Fastsocket

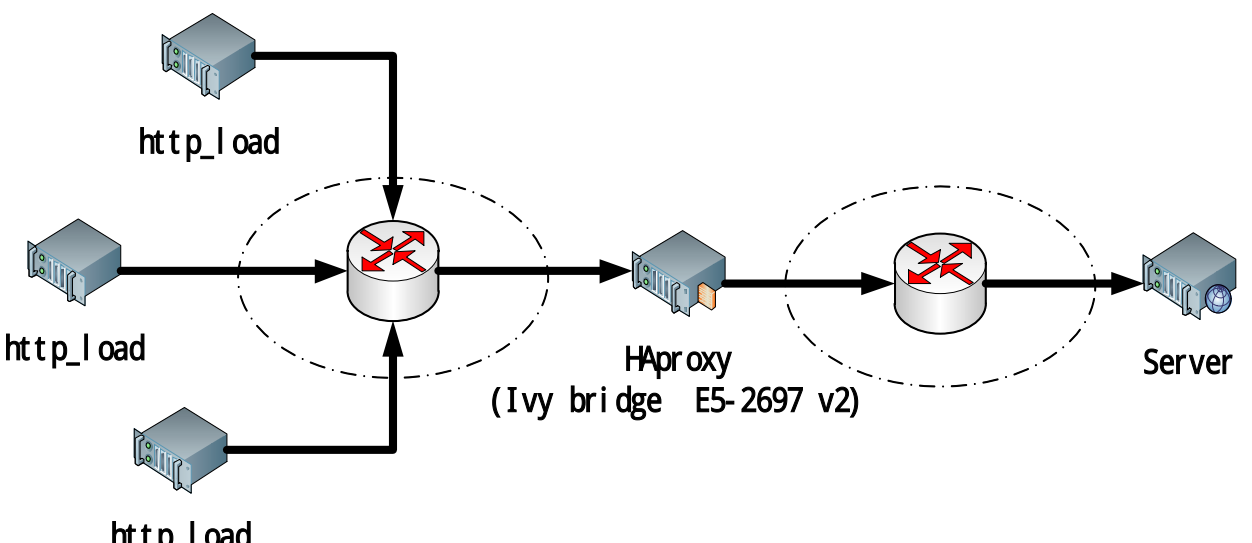
A Scalable Kernel TCP Socket

Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Junjie Mao

SINA & Tsinghua Univ

Testing Environment

- HAProxy: Open source TCP/HTTP loadbalancer.
- OS: CentOS-6.2 (Kernel : 2.6.32-220.23.1.el6)
- CPU: Intel Ivy-Bridge E5-2697-v2 (12 core) * 2
- NIC: Intel X520 (Support Flow-Director)
- Scenario: short TCP connections



Fastsocket

- Scalability Performance
- Single Core Performance
- Production System Feasibility
- Future Work

Socket API Problem

Socket API is fundamental for most network applications

- Kernel eats too much CPU cycles
- Less CPU cycles left to Application
- Call for more efficient kernel socket API

Performance on Multicore System

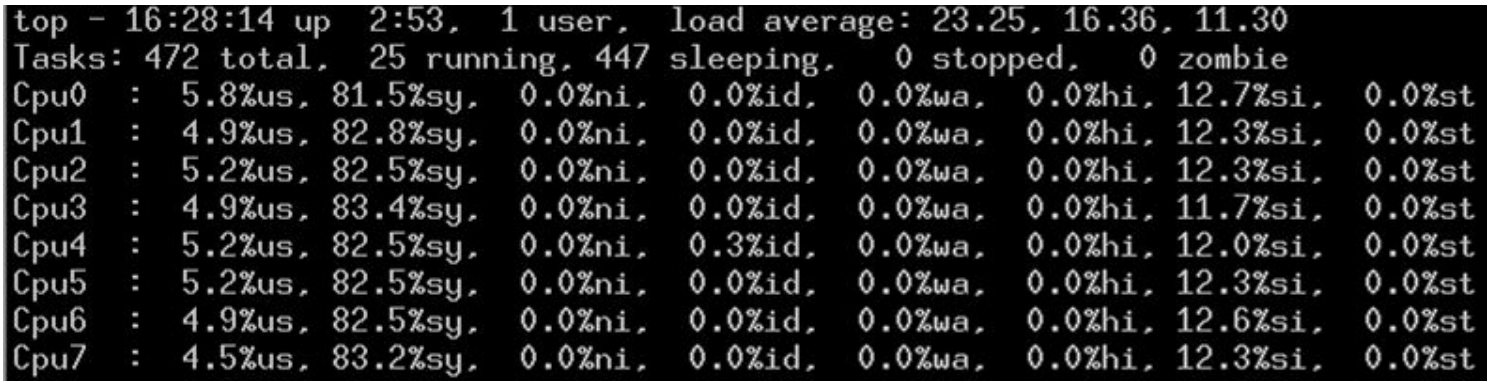
Theoretical Overall Performance:

