Reliability:
Reliable, Efficient, Secure Network Transport

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs434/

03/17/2021
Admin.: PS2

- Part 1 (reliable transport); Part 2 (congestion control); Part 3 (comparison)

proj-sol:

- 129 400 3045 FishThread.java
- 388 1457 12873 Node.java
- 51 167 1145 PingRequest.java
- 83 250 2106 SimpleTCPSeqSpace.java
- 181 605 5248 TCPManager.java
- 889 3088 26381 TCPSock.java
- 60 149 1316 TCPSockID.java
- 123 382 3866 TransferClient.java
- 147 500 5059 TransferServer.java
- 2051 6998 61039 total

proj:

- 129 400 3045 FishThread.java
- 341 1301 11313 Node.java
- 51 167 1145 PingRequest.java
- 132 460 3146 TCPSock.java
- 50 128 909 TCPManager.java
- 123 382 3866 TransferClient.java
- 147 500 5059 TransferServer.java
- 973 3338 28483 total
Admin

- I will schedule the time to meet with me on projects before PS1 grading. Please see meeting slots on canvas
Recap: Network Load Balancing (L4RR) with Direct Server Return (DSR)

Each real server **MUST** use VIP as an IP address: binded socket allows VIP, but not ARP reply to avoid race condition.
Recap: Simple Hash SS $\Rightarrow$ Consistent Hash SS

Consistent hash goal: after the connection $C$ is assigned to server $S_1$, $C$’s assigned server does not change after other node failures, as long as $S_1$ does not fail.

$$\text{hash0}(p) = 5$$

$$\text{hash}(p) = 7$$

**stateless router:** $\text{hash0}(p) \% |\text{LB}|$

if existing conn
use conn track
else
$$\text{servers}[\text{hash}(p) \% |\text{LB}|]$$
$$\text{consistent\_hash}(p)$$
Recap: Basic Ring Consistent Hash

- A key is assigned at its successor node: node with next higher or equal ID
- To better load balance, map a node to multiple points on the ring

Discussion:
- If S5 fails, will an assignment to a non-crashed server change?
- Do we still need connection table?

Key = hash(TCP-connection)
Recap: Load Balancing

- Google Maglev

```
function POPULATE
  for each i < N do
    next[i] ← 0
  end for
  for each j < M do
    entry[j] ← -1
  end for
  n ← 0
  while true do
    for each i < N do
      c ← permutation[i][next[i]]
      while entry[c] ≥ 0 do
        next[i] ← next[i] + 1
        c ← permutation[i][next[i]]
      end while
      entry[c] ← i
      next[i] ← next[i] + 1
      n ← n + 1
      if n = M then return end if
    end for
  end while
end function
```

- Akamai bin-packing algorithm (column 12 of Akamai Patent)

  every T second
  - Compute the load to each publisher k (called serial number)
  - Sort the publishers from increasing load
  - For each publisher, associate a list of random servers generated by a hash function
  - Assign the publisher to the first server that does not overload
Linux ipvs Server Selection Algorithms

- See ipvsadm manual (https://linux.die.net/man/8/ipvsadm)
- Many scheduling algorithms
  - **rr** - Robin Robin
  - **wrr** - Weighted Round Robin
  - **lc** - Least-Connection: assigns more jobs to real servers with fewer active jobs.
  - **wlc** - Weighted Least-Connection. This is the default.
  - **lblc** - Locality-Based Least-Connection: assigns jobs destined for the same IP address to the same server if the server is not overloaded and available; otherwise assign jobs to servers with fewer jobs, and keep it for future assignment.
  - **lblcr** - Locality-Based Least-Connection with Replication: assigns jobs destined for the same IP address to the least-connection node in the server set for the IP address. If all the node in the server set are over loaded, it picks up a node with fewer jobs in the cluster and adds it in the sever set for the target. If the server set has not been modified for the specified time, the most loaded node is removed from the server set, in order to avoid high degree of replication.
See ipvsadm manual ([https://linux.die.net/man/8/ipvsadm](https://linux.die.net/man/8/ipvsadm))

