Ultra Low Delay Switching for Networked Music Performance

George Baltas and George Xylomenos

Mobile Multimedia Laboratory, Department of Informatics
Athens University of Economics and Business
Athens 10434, Greece
g@gbaltas.com, xgeorge@aueb.gr

***Abstract*—Low latency is essential for videoconferencing applications such as presence and collaboration between remote participants. In modern videoconferencing systems, the Selective Forwarding Unit (SFU) has the role of transparently duplicating and forwarding media streams between participants, hence it must be able to process large volumes of incoming packets at very high rates. SFU performance is heavily affected by the network I/O mechanisms employed to move packets from the Network Interface Card (NIC) to user space, and then move the copies back to the NIC. Traditional mechanisms, such as POSIX sockets, are not designed for high performance networking and prove to be a major bottleneck in such scenarios, by increasing packet latency and undermining the SFU's scalability. In this paper, we present a novel SFU platform which was designed to handle the ultra-low latency requirements of Networked Music Performance (NMP) applications, that is, the collaboration of musicians in real time. We implement a prototype SFU based on POSIX sockets and outline its performance bottlenecks. To overcome them, we turn to the netmap framework for fast packet I/O, which provides direct but safe access to the NIC buffers. We argue that ultra-low latency videoconferencing is a natural application for netmap and thus design and implement a netmap-based SFU.**

***Index Terms*—Networked music performance, selective forwarding unit, delay**

I. Introduction

In *Networked Music Performance* (NMP) musicians located at different places perform together via network connections [1]. To achieve proper synchronization between artists, NMP requires a very low end-to-end delay threshold, hence NMP can be seen as videoconferencing optimized for very low delay. In modern videoconferencing systems, remote clients communicate with each other via real-time media packet streams. On the sending end, each client transmits one or more audio/video streams. Each user has a preferences profile, which marks a subset of the available streams as interesting to the user. On the receiving end, only streams included in the user's profile are received and presented. For example, one might want to hear all participants but only watch one of them. This translates to receiving audio streams from all clients but a single video stream from only one client. In terms of media quality, recent videoconferencing systems employ layered encoding schemes where layer 0 is the base layer and layers 1 to n are enhancement layers. Different quality layers are transmitted independently as separate streams, hence users can request the base layer and any number of consecutive enhancement layers they deem sufficient in terms of quality and acceptable in terms of bandwidth requirements.

In order to implement multiparty conferencing, a communication scheme between multiple clients is needed. The first option to use direct communication; each outgoing stream is sent to all other clients via multicast. This approach was found to reduce latency, but does not scale to multiple participants and requires network protocols that are not currently deployed on the Internet at large [2]. A second option is for each client to unicast each stream to each interested client; this does not require multicast support, but it requires a lot of processing (for replication) and bandwidth (for transmission) resources at the client. A third option is to employ a high performance machine as part of the videoconferencing infrastructure that handles, among other things, client communication. This machine, known in the literature as a *Selective Forwarding Unit* (SFU), is a server that relays data between participants [3]. In this approach, each client sends all of its streams only to the SFU, who is responsible for handling the rest. Assuming that the SFU is aware of client preferences, it can replicate and forward each stream only to the clients interested in it. While this involves stream replication at the SFU, it provides a centralized way to deal with packet replication and forwarding.

In legacy videoconferencing systems, a *Multipoint Conferencing Unit* (MCU) is used instead of the SFU. The MCU mixes the audio streams and transcodes the video streams received by all clients, so as to send a combined audio and video stream to each client. As a result, the MCU requires very large amounts of computational power and introduces significant delays for media processing, due to the need to decode and re-encode all media. In contrast, the SFU simply redirects incoming streams to clients, minimizing the introduced delay, hence it is far more suitable for NMP.

