Real Time on Ethernet using off-the-shelf Hardware

Jork Löser Hermann Härtig

TU Dresden, Germany

Abstract

Switched networks increasingly become commodity, replacing shared bus networks in LANs. Switched networks support simultaneous access using dedicated channels per attached node and reduce frame drops using buffers. We use these two properties to achieve lossless real-time data transfer at the network level. In this paper, we describe the model and our implementation in a real-time Operating System. This entirely software-based solution provides application-to-application real-time communication on standard hardware using UDP/IP as transport level protocol.

1 Motivation

To provide real-time communication we have to 1) guarantee timely delivery of data frames at the network level, to 2) prevent data loss at the network level and to 3) provide the real-time guarantees to user applications using an appropriate Operating System.

After the availability of ATM networks, that provide real-time transfer at the network level, it took quite some time until Operating Systems were able to provide this real-time transfer to the application level [2]. We show in this paper, that recent developments in the Ethernet technology allow us to use the popular and cheap Ethernet to transfer real-time data efficiently as well. Again it seems, that this potential is not used by OS implementations yet, regardless of significant advantages over ATM: Due to the high costs inherent to the complex ATM technology, ATM did not become as widely used as expected. This urges investigating other, more common network technologies, e.g., Ethernet. To provide real-time data transfer to the application level efficiently, we developed a real-time network stack running on our real-time operating system DROPS.

2 Hardware issues

We focus our work on the Ethernet technology, **the** commodity network for decades. Within the original bus-based Ethernet, *collisions* appear as a result of the the CSMA/CD technology. These collisions lead to automatic retransmissions, which in turn prevent to give tight bounds for the transfer time of data.

With *switched* Ethernet the bus-based data exchange turned into a star-based one: Every node has a pair of exclusive channels to transmit to and receive data from a central switch. The switch receives and forwards the data to the according destination. CSMA/CD is not used and the absence of collisions results in upper bounds for data transfer times.

Still, in cases of high load switches must drop frames. To prevent this dropping, let us investigate what high load means, i.e., when a switch actually drops frames. For the sake of simplicity, we concentrate on an output-buffered switch. Figure 1 shows a typical switch with receive channels (rx channels), control logic, buffer space and queued transmit channels (tx channels).

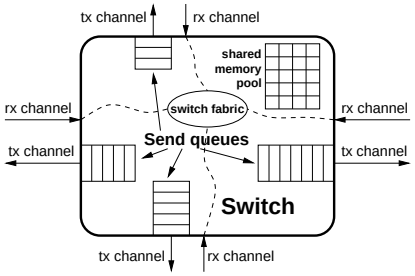

Figure 1: Queuing inside an output-buffered Switch. If queueing a frame is necessary, memory is allocated from a shared memory pool and assigned to the corresponding queue.

If a frame arrives for an output channel where the control logic is forwarding another frame to, the frame is queued. If all memory inside the switch is allocated for queued frames, the current frame must be dropped.

Hence, all that needs to be done to prevent dropping

of frames is to avoid the switching memory of being exhausted, i.e., to bound the output queues in length. This requires two conditions to be met: Firstly, the accumulated average rate of incoming traffic designated for one particular transmit channel must not exceed the traffic rate of the transmit channel. Secondly, the amount of data arriving in a particular time interval must be bound.

Formally, let B be the bandwidth of a channel, measured in number of maximum sized frames per second. Let N be the number of nodes sending to the according output channel, and let b_i the bandwidth (in frames) node i is allowed to send with. Let further M_i be the amount of memory (in frames) the according output queue in the switch is allowed to occupy on behalf of node i.

Using a (Λ_i/E_i) leaky-bucket traffic shaper [4] at the transmitter of each network node results in the desired bounding of the queue lengths. We set $\Lambda_i = b_i$ (the average bandwidth) and $E_i = M_i$ (the maximum burst size). The parameters b_i and M_i are determined based on user-requests for bandwidth and validated by an appropriate software-based reservation mechanism.

When determining values for M_i, minimizing message delays caused by queueing in the switch conflicts with minimizing the CPU consumption in the traffic shaper: Longer maximum queue length increase the maximum delays, smaller bucket sizes may require the traffic shapers to runs more often. When the bucket sizes of a connection are set proportionally to the bandwidth of that connection, the time to refill the bucket becomes a constant. With Fast Ethernet and a switch buffer capacity of 512KByte this refill time has an upper bound of 41.9 ms. Todays workstations are fast enough to run leaky bucket shapers with a refill time of 1 ms.

Note, that the proposed traffic shaping does not require to modify the switch or the node hardware. It can entirely be executed in software at the driver level. In contrast to token-based or time-slot mechanisms this scheme has the advantage that all nodes can perform their send operations independently and specifically unsynchronized after a connection is established. Obviously, the traffic shaping scheme can be extended to multiple connections at each node. Each connection has its own traffic shaper with an own set of parameters.

3 Implementation Issues

Our system is built on the Dresden Realtime OPerating System DROPS [1], a micro-kernel based system.

