Memory mapped netlink

Patrick McHardy <kaber@trash.net>

Netfilter Workshop 2011

Freiburg im Breisgau, Germany

Memory mapped netlink
Current state of affairs

- Netlink uses regular socket I/O
- Messages are constructed into a socket buffer's data area, then copied to userspace or vice versa
- Performance doesn't matter much in many cases since most netlink subsystems have very low bandwidth demands. Notable exceptions are nfnetlink_queue, ctnetlink and possibly nfnetlink_log.

Memory mapped netlink
Current state of affairs

- No need to be wasteful though, even low bandwidth subsystems can benefit from improved performance, especially during large dumps

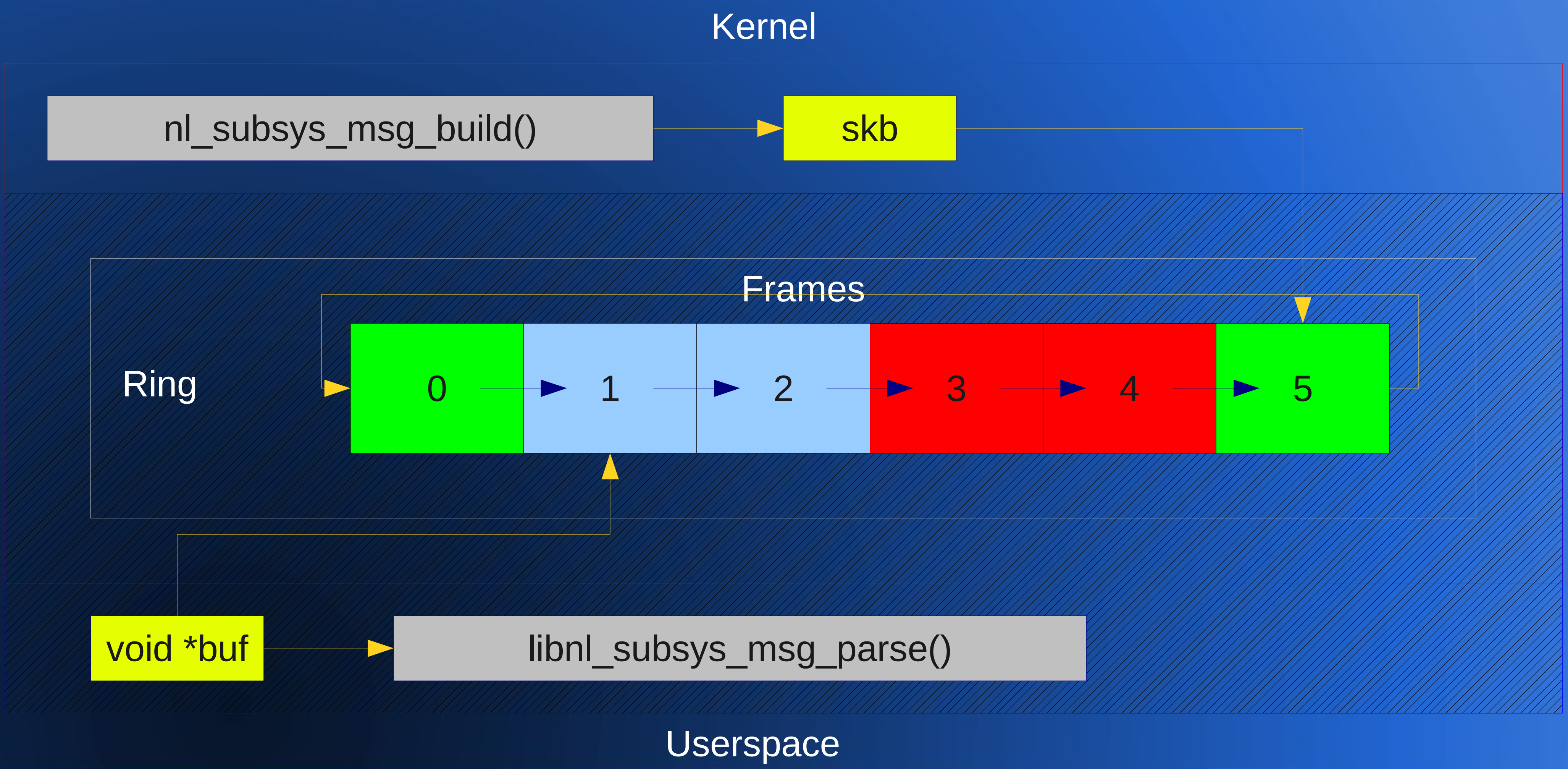
Memory mapped netlink Basic concept

- Have userspace set up a shared memory mapped circular ring (modelled after AF_PACKET, see Documentation/networking/packet_mmap.txt) for RX and TX
- In kernel->userspace direction, attach frames from ring to socket buffer heads and perform message construction directly into the memory mapped area. Userspace parses messages from that area.

