Message Passing Architecture in Intra-Cluster Communication

Xiao Zhang

Lamxi Bhuyan

<xzhang, bhuyan>@cs.ucr.edu

Outline

1 – Kernel-based Message Passing Architecture

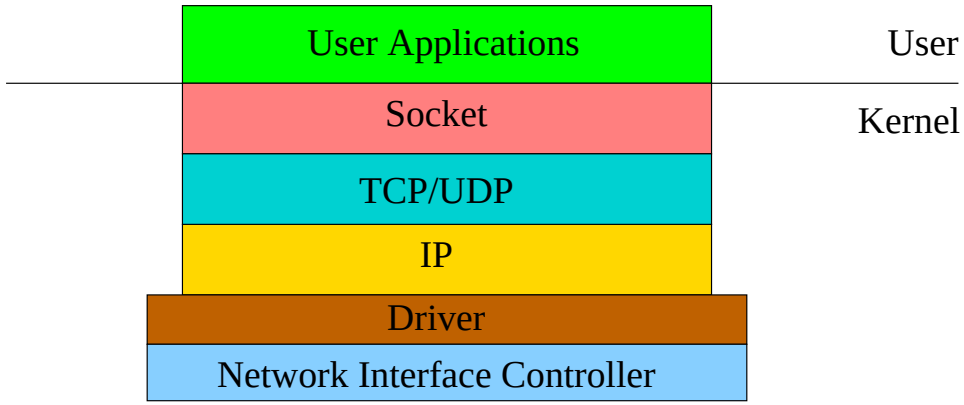


Figure 1: Traditional kernel-based message passing architecture for intra-cluster communication.

- User application initiates communication.
- Socket is the interface to user application.
- TCP and UDP provide end-to-end channel for users.
- IP is for routing.
- Driver/NIC perform transmit and receive.

User Applications:

```
s = socket(SOCK_DGRAM)
bind(s,localAddr)
...
Allocate 2 msg buf
...
connect(s,remoteAddr)
...
Prepare msg[1] for sending
send(s, msg[1])
recv(s, &msg[0])
...
close(s)
```

Echo client using UDP/IP socket

```
s = socket(SOCK_DGRAM)
bind(s,localAddr)
...
Allocate 1 msg buf
...
connect(s,remoteAddr)
...
...
recv(s, &msg)
send(s, msg)
...
close(s)
```

