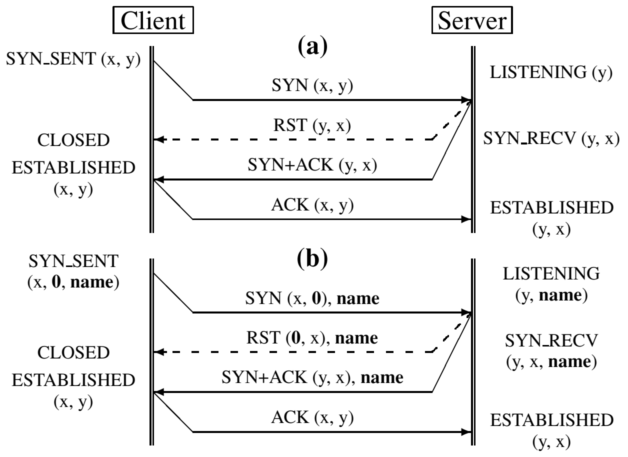

Figure 1: Standard 3-way handshake, using port numbers x and y (a) and extended 3-way handshake using a server port name (b). The slashed line represents an alternative server reply (RST segment) when the connection to port y or with the given name is rejected.

For applications binding only names to ports, and not fixed numbers, the TCP layer can allocate random port numbers on a per-request basis. The benefits of this protocol decoupling from fixed port numbers are (i) harder traffic eavesdropping and (ii) increased number of concurrent connections, as in [12].

Port names are resolved as soon as possible to allow clients and servers to use port numbers in the normal TCP stream exchanges. Consequently, we integrated the name resolution in the TCP synchronization phase, where clients and servers exchange initial sequence numbers and TCP options (see Fig. 1). The port name is specified in the SYN request, the name→number resolution is given in the subsequent SYN+ACK segment. Thereafter, both peers will always use the server port number in all segments exchanged in the TCP stream. Stateful firewalls should have no problems managing this, if properly updated to keep track of port number/name pairs.

The name resolution works as follows. The server host, when receiving a SYN with a port name, ignores the destination port number and looks for a socket in the LISTEN state bound to that port name. If such a socket is found, a SYN+ACK is sent to the client containing the name resolution, i.e. port name and number. Otherwise, a RST packet is sent with the unresolved port name.

3.2 Backward compatibility

Port name resolutions, expressed in SYN requests, should carry a null server port number to force a RST reply from standard TCP stacks. Though the TCP standard does not refer that 0 is an invalid port number, in practice it is not used; therefore, it can be used for differentiating old TCP stacks from new ones implementing port names. Protocol scrubbers [6] should be updated to include TCP destination port 0 as valid when port name option is present.

Port name resolutions are only provided to clients that request a connection to a TCP endpoint with a port name. Clients using the normal TCP connection request, i.e. using a server port number, do not get any name resolution when the port number is actually associated with a name. Likewise, they do not receive a port name in a RST reply. This is done for two reasons. First, the client didn't request a name resolution, so it should not get one. Second, for backward compatibility with current TCP stacks, clients using the actual number-based port addressing should observe the standard TCP behavior from servers (described in Fig. 1).

3.3 Port names in TCP segments

As above described, port names are only used in the TCP synchronization phase, therefore in SYN, SYN+ACK and RST segments. Thus, we have to conceive a way of adding arbitrary names to these segments and to signal that they should be used.

As referred in §2, adding a name to a TCP header is not a suitable solution, because it severely limits the length of port names. Instead, we decided to use the TCP payload to store the port name and to use a new TCP option to signal the presence and the length of the port name in the beginning of the segment payload.

TCP synchronization segments allow clients and servers to exchange data. Though this is not supported by the Berkeley sockets API, it is allowed by the XTI (X/Open Transport Interface) and is a correct behavior according to the standard [9]. RST segments usually do not carry any data in their payload but a standard amendment [2, §4.2.2.12] defines that they may carry a reason message in their payload. Thus, using the payload of synchronization and reset segments for exchanging port names is correct, according to the standards, though such data should not be delivered to upper layers.