*Single core performance * Available CPU cores*

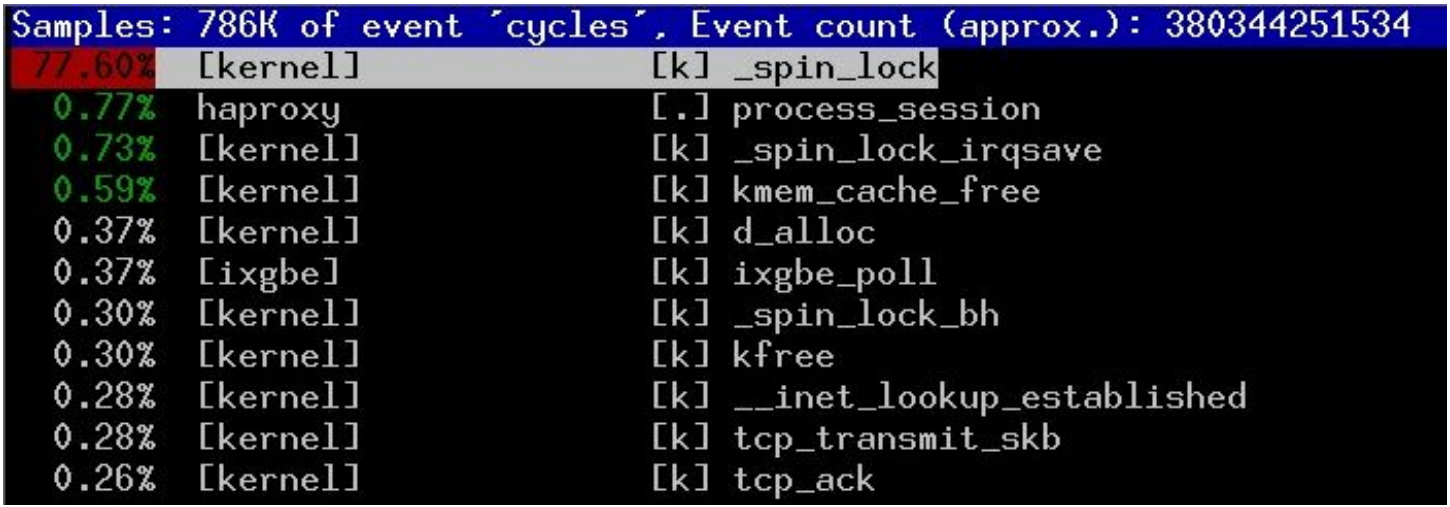
How close can we reach the limit: ***Scalability.***