Memory mapped netlink
Basic concept

- In userspace->kernel direction parse the messages directly from the memory mapped area

Memory mapped netlink
Basic concept



Memory mapped netlinkk
Basic concept

- Ring frames are of fixed size specified by userspace and contain data necessary for synchronizing processing between kernel- and userspace, meta data and the actual message data
- Synchronization happens by means of a status descriptor that specifies ownership of the frame and the operation to perform

Memory mapped netlink
Basic concept

- Frames to be processed by userspace are marked using NL_MMAP_STATUS_USER, after finishing processing userspace releases the frame back to the kernel by resetting the status word to NL_MMAP_STATUS_KERNEL
- Messages that exceed the ring data size are queued to the socket receive queue and userspace is instructed to invoke recvmsg() by setting the status word to NL_MMAP_STATUS_COPY

Memory mapped netlink Basic concept

- Similary, frames that originate from a subsystem not supporting memory mapped operation that are delivered to a memory mapped socket are queued and userspace is instructed to invoke recvmsg()
- Frames from userspace to the kernel are stored in the ring and their status word is marked using NL_MMAP_STATUS_SEND. To trigger processing on the kernel side, userspace invokes sendmsg() with msg.iov.iov_base = NULL.

Memory mapped netlink Implementation details

- Userspace ring setup is performed using setsockopt() calls for RX and TX rings (NETLINK_RX_RING/NETLINK_TX_RING) and a call to mmap() to map the resulting ring area into the process' address space:

  setsockopt(fd, NETLINK_{RX,TX}_RING, ...);

  ring = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

Memory mapped netlink Implementation details

- Message ordering details are problematic: socket buffer data area points to ring frames of receiving sockets, meaning socket lookup has to be performed before constructing messages, not as usual when delivering them. Result is that message order in the ring depends on allocation time, not delivery time. When order matters (nfnetlink_queue in some cases) construction and delivery is performed atomically, allocation usually isn't.

Memory mapped netlink Implementation details

- Since kernel netlink subsystems usually perform message validation before actually processing the contents, the contents of the shared memory area must not be allowed to be changed by userpace change after validation.
- Only a single mapping of the ring is allowed, also socket file descriptor must not be shared (f.i. through AF_UNIX sockets). When either condition is false, fallback to copying.

Memory mapped netlink Implementation details

- Because of the need to synchronize processing between kernel- and userspace, only unicast communication is supported.
- Conversion to use mmaped netlink in the kernel is easy: replace message allocation by netlink_alloc_skb(ssk, size, dst_pid, gfp_mask)
- Dump are using mmaped I/O automatically

Memory mapped netlink Implementation details

- For multicast ring could theoretically be mapped into multiple processes address spaces and descriptors be separated from data area. Synchronization would then depend on slowest of all processes though, one hanging process would break netlink communication for everyone sharing a ring.

Memory mapped netlink Implementation details

- Different approach to supporting multicast is to allow memory mapped I/O as long as only a single listener is subscribed to a multicast group, which is a very common case for high bandwidth netlink subsystems.

Memory mapped netlink Implementation details

- Similar to unicast case, message destination would have to be determined at allocation time instead of transmission time. Does not matter though since a process newly subscribing to a group can't know which message will be the first received one anyways.

Memory mapped netlink Implementation details

- Flow control during dumps with regular I/O is happening by checking socket queue receive space during recvmsg() call
- Obviously doesn't work with memory mapped I/O since no recvmsg() invocation. Currently done in netlink_poll() once userspace has (supposedly) processed all messages in the ring

Memory mapped netlink Implementation details

- In userspace support for testing has been added to libmnl
- ring setup: mnl_socket_set_ring(sk, rx_size, tx_size). Frame size is currently hardcoded.