As port names are transmitted in the segments' payload, they have a direct effect in the management of the stream sequence numbers. The sequence and acknowledge numbers exist to control the data stream between peers, guaranteeing byte order and resilience to data loss.

We decided to keep the semantics of sequence numbers during the synchronization using port names, i.e. names are seen as data exchanged in payloads (see table below), but they are removed from the data that is provided to upper protocol layers. This does not raise any coherence problem, since upper layers are not aware of sequence numbers. In other words, upper layers do not notice that the amount of data received in segments' payload is not the same amount of data they actually get.

Consequently, whenever a name resolution is required in a SYN segment, its reply, SYN+ACK or RST, acknowledges a sequence number equal to the client's ISN plus the port name length plus one. Similarly, the client's ACK will contain a sequence number equal to its ISN plus the server's port name plus one:

	TCP standard		TCP with port names	
Segment	seq	ack	seq	ack
SYN	ISN_c	0	ISN_c	0
RST	0	$\text{ISN}_c + 1$	0	$\text{ISN}_c + 1 + L$
SYN+ACK	ISN_s	$\text{ISN}_c + 1$	ISN_s	$\text{ISN}_c + 1 + L$
ACK	$\text{ISN}_c + 1$	$\text{ISN}_s + 1$	$\text{ISN}_c + 1 + L$	$\text{ISN}_s + 1$

where L is the length of the port name.

3.4 Caching of name resolutions

Some name services' clients maintain caches of name resolutions, which is common in DNS but not in RPC. We decided not to maintain caches in our TCP port name resolutions. Two main reasons justify our decision. First, name resolutions are piggybacked in the first two segments of a TCP synchronization, thus the overhead of name resolution is too reduced to justify the existence and management of caches in clients for increasing performance. Second, stalled cached resolutions can lead to wrong TCP connections that only applications can possibly detect, but not the TCP layer.

3.5 Managing port access restrictions

The TCP name service prevents services with unusual names to be discovered by port scanning tools, hiding them from people or tools that wish to exploit and not really use them. But in §3.1 we saw that port names may be associated to fixed port numbers. Thus, to enforce the name-based access control we need to disallow clients to connect server ports using only their number. So, servers must specify, when binding, if (i) port-based connections are permitted or (ii) only name-based connections are allowed. The latter allows the TCP layer to use random numbers for the server port on each connection request.

3.6 Denial of Service (DoS) issues

Name resolutions in servers require more time than simple port number lookups. However, the extra workload required to resolve port names does not prevent other clients from accessing the server or even local server applications to run. Thus, though name resolution flooding attacks may slow down servers, by itself they do not create a clear and well defined DoS scenario.

4 Implementation

Our TCP name service was implemented in a recent Linux kernel (2.6.22.9), using a Fedora 7 kernel source.

In Linux there are two separate implementations of TCP, one for IPv4 and another for IPv6. Since our implementation is mainly a proof of concept, we only updated the TCP version for IPv4. Nevertheless, we took into consideration some IPv6 issues, as for the naming of TCP endpoints, described in §4.7.

4.1 Socket related structures

Port names, either bound to local ports or to be resolved within connection handshakes, are stored in dynamically allocated, kernel space memory areas. Structures storing port numbers, `inet_sock` and `tcp_sock`, were updated to include an optional port name; the latter was also updated to include port access restrictions.

Some auxiliary structures, `tcp_request_sock` and `tcp_options_received`, were also enriched to maintain the length of the port name. This value simplifies the calculation of sequence numbers in TCP segments containing port names and helps locating the socket structure in hashed lists (see §4.3).