Echo server using UDP/IP socket

1.1 – How UDP send/receive work

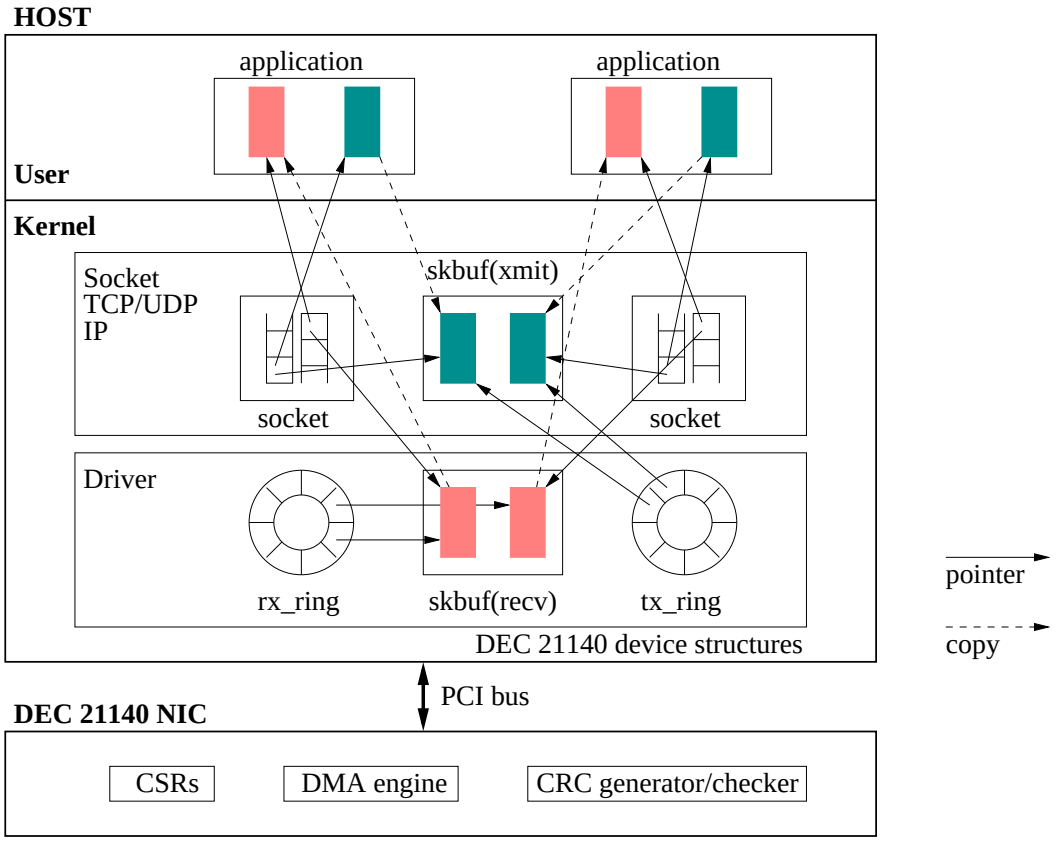


Figure 2: TCP/UDP/IP over DEC 21140 NIC (tulip)

UDP send operations

Socket:

- get the BSD socket structure.
- build `msghdr` structure for the sending message.
- pass control from BSD socket to INET socket.

UDP:

- get and verify the address. For connected socket, a flag is set to avoid routing.
- For unconnected socket, call routing function to get the routing table entry.
- fill in the UDP header (except checksum).

IP:

- segment large message if necessary.
- allocate a socket buffer (`skbuf`) and a data buffer for the sending message. The data buffer includes the message and udp/ip/ethernet headers. The ethernet header is 16-byte aligned. The data buffer is 128-byte aligned.
- fill in the IP header and calculate the IP header checksum.

IP (continue):

- copy the message from user space to `skbuf` and compute the message checksum at the same time.
- compute the UDP header checksum and TCP/UDP pseudo-header checksum.
- copy the UDP header to `skbuf`.

⇓

0 31

source ip address		
destination ip address		
0	Protocol	segment length

Driver(xmit):

- enqueue `skbuf` to `tx_ring`.
- trigger the immediate transmit.

NIC:

- DMA fetch `tx_ring` descriptor.
- DMA fetch data buffer.
- CRC checksum.
- put packet onto the wire.

After NIC complete transmission:

- send a interrupt signal to CPU.
- CPU calls the NIC interrupt handler to dequeue `skbuf` from `tx_ring` and free the buffer.

UDP receive operations

when NIC receives a packet:

- DMA the incoming packet to a `skbuf` pointed by `rx_ring` tail descriptor.
- send an interrupt to CPU.

Driver(recv):

- determine the packet protocol id.
- push `skbuf` into a queue.
- add the NIC to the poll list, and raise a soft interrupt to trigger the bottom-half of the receive process.

IP (in bottom half):

- check IP header (length, version, checksum).
- handle IP options if any.
- reassemble IP fragments if necessary.

UDP (in bottom half):

- check UDP header (length, checksum).
- find the INET socket that matches the address/port.
- enqueue `skbuf` into the socket receive queue.
- wake up a sleeping process waiting for packets.

Socket:

- get the BSD socket structure.
- build `msghdr` structure for receiving message.
- pass control from BSD socket to INET socket.

UDP (top half):

- If no packet arrives and `recv()` is a blocking call, the current process will sleep.
- After being waked up, copy and checksum the message from `skbuf` to user space.
- copy the sender's address to user space.

