Signaled Receiver Processing

José Brustoloni, Eran Gabber and Avi Silberschatz

Signaled Receiver Processing (SRP) is a new scheme for input protocol implementation that makes the system less susceptible to denial of service attacks and can improve application performance under high receive loads.
Most current systems process input protocols at interrupt level.

On BSD Unix, for example, a hardware interrupt happens when a packet arrives. The network interface driver passes the packet to ether_input, which places the packet in IP’s input queue and causes a software interrupt.

The system handles the software interrupt by dequeueing each packet and calling ip_input. Ip_input checksums the packet header and may submit the packet to preliminary processing, including firewall, network address translation (NAT), and IP options. The preliminary processing may drop, modify, or forward the packet. Ip_input then checks the packet’s destination IP address. If that address is the same as the host’s, ip_input reassembles the packet and passes it to a higher-level protocol, as indicated in the packet’s header (TCP, UDP, ICMP, IGMP, RSVP, IPIP, RAW). If the destination address is a multicast address, ip_input may submit the packet for multicast forwarding and also pass the packet to a higher-level protocol, for local delivery. In the remaining cases, ip_input may submit the packet for IP forwarding.

TCP’s and UDP’s input routines checksum the packet, find the protocol control block (PCB) according to the packet’s header (in particular, the destination port), append the packet to the respective socket receive queue, and wakeup processes waiting for the socket (or drop the packet if queue is full).

On receive calls, the receiving application waits for data in the socket’s receive queue (may block) and, after that, dequeues the data and copies it out.
The previous slide shows that BSD Unix processes all input protocols as a hardware or software interrupt, at a priority higher than that of any application. If the receive load is sufficiently high, the system may spend all of its time processing input protocols, and applications cannot make forward progress, a situation known as “receive livelock”.

Additionally, input protocol processing occurs even if the respective socket receive queue is already full. The CPU spent in such cases is wasted, since the packet will need to be dropped.

Finally, the time spent on input protocol processing is charged to the interrupted process, even if that process is unrelated to the incoming packet. Such incorrect accounting can make the CPU scheduler incapable of providing quality of service guarantees.
Lazy receiver processing (LRP) has been proposed for solving the problems mentioned on the previous slide.

In LRP, the network interface hardware or its driver perform early demultiplexing, that is, examine the packet header to determine the corresponding channel (each socket has an associated channel). The system then enqueues the packet in that channel’s queue and wakes up the processes waiting for it (or drops the packet if the queue is full).

On UDP receive calls, the receiving process performs the following loop while there is not enough data in the socket receive queue: wait for packet in channel (may block) and then dequeue the packet and call ip_input, which eventually causes the packet to be enqueued in the socket receive queue. The receiving process then dequeues the data and copies it out.

The same scheme would not work well for TCP because, for best throughput, the receiver has to send acknowledgements promptly, asynchronously from the application’s receive calls. LRP therefore uses a kernel thread that is scheduled at the receiving process’s priority and continuously waits for a packet in the channel, dequeues the packet, and calls ip_input. On TCP receive calls, the receiving process can then simply wait for data in the socket receive queue (which may block), dequeue the data, and copy it out.
Difficulties in using LRP

- What if kernel threads are not available (e.g., FreeBSD)?
- What if an application is busy and would like to defer TCP/IP input processing?
- What about preliminary packet processing (e.g., firewall, NAT)?
- Should IP forwarding be penalized for its CPU consumption?

In LRP, most input protocol processing is scheduled at the receiving application’s priority, preventing receive livelock and other scheduling anomalies.

However, LRP can also present some difficulties.

First, several contemporary systems (e.g., FreeBSD) do not support kernel threads, which are required by LRP.

Second, even if an application is busy, it cannot defer its TCP input protocol processing, which happens in a separate kernel thread.

Third, LRP’s early demultiplexing ignores preliminary processing, such as firewall and NAT. Such processing may modify the packet and require new demultiplexing.

Fourth, LRP does not properly acknowledge that some system protocol processing, such as IP forwarding, is ill-suited for scheduling as a time-sharing process that gets penalized according to its CPU usage.
Signaled Receiver Processing (SRP)

NW I/F HW or SW or after preliminary packet processing (e.g., firewall, NAT)
   REDMX: Determine PCB from (packet header, skip)
   Enqueue packet in PCB → socket → UIQ (unprocessed input queue)
   Wakeup processes waiting for socket or
   signal SIGUIQ to processes that have socket open

Default SIGUIQ: dequeue packet and call ip_input
   (but: application can catch and defer SIGUIQ processing)

Receiving process: dequeue packets and call ip_input
   wait for data in socket receive queue
   dequeue data and copy it out

System protocol processing (e.g., firewall or IP forwarding):
   - On Eclipse/BSD, processes with minimum CPU guarantee
   - pseudo-sockets

Signaled receiver processing (SRP) uses a reinvocable early demultiplexing function (REDMX), used by the network interface hardware or its driver and after each preliminary processing step (e.g., firewall or NAT). REDMX determines the PCB that corresponds to a packet based on the packet header and a skip argument, which expresses the preliminary processing steps already performed (if any). Each PCB points to a socket, and each socket has an unprocessed input queue (UIQ). The system enqueues each packet in the corresponding UIQ and wakes up processes waiting for the respective socket. If no processes are waiting, the system signals SIGUIQ to the processes that have the socket open.

The default SIGUIQ action is to dequeue the packet and call ip_input. This causes input protocol processing to occur in the context of the receiving processes, but possibly asynchronously to the application’s receive calls. The packet is then enqueued in the socket receive queue.

However, any process can catch its SIGUIQ signals and, for example, defer input protocol processing until a subsequent receive call.

On receive calls, the receiving process dequeues any packets from UIQ and calls ip_input. The process then waits for data in the socket receive queue (may block), dequeues the data, and copies it out.

On Eclipse/BSD, system protocol processing, such as IP forwarding, occurs in the context of processes that have minimum CPU shares (e.g., 40%), guaranteed by the operating system.
Advantages of SRP

• No kernel threads needed
• Application can catch SIGUIQ and defer protocol processing
• REDMX supports preliminary protocol processing (e.g., firewall, NAT)
• Eclipse/BSD’s CPU scheduler allows system protocol processing not to be penalized for CPU consumption

Therefore, SRP provides the following advantages in addition to LRP’s:
• No kernel threads are needed;
• Busy applications can defer protocol processing; and
• Full system functionality present in contemporary systems can be maintained, including preliminary protocol processing, such as firewall and NAT.

SRP was developed for Eclipse/BSD, a system that provides quality of service guarantees. On Eclipse/BSD, SRP realizes the additional advantage:

• System protocol processing, such as IP forwarding, is appropriately scheduled.

SRP can be useful in any systems (e.g., servers) subject to high receive loads.