Summarizing the above, the path of a media packet is as follows: A media snippet is captured by the corresponding hardware, encoded in an appropriate format and packetized. Then, the packet is transmitted to the SFU over the network. The SFU performs all required processing to transmit the packet to all interested clients. The clients receiving the packet read, decode and finally present the media snippet to the user. The goal of the Musinet project [4] is to minimize the delays

incurred in each part of that path, from capture and encoding, to replication, reception and playback. In this paper, we focus on the SFU, the centralized network I/O component used to receive and transmit packets between participants.

The remainder of this paper is structured as follows. In Section II we describe a baseline SFU implementation based on POSIX sockets, as well as a dummy client used for SFU testing, and discuss the performance bottlenecks of this approach. We then show in Section III how the netmap framework offers direct but safe access to the network interface, so as to bypass the bottlenecks introduced by sockets. In Section IV we describe our netmap-based SFU implementation and show how it avoids the limitations of the sockets-based alternative. We conclude and discuss future work in Section V.

II. A SOCKET-BASED IMPLEMENTATION

We begin by implementing a socket-based SFU to highlight the delay issues arising due to the particular demands of this application. At the transport level, communication between the clients and the SFU is achieved by sending and receiving UDP packets using two ports: A control port is used for signaling and a data port is used for exchanging media streams. All incoming or outgoing packets use a simplified version of RTP. In our implementation, each packet carries a 32 bit *Synchronization Source Identifier* (SSRC) field in its header. The SSRC uniquely identifies a stream within the session and consists of two parts: a 28 bit *client identifier* (cID) and a 4 bit *stream identifier* (sID). This structure allows us to easily determine the client that transmitted a given packet, a very frequent operation both at the clients and at the SFU.

In order for a client to participate in a session, it must first register with the SFU using the control port. During registration, the client receives a cID from the SFU, which can be used to generate a continuous range of SSRCs by appending an sID field for as many audio and video streams the client wishes to generate. When registering a client, the SFU updates an internal routing structure and initializes the client's preferences. Once registration is complete, the client can start sending control packets to update its preferences or unregister, and data packets for relaying to the other clients. In addition to the SSRC, all packets sent from a client include a timestamp and a flags field showing the packet type (control or data) and the payload type, e.g., video/base layer.

When using POSIX sockets, a packet must make a series of steps before being sent to the NIC for transmission. Initially, the packet payload resides within a user space buffer. For UDP packets, a system call such as `sendto()` is used to pass the payload to the host networking stack, which triggers a rather long and expensive kernel control path. The kernel is responsible for allocating a packet buffer, copying the payload there from user space, routing, building the headers and queuing the packet for transmission. The last step involves invoking the NIC driver and building a device compatible packet representation. On the receive side, the kernel processes incoming packets by allocating storage buffers, queuing the buffers for reception, and then passing the received packet up through the networking stack. During this process the packet is checked for errors, routed, stripped of its headers and appended to the socket receive queue. When the user issues a system call such as `recvfrom()`, the packet is removed from the corresponding socket queue, checked for errors and copied from kernel memory to a user-provided buffer.

A. Client application

In our platform, the client application is responsible for transmitting the user-generated data streams and receiving data streams from other clients. Unlike the SFU, the client application should run on a variety of operating systems using commodity hardware, therefore sockets are a good candidate for the client's network I/O. Our client implementation uses two types of threads: an engine thread and one or more packet generator threads. The engine thread is responsible for sending and receiving packets to/from the SFU. In our current implementation, received packets are logged and dropped; eventually, a real conferencing client will be modified to work with our SFU. The engine thread is also responsible for handling the session management packets exchanged with the SFU. A packet generator thread is responsible for producing a dummy packet stream for simulation purposes. When a packet is produced, the generator thread notifies the engine thread accordingly. One or more generator threads can be running in the same process and their traffic details (packets per second, packet size, and so on) are fully customizable to help performance testing. The client application also maintains counters for RX/TX packets/bytes and measures I/O speed, packet loss and average latency during the last second.

