Advanced Computer Networks

Andreas Biri, D-ITET 09.07.17

1. Networking

Demands on the network:
- Performance: latency, bandwidth
- Reliability, availability, security
- Operator: flexibility, manageability

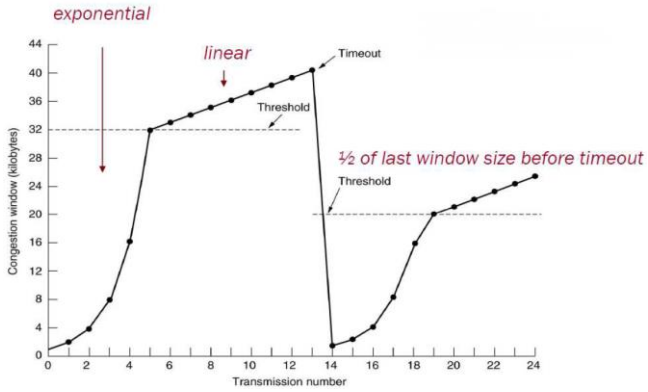
Latency only has small influence on big files
- large impact on transfer time for small files
- higher bandwidth, the more data is lost if the channel breaks down & buffers need to be increased drastically

Transmission Control Protocol (TCP)

- connection-oriented, stream-oriented (sequ. numbers)
- acknowledgments & retransmissions
- sliding window protocol & *maximum message size* (MMS)
- *slow start:* start with window size 1 and increase

Flow control: avoid overflow at receiver (sliding window)
Congestion control: avoid overflow at routers (ACKs)

Additive Increase / Multiplicative Decrease (AIMD)



Bandwidth-Delay product: "data currently in flight"
- data inside network, sent but not yet acknowledged

TCP Incast

In DC, request result in a scatter-gather traffic pattern
→ all returning responses (e.g. from storage servers) are synchronized and return simultaneously, overloading the switch buffers and resulting in dropped packages
→ packets will only get retransmitted after the Retransmission Timeout (RTO) of $200ms$ has passed
- especially influences systems which require all answers to continue its calculations (distributed storage, web search)

Networking demands

Flow-completion time: very important for the user (fast loading times), but mostly disregarded by operator
- usually, tries to optimize maximal throughput, link utilization & fairness (important to operator)

Flow fairness: allocate same rates to all the flows
- but: if I want more, I just add another simultaneous flow
- long flows (over many routers) lead to more congestion, but receive equal rate

End-to-end argument: logic should be outside the network
- if possible at end (e.g. top layer, end node), don't put complexity inside the network if application specific
- complexity of network is shared by everyone, even if we might not need it → only nodes that need should pay costs
- mostly, full functionality can only be achieved if they are done over entire path, therefore don't do it in between ("end-to-end check must be implemented anyway")

Fate-sharing principle: "Acceptable to lose state about entity, if at the same time entity itself is lost"
- "If entity dies, what's the point of knowing it's state?"
- example of end-to-end argument: store state at entities
- Transport level sync on host can be lost if host disconnect
- BGP: link failure leads to termination of announcements, and therefore the state itself is lost

Modularity & layering: each application should choose the setting best suited for its usage, no "general solution"

2. Data center (DC)

Scatter-gather traffic pattern

One requests starts a cascade of multiple requests, each with a very short response deadline (10ms)
Responses will be gathered and then returned to client
- hundreds of *memcache* fetches per request

Big Data analysis

- Hadoop, Spark, Database joins with lot of internal traffic
- 3 V's : **V**olume, **V**ariety, **V**elocity

Measurements

- gather header information at top-of-rack switch
- *Port mirroring:* get entire packages
- Log everything is not feasible, as too much
  → use Bloom filter & hashing to compress data
- Sampling: set criteria, e.g. "dump flow if above X"
  → might miss problems
- Replay scenario with logging ON
  → for "Heisenbugs", cannot figure them out this way

→ concentrate on bugs & reproduce them to dig down

What do we do if we have the data?
- Neutral networks to learn & predict traffic for TE
- Otherwise, not many applications

Traffic characteristics

DC: "machine to machine" traffic is several orders of magnitude larger than what goes out to the Internet

Disaggregated DC: requires extremely large bandwidth
- allows *Hardware as a Service* (HaaS) ("get what you need")

Depends very much on application, scale, network design
- FB: lot of intra-Cluster traffic, cache intra-DC
- Google: entirely non-local traffic
  → better load-sharing & reliability

Implications for networking
- Data center internal traffic is BIG
- Tight deadlines for network I/O (only small delays allowed)
  → if response takes too long, make another one
- Congestion & TCP incast: original TCP doesn't work well
- Network shared by applications with different objectives
  → SDN to adjust access rates, e.g. slow memory access
  → Fragmentation: solve problem with virtual machines
- centralized control at the flow level may be difficult
  → distributed control with centralized thinking (SDN)

DC size & location

Can have mega DC & small ones at multiple sites
- Mega: good for bulk data, easy management & costs
- Small: reliability (no single-point-of-failure),
  better latency, privacy
- Hybrid: answer part at small ones, heavy-hitter externally

High throughput networks
- want low latency at the same time
- support big data analysis / MapReduce tasks
- ease virtual machine placement
- freedom, easier management (don't worry about load)

Top-of-rack switch: *rearrangeably non-blocking*
- allow full BW between all ports of the rack

"Scaling up" vs. "scaling out"
- scaling up: *increase* switch capacity, new hardware
- scaling out: buy *more* switches, distribute load

"Big switch" approach: one piece of HW
- easy setup & maintenance
- very expensive, maybe not feasible
- single point-of-failure
- not well manageable or scalable

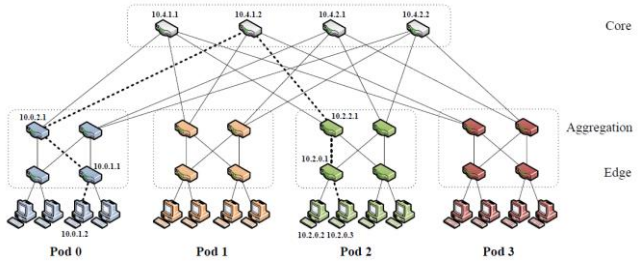
Distributed network approach: commodity HW
- use known good network structures
- existing small & cheap elements to build a large network