Memory mapped netlink Implementation details

- Reception:

```
mnl_socket_poll();
while (1) {
        hdr = mnl_socket_get_frame(sk, MNL_RING_RX);
        if (hdr->nm_status != NL_MMAP_STATUS_USER)
                break;
        <process frame>
        <release frame back to kernel: update status>
        mnl_socket_advance_ring(sk, MNL_RING_RX);
        }
```

Memory mapped netlink Implementation details

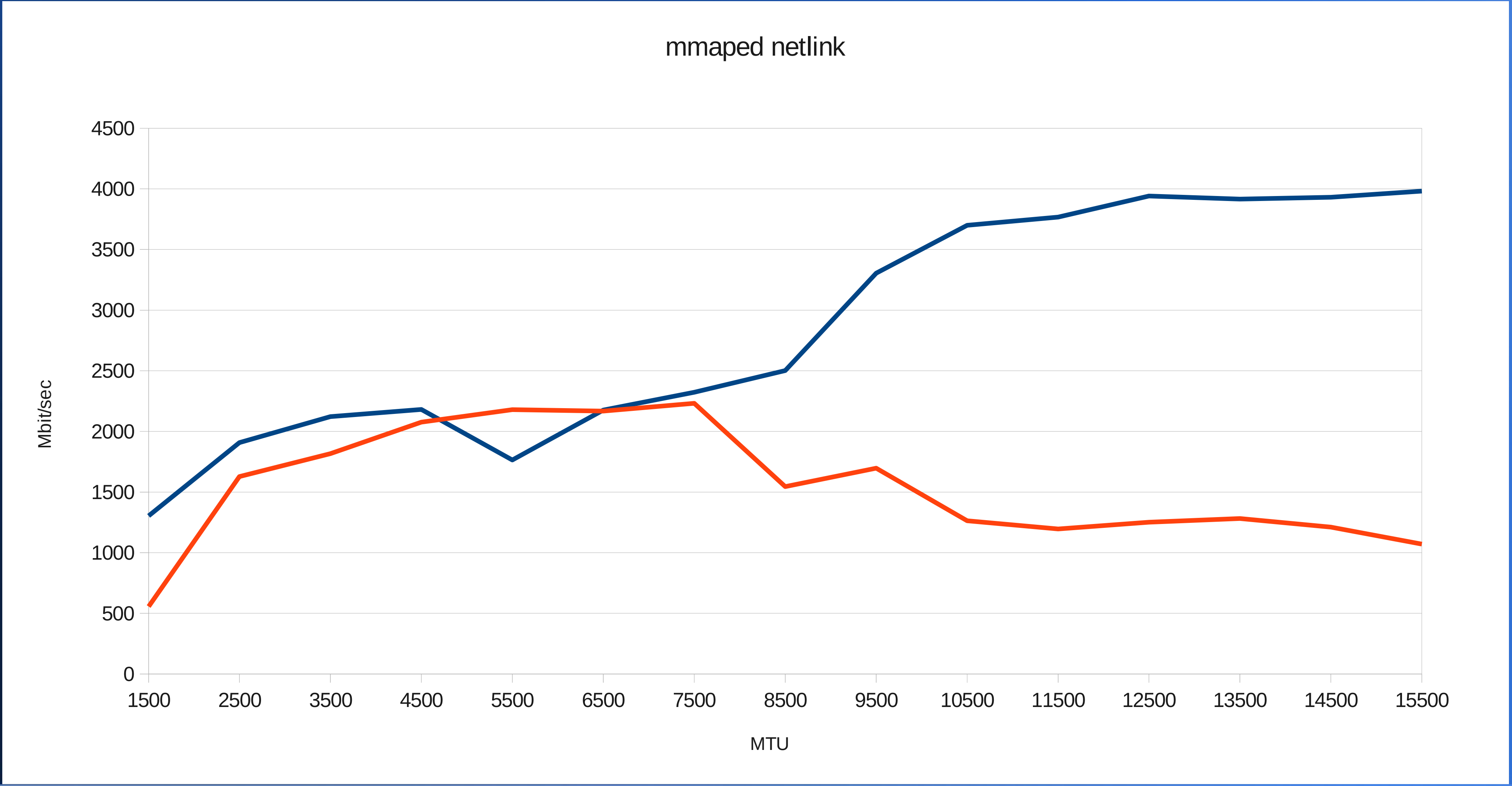
- Transmission:

```
hdr = mnl_socket_get_frame(sk, MNL_RING_TX);
if (hdr->nm_status != NL_MMAP_STATUS_KERNEL)
        <handle error>
<build msg>
hdr->nm_status = NL_MMAP_STATUS_SEND;
mnl_socket_sendto(sk, NULL, 0)
```