Latency, as measured by the client application, represents the time required for a captured frame to be sent to the SFU, from there to be forwarded to all interested clients and finally to be delivered to the client application. Since clients cannot be expected to use tightly synchronized clocks, especially in wide-area networks, we modified the SFU for testing purposes so that after forwarding a packet to all interested clients, it echoes the packet back to its origin client. When a client receives a packet with its own cID, it measures the difference between the current time and the timestamp in the packet header. Since both timestamps are produced by the same clock, their difference represents the *maximum* end-to-end latency observed for the current packet. This includes the SFU induced latency and can therefore be used as a performance metric for comparing different SFU implementations, assuming that network latency between clients and the SFU is fairly static, as in our controlled laboratory testing environment.

B. SFU application

The socket mechanism provides a high level of abstraction and hides many details involved in receiving and sending packets. Therefore, implementing a socket-based SFU is rather straightforward. When idle, the SFU listens on the control and data ports for incoming packets. Since packets can arrive at both ports, we use `select()` to multiplex these inputs. To assess SFU latency, we focus on how data packets are

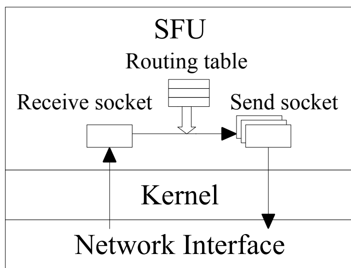

Fig. 1. A Socket-based SFU.

handled. When a data packet arrives, we can read it into a user-space buffer by calling `recvfrom()`; note that we can only read one packet per call and packets are read only in a FIFO manner. Once the packet is in the user-space buffer, we can begin the packet forwarding process. For every client that the packet needs to be sent to, we modify the source and destination address of *the same* user-space buffer (to avoid copying) and pass it to `sendto()`. Again, we can only send one packet at a time. When the packet has been sent to all its destinations, we can start processing the next packet. If none is available, we return to the idle state. An overall view of the application layer SFU is given in Fig. 1.

C. Performance bottlenecks

Even an optimized application layer SFU introduces delays on the order of 20 ms [5]. To reduce these delays, we need to look at all the sources of delay within the SFU. First, in order to send or receive a single packet using the socket mechanism, we have to make a system call; there are no system calls for sending or receiving multiple packets. Furthermore, in order to check whether incoming data is available or whether we can write outgoing data, we need an additional system call, such as `select()`, as there is no straightforward way to inquire how many packets are waiting in a socket receive queue. System calls however incur significant overhead cost and can greatly decrease performance. Typically, a system call requires at least a mode switch, a (partial) context switch, parameter passing between user and kernel space and parameter validation. Our prototype SFU requires one system call for input multiplexing, one system call for reading a packet and, in the case of a data packet i with d_i receivers, d_i system calls for transmission. In total, this creates a cost of $d_i + 2$ system calls per packet.

Second, although the SFU does not process the packet payload, the packet is extensively processed by the networking stack. When forwarding packets, very little processing is required, as each packet arrives with a header that has most of the needed information already in place. However, sockets do not allow applications to take advantage of existing information, as the networking stack strips the packet of its header before delivering it to user space applications. For each packet sent, layer headers are built from scratch; caching techniques may improve performance but still perform redundant processing. Moreover, both receive and transmit system calls trigger a kernel control path that includes many operations, such as allocating and initializing resources, accessing shared kernel data structures, and so on. In total, each incoming packet i needs to go through the networking stack $d_i + 1$ times to be forwarded to all its destinations.

Third, when receiving or sending a packet through standard system calls, the kernel does not pass the user space buffer through the networking stack, but rather copies the packet between user and kernel memory. Data copying itself does not necessarily degrade performance, as it can be used to allow processing multiple packets at a time. In our case however, sockets do not allow for batch processing, therefore packet copying is redundant, causing latency and wasting memory bandwidth. In order to forward a packet i to d_i destinations, $d_i + 1$ packet copies are needed.