Kernel Inefficiency

More than 90% CPU is consumed by Kernel



Synchronization Overhead

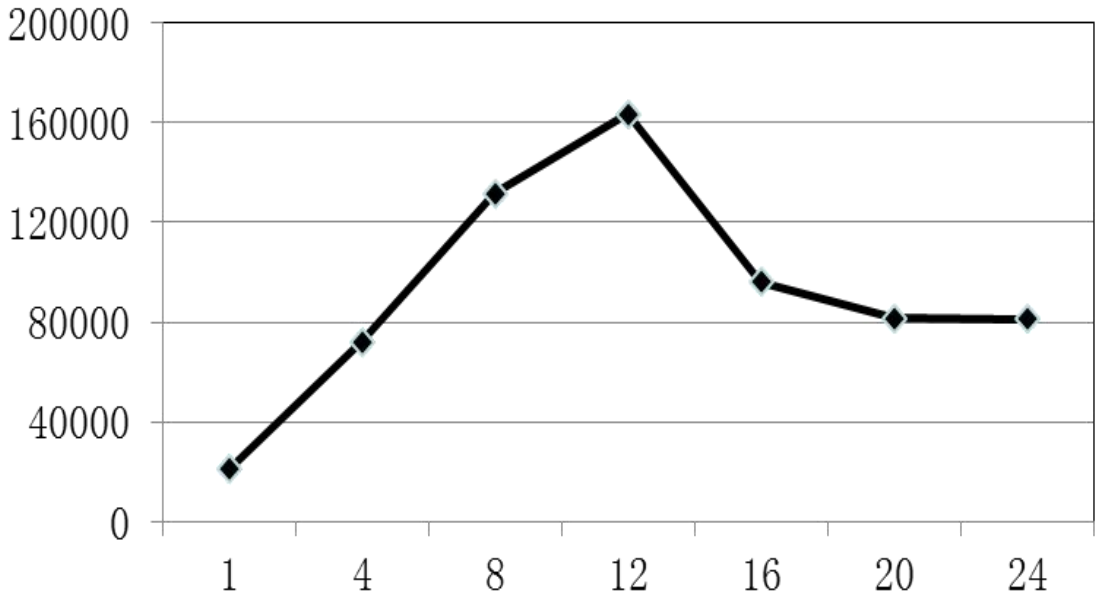


Almost 80% CPU time is spent on locking.

Scalability Problem

HTTP CPS (Connection Per Second) throughput with different number of CPU cores.

Scalability is KEY to multicore system capacity.



Dilemma

How to update single machine capacity?

- Spend more for more CPU cores
- Capacity dose not improve but get worse
- Just awkward

What to do

- Hardware Assistance

  Limited effects. (NIC offload feature)

- Data Plane Mode

  You need to implement your own TCP/IP stack. (DPDK)

  Lacks of kernel generality and full features.

  Other production limits.

- Change Linux kernel

  Keep improving but not enough. (Linux upstream)

  Need to modify application code. (Megapipe OSDI)