4.2 Name→number mappings

Since port names are just a step towards port numbers, some mapping table converting names to numbers must exist. This mapping only applies for local ports, as no caching of remote name-number bindings is required. We implemented this mapping by extending the `inet_hashinfo` structure to include an hashed variable based on a new structure named `portname_hashbucket`:

```
struct portname_hashbucket {
  spinlock_t         lock;
  struct list_head   list;
  unsigned short     port;
  char               * portname;
  unsigned short int portname_len;
};
```

An element is first added to this hashed variable when a socket is bound to a port name, in *inet_bind*. This is implemented by a new function, *inet_bind_hash_portname*, which extends the functionality of *inet_bind_hash*. An element is removed from the hashed variable when the socket referring it is eliminated in *inet_unhash*.

This variable is used for name lookup in two distinct occasions: (i) when binding a name to a port, in *inet_csk_get_port*, to check if it is not already being used, and (ii) in *__inet_lookup_listener*, to get the port number of a listening socket upon receiving a SYN with a port name.

4.3 Socket hashed lists

Linux implements several hashed lists to index sockets. One of them is *listening_hash*, containing `INET_LHTABLE_SIZE` lists of sockets involved in TCP connections. The actual list of a socket is given by the functions *inet_ehashfn* and *inet_sk_ehashfn*, which produce an integer from the source/destination addresses and port numbers. Thus, both local and remote port numbers are crucial for indexing sockets in *listening_hash*.

However, clients using name-based connections raise a problem: they don't have a remote port number when adding a socket in the SYN_SENT state, i.e. after sending a SYN segment with a port name. Thus, for these client sockets there is a temporary hashing within *listening_hash* until getting the name resolution. This temporary hashing applies only to sockets in the SYN_SENT state; after getting a correct SYN+ACK segment with the required name resolution, the remote port is used to relocate the socket, now in the ESTABLISHED state.

For the temporary hashing we used the same functions and replaced the server port number by the port name length. We could as well have used a null server port number but using the name length is more likely to improve, with no extra costs, the spreading of sockets (only in the SYN_SENT state) among the hashed lists when many port names are used.

4.4 Defining port access restrictions

Implementing TCP port access restrictions is accomplished by setting a new, TCP level socket option. We named this option `TCP_BIND_PORTNAME` and gave it the value 15. There are three different listening modes that can be set through this option: (**0**) port number only (current standard); (**1**) port name only (legacy connection requests are not allowed); and (**2**) port number and port name: both legacy and name-based SYN requests are accepted. This option is checked by *tcp_v4_rcv* function, upon the arrival of a SYN segment.

4.5 Port names in TCP segments

When a client application issues a connection request using a port name, the local TCP stack copies the port name from the `sin_portname` field of the provided `sockaddr_in_named` structure and updates internal variables as needed. The kernel will then send a SYN packet with the port name in the payload and a TCP option indicating the length of the port name, which corresponds to the given `sin_portname_len`. We used the value `0x45` for the new TCP option, which uses a 16-bit integer to communicate the port name length.

As the Linux TCP does not handle user data in synchronization segments, no modifications were required to prevent port names exchanged in SYN and SYN+ACK segments to be delivered as normal data to applications.

Linux has one special socket *tcp_socket* that is only used for sending RST segments by the function *tcp_v4_send_reset*. This function and *ip_send_reply*, called to generate the actual RST IP datagram, were extended to process further parameters, namely port names, and to handle port names in TCP segments.

4.6 IP fragmentation

Linux sets the Don't Fragment IP flag in TCP segments not containing payload, such as SYN and SYN+ACK, which is a correct behavior [8] since their packet length is bellow the fragmentation threshold of 68 bytes. But with our port name resolution their payload contains a (possibly large) port name, thus fragmentation must be allowed in those cases. The function *tcp_transmit_skb* was modified, when calling *ip_queue_xmit*, to allow IP fragmentation for this synchronization segments.

