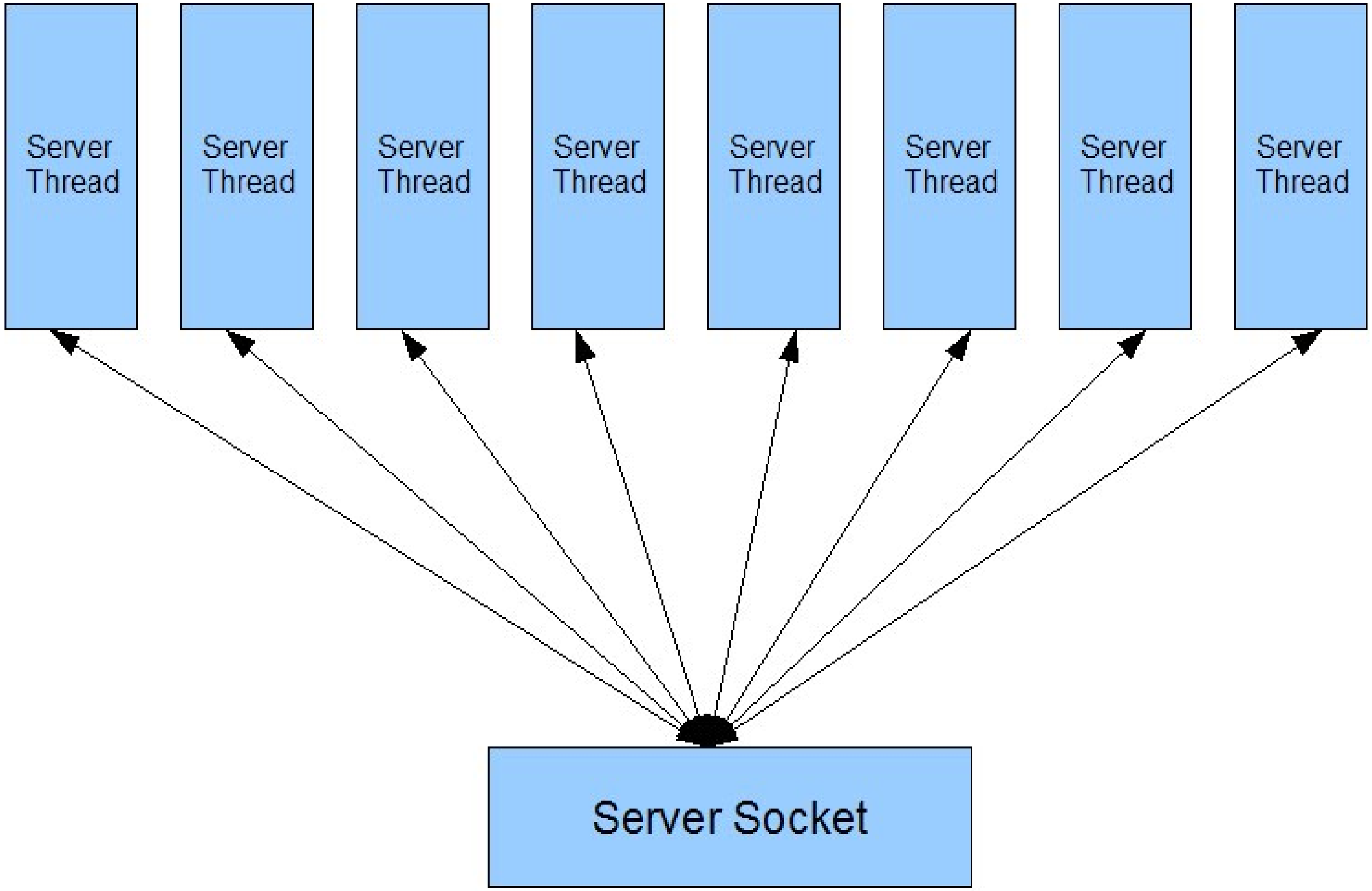
SO_REUSEPORT
Scaling Techniques for Servers with High Connection Rates