But how to distribute over the network?
- minimal latency, move only small data: keep rack local
- large data-set, latency unimportant: use load distribution to use entire network & spread the work

Oversubscription: hosts have reduced connectivity to hosts connected on other switches because of BW limitations

Hypercube: use cube of dimension d
- $degree(node) = d$
- $\#\ nodes = 2^d$
- $diameter = d$

Fat-tree network
- communication with any other host at full bandwidth (*rearrangeably non-blocking:* path set exists for full BW)
- leverage commodity hardware for cost-efficient solution
- backward compatibility to existing protocols



k-ary fat-tree: k-port switches and k pods

Core: $(k/2)^2$, each connected to $k/2$ aggr. sw
Aggregation: $k^2/2$, $k/2$ per pod
Edge: $k^2/2$, each connected to $k/2$ hosts

Hosts: $k^3/4$, each $(k/2)^2$ ECMP to any other

Furthermore, need to spread load & simplify routing
- e.g. specific IP address distribution:

$$10.podNr.switchNr.hostNr$$

Equal-Cost Multipath (ECMP)
- static load splitting among flows
- currently, implementations limited to 8-16 paths

Influence of path length
- packet travelling a short path consumes less capacity

$$\#\ flows * througput\ per\ flow * mean\ path\ length \leq total\ capacity$$

$$throughput\ per\ flow \leq \frac{total\ capacity}{\#\ flows * mean\ path\ length}$$

Could not use best-known degree-diameter graph
- lack of flexibility: need *identica*l switches
  no possibility to extend & increment

Jellyfish

Choose one random graph which connects top-of-rack switches with each other

Achieves close to optimal performances (!)
- on average, creates small mean path lengths

2.2 Data center routing

Routing: tells you *possible paths*
Forwarding: tells you how to *utilize those paths*

- exploit structure in the network
- incorporate routing information in node names
- store information in routing tables
- store maps of the network on devices

ARP: bad scalability (large tables, lots of traffic)

Methods to avoid loops
- spanning tree creation
- hop count to stop packets after certain time
- reverse path forwarding (RPF) check

However, the **spanning tree protocol (STP)** doesn't work:
- results in low throughput after horizontal scaling
- partition is easily possible (disconnected network)
- reduce bisectional bandwidth (not full BW anymore)
- throw away perfectly good links for nothing

Transparent Interconnection of Lots of Links (TRILL)

Link-state protocol between switches using all available links (store maps of the network at endpoints)
- layer 2 (requires no setup or IP addresses)
- source learning ("Where is unknown source from?")
- link-state protocol (each node first learns entire network)

First switch encountering package *encapsulates* it and sends it to correct egress switch, which *decapsulates* again

Shortest path routing: avoids loops (distance to destination always decreases)

Source routing: complete decision made at entry point (predefined path from beginning)

Virtual Layer 2 (VL2)

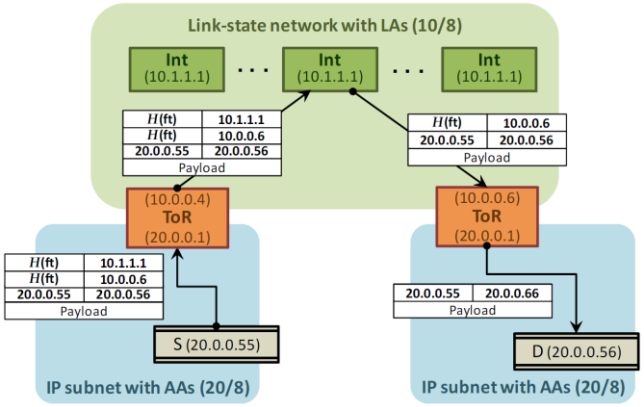
When moving VMs around, want to keep IP same

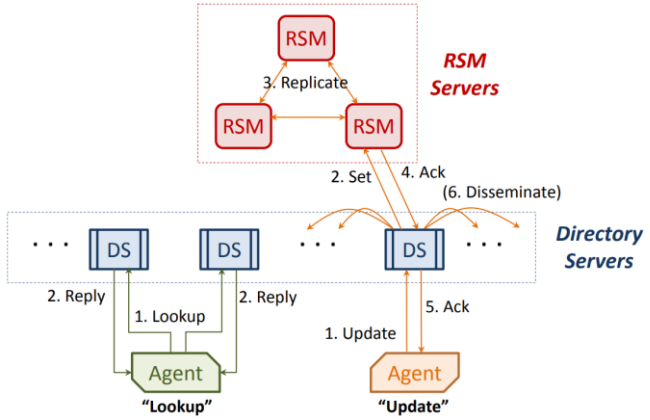
Clos network (different layers with different tasks)

VL2 agent (on server): acts as hypervisor
- intercepts outgoing packet
- sends to RSM and encapsulates it with LA

Name ≠ Location:
Locator Address (LA): "you can reach me here"
Application Address (AA): "my actual IP"

Network only knows locator address
Hosts / Apps only know application address
- Agent asks DS to translate so it can encapsulate





Border Gateway Protocol (BGP)

Prefix wants to be reachable
- AS appends its ID & announces that it knows a path
- other AS append their identifier and send it further

Path can be chosen based on various notions
- shortest path, cheapest path

Why should one use BGP inside a DC?
- Known to be unstable (oscillating paths)
- Converges very slowly, not adaptive to changes

But: it is well known & simple to use
- FB: centralized AS advertisement → craft network, choose parameters themselves & make it sophisticated

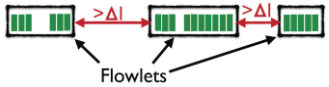
Equal cost multi-path (ECMP)

Multipath Routing: choose one out of many & distribute

Possible solution: choose one path uniformly at random
- packets take different path → requires packet reordering
- may result in "packet loss" (stuck in buffer / queue) and unnecessary re-transmissions

Use **hash functions** on packet header
- packets of same flow all take the same path
- need a different hash function for each level (else, each switch chooses same port nr for flow)
- hash collisions will remain constant (if two flows collide once, they will continue colliding) → doesn't guarantee load balancing

