Socket Emulation over a Publish/Subscribe Network Architecture

George XYLOMENOS, Blerim CICI
Athens University of Economics and Business, Patision 76, Athens, 104 34, Greece
Tel: + 30 210 8203693, Fax: + 30 210 8203686
Email: xgeorge@aueb.gr, blerim153@gmail.com

***Abstract*— The current drive towards new networking paradigms that make information the center of the network architecture, cannot ignore the fact that any new architecture will have to co-exist with the existing Internet for an extended period of time. In order for a new architecture to be globally deployed, it must ensure that existing applications will continue to operate, preferably without the need to even recompile them. As part of the Publish Subscribe Internet Routing Paradigm (PSIRP) project, we have explored the options for making existing applications based on the ubiquitous Sockets Application Programming Interface (API) compatible with the PSIRP prototype implementation, which is based on publish/subscribe principles throughout the protocol stack. We describe an emulator which mediates between the client/server socket calls and the publish/subscribe calls implemented by the PSIRP prototype. Our socket emulator allows existing applications, either TCP or UDP based, to run unmodified over an information centric network architecture that is radically different from the endpoint centric Internet architecture for which they were designed.**

***Index Terms*— TCP/IP, Sockets, Publish/Subscribe, PSIRP.**

I. INTRODUCTION

A large fraction of current Internet traffic is due to peer to peer content distribution applications [1], in which participants are solely interested in the exchanged data rather than in the endpoint addresses of their peers, indicating that the Internet is evolving from a network connecting pairs of end hosts to a substrate for information dissemination. There are many proposals for evolving or redesigning the Internet architecture based on an information centric paradigm, for example, the *Content Centric Networking* (CCN) [2] project and the *Publish/Subscribe Internet Routing Paradigm* (PSIRP) project [3]. The PSIRP project in particular is working on a network architecture based entirely on publish/subscribe principles, and its prototype implementation employs publish/subscribe concepts throughout the protocol stack [4]. In the publish/subscribe model, publishers announce available data, subscribers express their interests, and the network allows them to rendezvous for the exchange of data.

Any proposal that seeks to radically change the architecture of the Internet must plan to co-exist with the existing Internet for an extended period of time. In particular, in order to be deployed, a new architecture must ensure that it will be possible to execute existing applications on top of it. While many applications, especially content distribution ones, can reasonably be expected to be rewritten so as to operate optimally over an information centric architecture, there is a vast number of existing, endpoint centric, applications that will have to operate in some type of compatibility mode, preferably without the need to even recompile them. Since most existing Internet applications were written on top of the widespread Sockets *Application Programming Interface* (API) [5], the most direct way to make them compatible with a new architecture is to develop middleware to transparently translate Socket API calls to the underlying information centric calls offered by the new architecture.

In this paper we describe the design and implementation of a Socket API emulator for PSIRP, which allows unmodified Internet applications to operate on top of a native publish/subscribe protocol stack. In Section 2 we introduce the basic concepts of the PSIRP architecture and implementation. In Section 3 we discuss the different emulation options available and motivate our selection. In Section 4 we explain how IP addresses and socket calls are translated into PSIRP calls. Finally, in Section 5 we summarize our work and discuss future development directions.

II. PSIRP IMPLEMENTATION CONCEPTS

In the PSIRP prototype implementation, which is based on the FreeBSD operating system, publications are handled via a set of calls encapsulated in the `libpsirp` library [4]. In order for the reader to understand how publish/subscribe communication is achieved and appreciate the intricacies of this programming model, in this section we provide an introduction to the `libpsirp` concepts and calls. In the PSIRP architecture, the central entity is a *publication* which is made available by *publishers* to *subscribers*. The network provides mechanisms for publishers and subscribers to rendezvous in order for publications to be transported from the former to the latter. A publication is identified by a *Scope Identifier* (SId) and a *Rendezvous Identifier* (RId) [3]; the SId represents an information collection, while the RId represents an information item within this collection. For example, a user may publish a set of holiday pictures, each identified by an RId, within a scope representing his friends, identified by an SId. Note that there are no global endpoint identifiers whatsoever in PSIRP.