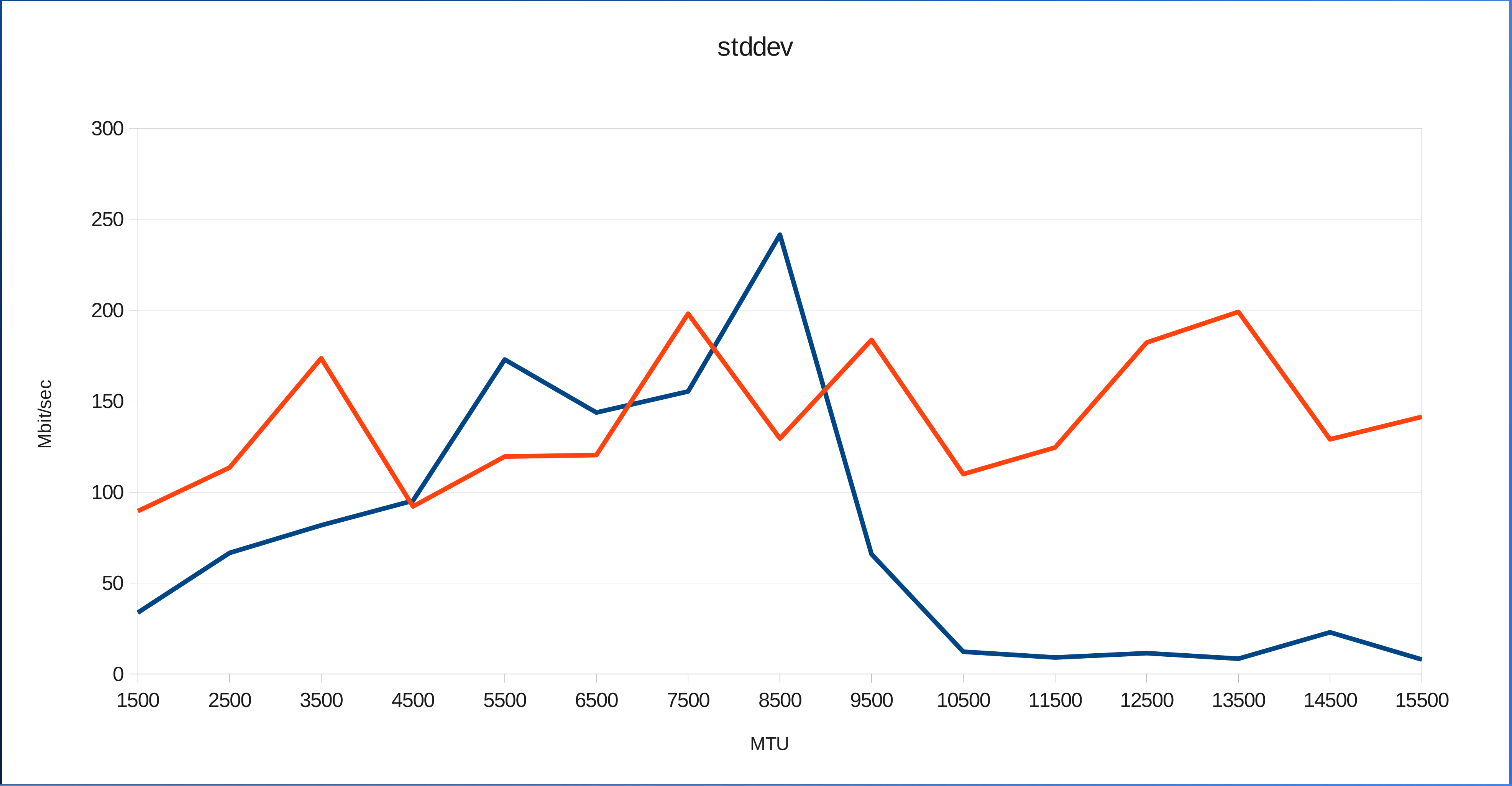
Memory mapped netlink Performance

- Due to lack of sufficiently fast hardware (10Gbit), unfortunately only loopback testing could be performed.
- Even better performance for real NICs expected since loopback traffic most likely benefits from CPU caching effects, decreasing copying costs.
- Scenario: iperf over loopback for different MTUs with nfnetlink_queue

Memory mapped netlink Performance



Memory mapped netlink Performance



Memory mapped netlink Performance

- Performance is roughly +200%-+300%, depending on MTU, also more stable (both with exceptions).
- For MTU of 4500-6500, variance peaks and performance goes down. Scheduling effects suspected (two CPUs, three demanding processes), needs test on real network.
- PPS test on Gbit NIC with 64b packets shows improvement of +400% (not included in graphs).

Memory mapped netlink
State of development

- Working well, no known bugs
- Flow control and socket buffer accounting still need a bit of work
- Submission to netdev will happen soon
- Code will be uploaded to kernel.org soon (today or tommorrow): git://git.kernel.org/pub/scm/linux/kernel/git/kaber/nl-mmap-2.6.git

Memory mapped netlink Thanks

- Harald for discussing the basic concept a couple of hundreds time with me
- Davem for the suggestion to use page mapping count to assure payload can't be modified after validation on transmission