Split traffic into **flowlets** (group packets) and send each flowlet on a separate path, separated by **flowlet gap**
- gap too short: packet-level switching (reordering)
  too long: flow takes long to complete



CONGA: edge based monitoring
- track all possible paths & choose path with best result

2.3 Multi-tenant DC

- better economy of scale through increased utilization
- improved reliability

Agility: "Use any server for any service at any time"

Traditionally: tenants in "silos" (give entire rack for it)
- poor utilization & inability to expand
- IP addresses locked to topological location
  (VM migration hard, as addressing gets difficult)

Key requirements:
- Location independent addressing
  (tenant's IP addresses can be taken anywhere)
- Performance uniformity
  (VM receive same throughput regardless of placement;
  e.g. not influenced by oversubscription)
- Security → granular VM layer security
  (**Micro-segmentation:** isolation at tenant granularity)

Network virtualization

Virtual layer 2 (VL2)

App / Tenant layer	• Application Addresses (AAs): Location independent • Illusion of a single big Layer 2 switch connecting the app
Virtualization layer	• Directory server: Maintain AA to LA mapping • Server agent: Query server, wrap AAs in outer LA header
Physical network layer	• Locator Addresses (LAs): Tied to topology, used to route • Layer 3 routing via OSPF / BGP

Location independent addressing: Application address (AA)
- separation of virtual & physical address

Performance uniformity
- Clos network: low oversubscription (uniform capacity)
- ECMP provides load balancing (traffic-oblivious routing)
- BUT: depends on TCP flows, tenants can still cheat

Security: DS allow/deny connections by resolving $AA \rightarrow LA$

"SDN": **logically centralized control & dynamic data paths**

Network virtualization platform (NVP)

Want to expose a certain virtual network topology to any physical network by using hypervisors as middle layer

Virtual device (switch, router) as table/map in software
- tenants can specify devices which are implemented as SW tables, e.g. with OpenFlow switches

Access control list (ACL): checks access permissions

Requires full virtual network state at every host with a VM
- use incremental state computation or multiple stages

Slow processing, as many table lookups
- do step-by-step lookup only for first packet for pipeline

Tunneling interferes with TCP Segmentation Offload (TSO)
- can add "fake" outer TCP headers to stop this

2.4 TCP in data centers

Have control over the entire network and can own version

UDP at DC: no congestion resolution or retransmissions
- lower latency than TCP
- creates less state (uses less memory) per client than TCP

Control is implemented at sender (*end-to-end principle*)
- Congestion window (CW): start at 1 TCP segment
- Receiver window (RW): set by Rx, maximal CW
- Slow start: first double after each CW, then + 1
- Timeout: reduce CW by half, then back to 1

Partition/Aggregate: first divide request, then gather
- MapReduce, web queries, social networks
- *Aggregators:* web servers; *Workers*: memcached servers

TCP Incast: lots of *synchronized* response to aggregators
- switch will run out of buffers & drop packets
- sender needs to wait for RTO and then retransmit
- need all the answers to a request before preceding
  → Incast probability increases with numbers of servers

Fine-grained timeouts (against TCP Incast)

In DC, we mostly have very low roundtrip times, wherefore a RTO of $200 - 400$ms is orders of magnitudes too large

1. Reduce RTO_{min} to reduce incast influence
2. No RTO_{min}, microsecond TCP (high accuracy)

Datacenter TCP (DCTCP)

Mice: short messages (50KB-1MB), query, coordination
Elephant: large flows (1MB-100MB), data update, backup

Many flows very small, but most bytes in large flows
- elephant flows overload switch buffers
- mice flows particularly influenced, as longer completion

Explicit Congestion Notification (ECN): mark packets in switches if they experience congestion
- scale TCP window proportionally to number of packets with ECN bit set ($8/10 \rightarrow -40\%$)

Switch: mark packets if queue length above threshold
Receiver: send ECN back with delayed ACKs
Sender: maintain running average of parked packets

$$a = (1 - g) * a + g * F, \qquad F: \#marked\ in\ last\ w$$

$$w' = \left(1 - \frac{a}{2}\right) w$$

Achieves full throughput with small footprint on switch (much smaller queues & no state in network)
- a close to 0: "low congestion"
- a close to 1: "high congestion"

Deadline-aware TCP

Deadline driven delivery (D^3)

Allocate bandwidth on switches to flows according to their deadlines (requires state inside the network)

Rate required to satisfy a flow deadline:

$$r = \frac{s}{d}, \qquad \boldsymbol{s}: flow\ size, \boldsymbol{d}: deadline$$

1. Application sends desired rate to switches
2. Switches allocate rates α based on traffic load
3. Sending rate for next RTT: $r = \min(\alpha_i)$

If switch capacity is exhausted, allocate *base rate* to rest

Flow quenching: terminate useless flow (expired, too large)

D^3 can support much more flows with deadlines than TCP
- exact size and timing of flow needs to be known
- violates end-to-end principle (put state into the network)
- need router support for this
- greedy rate allocation leads to **priority inversion**

Deadline-aware Datacenter TCP (D^2TCP)

Integrate deadlines to DCTCP
- per-flow state at end-hosts (not routers/switches)

Deadline factor d: large → close deadline

$$d = T_c/D$$

T_c : completion time needed with current window
D : remaining time until deadline expires

Penalty function: prefer flows with near deadline

$$p = a^d$$

$$w = \begin{cases} w * \left(1 - \frac{p}{2}\right) & p > 0 \\ w + 1 & p = 0 \end{cases}$$

Multipath TCP

Modern data centers provide many parallel paths
(e.g. fat trees)

Round-robin scheduling per packet
- different RTT and MTU on the different paths
- reordering may lead to multiple ACKs

ECMP: hash network information to select outgoing link
- preserves flow affinity
- may not use the links uniformly (hash collisions)
- ECMP doesn't include the actual size of the flow
- static → no information about current traffic

Multipath TCP (MPTCP)

Use many subflows per TCP flow, each on a random path
- path with least congestion receives most traffic data
- Sender asks Receiver whether he is MP capable
  (in Internet: often, unknown options are removed)

Subflow between different IPs or same with different port
- ECMP hashes different flows to different paths
- each subflow runs its own congestion control
- protocol then handles traffic volume over all subflows
- often, 4-5 subflows are sufficient

3. Software-defined networking

Networks are complicated: distributed and no clear paras
- only "knows and dials" to adjust
- low level of abstraction, lots of small variables

Equipment is proprietary (black box for rest)
- no innovation, cannot adapt to problems

Traditional networking: code rules
1. input rules ("for link A-D, to this…")
2. Inter-connected protocols
- "Does my distributed algorithm work as intended?"