Publications consist of data and metadata; data are transparently mapped to the memory space of the publishers and subscribers of a publication. A publisher creates a new publication by calling `int psirp_create(int size, psirp_pub_t *pub)`. This allocates a memory area of `size` bytes for the publication data, initializes a data structure for the publication and returns a handle of type `psirp_pub_t` to this structure. The publisher can call `caddr_t psirp_pub_data(psirp_pub_t pub)` to get a pointer to the memory block of the publication. When the publication is ready, it is made available to the kernel via the `int psirp_publish(psirp_id_t *sid, psirp_id_t *rid, psirp_pub_t pub)` call, which takes as parameters two structures of type `psirp_id_t` holding the desired SId and RId for the publication, as well as a handle to the publication. The kernel can then decide where to forward the publication to. If a publication with the same SId/RId already exists, then the new publication is assumed to be a new version of the original publication, therefore its version number is increased. The `int psirp_atoid(psirp_id_t *rid, const char *str)` call converts an SId or RId represented as an ASCII string to the internal format used in the `psirp_id_t` structures.

In order to subscribe to a publication with a specified SId/RId pair, a subscriber must call `int psirp_subscribe_sync(psirp_id_t *sid, psirp_id_t *rid, psirp_pub_t *pub, struct timeval *timeout)`. This blocks the subscriber for at most `timeout` (forever if `timeout` is 0); if a matching publication is found before the interval expires, a handle is returned to the latest version of the publication. The caller can distinguish new from old versions of a publication by asking for their version numbers via the `int psirp_pub_version_count(psirp_pub_t pub)` call. In order to retrieve previous versions of a publication, the subscriber must first call `int psirp_subscribe_versions(psirp_pub_t pub, psirp_pub_t *versions, int start_index, int max_count)`, which returns in the `versions` array up to `max_count` handles to earlier versions of publication `pub` starting from version `start_index`. The publication data can be accessed in the same manner as in the publisher side, that is, via the `caddr_t psirp_pub_data(psirp_pub_t pub)` call. Finally, the `void psirp_free(psirp_pub_t pub)` call frees the publication structure and unmaps the memory allocated for the publication.

III. Emulation options

In the Sockets API, a socket represents a communication endpoint, identified by an IP address and a TCP/UDP port. Communication takes place by having each application attach to a local socket and then perform calls on these sockets. The actual communication between sockets is achieved by exploiting the services of the TCP/UDP protocols, transparently to the applications. As shown in Figure 1.(a), the socket uses either TCP or UDP at the transport layer, the transport layer uses IP at the network layer, and IP uses some kind of link and physical layer protocols (such as Ethernet) for data transmission [5]. In contrast, in the PSIRP prototype, publish/subscribe applications talk to `libpsirp` which implements its own transport and network layer protocols directly on top of the physical and link layers, providing an entirely different publish/subscribe oriented API. The goal of the Sockets API emulator is therefore to translate between socket calls and `libpsirp` calls, despite their entirely different approaches.

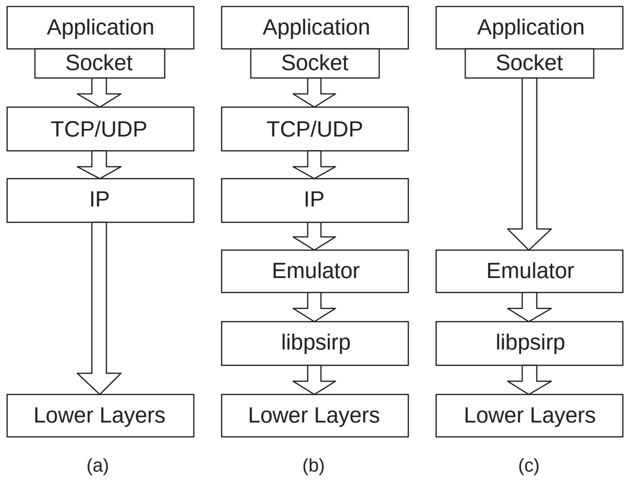

Fig. 1. Socket emulator structure: (a) standard TCP/IP stack, (b) network level emulation, (c) transport level emulation.

