Networking

Advanced Operating Systems
(263-3800-00)

Timothy Roscoe

Thursday 18 November 2010

Systems@ETH Zürich

Outline

Systems@ETH Zürich

- Introduction
  - Motivation
  - What the problem is
- Networking in BSD
- Multiplexing
- Lazy Receiver Processing
- Receive Livelock
- Demultiplexing techniques
  - Software demultiplexing
  - Hardware demultiplexing
- Structural approaches

Why does networking matter?

- Almost all interesting apps are now networked
  – Single-machine systems are now rare
- Only new h/w development to impact the OS since Unix
  – Focus of much OS research for 20 years
- Increasingly, network i/f is the only peripheral that matters
  – Disks increasingly appear to be the other side of a network
  – GPU increasingly less of a periphera

Too big a topic for one lecture!

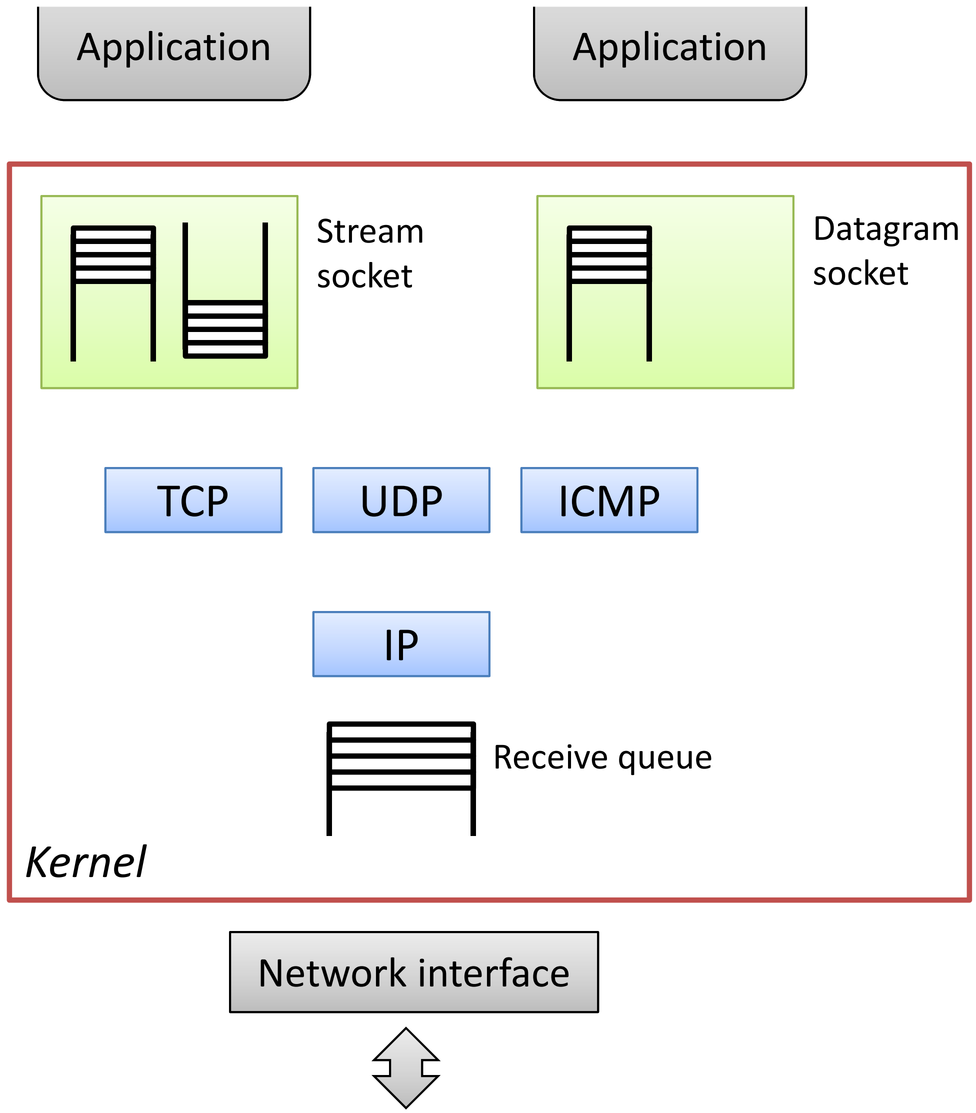
Systems@ETH Zürich

- Assume general familiarity with Unix-style networking
  - BSD sockets
  - Network interfaces, routing tables, etc.
- Will also assume you know about networks
  - Packet-oriented protocols
  - TCP end-system processing requirements
- Focus on:
  - General concepts
  - Specific techniques

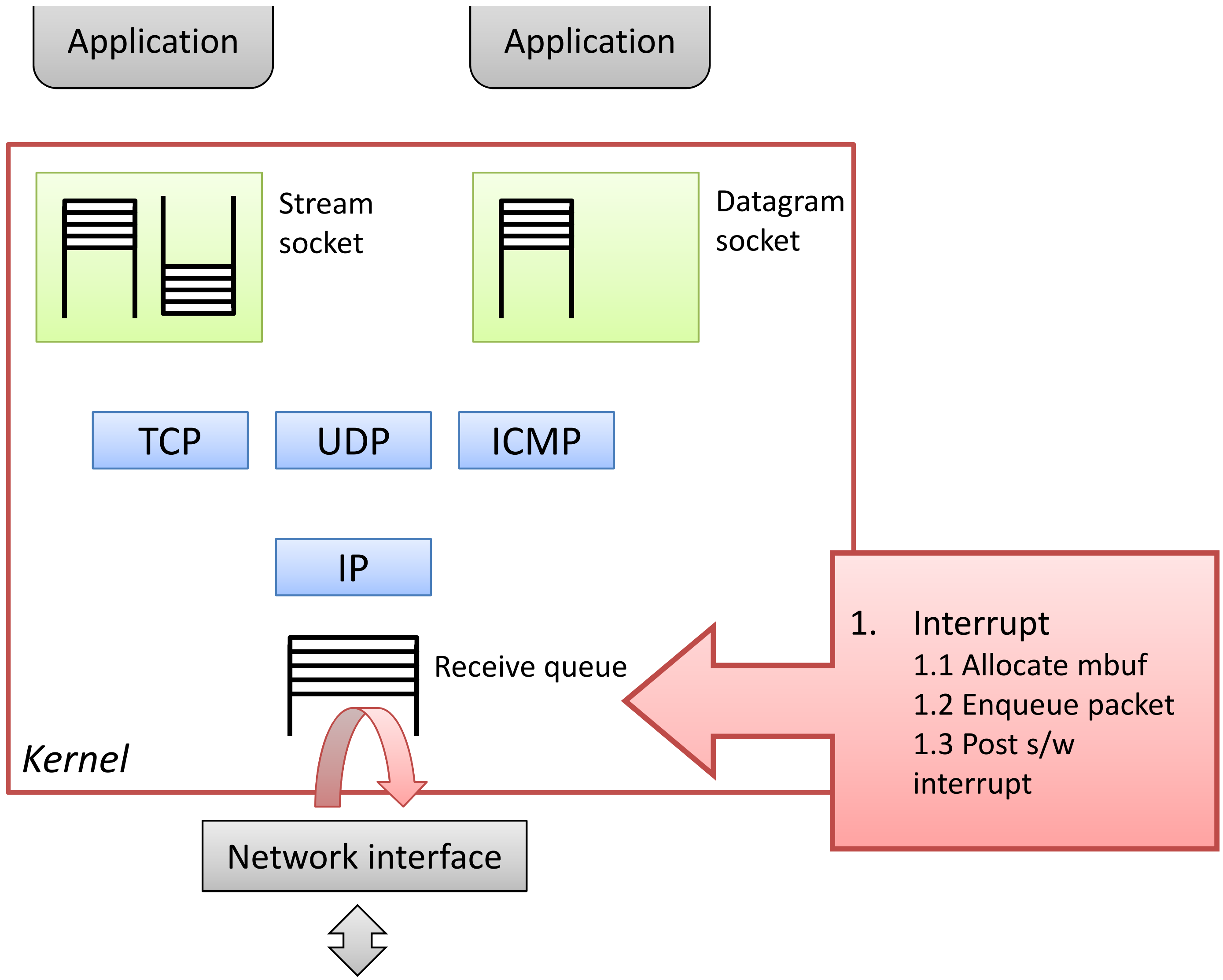
What is the problem?

- Multiplexing / demultiplexing
  - When? Where? How many times?
- Packet scheduling
  - Prioritizing flows, throughput vs. latency
- Buffering
  - Where is the memory coming from?
- Protocol processing
  - E.g. TCP
- Interaction with the rest of the OS
  - Scheduling, memory allocation