Ying Cai
ycai@google.com

Problems

- Servers with high connection/transaction rates
  - TCP servers, e.g. web server
  - UDP servers, e.g. DNS server
- On multi-core systems, using multiple servicing threads, e.g. one thread per servicing core.
  - The single server socket becomes bottleneck
  - Cache line bounces
  - Hard to achieve load balance
  - Things will only get worse with more cores

Scenario


Server Thread
Server Thread
Server Thread
Server Thread
Server Thread
Server Thread
Server Thread
Server Thread
Server Socket

Single TCP Server Socket - Solution 1

- Use a listener thread to dispatch established connections to server threads
  - The single listener thread becomes bottleneck due to high connection rate
  - Cache misses of the socket structure
  - Load balance is not an issue here

Single TCP Server Socket - Solution 2

- All server threads accept() on the single server socket
  - Lock contention on the server socket
  - Cache line bouncing of the server socket
  - Loads (number of accepted connections per thread) are usually not balanced
    - Larger latency on busier CPUs
    - It can almost be achieved by accept() at random intervals, but it is hard to decide the interval value, and may introduce latency.

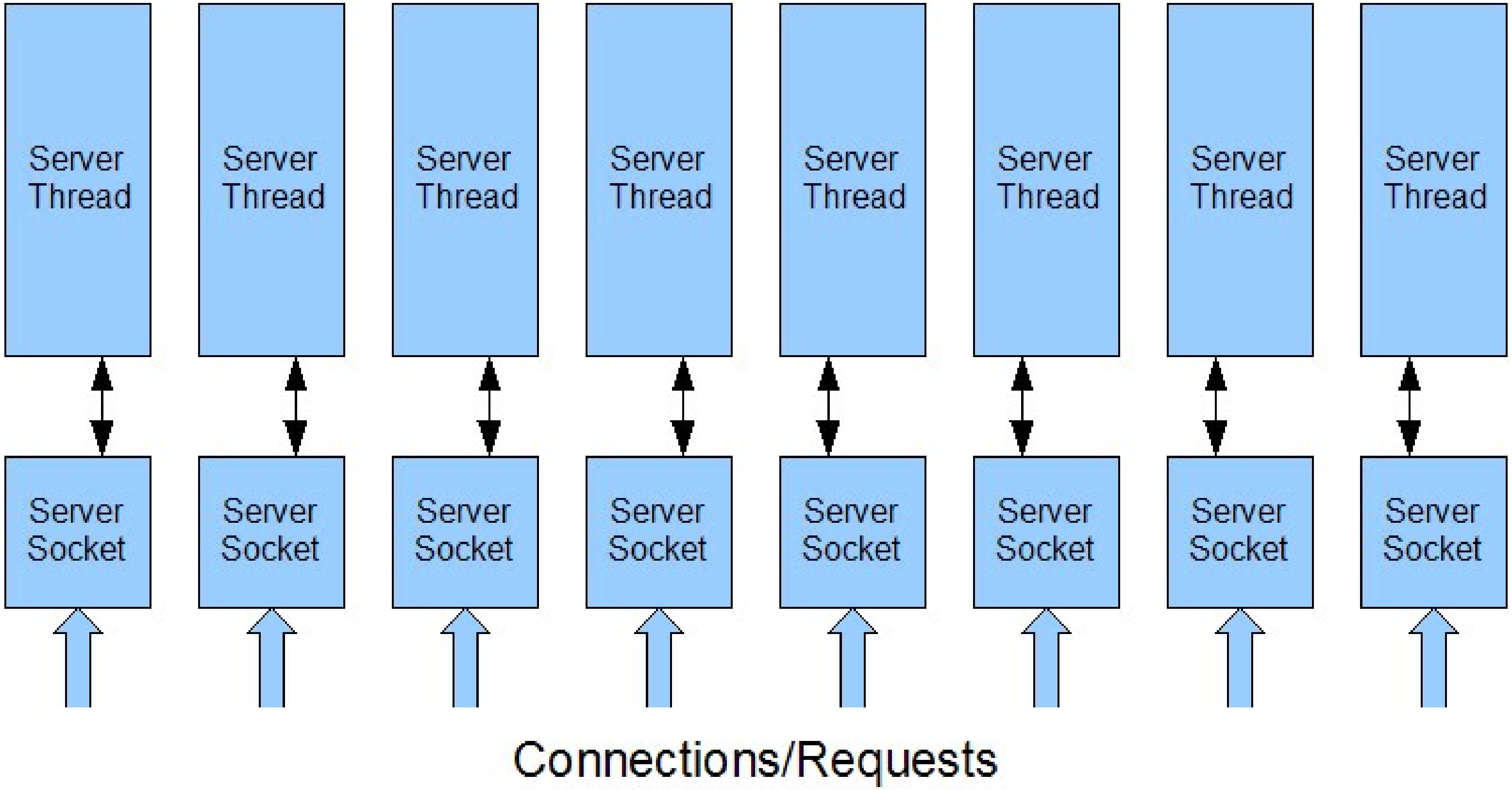
Single UDP Server Socket

- Have same issues as TCP
- SO_REUSEADDR allows multiple UDP sockets bind() to the same local IP address and UDP port, but it will not distribute packets among them. It is not designed to solve this problem.

New Socket Option - SO_REUSEPORT

- Allow multiple sockets bind()/listen() to the same local address and TCP/UDP port
  - Every thread can have its own server socket
  - No locking contention on the server socket
- Load balance is achieved by kernel - kernel randomly picks a socket to receive the TCP connection or UDP request
- For security reason, all these sockets must be opened by the same user, so other users can not "steal" packets

SO_REUSEPORT



How to enable

1. sysctl net.core.allow_reuseport=1
2. Before bind(), setsockopt SO_REUSEADDR and SO_REUSEPORT
3. Then the same as a normal socket - bind()/listen() /accept()

Status

- Developed by Tom Herbert at Google
- Submitted to upstream, but has not been accepted yet
- Deployed internally at Google
  - Will be deployed on Google Front End servers
  - Already deployed on Google DNS servers. Some test shows change from 50k request/s with some losses to 80k request/s without loss.