One approach, shown in Figure 1.(b) is to exploit an existing TCP/UDP/IP implementation to transform the socket calls to IP packets, and then exchange these packets via `libpsirp` calls. The advantage of this approach is that the emulator only has to provide a best effort service, analogous to that offered by IP. For TCP in particular, flow, congestion and error control are essentially provided by TCP, and the emulator only sees ready to transport IP packets. For UDP the benefits are not so clear, since the corresponding socket calls essentially map directly to UDP and IP packets. The disadvantage is that by treating PSIRP as a dumb transport, not only do we lose all the advantages of its redesigned architecture, but we also apply the IP specific TCP assumptions to an entirely different architecture. A similar approach has been found to be very detrimental for the performance of TCP applications on top of ATM networks [6]. In addition, going through the TCP/UDP/IP implementation represents a significant communication overhead for the emulator.

The other approach, shown in Figure 1.(c) is to translate each socket call directly to `libpsirp` calls. While this is roughly the same as above for UDP, for TCP it is considerably harder, as the emulator needs to provide semantics equivalent to those of TCP to the applications. For example, it has to deal with connection establishment and termination, as well as with flow, congestion and error control. However, in addition to avoiding TCP/UDP/IP overhead, in this manner the emulator can take full advantage of the facilities provided by `libpsirp`. For example, if the PSIRP prototype provided a reliable congestion controlled transport layer service for publish/subscribe networks, this could be used as a substitute for TCP. Despite the additional complexity, this approach will provide better performance in the long term, as it can take full advantage of the underlying publish/subscribe transport and

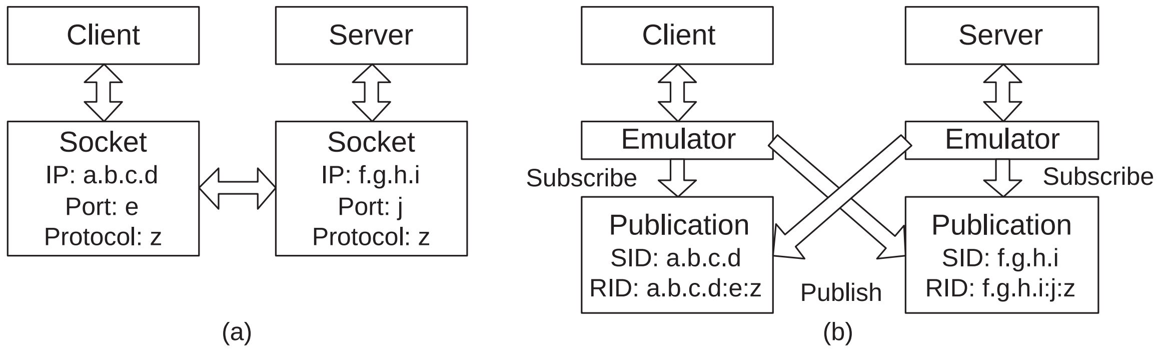

Fig. 2. Address translation: (a) standard TCP/IP socket, (b) emulated socket.

network layer implementations, therefore we have selected it for the emulator.

IV. Emulator Implementation

A. Mapping Addresses to Identifiers

Since PSIRP does not have a notion of endpoint identifiers, the Sockets API emulator needs a direct way of translating the TCP/UDP/IP addresses used by sockets to the SId/RId pairs used to exchange publications in PSIRP. The scheme that we have implemented is to create an SId for each machine based on its IP address and an RId for each socket in that machine by combining its IP address, its port number and the corresponding protocol (TCP or UDP). Therefore publishing to an SId translates to sending data to a machine, while publishing to an RId translates to sending data to a TCP or UDP port of that machine.

A client can establish communication with a server via the socket emulator based only on knowledge of the server's endpoint details, exactly as in the TCP/UDP/IP protocol suite. Say that a client with an IP address of `a.b.c.d` uses port `e` of protocol `z` to communicate with a server with an IP address of `f.g.h.i` using port `j` of protocol `z`, as shown in Figure 2.(a). The emulator translates the client to server messages to publications to the SId generated by `f.g.h.i` (the server's IP address) and the RId generated by `f.g.h.i:j:z`; the client publishes and the server subscribes to this SId/RId pair. In the server to client direction, messages are translated to publications to the SId generated by `a.b.c.d` (the client's IP address) and the RId generated by `a.b.c.d:e:z`; the server publishes and the client subscribes to this SId/RId pair. This arrangement is shown in Figure 2.(b). New messages sent in the same direction are represented as new versions of the same publication, thus allowing a sequence of packets to be transmitted in both directions. The SId/RId pair is directly generated from the above ASCII strings via the `int psirp_atoid()` `libpsirp` call.