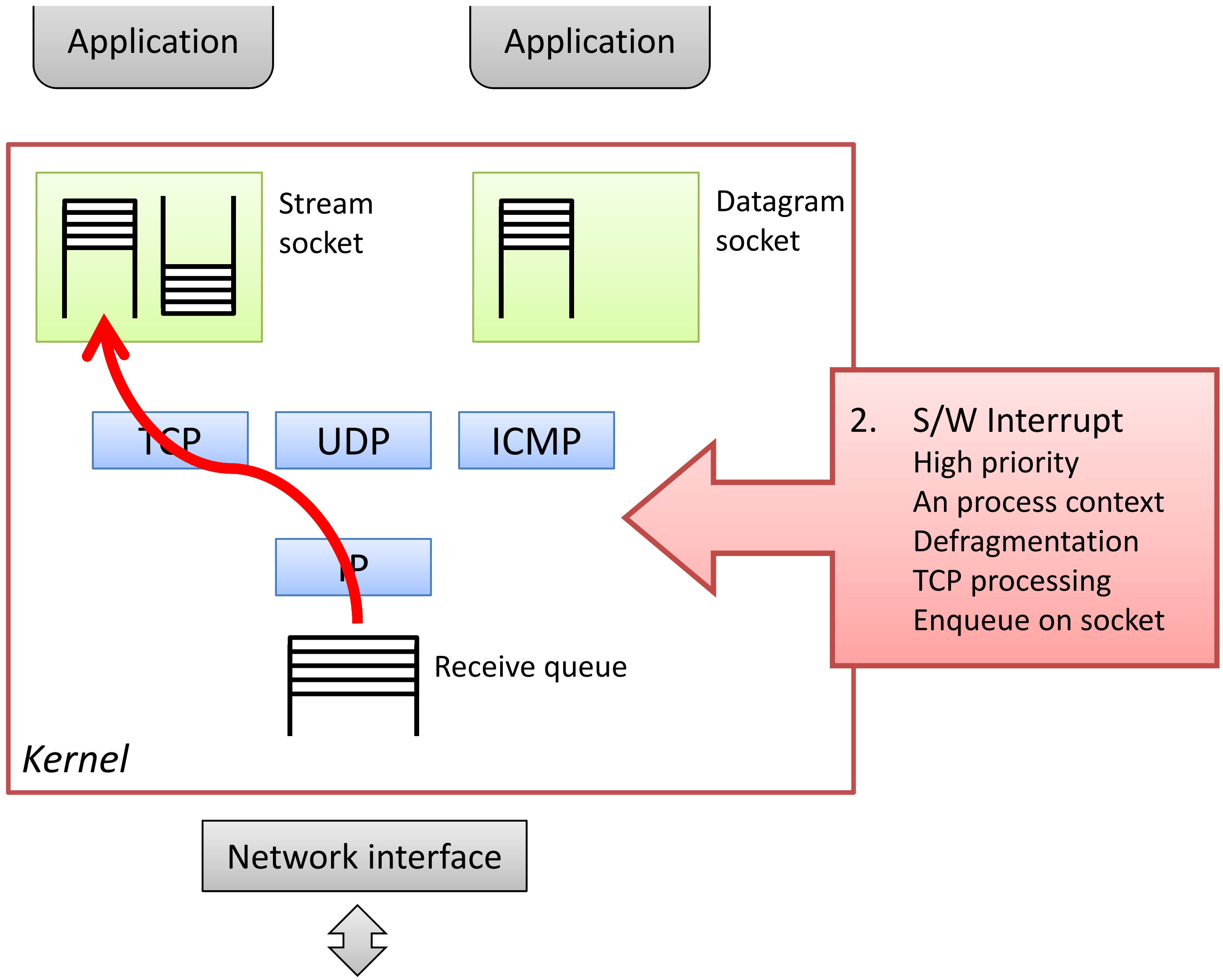
Receiving a packet in BSD



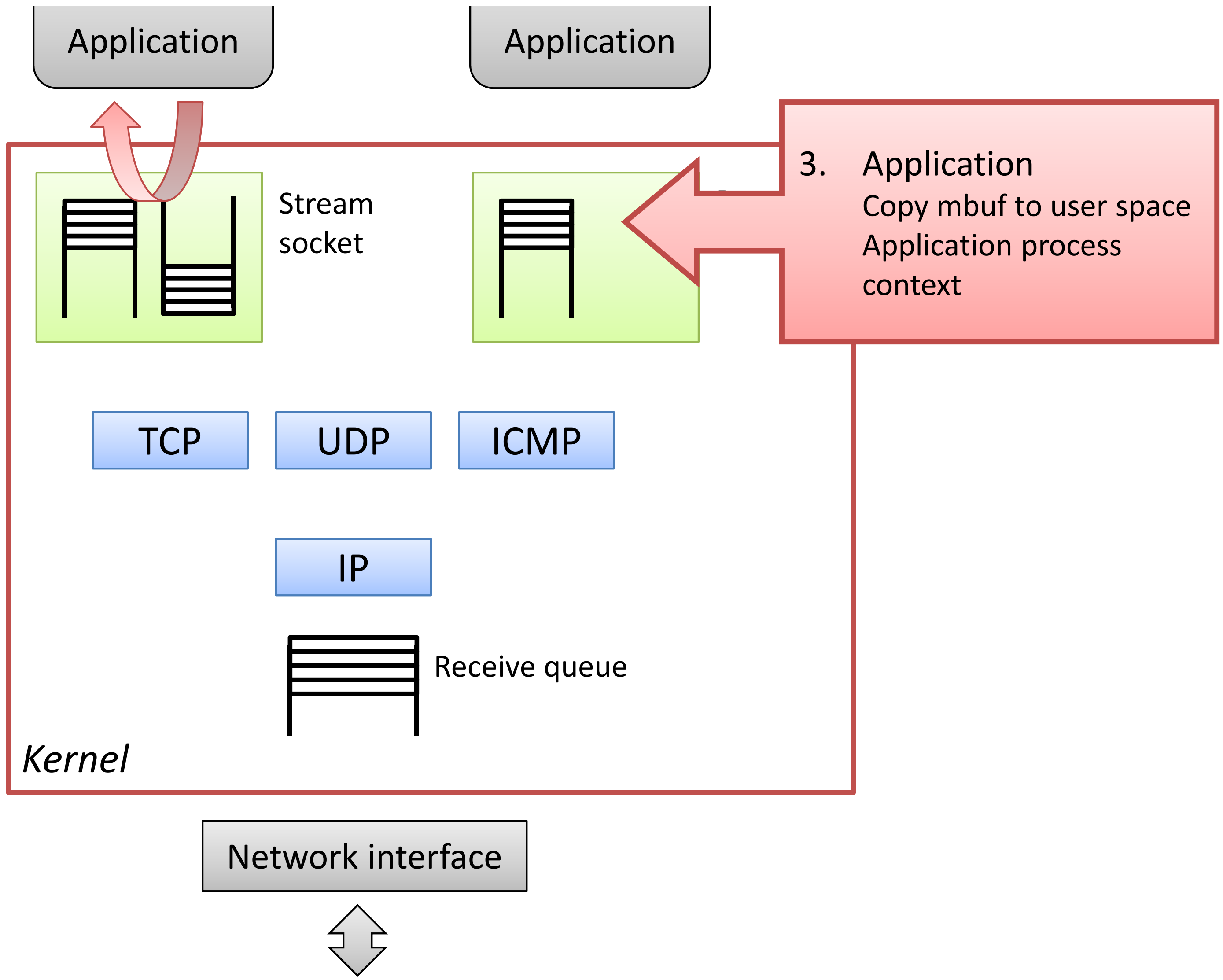
Receiving a packet in BSD



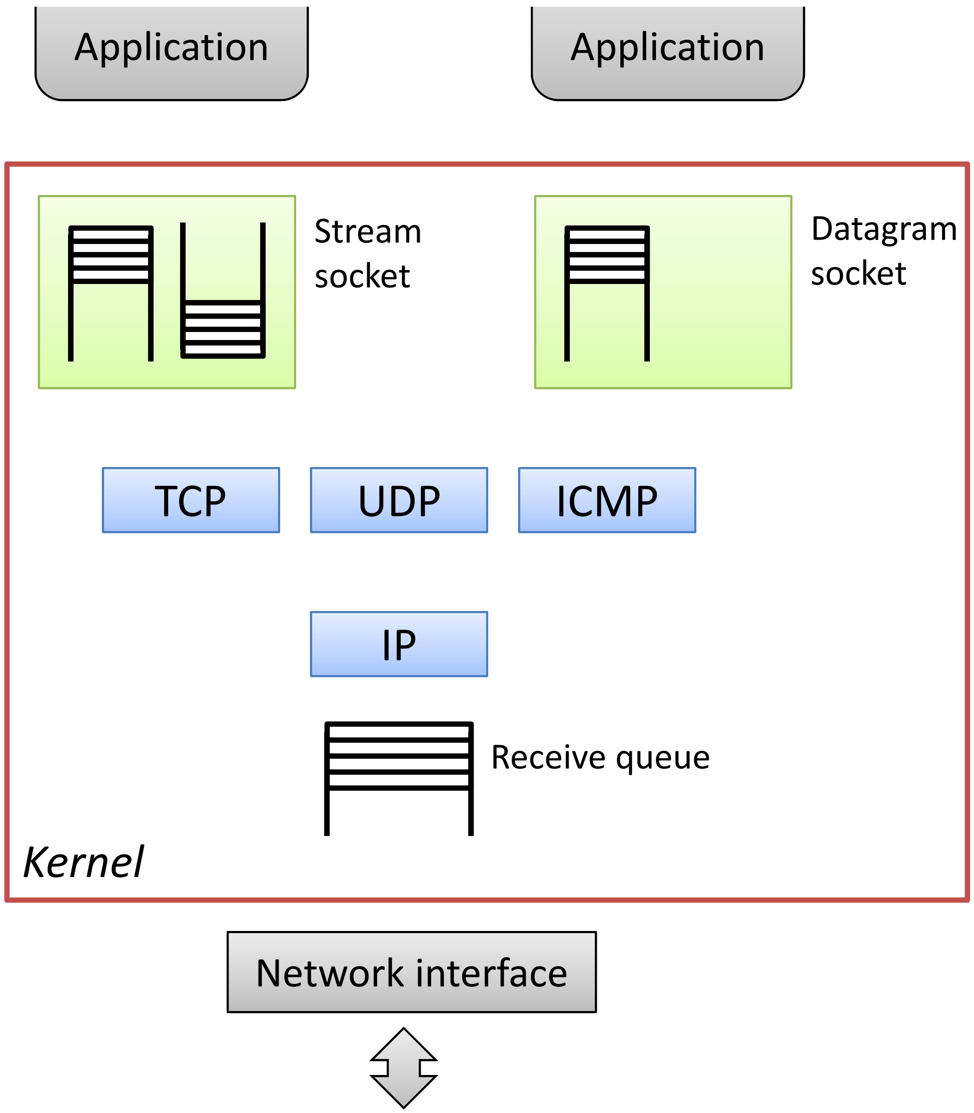
Receiving a packet in BSD



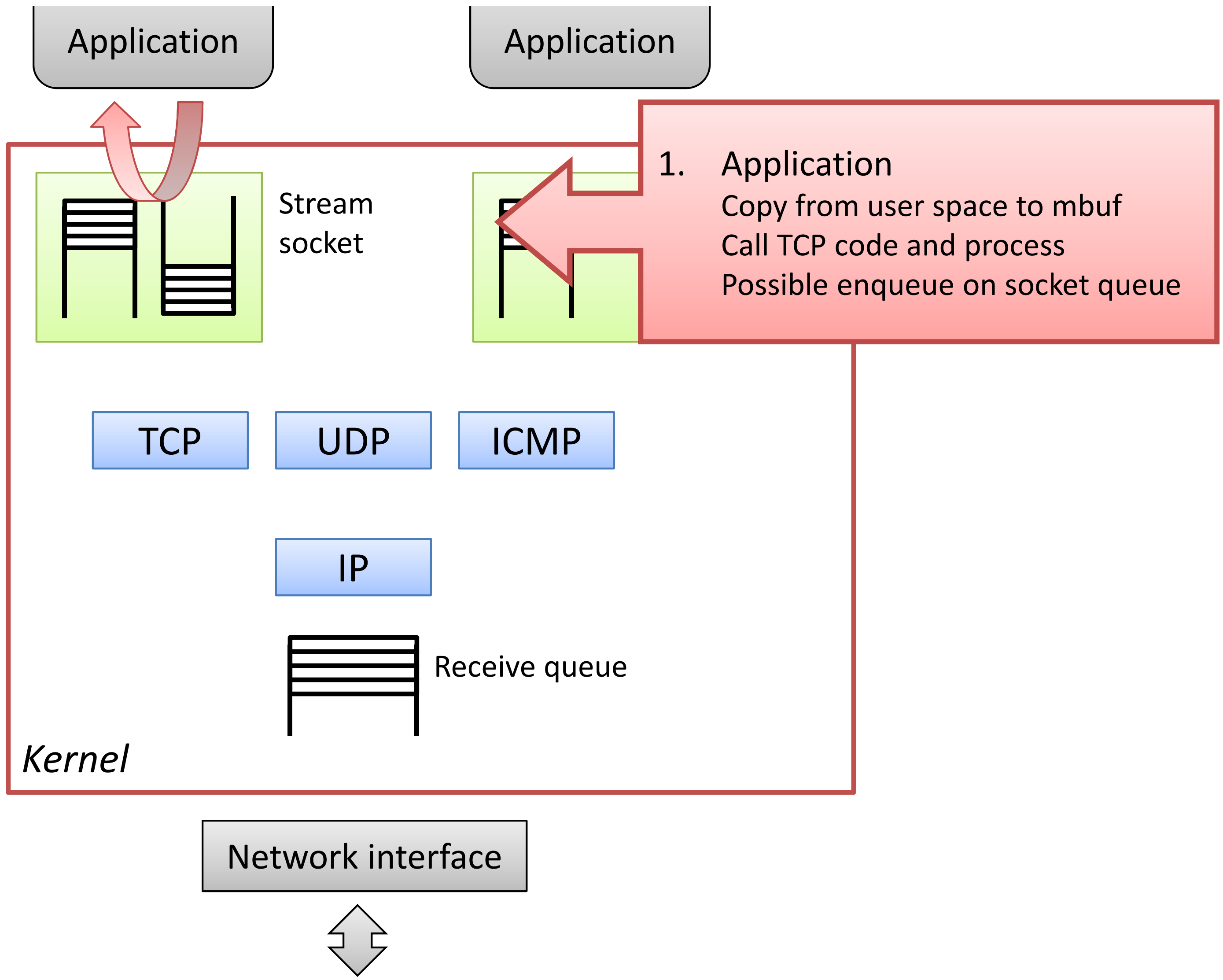
Receiving a packet in BSD



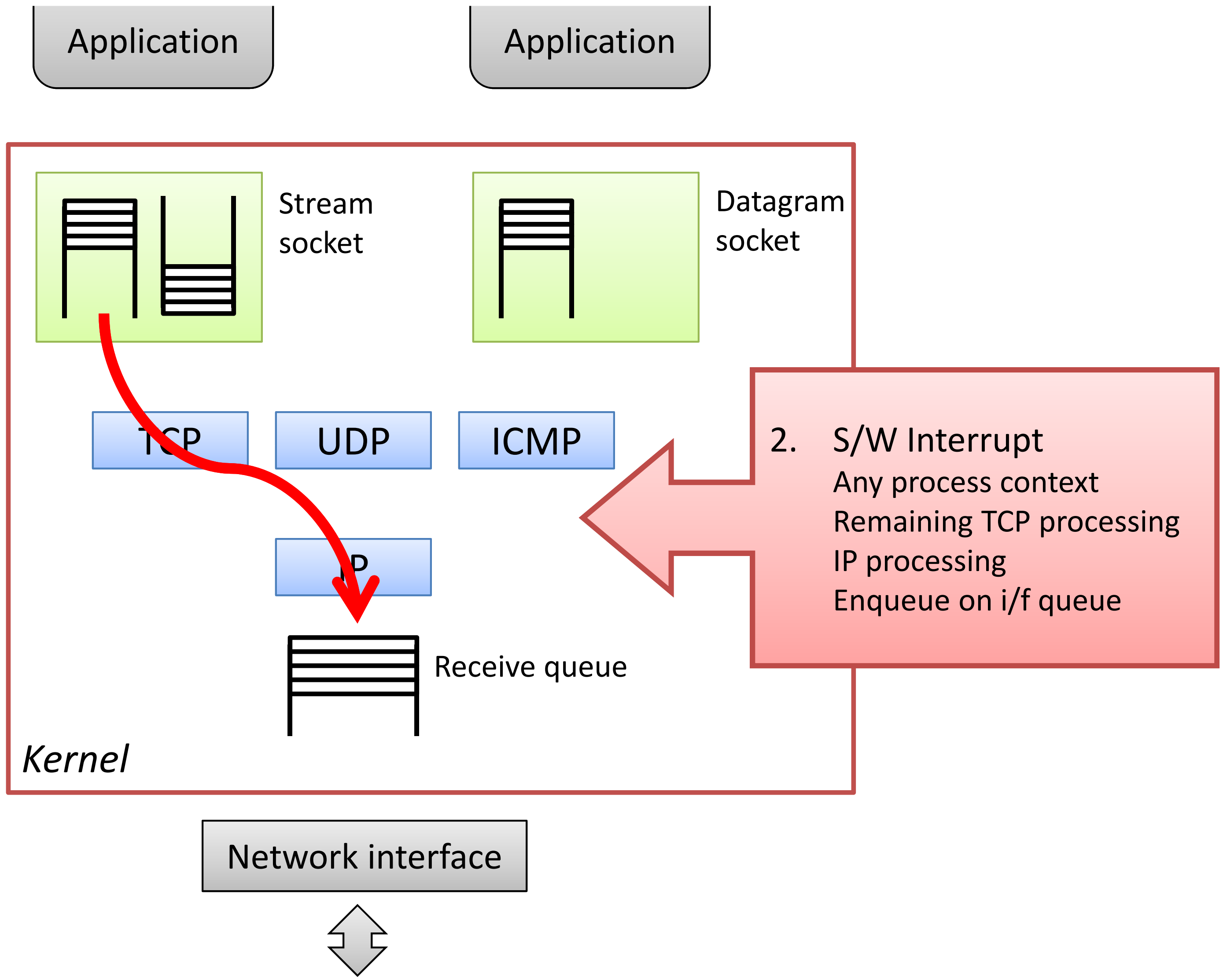
Sending a packet in BSD