Many scheduling algorithms

- **dh** - Destination Hashing: assigns jobs to servers through looking up a statically assigned hash table by their destination IP addresses.
- **sh** - Source Hashing: assigns jobs to servers through looking up a statically assigned hash table by their source IP addresses.
- **sed** - Shortest Expected Delay: assigns an incoming job to the server with the shortest expected delay. The expected delay that the job will experience is \((C_i + 1) / U_i\) if sent to the \(i\)th server, in which \(C_i\) is the number of jobs on the \(i\)th server and \(U_i\) is the fixed service rate (weight) of the \(i\)th server.
- **nq** - Never Queue: assigns an incoming job to an idle server if there is, instead of waiting for a fast one; if all the servers are busy, it adopts the Shortest Expected Delay policy to assign the job.
Outline

- Admin and recap
- Infrastructure supporting multiple network servers
  - Overview
  - Request routing using name/service based abstraction
  - Request routing using IP/URI based abstraction
  - Server selection (SS) algorithms
  - Request router reliability
Roadmap

- Many network/distributed systems use a control plane (e.g., RR) to manage the resource plane, but the control plane itself can often fail, resulting in a major challenge.

- Our goal (to build background):
  - See 2 key types of design handling control plane robustness
  - We will cover control plane reliability using consensus later when we cover cluster/cloud servers
Fully Distributed RR: Microsoft NLB

- No dedicated load balancer at all
- All servers in the cluster receive all packets
- All servers within the cluster simultaneously run a mapping algorithm to determine which server should handle the packet. Those servers not required to service the packet simply discard it.
  - Mapping (ranking) algorithm: computing the “winning” server according to host priorities, multicast or unicast mode, port rules, affinity, load percentage distribution, client IP address, client port number, other internal load information

Discussion

- Advantages and issues of fully distributed RR
Active/Backup RR

- At any time,
  - One RR (in a level) serves as the active RR;
  - Passive RRs monitor the active
  - One passive RR becomes active after detecting active failure

- Example: ha-keepalived
  - https://docs.nginx.com/nginx/admin-guide/high-availability/ha-keepalived-nodes/
Active/Passive Request Routers: HAProxy using VRRP [RFC3768]

Configuration on LB1/LB2

```
listen webfarm 192.168.1.1:80
  mode http
  balance roundrobin
  cookie JSESSIONID prefix
  option httpchk
  option forwardfor
  option httpchk HEAD /index.html HTTP/1.0
  server webA 192.168.1.11:80 cookie A check
  server webB 192.168.1.12:80 cookie B check
  server webC 192.168.1.13:80 cookie C check
  server webD 192.168.1.14:80 cookie D check
```

Configuration keepalived LB1/LB2

```
# Requires keepalived-1.1.13
# cheaper than pidof
# check every 2 seconds
# add 2 points of prio if OK

vrrp_script chk_haproxy {        
  script "killall -9 haproxy"
  interval 2
  weight 2
}

vrrp_instance VI_1 {         
  interface eth0
  state MASTER
  virtual_router_id 51
  priority 101
  virtual_ipaddress {        
    192.168.1.1
  }
  track_script {             
    chk_haproxy
  }
}
```

https://www.haproxy.org/download/1.2/doc/architecture.txt
Virtual Router Redundancy Protocol: Basic Ideas

- Virtual router
  - Specified by a virtual router identifier (VRID) and a set of associated IP addresses
  - Each virtual router ID has a corresponding (virtual) MAC: 00-00-5E-00-01-[VRID]

- Leader election among the physical routers to select a single master, who
  - Owns the given IPs of the virtual router
  - Receives and forwards packets for the IPes
Virtual Router Redundancy Protocol: Protocol and Msgs

<table>
<thead>
<tr>
<th>Version</th>
<th>Type</th>
<th>Virtual Rtr ID</th>
<th>Priority</th>
<th>Count IP Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auth Type</td>
<td>Adver Int</td>
<td>Checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP Address (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP Address (n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authentication Data (1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authentication Data (2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Periodical, multicast announcements to IP address 224.0.0.18 w/ IP protocol number 12
VRRP Sample Configuration

Legend:

- = Ethernet, Token Ring, or FDDI
H = Host computer
MR = Master Router
BR = Backup Router
* = IP Address
(IP) = default router for hosts
Discussion

