Design and Evaluation of a Socket Emulator for Publish/Subscribe Networks

George Xylomenos, Blerim Cici
Mobile Multimedia Laboratory & Department of Informatics
Athens University of Economics and Business, Patision 76, Athens, 104 34, Greece
Email: xgeorge@aueb.gr, blerim153@gmail.com

***Abstract*—In order for a Future Internet architecture to be globally deployed, it must ensure that existing applications will continue to operate efficiently on top of it. As part of the Publish Subscribe Internet Routing Paradigm (PSIRP) project, we have explored various options for making endpoint centric applications based on the Sockets Application Programming Interface (API) compatible with the information centric PSIRP prototype implementation. We developed an emulator that mediates between the client/server socket calls and the publish/subscribe PSIRP calls, transforming the exchange of packets to distribution of publications. To assess the overhead of our emulator, we measure the execution time of a simple file transfer application in native socket mode, in emulated socket mode and in native publish/subscribe mode.**

***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 [5], in which participants are solely interested in the exchanged data rather than in the endpoint addresses of their peers. This indicates 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) [3] project and the *Publish/Subscribe Internet Routing Paradigm* (PSIRP) project [2]. 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.

In order to be deployed, an information centric architecture must ensure that it will be possible to efficiently execute existing applications on top of it. While content distribution applications may be expected to be rewritten so as to operate optimally over an information centric architecture, a vast number of existing endpoint centric applications will have to operate in some kind of compatibility mode. Since most existing Internet applications were written on top of the Sockets *Application Programming Interface* (API) [7], the most direct way to make them compatible with a new architecture is to develop middleware to translate Socket API calls to the information centric calls of the new architecture.

In this paper we describe and evaluate 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. In Section 5 we evaluate our emulator by comparing the performance of an application in native socket mode, in emulated socket mode and in native publish/subscribe mode. Finally, in Section 6 we summarize our work.

Work supported by the ICT PSIRP project under contract ICT-2007-216173.

II. PSIRP Implementation Concepts

In the PSIRP prototype implementation, which uses the FreeBSD operating system, publications are handled via a set of calls encapsulated in the `libpsirp` library [4]. To understand how publish/subscribe communication is achieved, 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) [2]; 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.

Publications consist of data and metadata; data are mapped to the memory space of the publishers and subscribers. A publisher creates a new publication via `psirp_create()`. This allocates a memory area of a specified size for the publication data, initializes a data structure for the publication and returns a handle to this structure. The publisher can call `psirp_pub_data()` using that handle to get a pointer to the memory block of the publication. When the publication is ready, it is passed to the kernel via `psirp_publish()`, which takes as parameters the desired SId and RId for the publication, as well as a handle to it. The kernel can then decide where to forward the publication to. If a publication

with the same SId/RId already exists, the new publication is assumed to be a new version, therefore its version number is increased. A SId or RId in ASCII format is converted to the internal `libpsirp` format by `psirp_atoid()`.

A subscriber calls `psirp_subscribe_sync()` with a specific SId/RId pair to subscribe to a publication. This blocks the subscriber until a matching publication is found or until a timeout expires; in the former case, 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 `psirp_pub_version_count()`. To retrieve previous versions, the subscriber must call `psirp_subscribe_versions()`, which returns an array of handles to earlier versions of a specified publication. Finally, `psirp_free()` 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 perform calls on it. The actual communication between sockets is achieved by exploiting the services of the TCP/UDP protocols. 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 lower layer protocol (such as Ethernet) for data transmission [7]. In contrast, in the PSIRP prototype, publish/subscribe applications talk to `libpsirp` which implements its own transport and network layer protocols on top of the lower layers. The goal of the Sockets API emulator is therefore to translate between socket calls and `libpsirp` calls, despite their different approaches.

One emulation 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 IP packets. The disadvantage is that by treating PSIRP as a dumb transport, not only do we lose the advantages of its redesigned architecture, we also apply 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 [1]. 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 deal 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 transport service for publish/subscribe networks, this transport could be used instead of TCP. Despite the additional complexity, this approach will provide better performance in the long term, therefore we have selected it for the emulator.