1.2 – UDP critical path analysis

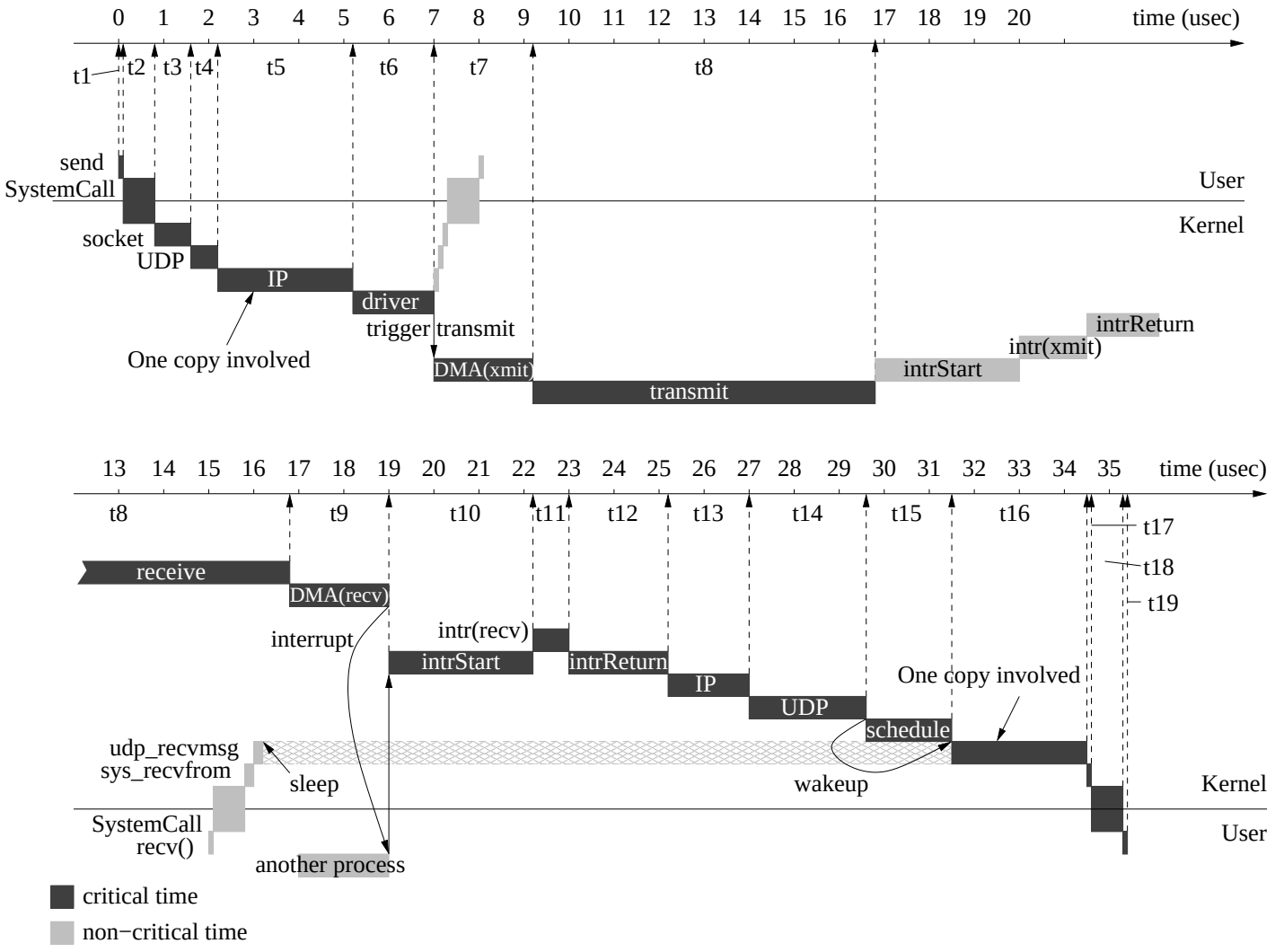


Figure 3: Time line of transferring 1-byte message using UDP/IP

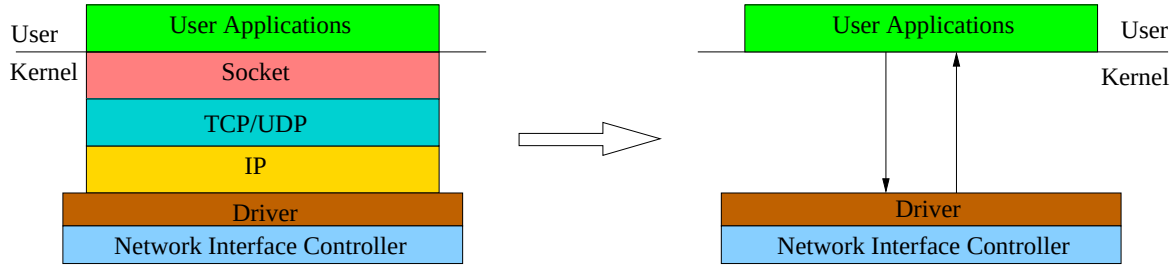
Categories	**Intervals**	**usec**	**percentage**
User application	$t_1 + t_{19}$	0.3	0.9
Protocol processing	$t_3 + t_4 + t_5 + t_6 + t_{11} + t_{13} + t_{14} + t_{16} + t_{17}$	14.5	41.0
OS system call overhead	$t_2 + t_{18}$	1.6	4.5
OS interrupt overhead	$t_{10} + t_{12}$	5	14.1
OS context switch overhead	t_{15}	2	5.6
DMA/transmit/receive	$t_7 + t_8 + t_9$	12	33.9
Total		35.4	100

Table 1: UDP critical time break down

Problem: **LARGE OVERHEAD!**

Goal: reduce software overhead to approach hardware limits.

Quick solution:



This idea is reasonable:

- Cluster network can be assumed to be reliable and secure. A thick procotol stack like TCP/IP is not necessary.
- An increasing number of applications are more sensitive to communication latency.

But,

- How to allow direct access yet provide protection?
- How to design an efficient, yet versatile programming interface?
- How to manage resources, in particular memory?
- How to fair share network without a kernel path?