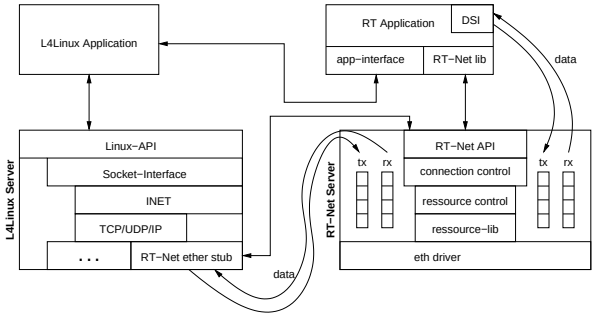

Figure 2: The node architecture.

DROPS runs real-time applications, which reserve the resources they need for proper operation. Remaining resources, including CPU cycles, memory and network bandwidth, can be consumed by best-effort applications. One of these best-effort applications is L^{4}Linux, a server offering the Linux kernel API to execute Linux applications. Consequently, L^{4}Linux utilizes a stub driver to access the network using our network stack.

Figure 2 shows the application model of our approach. An *RT-Net Server* directly interacts with the network interface card (NIC). The server shapes the outgoing traffic according to prior reservation and polices incoming traffic to avoid overload situations. It offers connection-oriented packet-based interfaces to its clients. This allows accounting of transmit traffic and early demultiplexing of received traffic, for real-time clients as well as for best-effort clients.

Best-effort clients normally implement IP-stacks, and hence transfer data-link layer frames to the RT-Net Server. In contrast to this, real-time clients are user applications operating at the transport layer. We use UDP/IP to transfer real-time data, and hence real-time connections are UDP/IP connections with fixed IP addresses and fixed UDP ports. The UDP protocol handling is done entirely at the RT-Net server. For the data exchange between the RT-Net server and the clients a zero-copy IPC protocol [3] is used. However, as we use standard NICs, receiving data requires one copy operation within the server.

3.1 Receiving Process

The receiving process runs in its own thread at interrupt priority inside the RT-Net server. Immediately after a frame is received from the NIC, early demultiplexing is used to find the appropriate receive-connection for that frame. To find a connection, the demux al-

gorithm checks the layer-3 protocol id (IP), the IP-protocol (UDP), the destination address and the destination port of a frame. This requires 2 compare-operations per frame and two additional compares for each real-time client and frame. If no real-time receiver is found, the frame is processed as a best-effort frame.

3.2 Sending Process

Contrary to the receiving process, the sending process is multithreaded, utilizing one thread per connection. Each thread waits for its client to provide a packet. If a packet is obtained on a real-time connection, it is encapsulated using appropriate UDP/IP headers. Note that this is a very fast operation, because the header information is mostly static for the packets of one connection. The connection is traffic-shaped then using a leaky bucket algorithm. Immediately after this, the packet is enqueued at the NIC.

Zero-copying is provided for both real-time and best-effort connections. For real-time connections, the RT-Net Server manages the shared memory used for the connection, which is a physically contiguous piece of memory. Hence, it can calculate the physical addresses of the data therein without effort, which it passes to the NIC. The prepended UDP/IP-headers are passed to the NIC using scatter/gather-techniques. Contrary to real-time clients, best-effort clients are trusted by the RT-Net Server. They pass physical addresses of their data to be send, which is directly passed to the NIC. Hence, the RT-Net Server has no need to access (and copy) best-effort send data.

Prior to establishing a connection at the RT-Net Server, a bandwidth reservation for the intended connection is required. Therefore, a management instance on a network-connected host is contacted, the *bandwidth manager*. The bandwidth manager assigns some amount of the switch buffer memory to each connection and ensures the switch memory not to be overbooked. It also ensures that the bandwidth reservations do not exceed the channel capacities.

3.3 Best-Effort Send Traffic

A problem specific to best-effort traffic is its sporadic burstiness. In contrast to real-time traffic, which uses bandwidth reservations based on prior analysis, best-effort traffic tends to be unpredictable. Moreover, best-effort traffic should utilize all remaining bandwidth, which is not used by real-time traffic. And last but not least, multiple best-effort senders in a network should be able to share the unoccupied bandwidth. Therefore, reserving a fixed bandwidth for each best-effort connection is not an option.

Instead, we reserve only a small amount of bandwidth for every best-effort send connection (i.e., an IP stack normally). If the best-effort sender realizes it needs a higher bandwidth, it tries to make an additional *one-shot reservation*. This one-shot reservation is valid only for a short period of time immediately after the reservation. During this time, the sender can transmit its data. If the time is over, and the sender still has to send a lot of data, it tries to make a reservation again.

When shaping the outgoing traffic at a node, we do not analyze where a best-effort traffic frame is sent to, currently. Therefore, the bandwidth manager takes care of all output queues of the switch when handling best-effort reservation requests.

3.4 Initial Sending

To cope with the problem of establishing the first connection of a node (which is used to establish further connections), we pragmatically reserve a very small amount of bandwidth for every node attached to the network.