Software-defined networks: add level of abstraction
1. High-level coding / programm
2. Automatically created rules by "Network OS"
3. Low-level API: Logically centralized controller ("Assembly")
   & Data plane API ("driver instructions")

Match + action tuple
1. Match specific set of packets (header)
2. Construct action which should be applied to it
3. Install (match, action) in a specific switch

Common primitives:
- match packets, execute actions
- topology discovery
- monitoring

Control plane: Logically centralized control view provided to multiple control apps as a database
Data plane: only applies rules and forwards packets

Opportunities:
- Open data plane interface (standardized allows for independence from vendor; can access device directly)
- Centralized controller (solve distributed problem once)
  → better (even optimal) solution, as overview
- Software abstraction on controller (reuse algorithms)

Challenges:
- performance and stability
- consistency in-between controllers & during updates
- incremental updates in case of failure difficult
- stateful middle boxes not yet implementable

Use-cases:
- cloud virtualization (separate virtual networks, flexibility)
- inter-datacenter traffic engineering (high utilization)
- **Special-purpose deployment with less diverse hardware**

4. Flow & contention control

Try to prevent congestion before it even occurs
- **pro-active** (TCP always reactive & therefore inefficient)
- with increasing bandwidth, more data gets lost reactively

Infiniband

Low latency (RTT: 1-2us), high bandwidth (up to 100Gbit/s)

Similar layers (all implemented in hardware):
- physical
- link: *credit-based buffer management, virtual channels*
- routing & transport: (un)reliable, connection/datagram

Key differences to TCP/IP:
- link-level flow control
- kernel bypass (write directly into memory)
- new network semantics (*not* compatible)

Ethernet

not designed with built-in flow-control (unlike Infiniband)

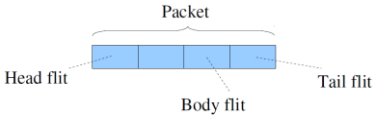
Priority Flow Control (PFC): "On/off" with priority classes
- coarse-grained (stops entire class, not just one channel)
- unfair: blocks all flows, even if no congestion there

Quantized Congestion Notification (QCN): layer 2
- add flow information on MAC packets & inform switches
- "ECN on layer 2" → does not cross subnet boundaries

4.1 Flow control

Determines resource allocation to packets
- channel bandwidth & buffer capacity (switch)
- deliver efficiently and with low, predictable latency

Flit: Flow control unIT
- packets are divided into flits by hardware same headers
- usually no extra headers (state saved in switches)
- much smaller than entire packet (mostly MTU sized)



Circuit switching: header reserves path for rest

Bufferless: just forward it on another path
- might be forced to drop & retransmit after timeout
- no guarantee that packet will arrive on other path
- valuable bandwidth used for packets which never arrive

Buffered flow control

decouple channel allocation in time

Store-and forward: send entire packet further
- channel & buffer allocation on a per-packet basis
- receive full packets & forward after complete reception
- high latency (need to wait for entire packet)

Cut-through: send individual flits as soon as they arrive
- channel & buffer allocation on a per-packet basis
  (need to allocate entire packet buffer at next switch)
- forward as soon as first (header) flit arrives
- low latency, but blocks for whole packet transmission

Wormhole: cut-through only on flits
- buffers are allocated on a per-flit basis
- low latency and efficient buffer usage
  (only need one flit buffer & one flit of channel BW)

Head-of-Line (HoL) blocking
Flit at the front of queue cannot go on, blocks all the rest

Virtual channel: multiple buffers per physical channel
- multiple virtual channels per physical channel
- separate buffer space prevents HoL blocking

Buffer management: communicate available buffer size
- *credit-based:* keep count of free buffers at downstream switchs & update if this changes
- *On/off:* downstream switches allows/stops stream
  (send "off" if buffer count below $F_{off} \geq t_{rt} * b \,/\, L_f$)

4.1 Flow coordination

Look at collective behaviour of flows by looking at jobs
Coordination between individual network transfers of a job

Problem: simply using more machines doesn't work
- more increase in communication time than decrease in computational time (takes even longer with more CPUs)

MapReduce: spread job, shuffle and aggregate again
- *Broadcast (Map):* one-to-many, partition work
- *Shuffle (Reduce):* many-to-many, aggregate results

Orchestra

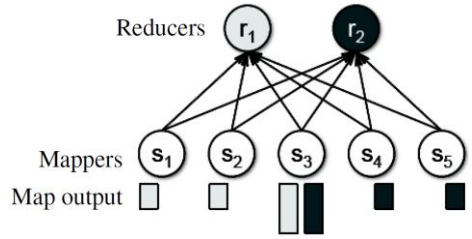
manage data transfers in clusters
- optimize at transfer level
- transfer: flows transporting data between two job stages

Coordination done through three control components:

Cornet: cooperative broadcast
- split data up into blocks & distribute them across nodes
- receivers of blocks become part of sender set
  (can choose sender in the same rack)
- large blocks (4MB), no incentives required, topology aware

Weighted shuffle scheduling: shuffle coordination
- divide output into buckets
- if buckets not equally shared over mappers, assign weights to each flow in a shuffle according to data

(e.g. node s_3 receives higher BW, as more data to transmit)



Inter-transfer controller (ITC): global coordination

5. End-host optimization

Networks became faster & latency decreased
CPU speed remained the same, but we got more cores

Network interface controllers (NIC) increasingly powerful
→ can offload encryption, TCP processing etc.