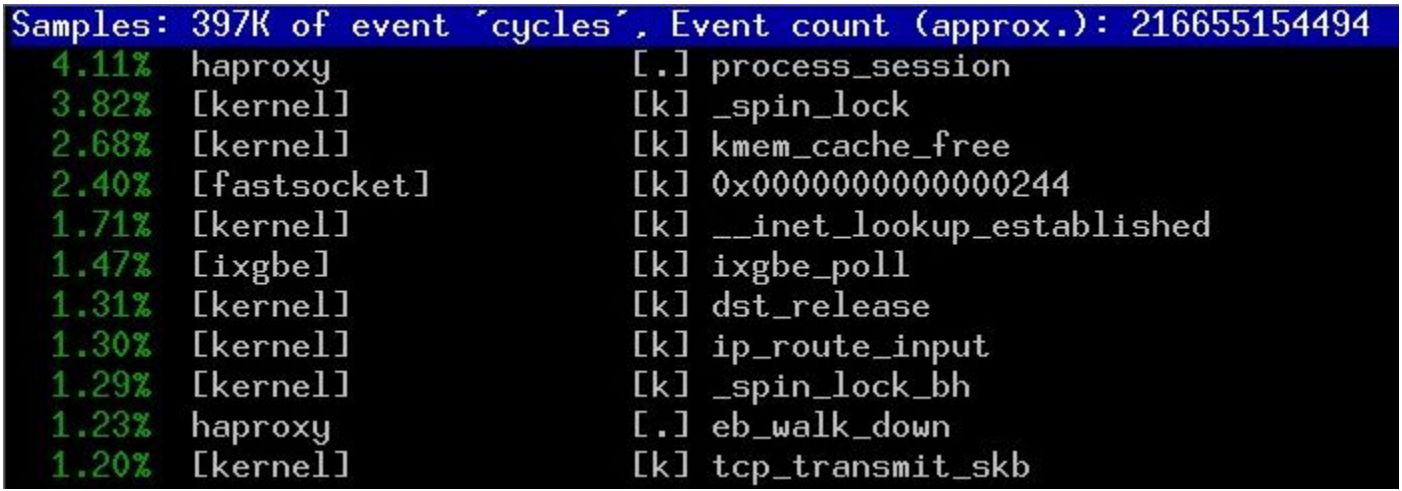
What to do

	Accept queue	Conn. Locality	Socket API	Event Handling	Packet I/O	Application Modification	Kernel Modification
PSIO [12], DPDK [4], PF_RING [7], netmap [21]	No TCP stack				**Batched**	No interface for transport layer	**No (NIC driver)**
Linux-2.6	Shared	None	BSD socket	Syscalls	Per packet	**Transparent**	**No**
Linux-3.9	**Per-core**	None	BSD socket	Syscalls	Per packet	Add option SO_REUSEPORT	**No**
Affinity-Accept [37]	**Per-core**	**Yes**	BSD socket	Syscalls	Per packet	**Transparent**	Yes
MegaPipe [28]	**Per-core**	**Yes**	lwsocket	**Batched syscalls**	Per packet	Event model to completion I/O	Yes
FlexSC [40], VOS [43]	Shared	None	BSD socket	**Batched syscalls**	Per packet	Change to use new API	Yes
mTCP	**Per-core**	**Yes**	**User-level socket**	**Batched function calls**	**Batched**	Socket API to mTCP API	**No (NIC driver)**

We Choose

- Change Linux kernel
  - BSD socket
  - table-level connection partition in TCP stack
  - guarantees connection locality for
    - passive connection
    - active connection.

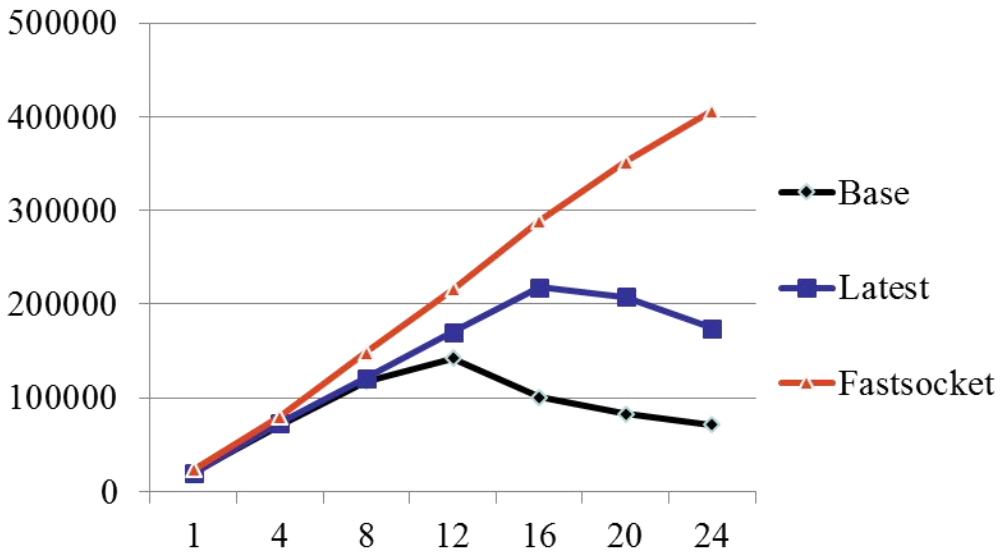
Fastsocket Scalability



Great! CPU is doing more for haproxy.