An alternative we have in mind is to use traffic prioritizing for the case that the used switch honors priority tagging. Analogously to ATM, all traffic that is sent conforming to a reservation, is marked with a high priority. Other traffic is send with a low priority. While traffic shaping is still required for all reserved connections, prioritizing has the following advantages: The initial traffic to establish the first connection can be sent with a low priority. Hence, we do not need to reserve that small amount of bandwidth for every potentially sending node in the network. Also, the best-effort traffic that exceeds the best-effort reservation could be sent with a low priority. In the case that bandwidth is left, the best-effort traffic passes the switch successfully. In the other case, it is discarded. Unfortunately, TCP/IP performance suffers dramatically from frequently dropped frames. It is on our agenda to look for a solution to this, currently we do not use traffic prioritizing.

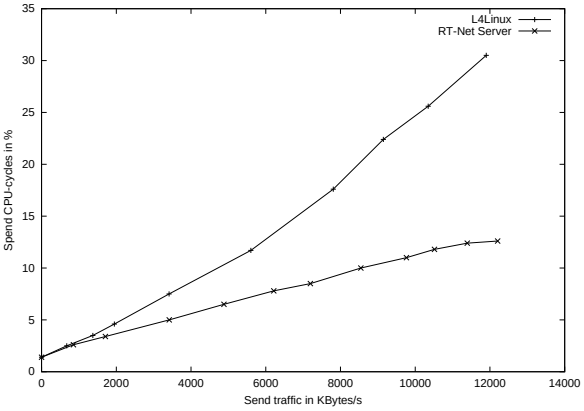

Figure 3: CPU cycles spent for sending data.

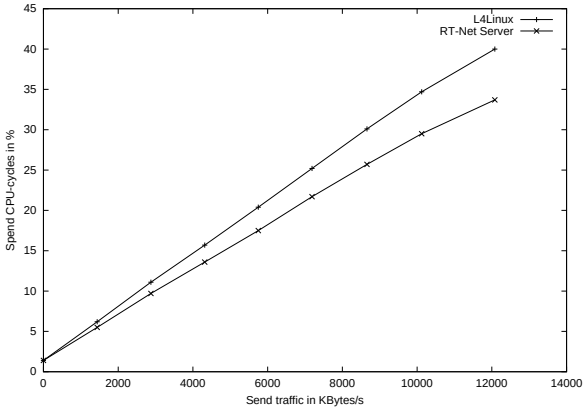

Figure 4: Used CPU cycles when receiving data.

4 Measurements

We measured the CPU time spend by DROPS and our network stack depending on the network load of real-time applications. We set up a send-connection which sends data with a bandwidth according to given reservations. During the experiment, we vary the reservation and hence the bandwidth. The traffic shaper uses a shaping interval of 1 ms all the time: If the bucket becomes empty, it delays frames for at least 1 ms to allow the bucket to fill. To investigate what our scheme for real-time on Ethernet actually costs, we compare the performance of our real-time stack with that of the original L^{4}Linux implementation. For that, we used an Linux application sending UDP datagrams. We sent bursts of different sizes and used `usleep` system-calls of 10 ms between these bursts. The achieved bandwidth was calculated of the amount of data being sent and the elapsed time. The spend CPU cycles in the real-time case and the L^{4}Linux case was measured with a low-priority (i.e., niced) process consuming spare CPU cycles and averaged over a 10 seconds: The less CPU time it got, the more CPU time was spend in DROPS and the network stack, resp. L^{4}Linux.

All experiments were done on an Intel Celeron Processor with 900MHz and 128KByte second level cache. The Fast Ethernet NIC uses shared ring-buffers to communicate with the host. The host writes send or receive descriptors into these rings and the NIC uses PCI-DMA to transfer the data of a frame. Figure 3 shows the time spent for sending data when using the traffic-shaping real-time stack resp. when using original L4Linux (100% correspond to 900 Mio cycles).

To measure the impact of receiving data, we used a similar setup. Here we offered a load to the host, which was consumed either by an real-time application, or, for the original L4Linux, by an user-process receiving the data. Figure 4 indicates the CPU cycles used for these cases.

As you can see, our real-time stack consumes less CPU-cycles than the L^{4}Linux IP-stack implementation. This is mainly due to the small overhead our network stack imposes for data transfer in contrast to L^{4}Linux, which copies the data between the user application and the kernel and executes more code in its network stack. The performance difference for the receive direction is mainly due to the early demultiplexing, which saves a lot of checks and queuing operations compared to the original L^{4}Linux.

References

[1] Dresden Realtime OPerating System. Project overview: `http://os.inf.tu-dresden.de/drops/`.

[2] Martin Borriss and Hermann Härtig. Design and implementation of a real-time ATM-based protocol server. In *19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998.

[3] Jork Löser, Lars Reuther, and Hermann Härtig. Position summary: A streaming interface for real-time interprocess communication. In *8th Workshop on Hot Topics in Operating Systems (HotOS)*, Elmau, Germany, May 2001. A comprehensive Tech Report is available from URL: `http://os.inf.tu-dresden.de/~jork/dsi_tech_200108.ps`.

[4] J. S. Turner. New Directions in Communications (or Which Way to the Information Age?). *IEEE Comm. Magazine*, 24(10):pp. 8–15, October 1986.