- Advantages and issues of VRRP based design
Recap: The Service Discovery, Request Routing Infrastructure Supporting Multiple Servers Journey

- How to make service direction transparent to user
  - DNS for name abstraction
    - One of the best designed protocols: hierarchical name space and hierarchical delegation, native caching, flexible recursive and iterative query, UDP to reduce overhead but also support TCP, same query and response format, interesting <length><content> encoding and pointer for compression, proactive answers of anticipated queries (server push) reduce # queries on server and latency on client;
    - mDNS, DNS SRV, DNS-SD handle totally distributed SD
  - How to become more responsive, more flexible, but still transparent
    - L4/L7 (on path) RR
  - How to handle the VIP mismatch problem of L4 RR
    - L4 NAT (rewrite)
  - How to reduce rewrite overhead
    - L4 DSR

- How to compute server assignment/scheduling
  - simple hash += conn tracking += consistent hash; Maglev, Akamai hash assignment

- RR reliability: fully distributed, active/passive VRRP protocol
Outline

- Admin and recap
- Infrastructure supporting multiple network servers
- Reliable, efficient, secure network transport
Overview: Transport Layer

- Provide *logical communication* between app’s processes
  - We take a broader view that HTTP can also be considered as a transport protocol to carry request/response messages for higher layer applications

- Transport protocols run in end systems

- Transport vs. network layer services:
  - *Network layer*: data transfer between end systems
  - *Transport layer*: data transfer between processes
    - relies on, enhances network layer services

Discussion: your wish list of services to be provided by a transport layer.
Roadmap: Transport Services We Plan to Cover

- Multiplexing (socket API)
- Transport: sliding window protocol, connection management for reliable transport
- Resource control
  - End host flow control (not to overwhelm endhost)
  - Network congestion control (efficient, fair use of network-wide resource)
- Security: confidentiality, integrity, authenticity of transport
- Mobility management
Roadmap: Main Related Protocols

- TCP (1981)
- MP TCP CC (2011)
- BBR (Google; 2016)
- TLS (2008, 2018)
- HTTP/2 (2015)
- QUIC (draft, last updated Feb. 2021)
- HTTP/3 (draft; last updated Feb. 2021)
- Reliable, Scalable Datagram (2020)
Transport Services and Protocols

- Reliable, in-order delivery (TCP)
  - multiplexing
  - reliability and connection setup
  - congestion control
  - flow control

- Transport-layer security (TLS)

- Unreliable, unordered delivery: UDP
  - multiplexing

- QUIC (Quick UDP Internet Connection)

- HTTP/1.1, HTTP/2

- HTTP/3
Roadmap: Class Scheduling

- **Class 1** (today; ready for PS2/part 1):
  - transport layer services
  - single data stream connection reliability
    - sliding window protocol
    - connection management

- **Class 2** (ready for PS2 part 2)
  - Basic CC (AIMD), Reno, Cubic, MP-TCP, Vegas, BBR

- **Class 3 & 4** (ready for PS2 part 3)
  - TLS, HTTP/2, QUIC, HTTP/3
Outline

- Admin and recap
- Infrastructure supporting multiple network servers
- Reliable, Efficient, Secure Network Transport
  - Overview
  - Basic, reliable transport, TCP
Basic, Reliable Transport

- Sender sends data to receiver, with each data unit assigned a sequence #
- Receiver sends acknowledgement (or nack) for each data unit
Discussion: Basic, Reliable Transport Design Space

- Sender driven vs receiver driven
- Uni-directional vs bi-directional
- In-order delivery vs not-in-order delivery
- Window control vs rate control
- ...
Basic, Reliable Transport Instantiation: Sender-Driven, Uni-Directional In-Order Delivery, Sliding Window Protocol

Sender event:
- k-bit seq # in pkt header from sender to receiver
- Assign new seq# to new data
- Maintain data in window buffer

Retransmit event:
- detect loss, retransmit lost data

Receiver event
- If receive in-order packet
  - Deliver to app
- Provide feedback
TCP Instantiation of Basic, Reliable Transport: Extend Previous to Bi-Directional