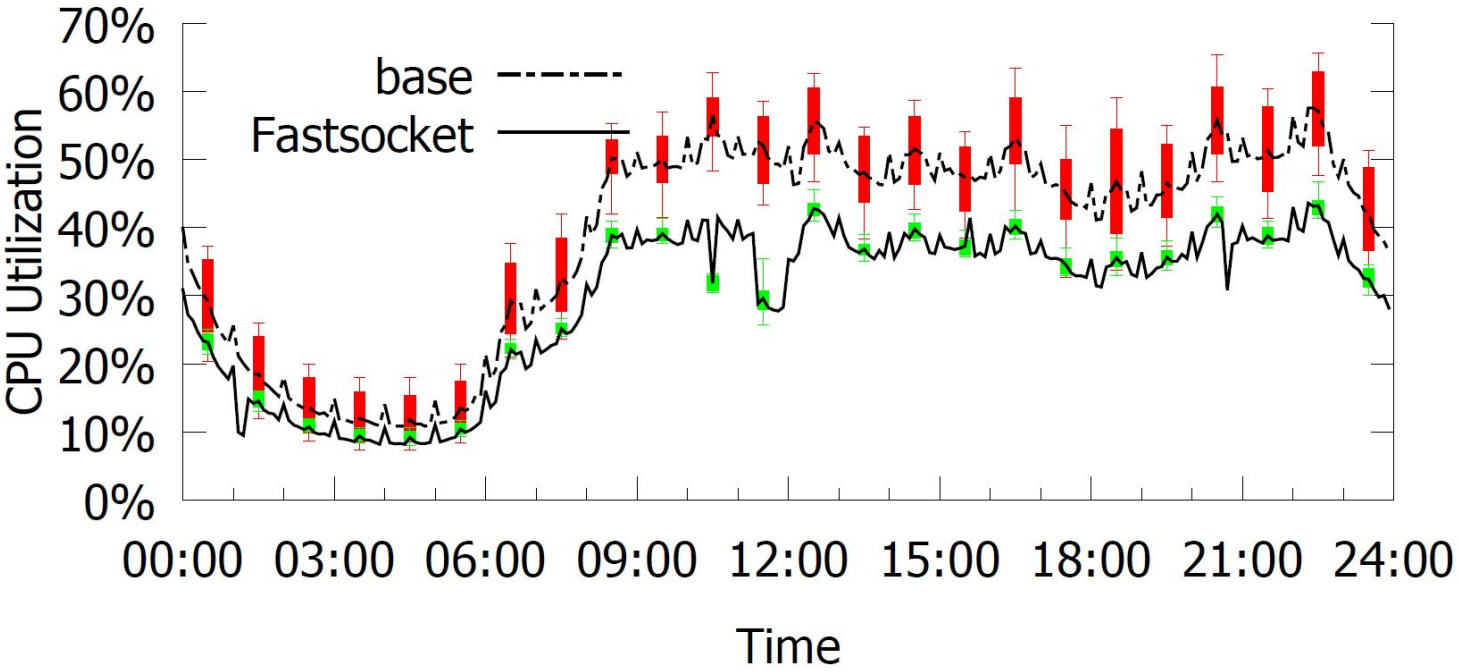
Fastsocket Scalability

Scalability is key for multicore performance.