Key challenges:
- *Scaling:* fast network & many cores require parallelization
- *Latency:* dominated by software processing,
  multiplied because of multi-tier architectures
- *CPU load:* packet processing & copy needs comp. power

Descriptor rings

1. OS gets buffer, fills it and hands it over to device
2. Device sends data and frees buffer
3. signals OS with interrupt & gives memory back

If one of the parties has no more space:
- Device: start discarding packets & signal in register
- CPU: signal device to interrupt when again free buffer

Receiver-Side Scaling (RSS)

Use multiple Rx/Tx queues for a single NIC
- one queue per core allows for fast processing

NUMA: *Non-uniform memory access*
(some memory access is faster than other, global memory)

NUMA node: group of cores with very fast local memory
- always want to process data with cores near to memory

Rx: Incoming packet is hashed and added to one queue
- each queue can then interrupt a corresponding core
  (should be in same NUMA node for best performance)

Transmit side: can use multiple transmit queues
- each core has own Tx queue → no locks / synchronization
- Performance isolation: NIC can schedule without CPU

Scaling with cores & performance isolation
Reduced CPU load (synchronization, load distribution)

TCP Offload

Move TCP/IP processing to the NIC
- implement TCP state machine in hardware

- Less host CPU cycles for processing & checksums
- Fewer CPU interrupts & memory copies
- can even offload expensive features s.a. encryption

TCP Offload Engines (TOEs)
- impose complex interfaces & management overhead
- only useful for longer connections (only small gain, and management overhead might overwhelm savings)
- implemented in hardware, no bug fixes & changes (OS doesn't control it; no new protocols s.a. DCTCP)

Nowadays, have many cores, which favours parallelization
- network processing is hard to parallelize

Applications: Storage-server access, cluster interconnection
- very high bandwidth
- low end-to-end latency
- long connection durations & not too many

Kernel bypass (no context switches)

Currently, a large part of the delay come from the OS
- data has to switch: $Kernel\ space \leftrightarrow User\ space$
- system call overhead
- multiple memory copies

User-level networking: remove overhead
- map individual queues directly to applications
- allow applications to poll queues
- requires hardware support to validate data & demultiplex messages to the applications
- Kernel only used during connection setup/teardown for mapping queues to the applications & interrupts

Use message queues (*Send, Recv, Free*) to get buffers,
write into them and hand them over to the NIC

Remote Direct Memory Access (RDMA)

"One-sided operation": only client actively involved
- requires buffer advertisement prior to data exchange
- need to check whether other party is allowed access
- **much faster for small message sizes**

RDMA Write:
- "Where should data be taken from *locally*?"
- "Where should it be placed *remotely*?"

RDMA Read:
- "Where should it be taken from *remotely*?"
- "Where should it be placed *locally*?"

Message passing also possible ("send"/"recv")

Implementations: vendors write own drivers & libs
- *Infiniband* (Compaq, HP, IBM, Intel, Microsoft, Sun)
- *iWARP*: RDMA over offloaded TCP/IP (custom NICs)
- *RoCE:* send directly over Ethernet (no TCP anymore)

Open Fabrics Enterprise Distribution (OFED)
- *Device driver:* implement allocation of message queues on device
- *User driver:* provide access to message queue from user space

"verbs": common application interface
- register application memory (together with OS)
- create a queue pair (QP; send + recv)
- create a completion queue (CQ)
  (element gets added after operation has completed)
- Send/Receive/Read/Write data
  Work-request element (WQE): buffer pointer + op type

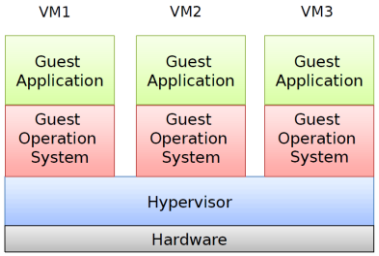
RDMA needs to do trade-off between polling & interrupts
(latter larger overhead, as context switch)

Doorbell batching: put multiple messages on the queues before notifying the NIC

6. Network virtualization

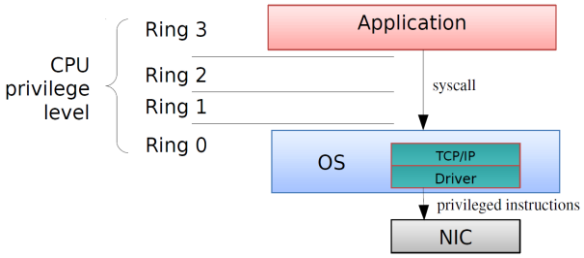
Multi-core: double number of cores every 18 months

Hypervisor manages VM access to hardware
- used for relaying interrupts & other commands



x86: 4 CPU privilege levels (Ring 0 – 3)
- *privileged instructions:* read or write I/O instructions

Ring 3: Application uses syscall & trap into kernel
Ring 0: OS drivers issue PCI commands
sensitive & hardware-related *privileged instructions*
(Kernel protects HW from "bad" programms)



6.1 Full Device Emulation

Guest OS unaware that it is being virtualized

Hypervisor emulates device at the lowest level
- privileged instructions from driver trapped by hypervisor

Ring 1: Guest OS, communicates with traps in *Ring 0*

- no changes to guest OS required (can use normal drivers)
- very inefficient & complex (lots of calls & interrupts)

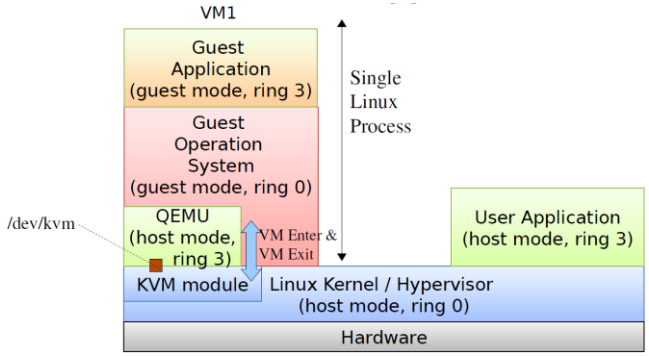
Difficulty with historical x86 virtualization
privileged instructions executed in Ring 3 may result in:
- a fault
- nothing
- process issues trap indicating it wants code in Ring 0

Last case is intended in virtualization; silently failing instructions make implementing virtualization difficult (need execution, but trap is not called and therefore instruction is simply ignored)

Intel Virtualization Technology VT-x
Duplicate all rings*: root & non-root* mode with 4 rings each