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

---

**Host A**
- Seq=42, ACK=79, data = ‘C’
- Host ACKs receipt of ‘C’, echoes back ‘C’

**Host B**
- Seq=79, ACK=43, data = ‘C’
- Seq=42, ACK=80

**User**
- User types ‘C’

**Simple telnet scenario**
- Host ACKs receipt of echoed ‘C’
- Host ACKs receipt of echoed ‘C’
Key Design Issues/Points

- Loss detection (timeout or additional mechanism)

- Connection management
  - Initial seq#
  - Resource release

- Window size
TCP Timeout Estimation/Basic Loss Detection

- Why is good timeout value important
  - Why is “too short” bad?
    - premature timeout
    - unnecessary retransmissions => many duplicates
  - Why is “too long” bad?
    - slow reaction to segment loss
- Ideal timeout and how to estimate?
TCP Retransmission Timeout (RTO) Alg

Problem:
- Ideal timeout = RTT, but RTT is not a fixed value
- Possibility: using the average of RTT, but this will generate many timeouts due to network variations

TCP solution:

\[
\text{Timeout} = \text{EstRTT} + 4 \times \text{DevRTT}
\]
TCP EstRTT and DevRTT Computation

- Exponential weighted moving average
  - influence of past sample decreases exponentially fast

\[
\text{EstRTT} = (1-\alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT}
\]

- SampleRTT: measured time from segment transmission until ACK receipt
  - typical value: \(\alpha = 0.125\)

\[
\text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta |\text{SampleRTT}-\text{EstRTT}|
\]
  (typically, \(\beta = 0.25\))
An Example TCP Session
**TCP Fast Retransmit**

- **Question:** Can we detect loss faster than RTT?

- Detect lost segments via duplicate ACKs
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - resend segment before timer expires
Triple Duplicate Ack

Packets

1 2 3 4 5 6 7

Acknowledgements (waiting seq#)

2 3 4 4 4 4
Fast Retransmit:

event: ACK received, with ACK field value of y
if (y > SendBase) {
  ...
  SendBase = y
  if (there are currently not-yet-acknowledged segments)
  start timer
  ...
}
else {
  increment count of dup ACKs received for y
  if (count of dup ACKs received for y = 3) {
    resend segment with sequence number y
    ...
  }
}
TCP: reliable data transfer

Simplified TCP sender

---

00 sendbase = initial_sequence number agreed by TWH
01 nextseqnum = initial_sequence number by TWH
02 loop (forever) {
03    switch(event)
04        event: data received from application above
05            if (window allows send)
06                create TCP segment with sequence number nextseqnum
07                if (no timer) start timer
08                pass segment to IP
09                nextseqnum = nextseqnum + length(data)
10            else put packet in buffer
11        event: timer timeout for sendbase
12            retransmit segment
13            compute new timeout interval
14            restart timer
15        event: ACK received, with ACK field value of y
16            if (y > sendbase) { /* cumulative ACK of all data up to y */
17                cancel the timer for sendbase
18                sendbase = y
19                if (no timer and packet pending) start timer for new sendbase
20                while (there are segments and window allow)
21                    sent a segment;
22            } /* end of loop forever */
TCP Reliability Optimizations

- TCP includes many tune/optimizations, e.g.,
  - the “small-packet problem”: sender sends a lot of small packets (e.g., telnet one char at a time)
    - Nagle’s algorithm: do not send data if there is small amount of data in send buffer and there is an unack’d segment [not required in PS2]
  - the “ack inefficiency” problem: receiver sends too many ACKs, no chance of combing ACK with data
    - Delayed ack to reduce # of ACKs/combine ACK with reply [not required in Assignment 2]
## Offline Read: TCP Ack Generation Rules

[RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. # . Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
Key Design Issues/Points

- Loss detection/timeout

- Connection management
  - Initial seq#
  - Resource release

- Window size
Backup Slides
TCP Developed in Time

1974
TCP described by Cerf & Kahn
IEEE Trans Comm

1975
Three-way Handshake
Tomlinson
SIGCOMM75

1981
TCP RFC 793/791

1983
TCP impl.
BSD 4.2

1984
Nagel Alg

1986
Kahn Alg to estimate RTT

1988
Fast Retransmit
BSD 4.3 Tahoe
SIGCOMM88

1990
Delayed ACK
BSD 4.3 Reno
SIGCOMM88

1996
SACK TCP
Selective Ack