IV. Emulator Implementation

A. Mapping Addresses to Identifiers

Since there is no notion of endpoint identifiers in PSIRP, the Sockets API emulator must translate the TCP/UDP/IP addresses used by sockets to the SId/RId pairs used to exchange publications in PSIRP. The scheme that we 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 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 port of that machine.

A client can communicate with a server via the socket emulator based only on the server's endpoint details, exactly as in TCP/UDP/IP. 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`. 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`. This arrangement is shown in Figure 2.(b). New messages sent in the same direction are represented by new versions of the same publication, thus allowing sequences of packets to be transmitted.

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. Figure 3 shows how Datagram calls are emulated; dotted arrows show how Socket calls are mapped to emulator actions, while 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, 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. 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

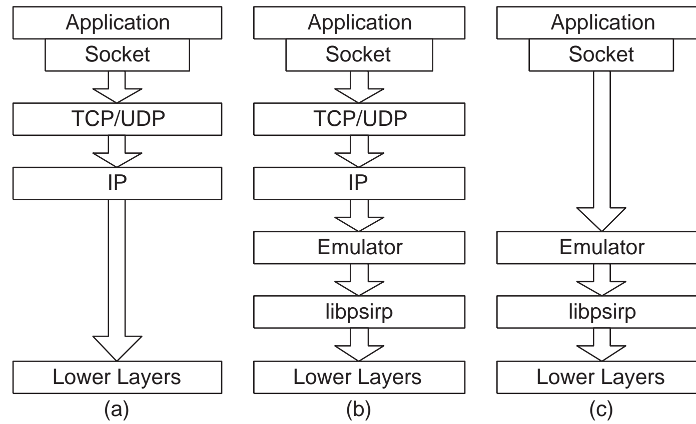

Fig. 1. Socket emulator structure: (a) standard TCP/IP stack, (b) network level emulation, (c) transport level emulation.

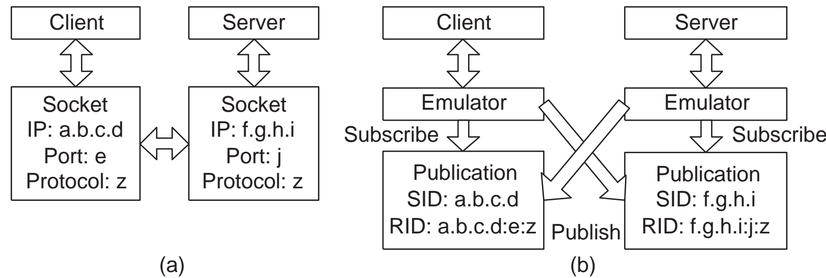

Fig. 2. Address translation: (a) standard TCP/IP socket, (b) emulated socket.

field the IP address and UDP port 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. 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, as explained above. The behavior of the client is symmetric; the only difference is that the client must know in advance the IP address and UDP port of the server to issue the first `sendto()` call.

C. Stream Socket Calls

Figure 4 shows how Stream calls are emulated. The `socket()` and `bind()` calls (the latter is 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, since in Stream sockets data transfer calls do not indicate addresses, unlike in Datagram sockets. 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 however 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 to handle additional connection requests. When `accept()` is called to indicate that the server is ready to receive a new connection request, the emulator 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 to initiate a connection, 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.