III. Network interfaces and netmap

Based on the above discussion, it is clear that going through the network protocol stack is counterproductive if we simply want to replicate packets with minimal changes to their headers. In order to bypass the protocol stack, we can access the *Network Interface Card* (NIC) directly, but this requires access to the kernel. An alternative is to use the netmap framework [6] to directly access packets from user space applications in a safe manner. This section explains how this works.

A. How NICs work

NICs communicate with the host computer over a bus like PCI, transferring data using a DMA engine. Usually NICs provide a fixed-size memory block on-chip (e.g. 64 KB), used to allocate a receive and a transmit FIFO queue to temporarily store incoming and outgoing packets, respectively. The capacity of each queue is not static, and can be configured according to system requirements. This packet buffering is necessary to minimize the possibility of packet loss or corruption, as well as to optimize performance in high-rate workloads.

On the host side, the NIC device driver maintains data buffers in host memory. Note that the host data buffers do not mirror the NIC receive/transmit FIFOs. The ownership of the host buffers is divided between the driver and the device. A receive buffer owned by the device is used to store incoming packets. When a packet fills the buffer, the ownership is transferred to the driver, to be eventually returned to the device when the packet is handled. In a similar way, a transmit buffer owned by the driver is used to store outgoing packets. When the buffer is filled by the driver, the ownership is transferred to the device until the data is moved to the device.

Each driver data buffer is associated with a buffer descriptor, a data structure that contains the buffer address and various other fields used by the NIC to store information. The descriptors of receive buffers are organized as a ring (a circular queue) residing in host memory, forming the receive descriptor ring. The same applies for the transmit buffers, which form a transmit descriptor ring. In order to track buffer ownership, each ring is split using two indexes: head and tail. The device driver owns descriptors and their corresponding buffers in the

range [head...tail - 1], while the device owns descriptors and their corresponding buffers in the range [tail...head - 1].

The NIC uses its on-chip memory to fetch and write back receive and transmit descriptors from the receive and transmit descriptor ring in host memory. To manage these rings, the NIC offers a set of programmable registers for each ring, which store the host memory address of the descriptor ring, the length of the ring, and the head and tail pointers. An on-chip DMA engine is used to transfer data between the NIC and host memory. On the receive path, the DMA engine transfers packets from the receive FIFO to the host buffers and updates the host receive descriptors. On the transmit path, the DMA engine transfers packets from host buffers to the NIC transmit FIFO and updates the host transmit descriptors.

B. *How netmap works*

Netmap is a framework that enables user space applications to perform high-rate packet I/O and potentially handle millions of packets per second [6]. Netmap focuses on reducing the time required to receive or send a packet from/to the wire and offers an efficient, well-integrated and device-independent framework geared towards packet-intensive applications, such as network monitors, firewalls and traffic generators. Many similar frameworks exist, each with different characteristics. Some run completely in the kernel, a constrained environment for developing user space code. Others run completely in user space, taking the role of the device driver, thus exposing critical kernel data structures and device registers to user space, which impairs system security and reliability. Netmap uses a hybrid approach; the kernel space is used for providing synchronization and the user space is employed to run the packet processing code.

To provide low-level packet access, netmap transparently disconnects the host networking stack from the packet datapath. Clients cannot crash the kernel, because critical data structures and device registers are not exposed to clients. Furthermore, clients cannot inject malicious or invalid memory buffer addresses into netmap, because all client requests are validated by the kernel. As far as portability is concerned, netmap does not depend on device-specific features, supports the standard `select()` and `poll()` system calls for event notification and requires minimal driver modifications.

Netmap offers significant speedups over traditional APIs. To achieve this, several ideas from previous works are used. First, netmap removes the need for managing packet buffers by using a NIC-friendly but device-independent metadata representation. Second, memory buffers are allocated only once in linear memory, eliminating the need for dynamic memory management. Third, all data-copy costs are eliminated by allowing direct and safe access to packets.