B. Datagram Socket Calls

Sockets come in two varieties: Datagram sockets, implemented on top of UDP, and Stream sockets, implemented on top of TCP. In this subsection we explain how Datagram socket calls are emulated, while the next one deals with Stream socket calls. In Datagram sockets, each endpoint uses `socket()` to create a communication endpoint and `bind()` to assign an address to it, consisting of an IP address and a UDP port. Then, messages can be sent to another socket via `sendto()` which takes as parameters the data to send and the address of the remote socket. Messages are received via `recvfrom()` which returns the data received and the address of the remote socket.

Figure 3 shows how Datagram calls are emulated; the dotted arrows show how Socket calls are mapped to emulator actions, while the solid arrows show the publications exchanged between machines. The server first calls `socket()` to create a data structure for its communication endpoint and get a handle to it for later use; this translates to the creation of an equivalent data structure in the emulator. In order for the socket to become accessible to clients, the server calls `bind()` to assign an IP address and a UDP port to the socket; the emulator uses this information to calculate an SId/RId pair for incoming data and stores both the socket address and the PSIRP identifiers in its own structure. The client performs the exact same calls before communication.

In order to receive data via the socket, the server issues the `recvfrom()` call on the socket, which is translated by the emulator to a `psirp_subscribe_sync()` call on its incoming SId/RId pair. In order to distinguish consecutive packets, the emulator ensures that each `recvfrom()` call returns the next version of the same publication; the last version number seen is stored in the socket structure. Each publication contains in its metadata field the IP address and UDP port stored in the socket from which the message was sent. The emulator passes these data to the server via the return parameters of the `recvfrom()` call, so that the server may later use them to send replies to the client. The `sendto()` call is translated by the emulator to a `psirp_publish()` call on the outgoing SId/RId pair generated by the IP address and UDP port provided by the caller in the socket call. In addition, the IP address and UDP port stored in the socket structure of the sender are inserted as metadata in the publication, in order for the receiver to be able to reply, as explained above. The behavior of the client is completely symmetric; the only difference is that the client must know in advance the IP address and UDP port of the server, so that it may issue the first `sendto()` call.