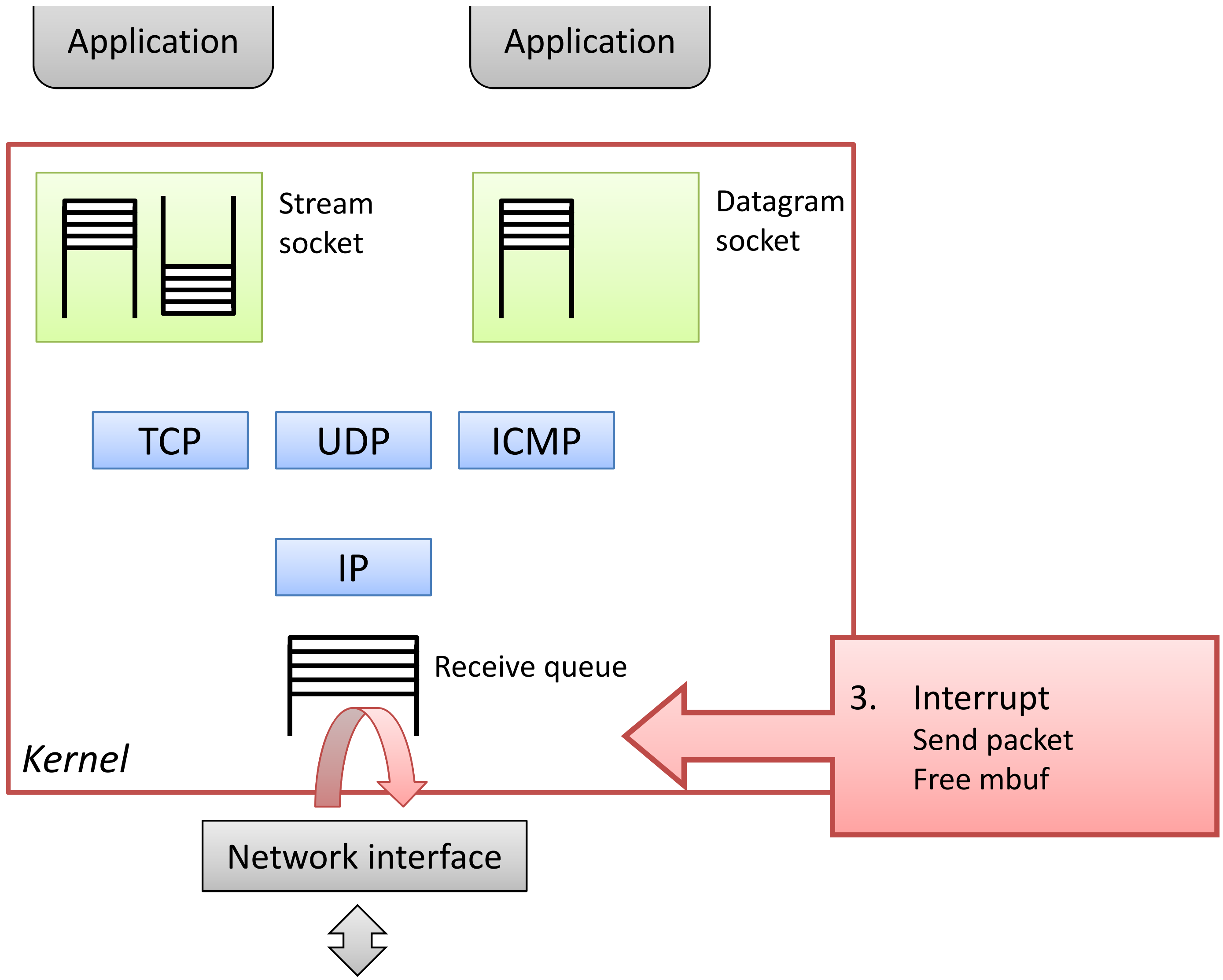
Sending a packet in BSD



Sending a packet in BSD



Sending a packet in BSD



A note on terminology

- In Unix (and most systems):
  - Top half: called from user space (syscalls, etc.)
    - Per-process stack, synchronous traps, etc.
  - Bottom half: hardware and software interrupts
    - Dedicated stack, asynchronous w.r.t. top half, etc.