2 – User-Level Message Passing Architecture

In *user-level message passing architecture*, the operating system kernel and its centralized networking stack are removed from the critical communication path in a protected fashion, providing user applications a *user-level network interface*, through which users can directly access to their network interface. The operating system is only involved in setting up the communication.

In this way, the communicating parties can avoid intermediate copies of data, interrupts, and context switches in the critical path, thus greatly decreasing communication latency and increasing network throughput.

2.1 – VIA Overview

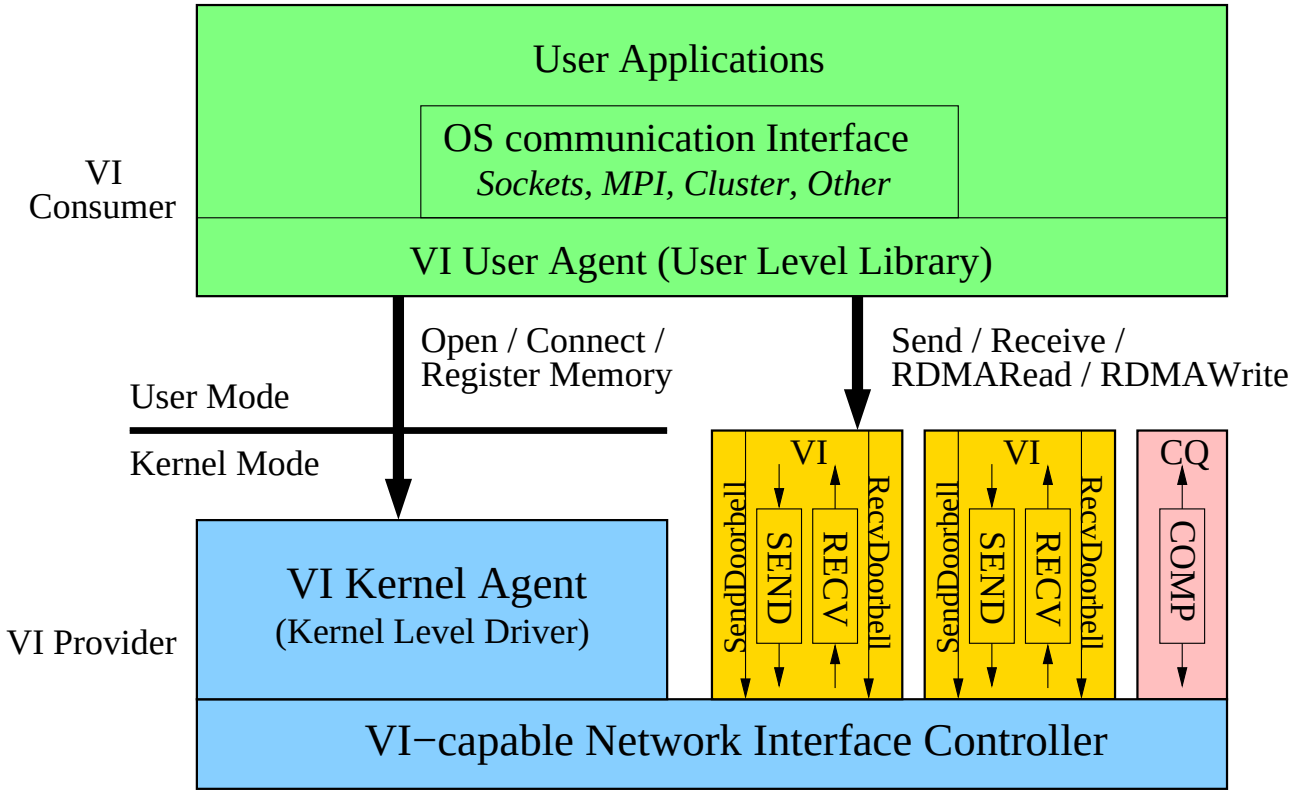


Figure 4: The Virtual Interface Architecture Model

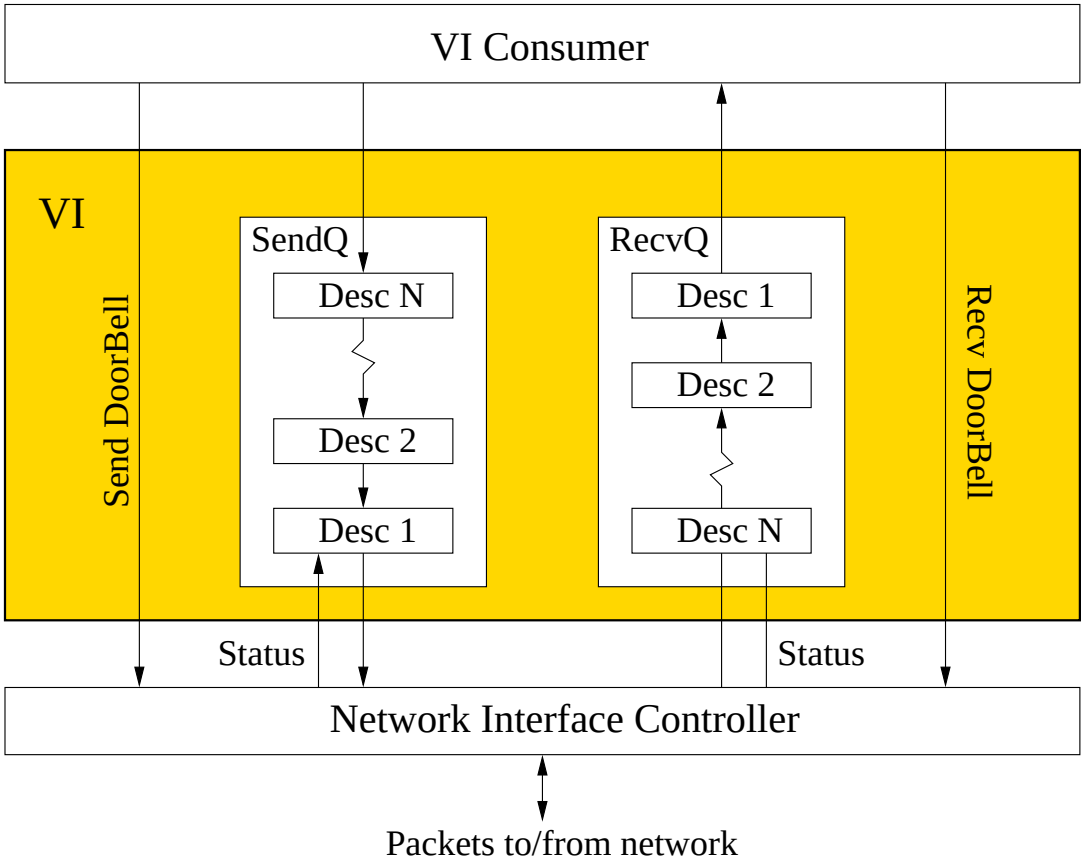


Figure 5: A Virtual Interface

```
VipOpenNic(&nicHand)
...