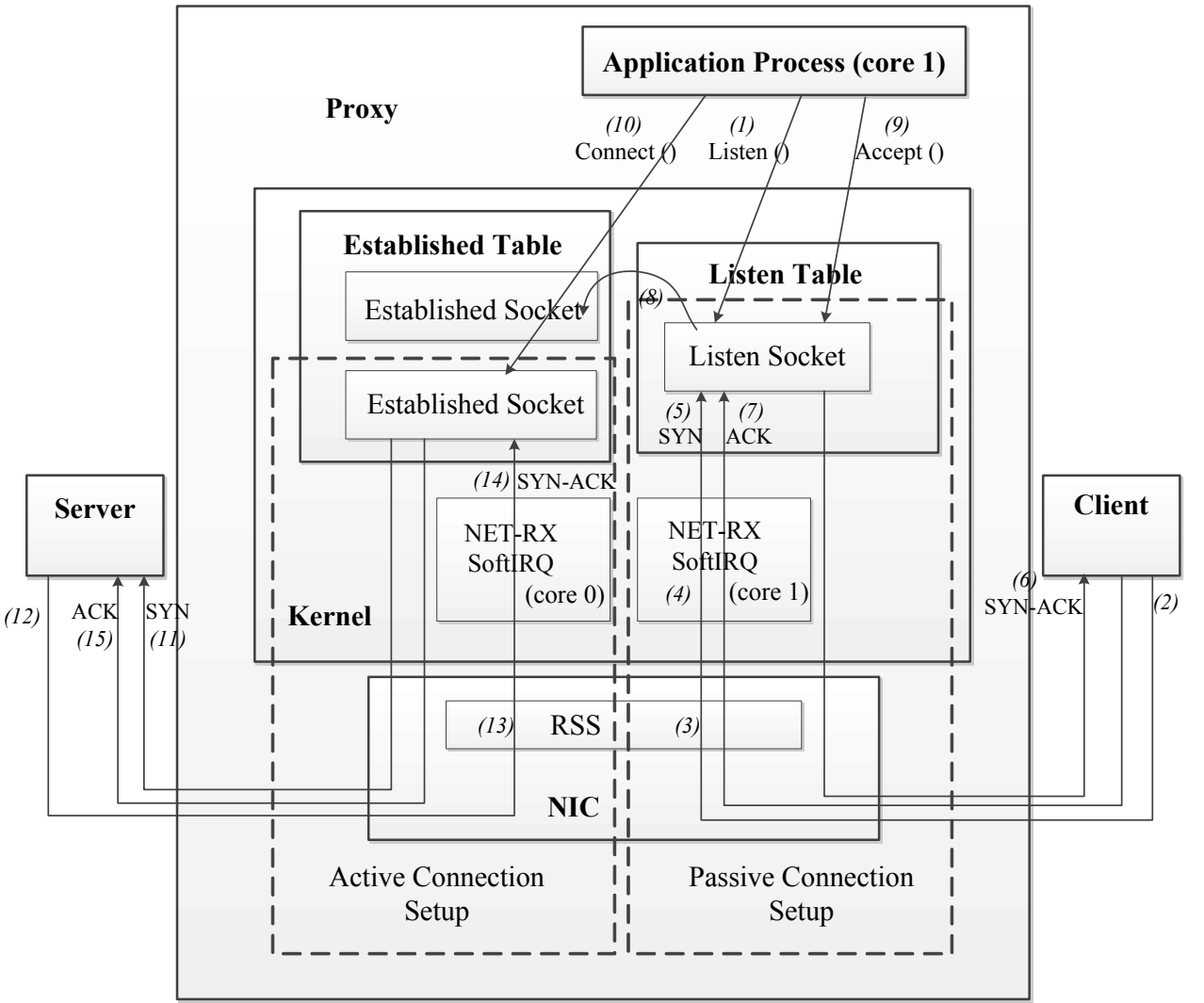
Fastsocket: 5.8X Base and 2.3X Lastest.

Production System Evaluation



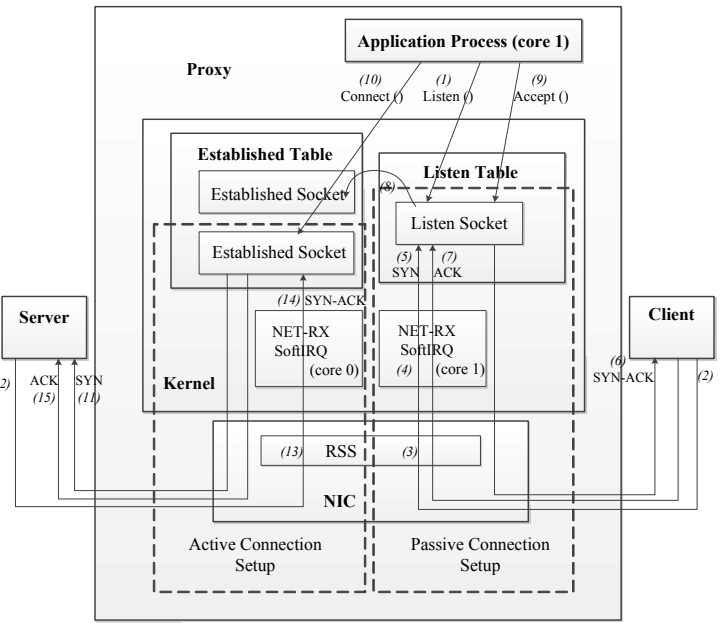
Two 8-core HAProxy (HTTP proxy) servers handling same amount of traffic, with and without Fastsocket.