A note on terminology

- In Unix (and most systems):
  - Top half: called from user space (syscalls, etc.)
    - Per-process stack, synchronous traps, etc.
  - Bottom half: hardware and software interrupts
    - Dedicated stack, asynchronous w.r.t. top half, etc.

- In Linux ☹
  - Top half: hardware interrupts
  - Bottom half: software interrupts

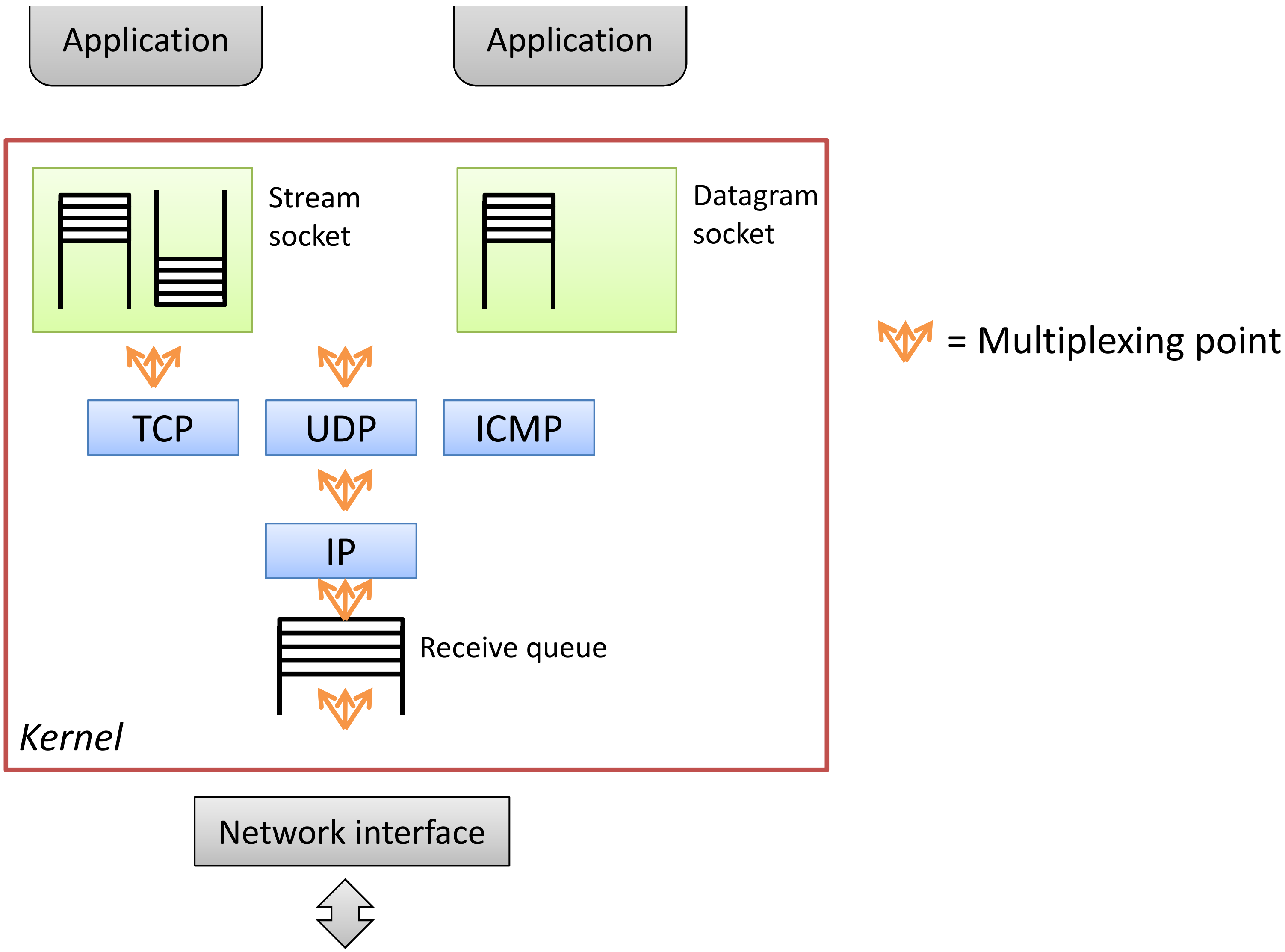
Multiplexing

- Fit traffic streams on a single channel
- Occurs at most levels of IP stack
- *Protocols* specify with headers/encapsulation
- Operating Systems also have to *schedule* messages
  - Which waiting packet do I send now?
- Examples:
  - IP, IPX, AppleTalk, etc. over Ethernet
  - TCP, UDP, ICMP, etc. over IP
  - Multiple TCP connections (ports)
  - ...

Demultiplexing

- Where to direct a packet from a lower layer
  - Lowest level: (physical!) network interface
  - Highest level: socket (for Unix/Windows...)
- Traditional approach: demux at multiple layers
  - IP, TCP, etc.
  - Extreme case: run each layer in own thread/process

Sending a packet in BSD



Layered vs. Early Multiplexing

[Tennenhouse, 1989]

Systems@ETH Zürich

- "Layered Multiplexing Considered Harmful"
- Context: multimedia (soft realtime) networked applications
- Most multiplexing points have no knowledge of:
  - Application-level destination
  - Application-level flow
  - Application-level requirements for Quality of Service
- Result: QoS Crosstalk: applications interfere with each other
  - Easy to demonstrate with live audio/video, and big file transfer as cross traffic.

So why multiplex in layers?

- Advantages:
  - Modularity, simplicity
  - Efficiency: very simple demux functions
  - Low CPU overhead, high utilization
- Context: slow machines, fast networks
- Avoid crosstalk by scheduling at each mux point?
  - Impractical (complexity, slow)
  - Doesn’t work (still can’t distinguish flows)

Alternative: low-level multiplexing

- Multiplex ***low***, multiplex ***once***
- Receive path:
  - Identify application dest of packet as soon as possible
  - Schedule packet processing according to application
- Transmit path:
  - Queue all packet processing separately by application
  - Only multiplex at the point of transmission (NIC or driver)

Lazy Receiver Processing

[Druschel and Banga, 1996]

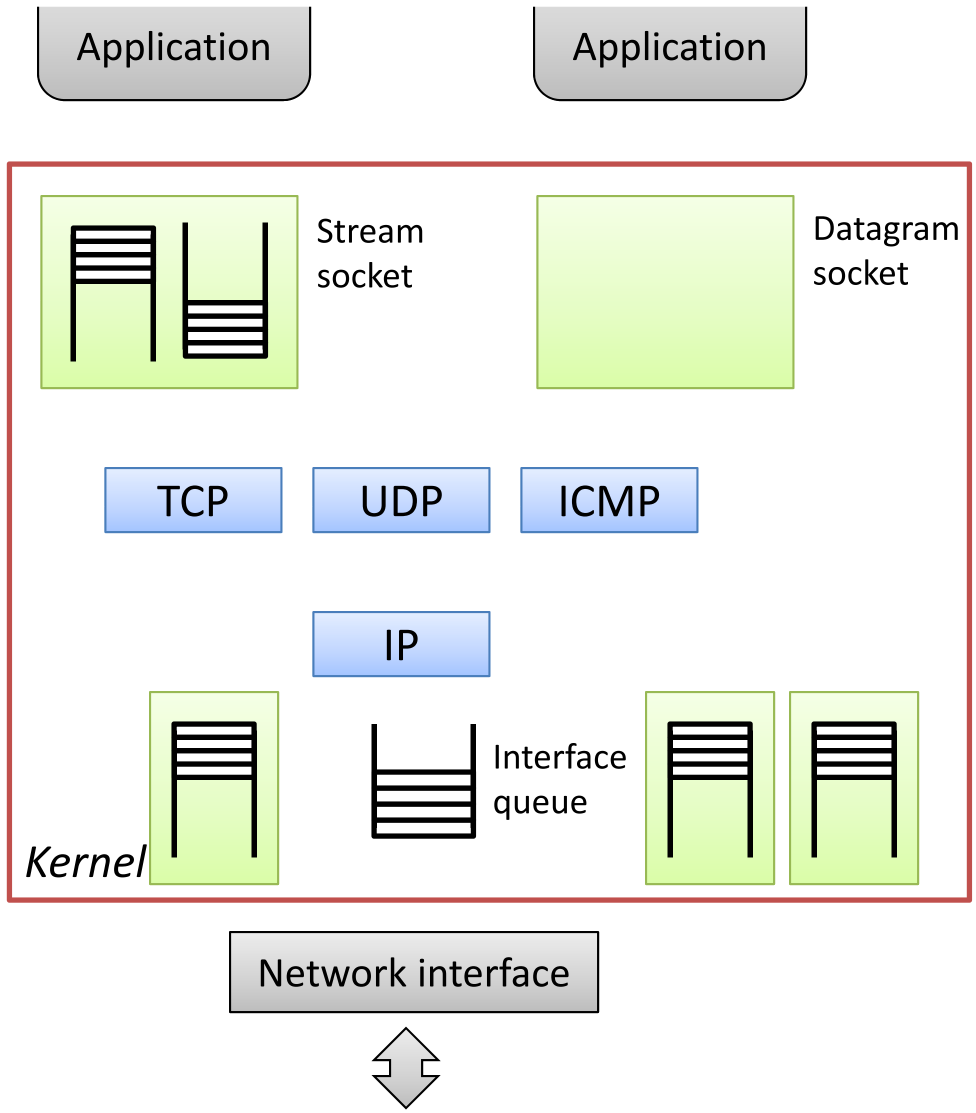
- BSD has "eager receiver processing": priorities are:
  1. capture and store packets in memory
  2. protocol processing of packets
  3. run the application
- No effective load shedding
  – Packet dropping can only happen late
- No traffic separation
  – Consequence of layered multiplexing ...
- Inappropriate resource accounting
  – Which application is doing the work in an interrupt?
  – The network is scheduling the computer!