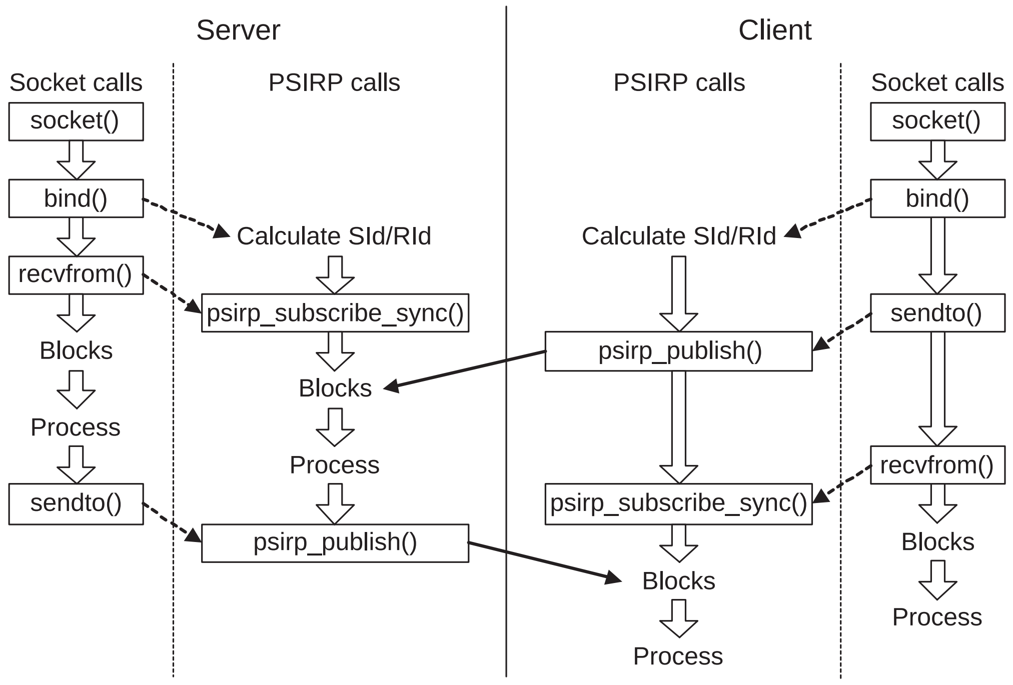

Fig. 3. Datagram socket calls

C. *Stream Socket Calls*

In Stream sockets, each endpoint also uses `socket()` to create a communication endpoint and `bind()` to assign a local address and port to it; clients may omit `bind()`, letting the system select a local address and port. The server then calls `listen()` to create a queue to store incoming connection requests while a previous connection is being handled. Finally, `accept()` is called to wait for an incoming connection on the server side; when it returns, it points to a new socket connected to a client, that is, a socket containing both local and remote endpoint addresses. The client on the other hand calls `connect()` with the address of a remote server, in order to connect to that server; when it returns, the existing socket is connected to the server. Data can then be sent by calling `send()`, while the `recv()` call returns received data; these calls have no need for endpoint addresses, as these are stored in the sockets.

Figure 4 shows how Stream calls are emulated. The `socket()` and `bind()` calls (optional on the client side) operate exactly as in the Datagram case, leading to the calculation of an SId/RId pair for incoming publications at each endpoint. Only the structure created in the emulator is different: a connected Stream socket must store both local and remote endpoint address and SId/RId pairs, while Datagram sockets only need the local part. The `listen()` call is only used for housekeeping: it creates a list for storing incoming connection requests until the emulator can service them.

The main differentiation from a Datagram Socket is that in a Stream socket a new socket needs to be created on the server side when a connection is established, leaving the original socket for additional concurrent connections. In the emulator, when `accept()` is called the server calls `psirp_subscribe_sync()` on its incoming SId/RId pair in order to receive the next connection request. On the client side, when `connect()` is called the emulator first uses the IP address and TCP port passed to that call, which the client knows in advance, to calculate the SId/RId pair of the server and then calls `psirp_publish()` to send it an empty publication, containing as metadata its own IP address and TCP port. Finally, the client calls `psirp_subscribe_sync()` on its incoming SId/RId pair and waits for a reply from the server.

On reception of the client's publication at the server, the emulator creates a new socket structure, using the local endpoint address from the existing socket and the remote endpoint address from the publication metadata. The server calculates the SId and RId for each endpoint as usual, but then it XORs the original local and remote RId and stores the result as its new local RId, which will be used for incoming data. As a result, connected sockets are differentiated in the server from unconnected ones as they use both endpoint addresses to calculate the RId for incoming data, exactly as in a connected TCP socket. Finally, the server calls `psirp_publish()` to send an empty publication to the client's incoming SId/RId pair. When this publication is received by the client, the client's socket structure is also updated by calculating the new incoming SId/RId pair of the server as above and the `connect()` call returns.

At this point connection establishment is complete, and either side can use the `send()` and `recv()` calls to send and receive data, respectively, as in the Datagram socket case. Due to the modified server RId used for connected sockets, there is no confusion between publications to connected sockets, which represent user data, and publications to unconnected sockets, which represent connection establishment packets.