Guest ≙ *non-root* mode, **Host** ≙ *root* mode

Protected instructions executed in *guest mode Ring 0* generate **traps** that can be checked in *host mode* (guaranteed access, not possibility of "nothing" as in x86)



KVM hypervisor: **K**ernel-based **V**irtual **M**achine

QEMU: hardware simulation (API for emulated drivers)

1. Start new guest OS → start QEMU process
2. QEMU interacts with KVM: allocate memory, start guest OS in *guest mode Ring 0* (use HW support)
3. I/O request from guest OS traps into KVM
4. KVM forwards requests to QEMU for emulation

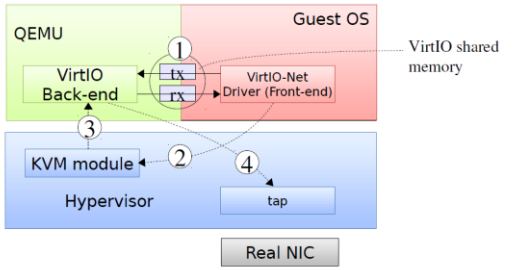
6.2 Paravirtualization

Guest OS aware that it is being virtualized
→ runs special **paravirtual device drivers**

Hypervisor cooperates with guest OS through interfaces:
- *Paravirtual driver*: "Frontend driver"
- *Interfaces:* "Backend driver"

- Better performance (knows it's a VM & doesn't trap everything; can e.g. do batch requests)
- Requires changes to the guest OS (special drivers)
- requires hypervisor involvement, e.g. interrupt relaying

VirtIO: I/O virtualization framework for Linux
- split driver model: front-end & back-end drivers
- APIs for front-end and back-end to communicate
- lot of overhead because of switching kernel/user space



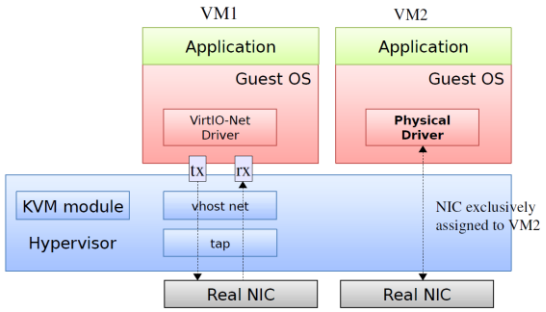
1) VirtIO-Net driver adds packet to shared VirtIO memory
2) VirtIO-Net driver causes trap into KVM
3) KVM schedules QEMU VirtIO Back-end
4) VirtIO back-end gets packet from shared VirtIO memory and emulates I/O (via system call)
5) KVM resumes guest

Vhost: improved VirtIO backend (see VM1 next page)
- put VirtIO emulation code into the kernel (no more in QEMU, which requires system calls)

6.3 Passthrough / Direct Assignment

Directly assign NIC to VM (no more hypervisor)

As VM has exclusive NIC, can use **physical driver**
(no special driver required, can directly talk to hardware)

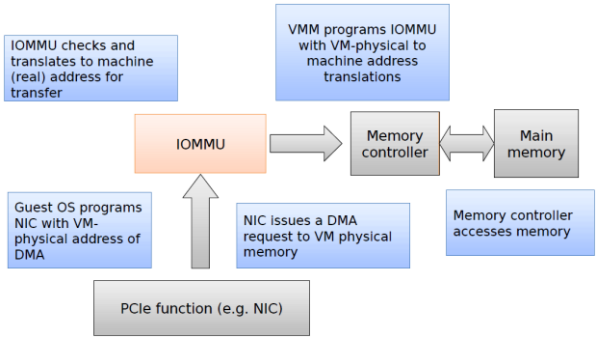


- VM tied to specific NIC hardware
  (makes VM migration more difficult, as HW related)
- VM physical addresses for DMA are host virtual addresses
  (might cause security issues, as belong to other VMs)
- need a different NIC for each VM

IOMMU: "Memory management unit (MMU) for DMA"
- solves security issue: translate & validate DMA requests

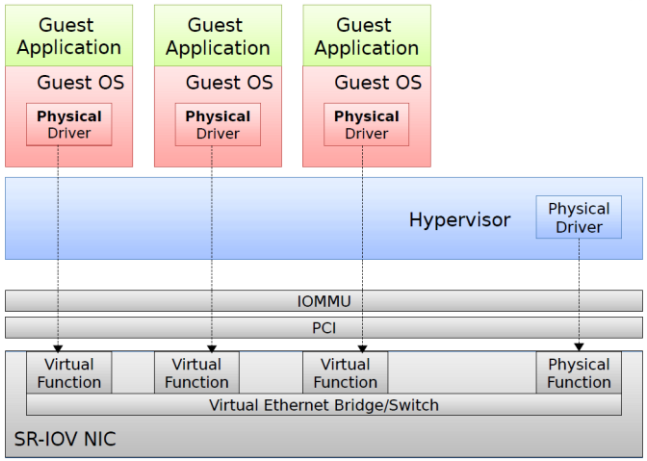
Virtual address: address in application of guest OS
Physical address: hardware address seen by guest OS
(physical address in the virtual machine)
Machine address: real hardware address on physical device
(as seen by hypervisor)

Virtual Machine Monitor (VMM): hypervisor



Single-Root-IO-Virtualization (SR-IOV):
display physical device as multiple virtual ones

Dynamically create new PCI devices
- *Physical function (PF):* original device, full functionality
- *Virtual function (VF):* extra device, limited functionality



6.4 Inter-VM communication

Internal communication
know VMs are on same physical machine → use hypervisor
- copy data into memory, then hand over
- low latency (1 software copy)
- uses host CPU cycles for networking (not wanted)
- easy to upgrade (SW) & fully supportive for OpenFlow

External communication
can separate from host computer, but more latency
- reduce CPU requirements, faster TCAMs on switch
- can integrate into network management policies
  (e.g. can use OpenFlow for policies between hosts)

- *External switch*: simplifies configuration, as all switching controlled by the network