VipCreateVI(&viHand)
...
Allocate 3 desc and 3 msg
Register 3 desc and 3 msg
desc[i].buf ← msg[i]
...
VipostRecv(viHand, &desc[1])
descp ← &desc[2]
VipConnectRequest(viHand,lAddr,rAddr)
...
Prepare msg[0] for sending
VipPostSend(viHand,desc[0])
VipSendWait(viHand,&descp2)
VipPostRecv(viHand,descp)
VipRecvWait(viHand,&descp)
...
VipDisconnect(viHand)
Deregister desc and msg
VipDestroyVI(viHand)
```

Echo client using VIA

```
VipOpenNic(&nicHand)
...
VipCreateVI(&viHand)
...
Allocate desc[2] and msg[2]
Register desc[2] and msg[2]
desc[i].buf ← msg[i]
...
VipPostRecv(viHand,&desc[0])
descp ← &desc[1]
VipConnectWait(nicHand,lAddr,rAddr,&connHand)
VipConnectAccept(connHand,viHand)
...
VipPostRecv(viHand,descp)
VipRecvWait(viHand,&descp)
VipPostSend(viHand,descp)
VipSendWait(viHand,&descp)
...
VipDisconnect(viHand)
Deregister desc and msg
VipDestroyVI(viHand)
```