Lazy Receiver Processing Ideas

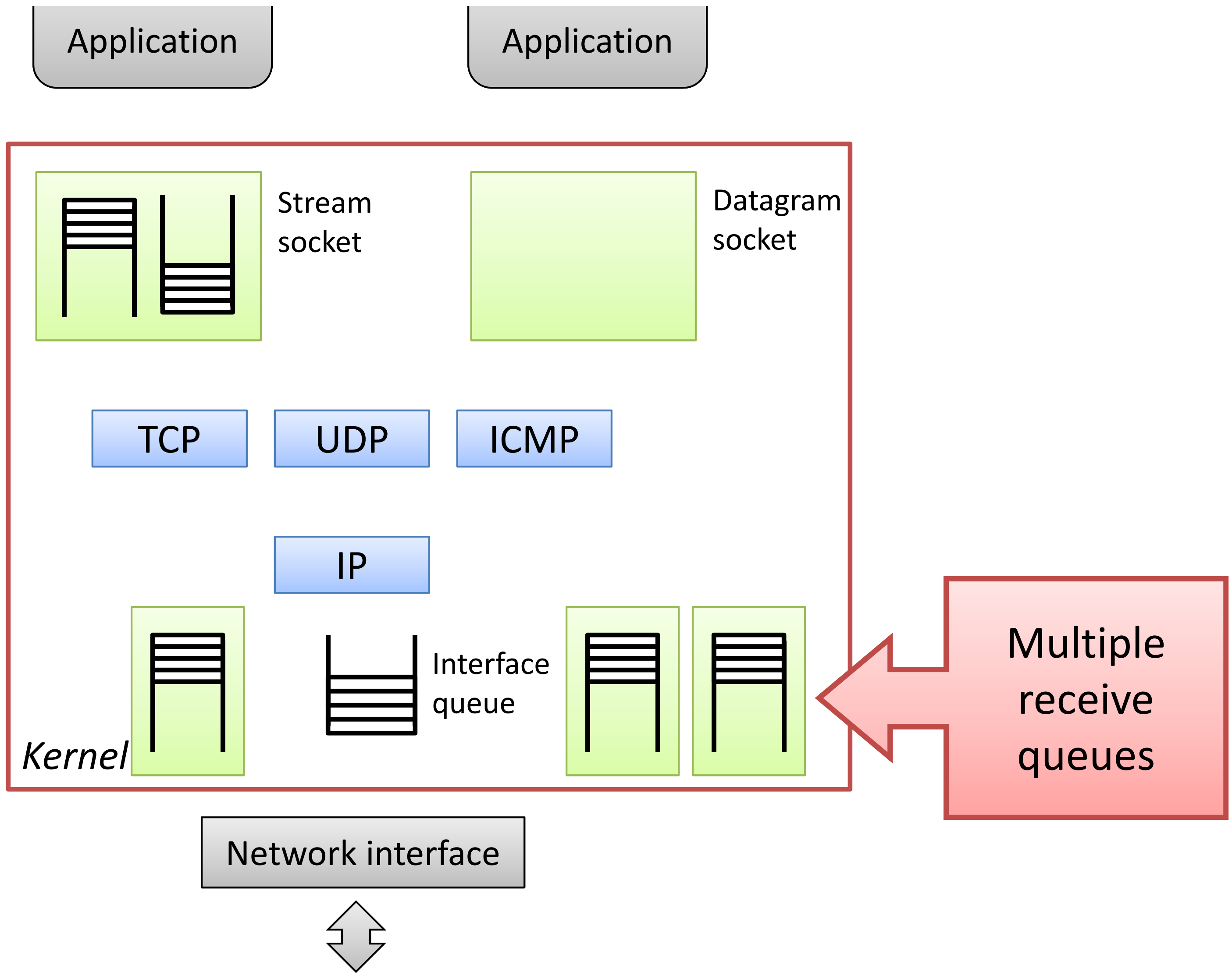
[Druschel and Banga, 1996]

1. Use early demultiplexing
2. Queue incoming packets separately
3. Tail drop on per-socket queues
4. Schedule each packet along with its application
5. Account processing time to each application