4.7 Using TCP port names

For binding names to TCP ports and to express port names when connecting to them, we created two new structures by extending `sockaddr_in` and `sockaddr_in6` structures, for IPv4 and IPv6, respectively. The new types were created by adding two extra fields: a pointer to the port name and the port name length, see below.

```
struct sockaddr_in_named {
  sa_family_t    sin_family;       /* AF_INET_NAMED */
  in_port_t      sin_port;         /* port number   */
  struct in_addr sin_addr;         /* IP address    */
  unsigned char  sin_zero[2];
  uint16_t       sin_portname_len; /* Port name len */
  char __user    *sin_portname;    /* Port name     */
};
```

Furthermore, we created two new address families for using with these new structures: `AF_INET_NAMED` for IPv4 and `AF_INET6_NAMED` for IPv6. At kernel level, these new address families are used solely for identifying the type of naming structures provided by client applications; for all other family tagging requirements, it uses the families `AF_INET` and `AF_INET6`.

The function *inet_bind* was extended to support binding to a local port name. Similarly, the functions *tcp_connect* and *tcp_v4_connect* were extended to support the connection to named ports.

5 Experience

For evaluating the features described in this proposal, we implemented some basic TCP client/server programs using the new `sockaddr_in_named` structure and ran them in a server with our modified Linux kernel. We also patched some well known applications, such as Apache2 web server and `netcat`. Apache2 was partially patched to bind to a specific port name; `netcat` was used as a basis to build a command line application supporting both port number and port name bindings.

All possible combinations of server port name bind modes with client connection methods were tested successfully within and between machines with the standard and enhanced TCP stacks, as shown in the table below. Clients with old TCP stacks can only connect to port numbers (cases 3 and 4) and servers with old TCP stacks can only handle connection requests including a valid port number (cases 2 and 4). Clients with new TCP stack (cases 1 and 2) can either connect by port number or name but the latest will only be fully understood by the new TCP port name aware stack (case 1).

		Server-side stack	
		new	old
Client-side stack	new	$\surd^{1}$	$\surd^{2}$
	old	$\surd^{3}$	$\surd^{4}$

6 Conclusions

In this paper we presented a simple name service for TCP ports. This name service allows TCP clients to identify services by port name, instead of port number, which is more user-friendly. The name service extends TCP synchronization segments, conforms with the TCP standard and is compatible with existing TCP implementations. Resolution of port names requires additional processing but does not create an opportunity for DoS attacks.

A security advantage of TCP port naming is that it allows services with unusual names, known only by small communities, to remain undetected by port scanners.

Our prototype implementation confirmed the compatibility with other TCP implementations. Furthermore, we were able to maintain compatibility with legacy systems, kernels and applications.

References

[1] BARHAM, P., HAND, S., ISAACS, R., JARDETZKY, P., MORTIER, R., AND ROSCOE, T. Techniques for Lightweight Concealment and Authentication in IP Networks. Tech. Rep. IRB-TR-02-009, Intel Research Berkeley, 2002.

[2] BRADEN, R. Requirements for Internet Hosts – Communication Layers. RFC 1122, IETF, Oct. 1989.

[3] GULBRANDSEN, A., VIXIE, P., AND ESIBOV, L. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, IETF, Feb. 2000.

[4] KRZYWINSKI, M. Port Knocking: Network Authentication Across Closed Ports. *SysAdmin Magazine*, 12 (2003), 12–17.

[5] LOTTOR, M. TCP port service Multiplexer (TCPMUX). RFC 1078, IETF, Nov. 1988.

[6] MALAN, G. R., WATSON, D., JAHANIAN, F., AND HOWELL, P. Transport and application protocol scrubbing. In *INFOCOM (3)* (2000), pp. 1381–1390.

[7] MOCKAPETRIS, P. Domain names – implementation and specification. RFC 1035, IETF, Nov. 1987.

[8] POSTEL, J. Internet Protocol. RFC 791, IETF, Sept. 1981.

[9] POSTEL, J. Transmission Control Protocol. RFC 793, IETF, Sept. 1981.

[10] REYNOLDS, J. Assigned Numbers: RFC 1700 is Replaced by an On-line Database. RFC 3232, IETF, Jan. 2002.

[11] REYNOLDS, J., AND POSTEL, J. Assigned Numbers. RFC 1700, IETF, Oct. 1994.

[12] TOUCH, J. A TCP Option for Port Names. Internet draft (expired), IETF, Apr. 2006.