Echo server using VIA

2.2 – How M-VIA send/receive work

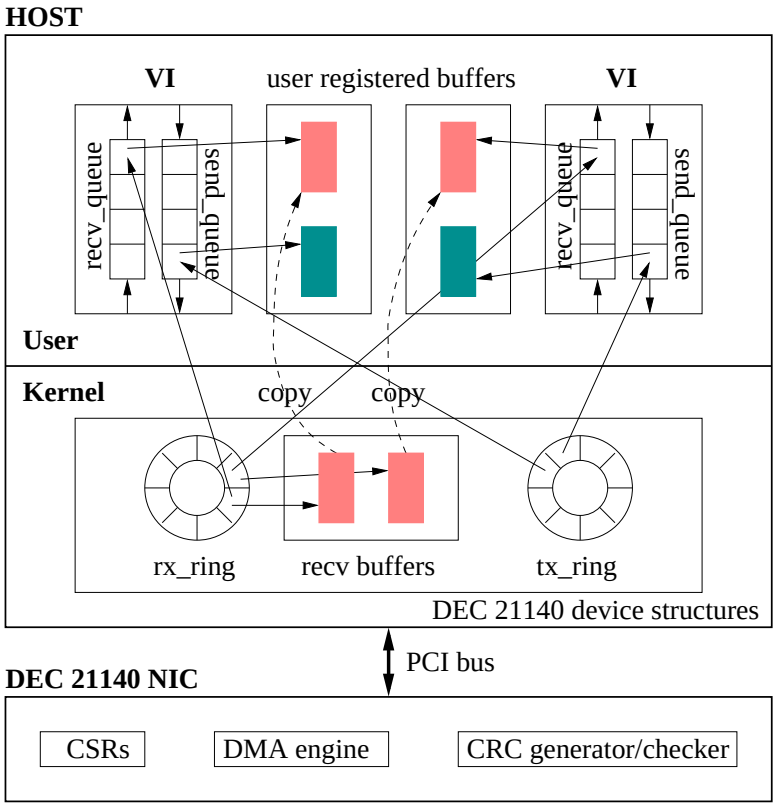


Figure 6: M-VIA over DEC 21140 NIC (tulip). M-VIA is a modular VIA implementation for Linux over Ethernet.

M-VIA send operations

VipPostSend:

- enqueue a send descriptor to the sending queue.

Driver(xmit):

- prepares send header, waits for device stop, gets the next send decriptor entry, and fill in the header.
- segments large packets into chunks.
- enqueue each chunk to `tx_ring`.
- trigger the immediate transmit.

NIC:

- DMA fetch `tx_ring` descriptor.
- DMA fetch data buffer.
- CRC checksum.
- put packet onto the wire.

After NIC complete transmission:

- send a interrupt signal to CPU.
- CPU calls the NIC interrupt handler to dequeue head entries from `tx_ring`, mark the descriptor complete and wake up a block process.

VipSendWait:

- check the head descriptor of the sending queue for N times. If the head descriptor is marked complete, remove it from the sending queue and return success.
- otherwise, sleep.
- after being waked up, check the head descriptor again. If the head descriptor is marked complete, remove it from the sending queue, and return success. Otherwise, return error.

M-VIA receive operations

VipPostRecv:

- enqueue a recv descriptor to the receiving queue.

when NIC receivess a packet:

- DMA the incoming packet to a buffer pointed by `rx_ring` tail descriptor.
- send a interrupt signal to CPU.

Driver(recv):

- get a corresponding VI structure from the received packet.
- check the sequence number of the received packet.
- copy the packet from `rx_ring` to VI specified buffer.
- dequeue head entry from `rx_ring`.
- mark the descritpor complete.
- wake up a block process.

VipRecvWait:

- checking the head descriptor of the receiving queue for N times. If the head descriptor is marked complete, remove it from the receiving queue and return success.
- otherwise, sleep.
- after being waked up, check the head descriptor again. If the head descriptor is marked complete, remove it from the receiving queue, and return success. Otherwise, return error.