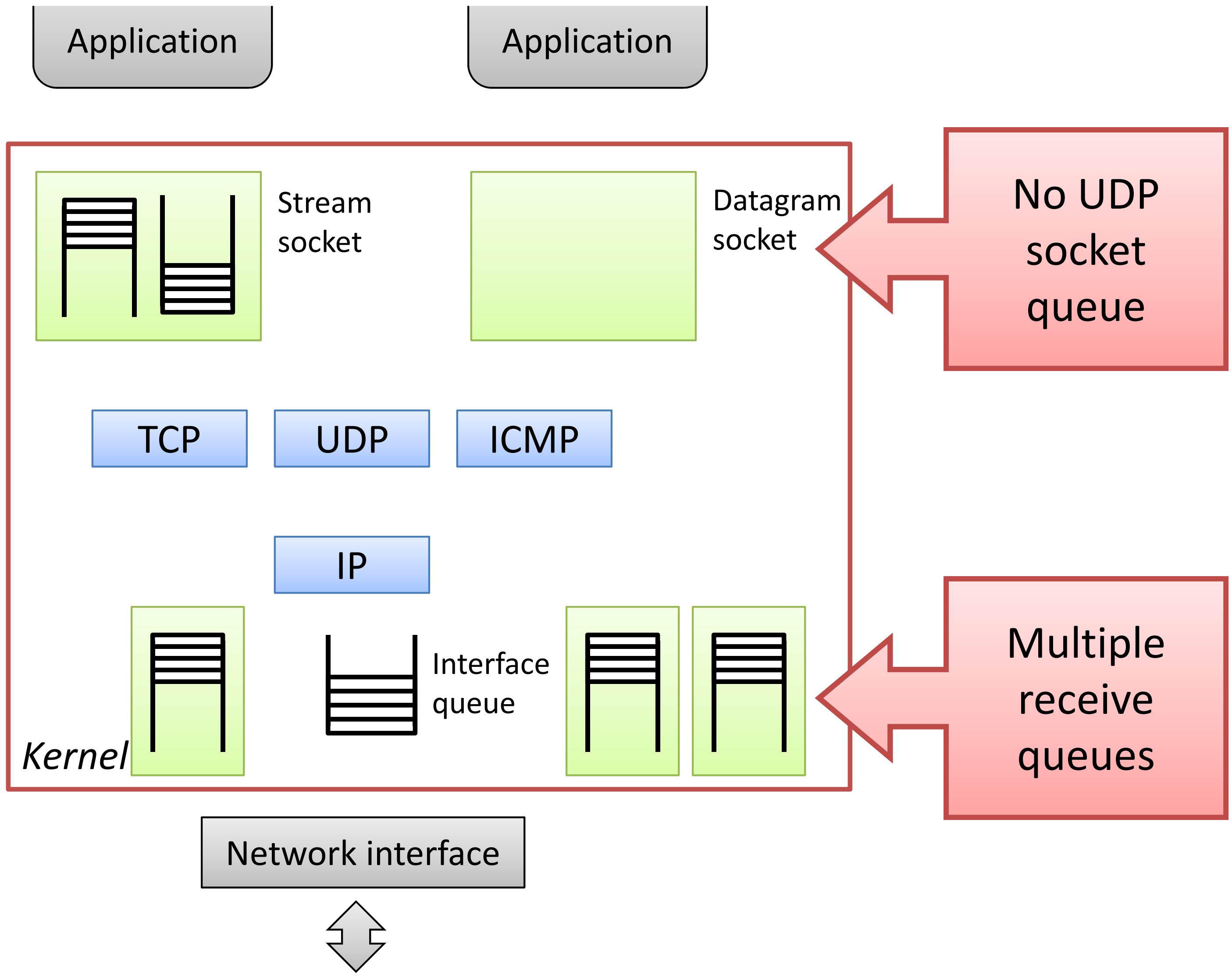
LRP in practice



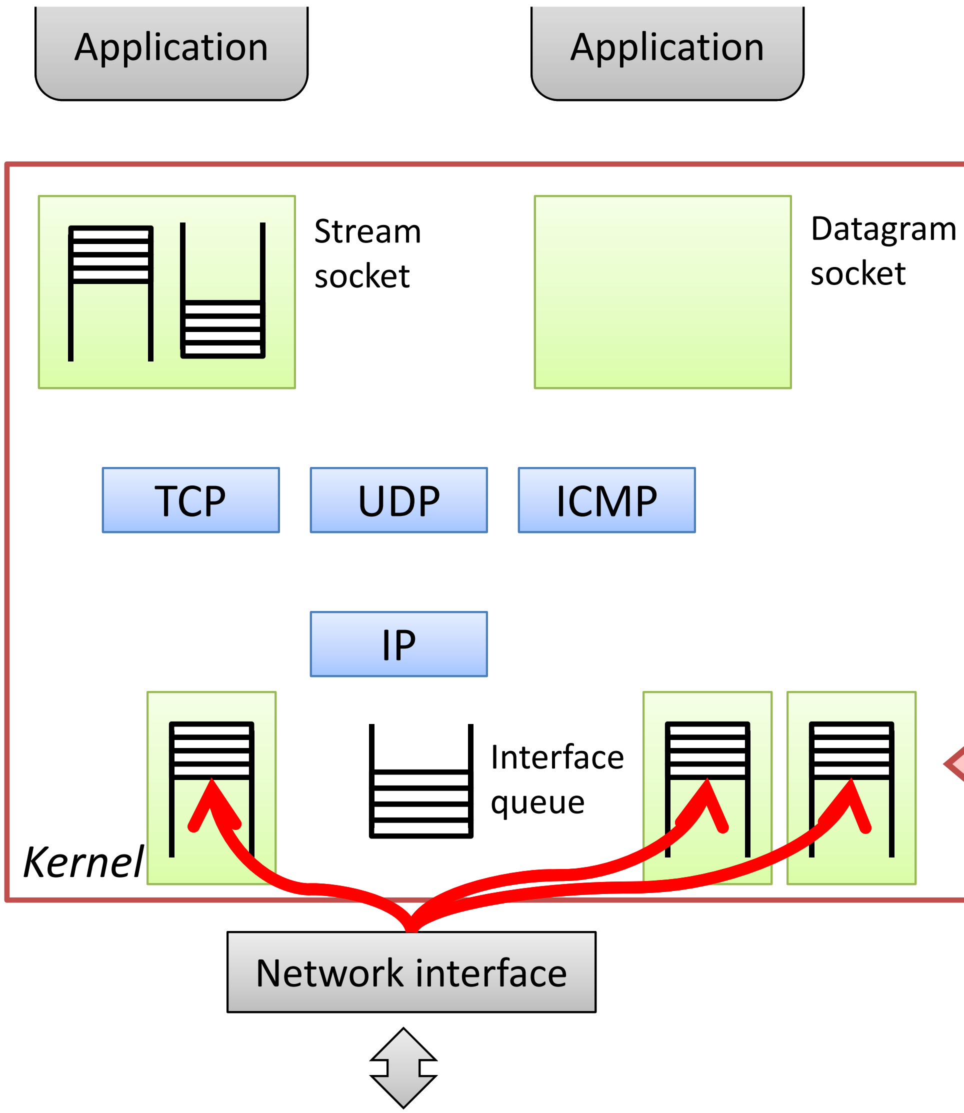
LRP in practice



LRP in practice

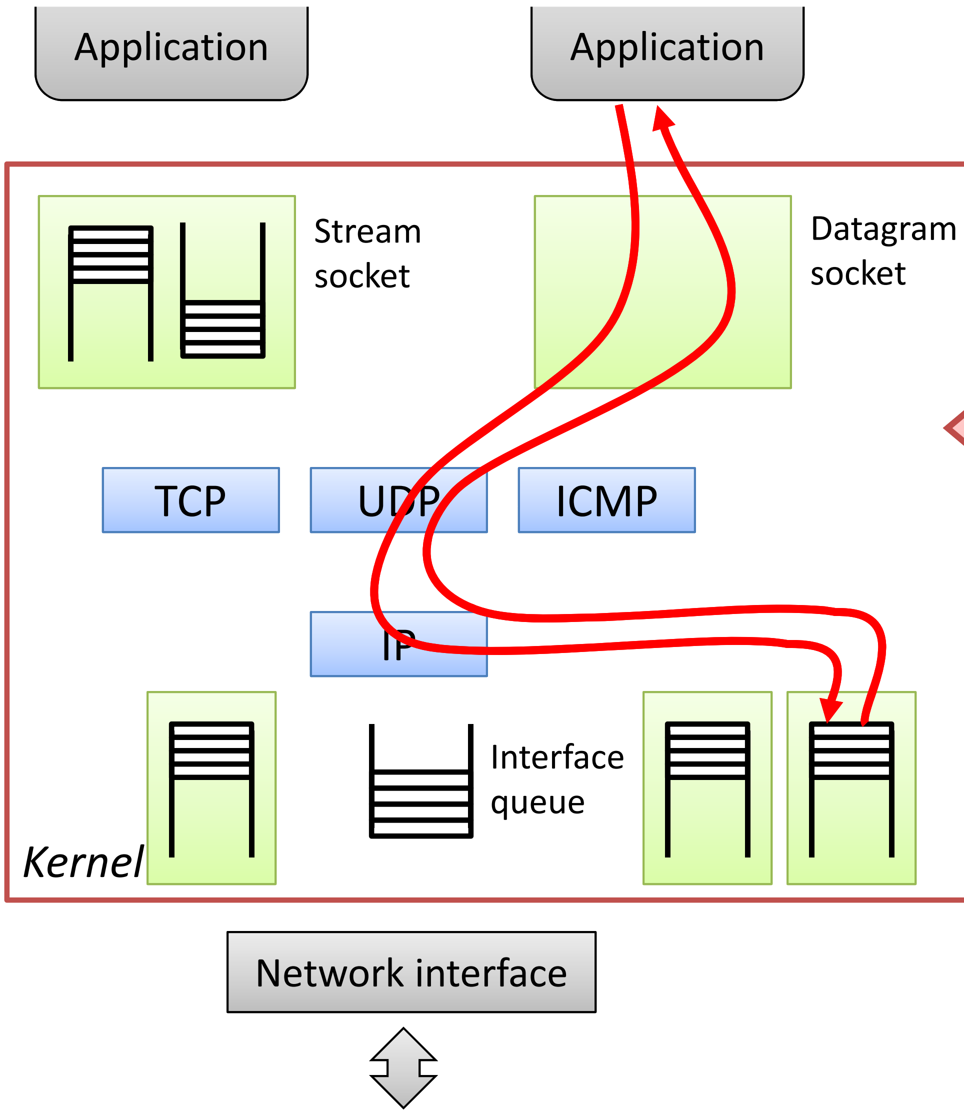


LRP in practice



NIC demuxes
- Once (to a socket)
- Early (before processing)

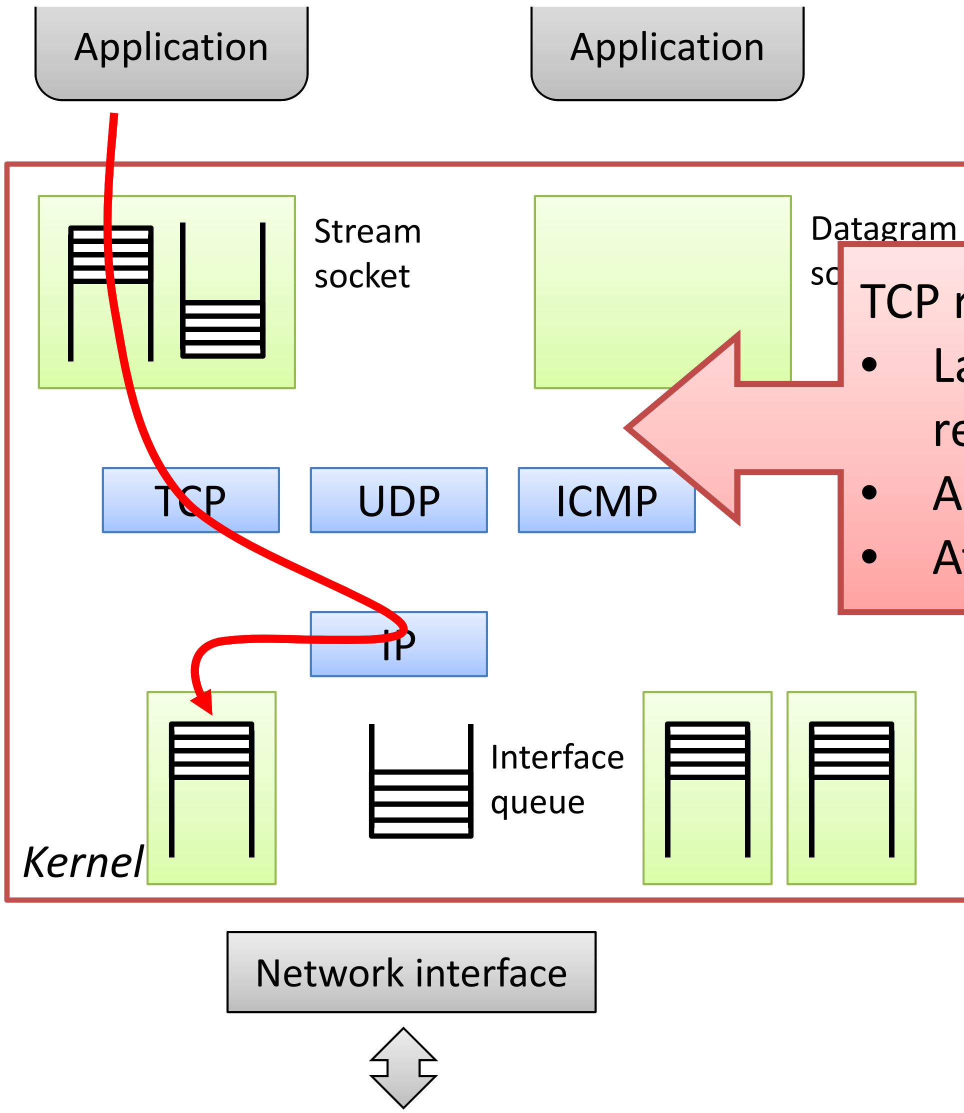
LRP in practice



UDP Processing

- Lazy (when application requests it)