work flow for a proxy on a TCP connection



work flow for a proxy on a TCP connection

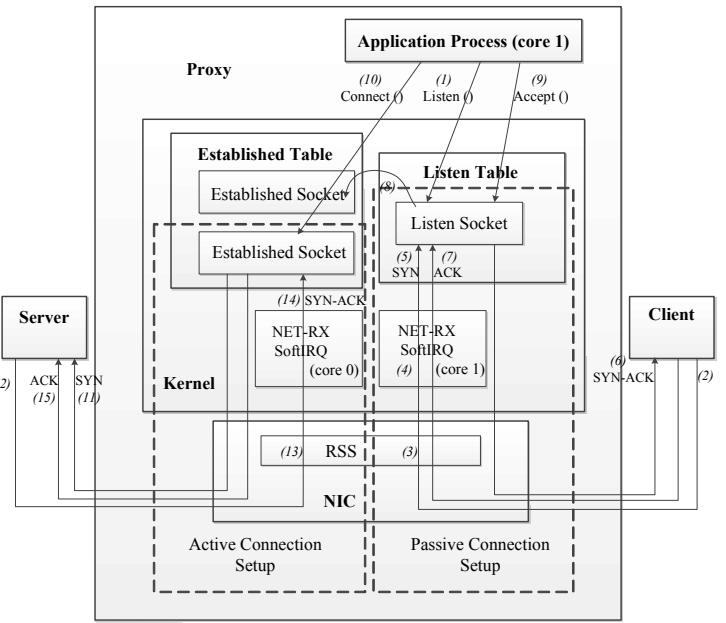
Passive Connections



(1) In Linux kernel, listen socket is used to represent a service port (e.g. 192.168.0.1:80) to wait for incoming connections. Upon start,the proxy application invokes listen() system call, and while inside kernel, a listen socket is created and inserted into a global hash table called the listen table.

work flow for a proxy on a TCP connection

Passive Connections

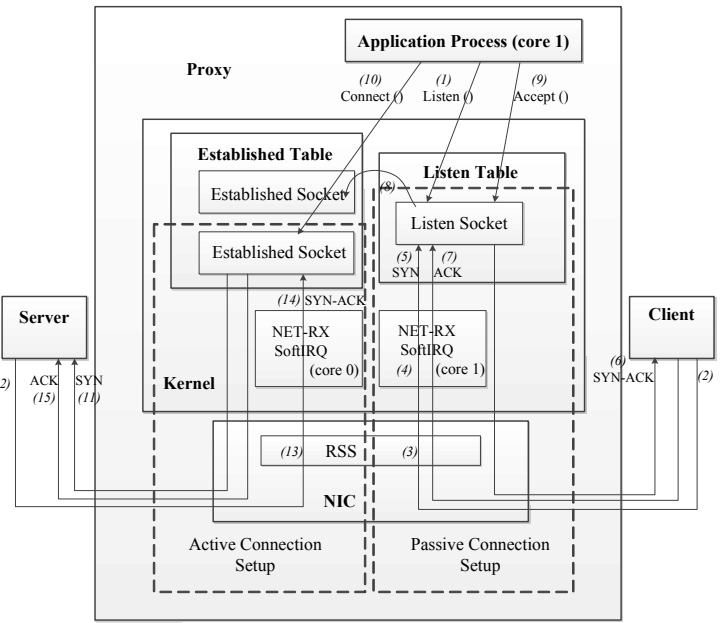


(1) In Linux kernel, listen socket is used to represent a service port (e.g. 192.168.0.1:80) to wait for incoming connections. Upon start,the proxy application invokes listen() system call, and while inside kernel, a listen socket is created and inserted into a global hash table called the listen table.