Netmap uses three main structures: `netmap_if`, `netmap_ring` and packet buffers. A `netmap_if` represents a network interface and holds the number of rings the interface has and an array of relative offsets to the corresponding `netmap_ring` structures. A `netmap_ring` structure represents a RX or TX ring that operates similarly to the NIC rings described above. Each ring holds information tracking its state, such as the size of the ring, the number of available slots, the current slot and so on, and an array of ring slots that hold packet buffer information. Each slot contains a payload length field, a flag field and a index used to access the corresponding packet buffer memory. Packet buffers have a fixed size and are preallocated when the interface is put into netmap mode, thus saving the cost of per-packet allocation. All netmap structures are allocated in linear memory, allowing packet forwarding between interfaces without packet copying.

Using netmap is straightforward. The netmap client first puts the NIC in netmap mode and mapping the netmap device memory to the process address space. The client can then start sending or receiving packets. We can synchronize the client and kernel rings by issuing `ioctl()` calls. Synchronization between the NIC and the client is implemented by using the standard `select()` and `poll()` system calls. When using netmap, the client application must also handle packets coming from or going to the host stack, otherwise other applications cannot send or receive packets. To facilitate this, netmap provides an additional *host* RX/TX ring pair. Packets sent by other applications are passed to the host RX ring and from there it is the netmap client's responsibility to send them to the network. In the same manner, incoming packets are first passed to the netmap client, which, if needed, passes them to the host TX ring to be received by other applications.

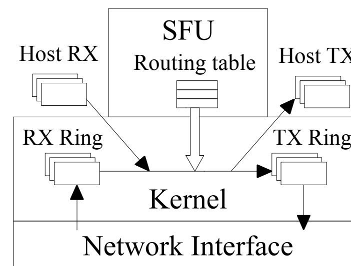

Fig. 2. A netmap-based SFU.

IV. A NETMAP-BASED IMPLEMENTATION

While techniques to improve the performance of our prototype socket-based SFU do exist, a fundamentally different approach is needed to achieve the ultra-low latency required by NMP. Our approach is to exploit the netmap I/O mechanism for low-level packet access, so as to avoid the performance bottlenecks due to per-packet overheads. For this reason, we have implemented a prototype netmap-based SFU, which operates as shown in Figure 2 [7]. This is bound to achieve significant speedups, because netmap offers direct access to low-level implementation details, creating the opportunity for ad-hoc fine-tuning. Of course, the design and implementation of the SFU is more complex than a standard socket-based implementation. Furthermore, since the host networking stack is partially disconnected, the SFU has to deal with raw packets without kernel support and must act as a bridge between the host networking stack and the network.

When in idle state, the SFU waits for incoming packets in the (device) RX and host RX rings. Waiting is implemented via the standard `poll()` function, which in turn calls a netmap-specific poll implementation. Packets available in the host RX ring, i.e. sent by other applications, are moved to the TX ring with zero-copy forwarding by simply swapping the netmap buffer indexes between the receive and transmit slots. Packets arriving at the RX ring, i.e. from the network, must be handled by the netmap client in the case of a packet destined for the SFU, or delivered to the host networking stack in any other case. In the latter case, the packet is moved to the host TX ring, again by using zero-copy forwarding between rings. If the packet is destined for the SFU, the packet processing functions of the SFU are invoked. Checking for the packet (in fact frame) destination, requires accessing the packet headers, which roughly translates to one main memory lookup.

Because of the high-rate of incoming packets, when `poll()` returns, there are usually multiple packets waiting in the RX ring. This allows for naturally implementing batch processing. The number of packets in the RX ring made available to the client by a single system call (RX-batch size) is impacted by server load; under high load there are many packets waiting to be processed, whereas when the load is low the RX ring is almost empty. When `poll()` returns, the SFU handles all packets that are available in the RX ring in a FIFO manner. The first step of this process is moving the packet from the RX to the TX ring using zero-copy packet forwarding between rings. Once the packet is moved, it needs to be sent to d_i destinations. We implemented this using the most straightforward approach; fill the next $d_i - 1$ buffers with a copy of the original packet. Using in total d_i slots, we can send the packet to all its destinations. This packet handling process is repeated for each packet in the RX-batch. When all packets are handled, we return to the idle state.