When the server receives the client's publication, 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. 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. 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

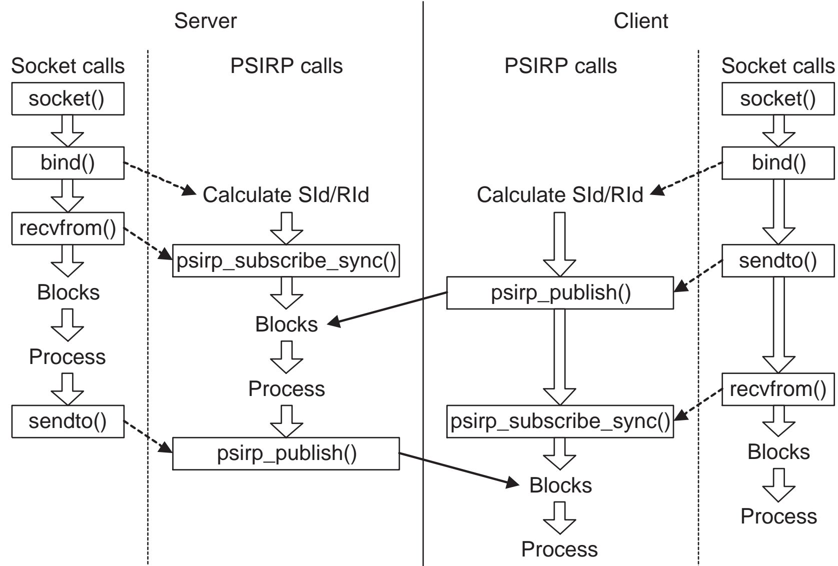

Fig. 3. Datagram socket calls

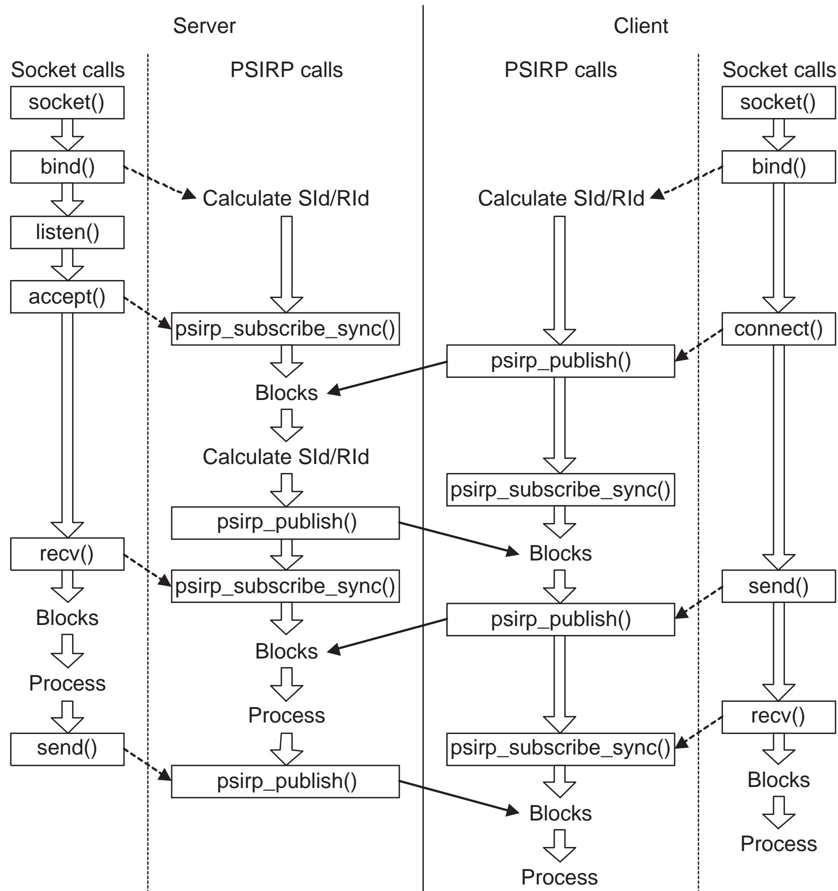

Fig. 4. Stream socket calls.

either side can use the `send()` and `recv()` calls to send and receive data, respectively, without indicating a destination address. Due to the modified server RId used for connected sockets, there is no confusion between publications to connected sockets (data) and unconnected sockets (connection requests).