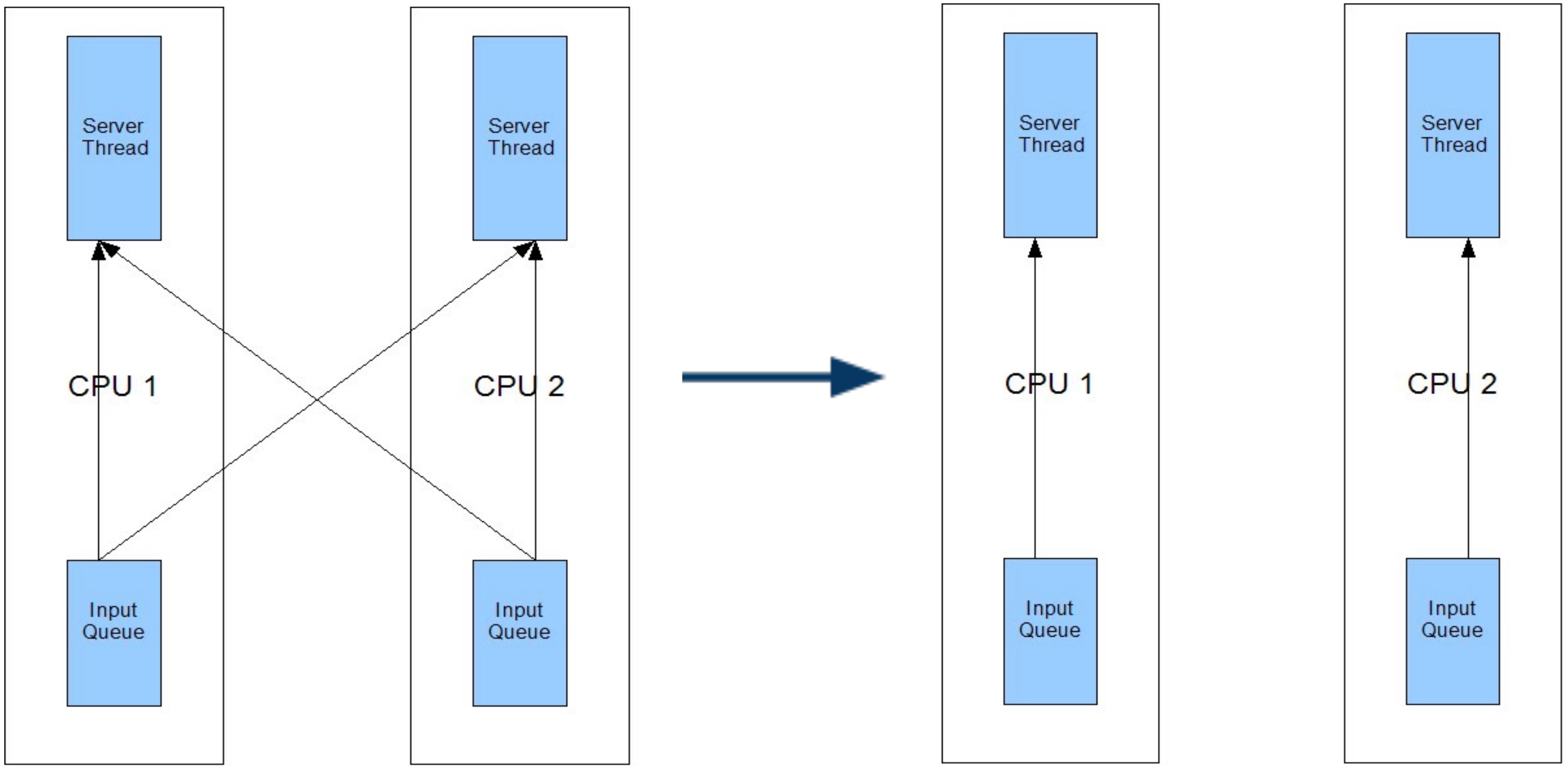
Known Issues - Hashing

- Hash is based on 4 tuples and the number of server sockets, so if the number is changed (server socket opened/closed), a packet may be hash into a different socket
  - TCP connection can not be established
- Solution 1: Use fixed number of server sockets
- Solution 2: Allow multiple server sockets to share the TCP request table
- Solution 3: Do not use hash, pick local server socket which is on the same CPU

Known Issues - Cache

- Have not solved the cache line bouncing problem completely
  - Solved: The accepting thread is the processing thread
  - Unsolved: The processed packets can be from another CPU
    - Instead of distribute randomly, deliver to the thread/socket on the same CPU

Silo'ing


Server Thread
Server Thread
CPU 1
CPU 2
Input Queue
Input Queue
Server Thread
Server Thread
CPU 1
CPU 2
Input Queue
Input Queue

Interactions with RFS/RPS/XPS-mq - TCP

- Bind server theads to CPUs
- RPS (Receive Packet Steering) distributes the TCP SYN packets to CPUs
- TCP connection is accept() by the server thread bound to the CPU
- Use XPS-mq (Transmit Packet Steering for multiqueue) to send replies using the transmit queue associated with this CPU
- Either RFS (Receive Flow Steering) or RPS can guarantee that succeeding packets of the same connection will be delivered to that CPU

Interactions with RFS/RPS/XPS-mq - TCP

- RFS/RPS is not needed is RxQs are set up per CPU
- But hardware may not support as many RxQs as CPUs

Interactions with RFS/RPS/XPS-mq - UDP

- Similar to TCP

Interactions with scheduler

- Some scheduler mechanism may harm the performance
  - Affine wakeup - too aggressive in certain conditions, causing cache misses

Other Scalability Issues

- Locking contentions
  - HTB Qdisc

Questions?