LRP in practice



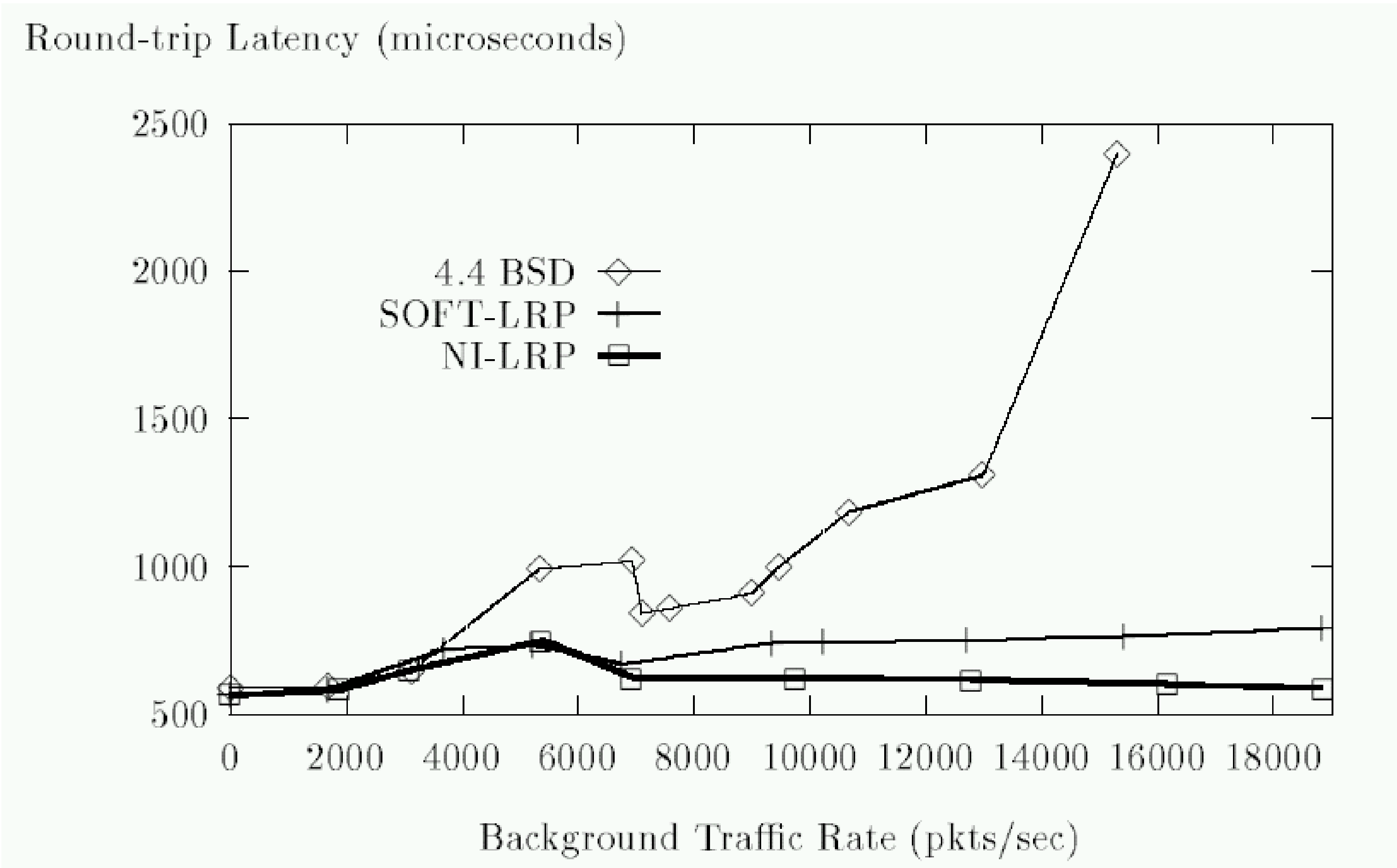
TCP receive

- Lazy (when application requests it)
- Accounted to dest. Process
- At dest. priority

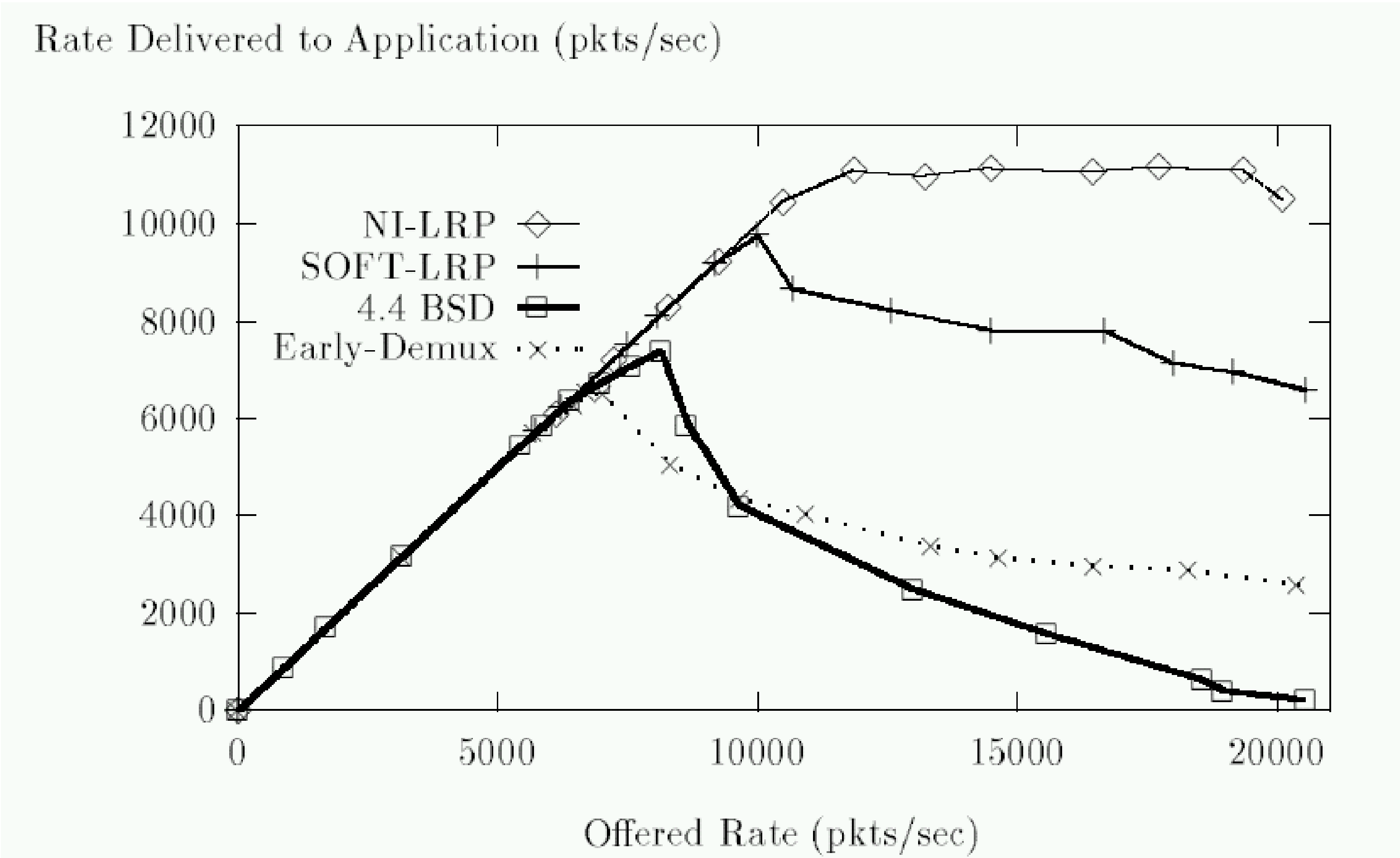
How to demux early?

- [G&B] used ATM: demux on Virtual Circuit Identifiers
  - Most ATMcards had per-VC queues, inc. the SBA-200
  - Requires one TCP connection per circuit
  - Also a good idea, but a story for another time...
- Ethernet cards (mostly) can’t do this...
  - “SOFT-LRP”: demux in interrupt handler
  - Adds latency, which is hard to account for
- See later ...

LRP Isolation



LRP stability



Livelock

- Graph shows ***livelock*** in action
  - As load increases, throughput **decreases**
  - Interrupts and processing overwhelm system
- Early demux alone is insufficient to prevent livelock
  - But helps somewhat under heavy load
- SOFT-LRP delays the onset
- NI-LRP (hardware) almost eliminates it

Eliminating livelock

[Mogul and Ramakrishnan, 1997]

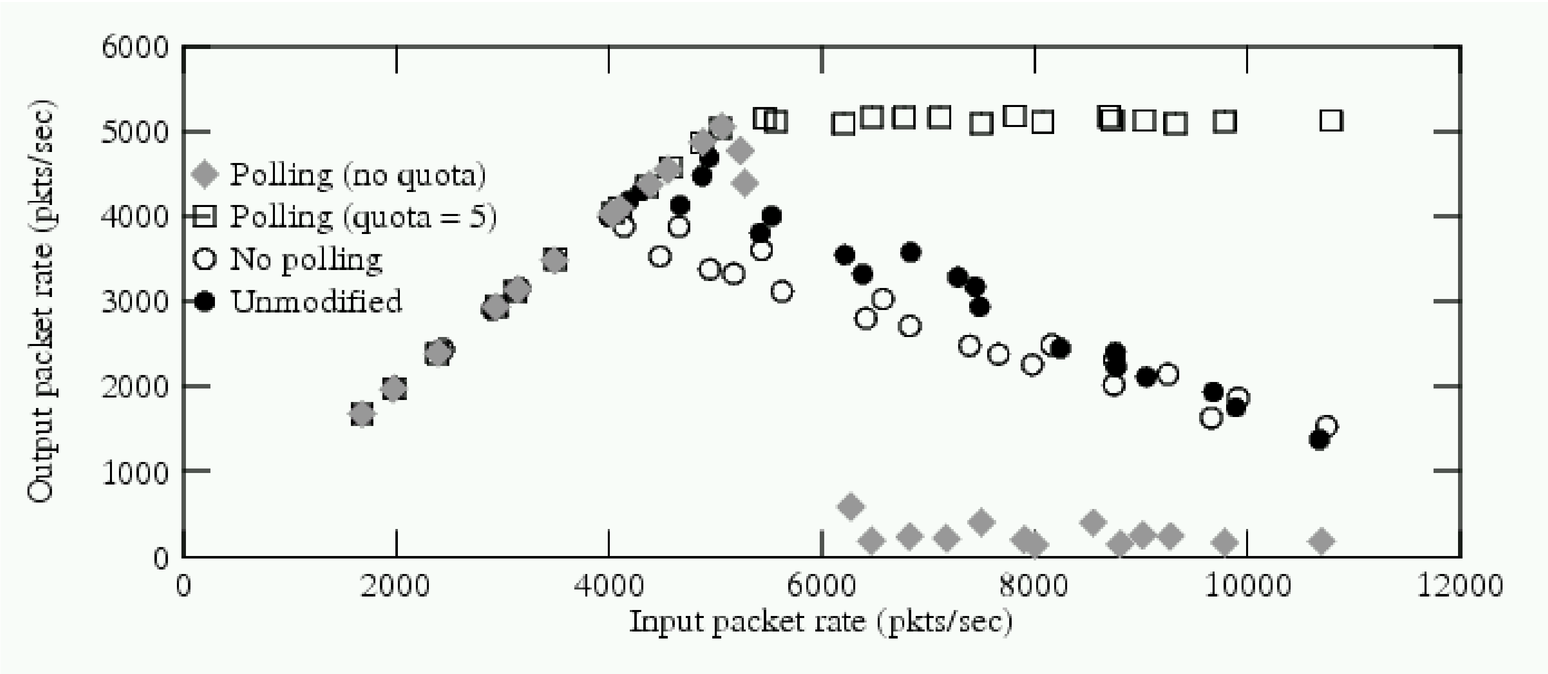
- Particular problem for:
  - Web servers, file servers, etc.
  - Network monitoring applications
  - Host-based routing
  - DoS resilience
- Throughput can drop to almost zero
  - E.g. no time for output processing
- Experience driven: NASDAQ, Election web servers

Approach:

[Mogul and Ramakrishnan, 1997]

- Avoid livelock:
  - Only use interrupts to initiate polling, then disable
  - Round-robin polling for input event sources
  - Schedule packet processing properly
  - Drop packets early
- Maintain performance:
  - Re-enable interrupts when no work pending to keep latency low
  - Buffer bursts in the receiving interface
  - Eliminate the IP input queue

Results



Output packet rate (pkts/sec)
Input packet rate (pkts/sec)
Polling (no quota)
Polling (quota = 5)
No polling
Unmodified
0
1000
2000
3000
4000
5000
6000
0
2000
4000
6000
8000
10000
12000

Software demux: basic idea

- **Packet filters**: old idea for inspecting the network
  - Each filter has an associated socket.
  - When a packet arrives, every filter is run on the packet
  - If the filter “passes” the packet, it’s delivered to the socket
- Not *quite* the same as a demultiplexer ...