V. Performance Assessment

In order to assess the overhead of the Socket API emulator, we implemented a *Trivial File Transfer Protocol* (TFTP) [6] application which transfers files using a simple stop and wait protocol over a Datagram socket. This application can execute either over a native UDP/IP implementation or over our emulator. Since the emulator operates over the PSIRP prototype implementation which introduces its own overhead, we also wrote a native `libpsirp` version of the TFTP client and server, by manually replacing socket calls with the corresponding publish and subscribe calls, as explained in the previous section. By executing the same experiments with each TFTP version we can assess the overhead incurred by `libpsirp` when a socket application is ported to it, and the additional overhead incurred by our emulator to run unmodified socket applications.

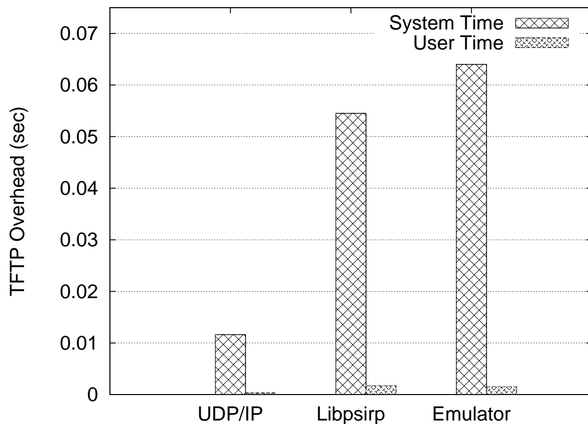

Fig. 5. User and System overhead for each TFTP version.

Due to limitations in the PSIRP prototype, we use the TFTP client and server to transport a small file consisting of 98 data packets with 1 KByte data payloads over an Ethernet. In all versions, a TFTP header is attached to each data packet, and then either UDP/IP headers, PSIRP headers or Socket API emulator metadata and PSIRP headers are added to each packet. We use the `time` command to compute the user space and system space computation time incurred by each version. Due to the coarse (10 ms) granularity of these timers, in each experiment we perform 3 transfers back to back to reduce zero timings; we repeated each experiment 25 times. In Figure 5 we show the average user and system time for the TFTP versions considered: the native socket version, the native `libpsirp` version and the emulated socket version.

Despite the coarse timer granularity, it is clear that the PSIRP prototype is slower than the native UDP/IP stack: even the native `libpsirp` TFTP version is much slower than the UDP/IP version. This is not surprising considering that this is an early prototype, while the UDP/IP stack is being optimized for 20 years. On the other hand, the emulated socket version is only 17% slower than the native `libpsirp` version, indicating that even though the emulator is unaware of the nature of the application, the automated translation between socket and `libpsirp` calls is not very costly. This is very encouraging, as it means that as the PSIRP prototype implementation becomes more optimized, the performance of the emulated socket applications will also improve accordingly.

VI. Conclusions

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. Our preliminary performance evaluation indicates that the performance overhead introduced by the emulator is quite low, on the order of 17%, thus it is a reasonable option for the execution of socket applications over PSIRP.

References

[1] D.E. Comer and J.C. Lin. TCP buffering and performance over an ATM network. *Internetworking: Research and Experience*, 6(1):1–13, March 1995.

[2] N. Fotiou, G.C. Polyzos, and D. Trossen. Illustrating a publish-subscribe Internet architecture. In *Proc. of the 2nd Euro-NF Workshop on Future Internet Architectures*, June 2009.

[3] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N. Briggs, and N. Braynard. Networking Named Content. In *Proc. of the ACM CoNEXT*, pages 1–12, 2009.

[4] P. Jokela and J. Tuonnonen. Progress report and evaluation of implemented upper and lower layer. PSIRP Deliverable 3.3, June 2009.

[5] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet service providers fear peer-assisted content distribution? In *Proc. of the Internet Measurement Conference (IMC)*, pages 63–76, 2005.

[6] K. Sollins. The TFTP protocol (revision 2). RFC 1350, July 1992.

[7] W.R. Stevens. *UNIX Network Programming: Networking APIs*, volume 1. Prentice Hall, second edition, 1998.