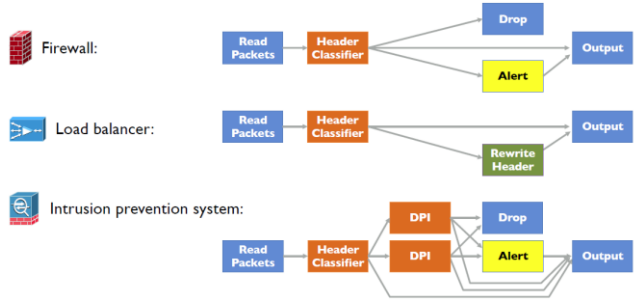
- *NIC:* requires less latency than external switch

7. Network functions virtualisation

Middleboxes: change your data inside the network
- Ad insertion, WAN accelerator, QoE monitor, firewall

Violate the end-to-end & fate-sharing principle: e.g. proxy can die, but connection is still up and running

Wasteful replication of functionality
- do same function multiple times, but cannot aggregate, as middleboxes are treated as black-boxes



Middleboxes are treated as black-boxes
- monolithic, and therefore hard to understand & debug
- long deployment timelines & no standards, vendor lock-in

Virtualization: instead of non-commodity hardware from specific vendors, which require physical installation at site, use same standard servers, storage & switches and build application inside them → **software middleboxes**

Middlebox consolidation: multiplexing of functionality can yield benefits, as peak loads could be distributed and therefore do not waste as much resources for worst-case

Hyper app: set of middlebox functionality

Network-wide coordination: can share load over boxes, each doing part of the function (e.g. filter part of IP range)

Waypointing: send all traffic through particular point in net

Software-defined Middleboxes: logically centralized controller and network functions as OpenBox apps
- expensive to save session state for a long time

8. Wide-area networks (WAN)

WAN: Wide-area LAN → inter-DC network

8.1 WAN routing

Datacenters on multiple locations:
- data availability & latency (locality)
- load balancing (high load where daytime)
- local data laws (e.g. don't want to route through US)

Hybrid public-private operation
- *public:* offload peak loads to extern assets
- *private:* normal traffic over own backbone / resources

Normal traffic: queuing delays ≫ transmission delays
(delays mostly from being stuck in queue)

WAN: mostly only transmission delay, not a lot of queuing
- more point-to-point connections
- higher degree of flexibility / manageability than Internet
- dedicated connectivity between small set of end-points

Multiprotocol Label switching (MPLS)
- "If you have this label, go to this next node"
- label corresponds to tunnel, tell packet where to forward
- link-state protocol: setup tunnel, reserve BW & flood info
- only ingress & egress node read the packet

Problem with standard approach:
- no global notion of management / centralized controller
- complex (influence by artificially adjusting parameters)
- inefficient (don't separate between different traffic)
  background: analytics, not latency-sensitive
  Non-background: "normal traffic", latency-sensitive

New technology should do **traffic engineering (TE)**
- leverage service diversity (some data tolerates delay)
- centralized TE using SDN
- dynamic reallocation of bandwidth (optimization)
- edge rate limiting (limit input rate, not inside network)

B4 (Google)

Use SDN for world-wide traffic engineering

Quagga: enables fine software control over routing (instead of HW)

eBGP: external BGP in-between ASes
iBGP: internal BGP for nodes inside the same AS

BGP routing as "big red switch"

Edge rate limiting: if experience increasing delay, just reduce producer rate → only need shallow buffers (cheap)

Safeguard: backup controller who takes over if Master dies

Aggregation: sort flows into groups → route them together

Traffic engineering (TE) instead of just shortest path (OSPF)
- **centralized TE servers with global view**
- per **QoS** traffic engineering (TE)
  (per app loss/latency/throughput considerations)
- put TE flows into the flow tables of the switches with higher priority than BGP flows (can switch if mistake)
- large throughput increase requires less bandwidth

Lessons learned from B4

Controller produces rules faster than switch can process
→ rules get queued and only applied with delay
→ Flow rules cause Head-of-line blocking of packets
causes timeouts & new rules because of this (worsens)

Solution: separate packet IO & flow request queues & prioritize packets (send only 1 rule every N packets)
- if rule already superseded, drop it already in the queue

Worst problem is *unstable mastership*: Slave receives no more heartbeats from master and claims ownership
→ some switches follow different controllers, mastership switches often and causes dropped packages & instability

Solution: make smaller domains & master election within domain, also removes single point of failure at each site

SWAN (Microsoft)

- Don't change in one step, but gradually
- Use free 10% slack on link to gradually switch flows (if not used, give slack to background tasks for utilization)

Other options

- for fixed, continuous flows between clients, just start with a large congestion window to not ruin TE
- control one end: DNS manipulation & load balancing
- try controlling two ends (e.g. Google Chrome) for more detailed and direct manipulation & traffic engineering

8.2 WAN congestion control

TCP needs to estimate the link capacity using ACKs:
- prevent losses by resending dropped packages
- help probing & setting the rate of transmission

Problems with TCP:
- Multiplicative decrease very drastic
- only works for long flows
- for losses not caused by congestion (e.g. physical layer), don't need to adjust the sending rate (but cannot know)
- need to overshoot first to find correct rate (create drops)

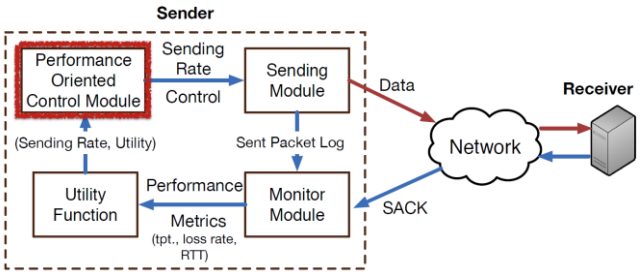
TCP uses a model of the network and creates rules for it
- follow low-level, hard-wired mapping $event \rightarrow reaction$
- works well if assumptions are correct
- cannot differentiate between multiple reasons for drops

PCC (Performance-oriented congestion control)

Black-box, online learning to do congestion control
- try to vary around found solutions to improve rate

Define **utility function** u depending on traffic metrics
- throughput, loss rate, latency

1. Set intended utility function after black-box network
2. Send at different rates and observe the utility function
3. Adjust your rate to reach maximal utility