Software demux: basic idea

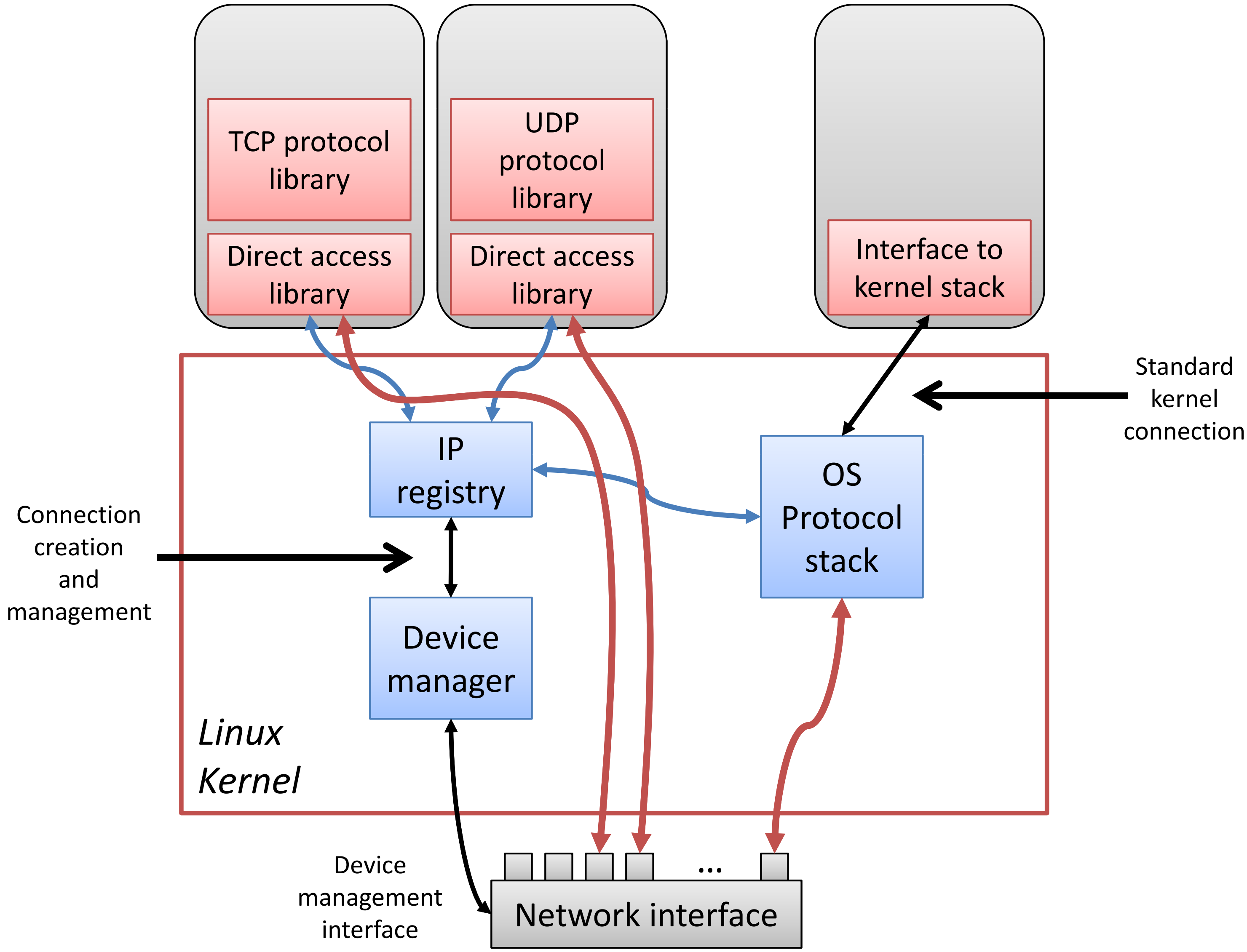
- Installation:
  - Write program in simple, high-level language
  - Compile to simple byte code (no jumps, etc.)
  - Hand to the kernel
  - Kernel installs
- Issues:
  - Slower than hard-coded demux (e.g. Linux main stack)
  - Or is it? c.f. DPF: efficient compilation of multiple filters

Hardware demux: Arsenic

[Pratt and Fraser, 2001]

- Early example of a User-Accessible network interface
  - for connectionless packet networks
  - c.f. UNet, VIA, RDMA, etc. (circuits are easy :-)
- Hardware can present as many devices
  - One per user process $\Rightarrow$ can map into userspace
  - “User-safe device”
- Kernel creates flows, installs filters
  - Run packet filters on the card
  - Compiled from HLL into MIPS machine code
  - Data DMAs direct into user-space buffers
- Original OS protocol stack used for “everything else”
  - Permanent default filter $\Rightarrow$ OS stack

Arsenic architecture



Direct access library

Systems@ETH Zürich

- Direct Access Library
  - Creates “connections” by calling protocol registry
  - Registers *(virtual address, length) tuples with kernel*
  - Supples NIC with empty receive buffers & descriptors
  - Fills transmit buffers & descriptors
- Device Manager
  - Fields NIC interrupts, translates to user-space signals
  - Creates and tears down “connections”
  - Compiles and installs filters on the NIC

Critique: why sockets?

- Amost the only OS Network API in widespread use
- Despite this, many criticisms:
  - No data placement control: lots of copies, data can't be aligned
  - No queuing: only one outstanding read request at a time
  - Poor asynchronous communication (upcalls): see dispatch!
  - TCP is completely abstracted by the API: can't tweak or look inside
- Some of these can be worked around
  - e.g. **`setsockopt`** or **`ioctl`**
- Data placement / queueing remains problematic

Structural approaches: Nemesis

[Black et al., 1997]

- Observation: Considerable complexity introduced to a legacy kernel.
- Ground-up reimplementation considerably simpler.
  - Nemesis put all protocol processing in application
  - ATM hardware or BPF software for demultiplexing
  - Per-application outbound filters for sending packets (transmit multiplexing)
  - Out-of-band servers for connection setup/teardown
- Much better QoS isolation

Exokernel

[Ganger et al., 2002]

- Similar to Nemesis, but demonstrated benefits of collapsing application and network stack together
- “Cheetah” web server used its own TCP stack, avoiding sockets
  - Merged TCP and HTTP processing.
  - Merged TCP retransmission pool and file cache!
  - “Knowledge-Based Packet Merging”, e.g. setting FIN on last data packet, delayed ACK for HTTP requests.
  - Precomputing packet checksums on disk files

Exokernel

[Ganger et al., 2002]

- Other example: Webswamp (needed to benchmark Cheetah!)
- Web load generator using 3 specializations:
  - Count-Only Data Reception (don't bother looking at received data)
  - Skipping Checksum Verification (fine for testing)
  - Knowledge-Based Packet Merging, again: don't ACK the server's SYN, discard the final ACK/FIN bit.
- Throughput increased by 5 for small documents

Scout

[Mosberger and Peterson, 1996]

Systems@ETH Zürich

- Very different attempt to deliver QoS & Isolation in the network stack
- **Paths** abstract data flow in the application
  - Across protection domains, processes, etc.
  - Comprised of processing with queues at each end
- Scheduler schedules **paths**, not **processes**
- Extends many ideas from classical networking into the OS
  - Queueing theory and techniques
  - Routing and circuits

What happened to this?

- Multiple queues for high-speed NICs are now commonplace
  - Though usually not enough yet for individual flows
- Receive-Side Scaling (RSS):
  - Multiple queues to exploit MP parallelism for scaling
- Interrupt coalescing is standard
- RDMA:
  - Circuit-based early demux
  - User-level signalling (bypasses kernel)
  - Surprising: setup overhead dominates
  - Niche market: HPC clusters

So what now?

- First, note:
  - L4 **completely ignores** networking!
  - Protocol stack is just another server process
- And Barrelfish?
  - Inside of the computer resembles a network, now, too
    $\Rightarrow$ NIC actually functions more as a **router/firewall**
  - But: must also parallelize flow processing!
  - Probably combination of:
    - Scout
    - RouteBricks [Dobrescu et al., 2009].

References

- Black, R., Barham, P. T., Donnelly, A., and Stratford, N. (1997). **Protocol implementation in a vertically structured operating system**. In LCN ’97: Proceedings of the 22nd Annual IEEE Conference on Local Computer Networks, pages 179–188, Washington, DC, USA. IEEE Computer Society.
- Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. (2009). **Routebricks: exploiting parallelism to scale software routers**. In SOSP ’09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pages 15–28, New York, NY, USA. ACM.
- Druschel, P. and Banga, G. (1996). **Lazy receiver processing (lrp): a network subsystem architecture for server systems**. In OSDI ’96: Proceedings of the second USENIX symposium on Operating Systems Design and Implementation, pages 261–275, New York, NY, USA. ACM.
- Ganger, G. R., Engler, D. R., Kaashoek, M. F., Hector M. Brice n., Hunt, R., and Pinckney, T. (2002). **Fast and flexible application-level networking on exokernel systems**. ACM Trans. Comput. Syst., 20(1):49–83.

References

- Mogul, J. C. and Ramakrishnan, K. K. (1997). **Eliminating receive livelock in an interrupt-driven kernel**. ACM Trans. Comput. Syst., 15(3):217–252.
- Mosberger, D. and Peterson, L. L. (1996). **Making paths explicit in the Scout operating system**. SIGOPS Oper. Syst. Rev., 30(SI):153–167.
- Pratt, I. and Fraser, K. (2001). **Arsenic: A user-accessible gigabit ethernet interface**. In Proceedings of INFOCOM2001, pages 67–76. IEEE Computer Society.
- Tennenhouse, D. L. (1989). **Layered multiplexing considered harmful**. In Proceedings of the 1st InternationalWorkshop on High-Speed Networks.