2.3 – M-VIA critical path analysis

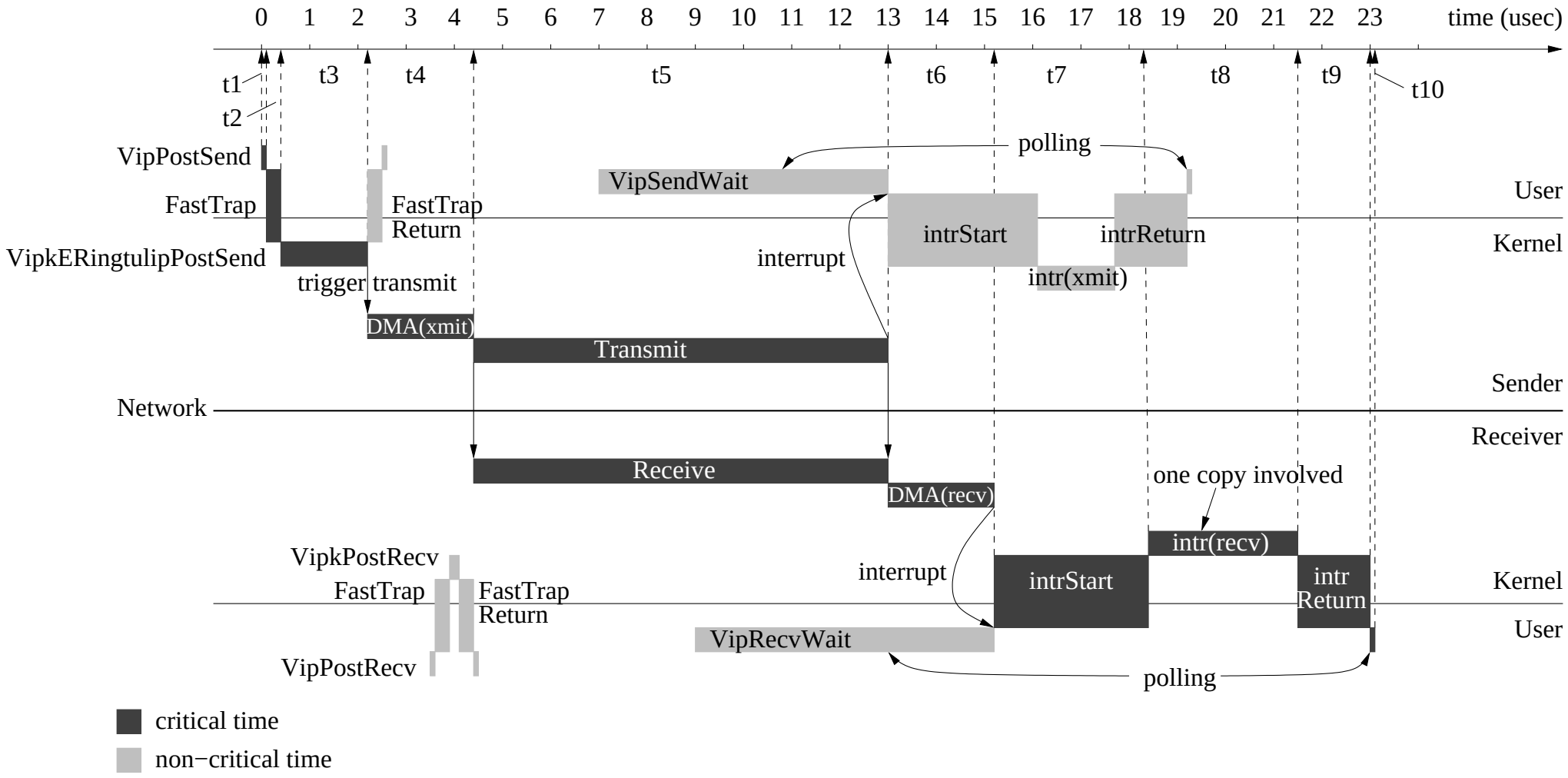


Figure 7: Time line of transferring 1-byte message using M-VIA

Interval	**Description**	**Time**	
		cycles	usec
t_1	user post send descriptor	31	0.08
t_2	Fast trap	138	0.34
t_3	device send (first chunk)	733	1.83
$t_4 + t_5$	DMA(xmit)/transmit		$10.1 + 0.08s$
t_6	DMA (receive)		???
t_7	interrupt start	1268	3.16
t_8	interrupt recv processing		$3.1 + 0.272\lfloor(s + 25)/32\rfloor$
t_9	interrupt return	627	1.57
t_{10}	user receive (polling)	20	0.05