Do randomized controlled trials around found point
- vary $\pm\,\varepsilon$ and choose better (or higher, if both the same)

PCC is still not using cooperation between parties
- for some utility function (e.g. throughput), we still converge to a fair, efficient *Nash equilibrium*
- does not need AIMD, as looks at real performance (in comparison to TCP, which deviates from convergence)

PCC offers much more stable performance than TCP
- good to use in CDN backbone, inter-datacenter networks
- much better throughput for satellite & dedicated nets
- very good for rapidly changing networks (but requires *convergence*; might not work for fast changes)

PCC (default) is **not TCP friendly** and starves its traffic
- need to use different utility functions to provide fairness

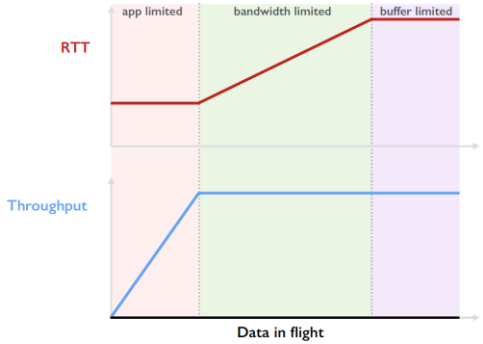
BBR: congestion-based congestion control

Try to estimate better network model for no congestion (find bottleneck BW & round-trip propagation)

Want to operate at bottleneck BW, but with minimal RTT
→ right between app & bandwidth limited regime (maximal BW, minimal RTT, buffers are starting to fill)

However, TCP operates at BW and buffer limited border
- there, buffers overflow and packets start dropping

$$\text{Data in flight} = bandwidth * delay\ product$$
$$= Throughput * RTT$$



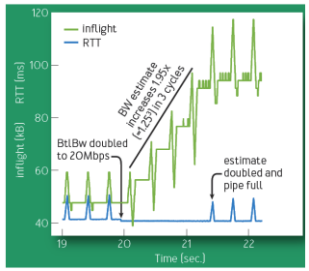
This point is optimal but unreachable
- cannot measure both values simultaneously (either RTT in app-limited regime, or BW in others)
- RTT increase due to buffer filling or other factors? (e.g. using different path)

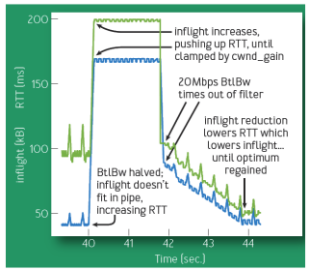
BBR synchronizes parties: buffers fill → stop rate increase
(TCP also synchronizes: if buffers full, all reduce drastically)

Operates with two target conditions:

1. **Full pipe:** $in - flight = RTT_{min} * BW_{bottleneck}$
2. **Rate balancing:** buffer fill rate = emptying rate

(Need second condition to spread data over time)





Constantly probe $BW_{bottleneck}$ around $[1, 1.25, 0.75]$ BW

Occasionally, drop rate to see an empty queue again
(also, at beginning drain queues again created in startup)

- very stable even under losses
- for multiple flows, results in fair share
- Goodput can result in less goodput with large buffers

9. Content Distribution Networks

Users and their experience depend on latency

Caching: Try to keep content near clients

Static caching can however lead to various problems:
- volume & diversity of content requires huge storage
- dynamic content (personalized, often updated)
- often encrypted

CDN: allows quick and globally distributed access to data
Located near their customers around the world at:
- *ISPs*: low latency for clients + don't have to pay own ISP for transporting data, can deliver directly
- *Internet exchange points:* good management & high BW

Spread contents server

Replicate entire services, not just caching, by interacting with backend servers

- Increased reliability, as no single-point-of-failure
- load balancing
- lower latency & traffic costs over entire internet

Network the sites and the origin

- own cables & protocols with Google's Quik

Overlay routing: route over another node and not directly
- internet routing cannot be to your advantage ("Triangle inequality violation" direct way not fastest)
- don't want to relay traffic & pay twice to ISP for traffic
- maintain map of RTT times & evaluate whether worth it

Persistent connections: CDN keep open TCP connections
- immediately large TCP window, no slow start
- no initial handshake required → less latency
- for SSL, even larger impact; but need to trust CDN node

Direct clients to appropriate servers

1. Client asks server for website & receives html file
2. Directly loads javascript, pictures etc. from CDN

DNS manipulation to locally customize DNS resolution and allow user to access best server for his location
- does not work if user doesn't use local DNS resolver (nowadays, additional arguments to solve this)

Anycast addresses: multiple servers answer to same address & nearest (fastest BGP link) is chosen
- announce same BGP prefix from multiple servers
- no fine-grained control to choose which server

CDN broker: send users to best CDN
- app asks broker in real-time: "Which CDN is best now?"
- broker can also chose small CDNs if they offer the best performance instead of always the same one, e.g. Akamai

10. Various

TCP Segmentation Offload (TSO)

Network interface controller (NIC) takes load off CPU and slices payload into package sizes (MTUs)

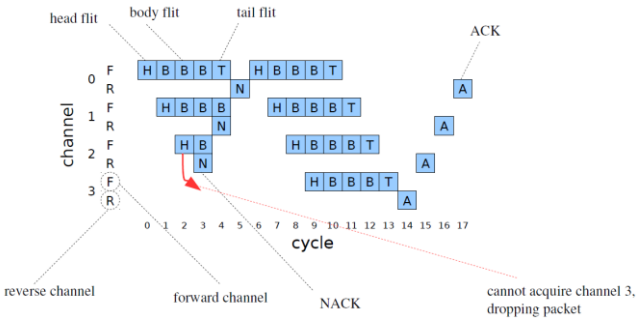
Valiant load balancing (VLB)

Explicitly send traffic via a specific node for load-balancing

TCP Fast retransmit

After having received three duplicate ACKS (i.e. four ACKs for the same packet), the sender decides the packet got dropped and is not simply arriving in a different order and therefore retransmits the packet before the RTO is over

Time-space diagram



Open Shortest Path First (OSPF)

Link-state routing inside a single *Autonomous System* (AS)
- create entire network and calculate shortest paths
- guarantee loop free connections
- dynamic load balancing between equal-cost links