Packets are not actually scheduled for transfer to the NIC until the kernel netmap ring is updated via an `ioctl()` call. The number of packets that are made available to the NIC driver via a single system call (TX-batch size) is critical for SFU performance. If the batch size is too small, the overhead costs, such as making the system call, setting up the DMA transfer and so on, are not amortized and become excessive. If the batch size is too big, the large packet bursts cause too much idle time in the memory bus and the NIC. Our current implementation schedules packets for transmission only when a user-specified batch size is met, or when the TX ring is full. In the future, we will expand on this idea and use a dynamic TX-batch size based on the current NIC workload.

After replicating an incoming packet, the UDP, TCP and Ethernet headers need to be updated for each outgoing packet to be properly received at the client. At the UDP level we only need to update the destination port. At the IP level, we need to update the source (our own) and destination address, the latter involving one lookup. The IP checksum is also updated incrementally. Finally, at the Ethernet level, we need to update the source (our own) and destination address. This processing takes place as the packet is copied into a TX buffer.

By using netmap we can reduce or even eliminate packet copying when receiving and transmitting packets. As described above, in our implementation we do make additional packet copies so as to implement batch transmission processing. However, we can avoid all packet copying, if we choose to send packets using a TX-batch size of 1 packet, modifying the same packet for transmission in each new round. This can be achieved with minor modifications in our implementation. This strategy however could cause system performance degradation due to the elimination of batch processing. There is an obvious tradeoff between the number of copies and the number of transmission rounds needed to reach a given number of recipients, which we intend to explore in the future.

V. Conclusion and Future Work

We presented an SFU implemented using a socket-based scheme and discussed the performance bottlenecks arising out of the limitations of the socket interface. We then explained why netmap and related frameworks are a natural application for network I/O in SFUs and pointed out the possible performance gains. We finally presented an SFU implementation based on netmap, which avoids socket calls in order to reduce copying and context switching. We are currently performing detailed experiments with the two implementations in order to measure the latency gains of the netmap-based SFU under different assumptions. Building on this work, in the future we will focus on the SFU internals and how they can be optimized to further improve system performance and scalability. One such direction is allowing packet reordering in the SFU, so as to allow packets to be transmitted without any copying, by reusing the same buffer for all transmissions and mixing packets from different streams to create large TX-batches. Another direction is dynamically setting the RX and TX batch sizes so as to optimize delay, depending on the traffic load and the expected delays due to copying.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALES.

References

[1] A. Renaud, A. Carôt, and P. Rebelo, "Networked music performance: State of the art," in *Proc. of the AES International Conference*, 2007.

[2] C. Stais, Y. Thomas, G. Xylomenos, and C. Tsilopoulos, "Networked music performance over information-centric networks," in *Proc. of the IEEE IIMC*, 2013.

[3] A. Eleftheriadis, R. M. Civanlar, and O. Shapiro, "Multipoint videoconferencing with scalable video coding," *Journal of Shejiang University SCIENCE A*, vol. 7, pp. 696–705, 2006.

[4] The MusiNet project. [Online]. Available: http://musinet.aueb.gr/

[5] VidyoRouter datasheet. [Online]. Available: http://www.vidyo.com/wp-content/uploads/DS-VidyoRouter.pdf

[6] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *Proc. of the USENIX ATC*, 2012.

[7] G. Xylomenos, C. Tsilopoulos, Y. Thomas, and G. C. Polyzos, "Reduced switching delay for networked music performance," in *Packet Video Workshop (Poster Session)*, 2013.