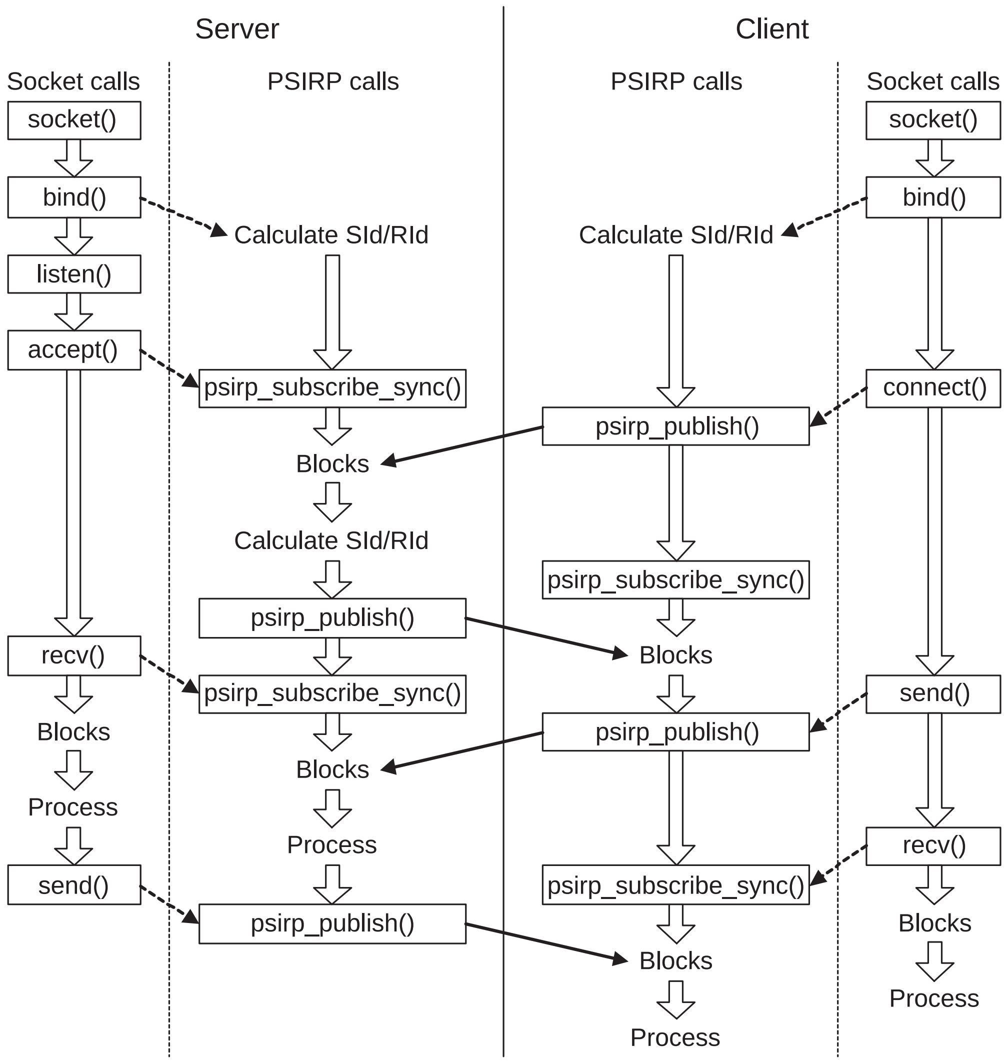

Fig. 4. Stream socket calls.

V. Conclusions and Future Work

We have presented the design and implementation of a Sockets API emulator for the publish/subscribe oriented prototype implementation of the PSIRP architecture. This emulator translates the socket calls used by existing Internet applications into the calls provided by the `libpsirp` library of the PSIRP prototype implementation. While the implementation details are specific to PSIRP, the emulation options and, to some extent, the emulator design, are relevant to other information centric architectures, where content rather than endpoints is at the center of the network architecture.

Regarding future work, many emulator features are still work in progress. For example, the `close()` calls have not yet been implemented for Stream sockets, therefore there is no connection release signaling, and handling of concurrent connection requests is still incomplete. Most importantly, the emulator inherits some of the limitations of the PSIRP prototype, therefore it will have to be further extended following the prototype itself. For example, in the current prototype only a UDP like service is provided for publication delivery, with the emulator using it for both Datagram and Stream sockets; as the publication transport services are further developed, the emulator will need to be modified so as to match TCP semantics as closely as possible.

References

[1] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should Internet service providers fear peer-assisted content distribution?," in *Proc. of the Internet Measurement Conference (IMC)*, pp. 63–76, 2005.
[2] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and N. Braynard, "Networking Named Content," in *Proc. of the ACM Int. Conference on emerging Networking EXperiments and Technologies ACM (CoNEXT)*, pp. 1–12, 2009.
[3] N. Fotiou, G. Polyzos, and D. Trossen, "Illustrating a publish-subscribe Internet architecture," in *Proc. of the 2nd Euro-NF Workshop on Future Internet Architectures*, June 2009.
[4] P. Jokela and J. Tuonnonen, "Progress report and evaluation of implemented upper and lower layer." PSIRP Deliverable 3.3, June 2009.
[5] W. Stevens, *UNIX Network Programming: Networking APIs*, vol. 1. Prentice Hall, second ed., 1998.
[6] D. Comer and J. Lin, "TCP buffering and performance over an ATM network," *Internetworking: Research and Expreinece*, vol. 6, pp. 1–13, March 1995.