Table 2: M-VIA critical time breakdown

2.4 – M-VIA performance evaluation

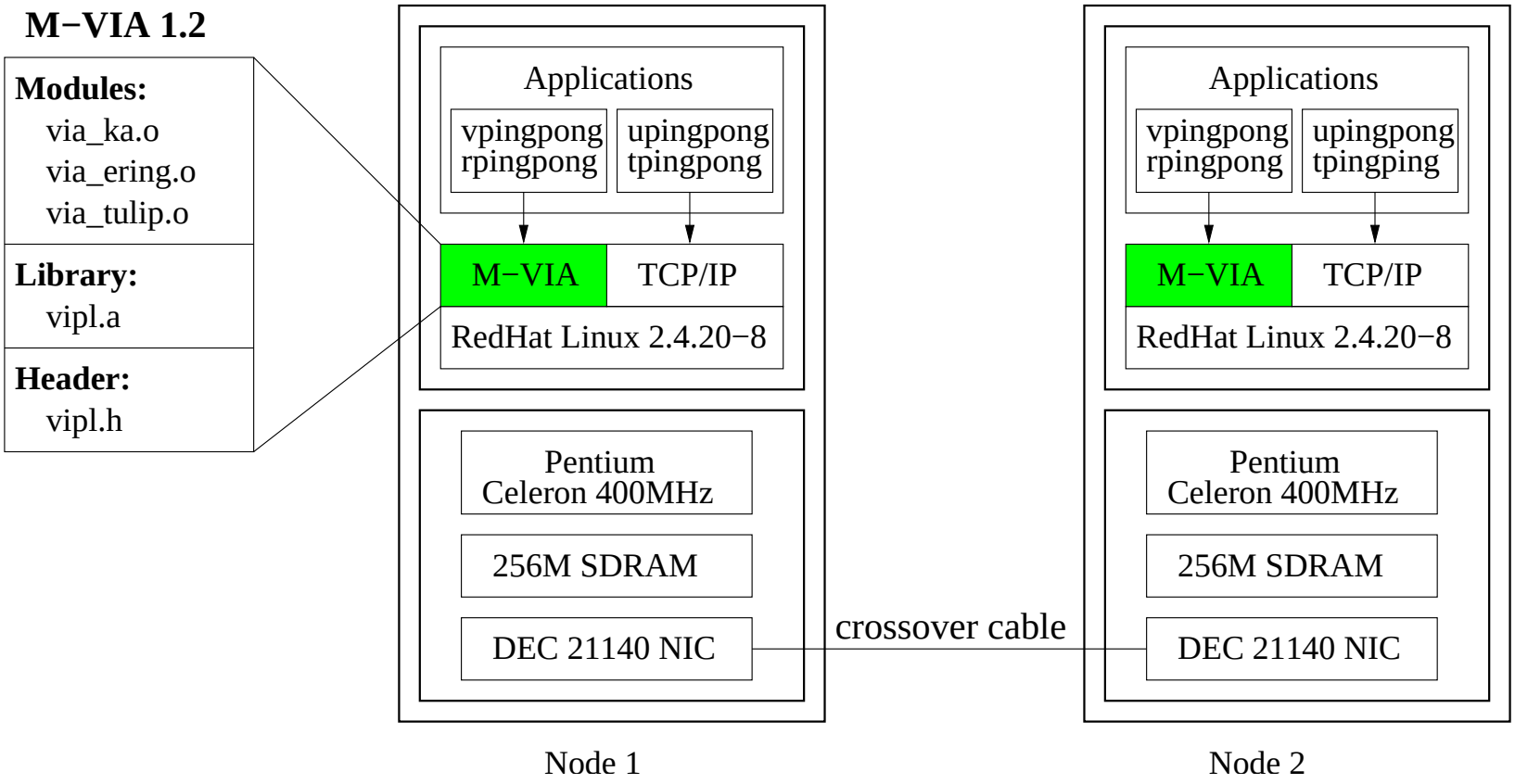


Figure 8: Experimental Setting

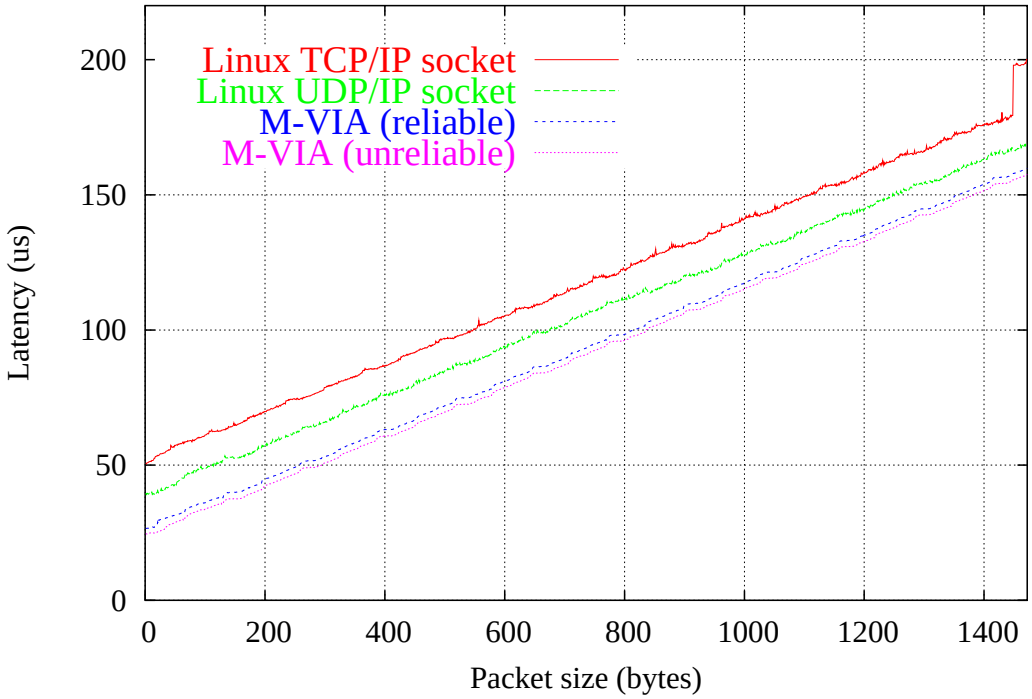


Figure 9: One-way lentency vs. message size

- entirely remove socket, TCP/UDP and IP layers.
- zero copy when sending. Sending overhead is constant.
- use fast trap instead of normal system call.
- use polling instead of interrupt to avoid context switch.

3 – Conclusion

- Traditional kernel-based communication has large software overhead.
- User-level communication is a good solution for intra-cluster communication.