(2) To establish a connection to the proxy server, a client sends a SYN packet to start a three/ way handshake connection setup.

work flow for a proxy on a TCP connection

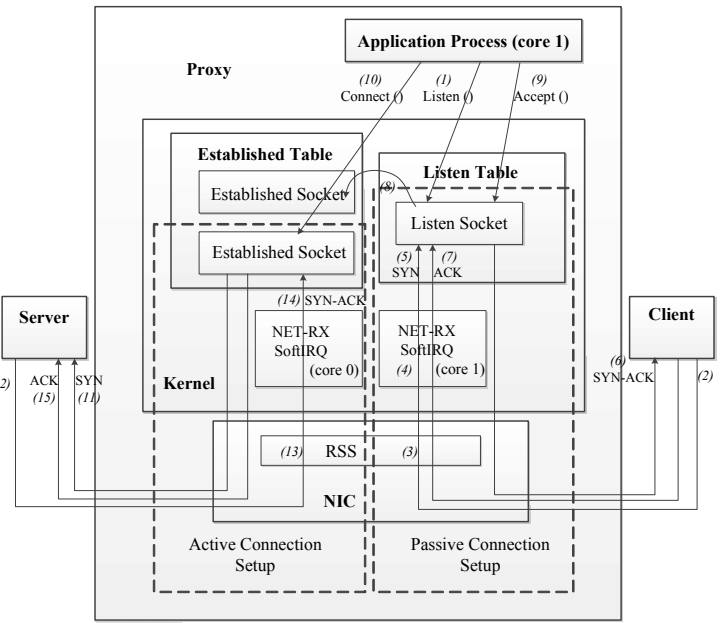
Passive Connections



(3/ 4) When the SYN packet arrived at the proxy NIC, either the hardware NIC RSS (Receive Side Scaling), or some software Mechanism. When mention software mechanism selects a CPU core to process the packet. The selection is based on the hash value of source IP address, destination IP address, source TCP port and destination TCP port, therefore, the rest incoming packets of this connection would always be delivered to the same CPU core.

work flow for a proxy on a TCP connection

Passive Connections

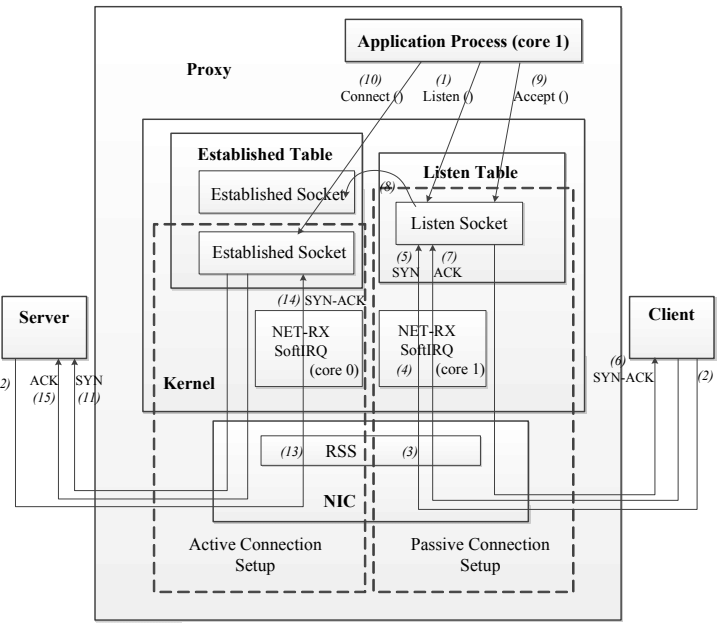


(5/ 6) The SYN packet matches the listen socket and is recorded as a half connection in the listen socket, and a SYN ACK packet is sent back to the client.

(7/ 8) When the proxy receives the ACK from the client, the half connection is moved into a accept queue of the listen socket and a
new established socket representing the established connection is created and inserted into another global hash table, called established table.

work flow for a proxy on a TCP connection

Passive Connections

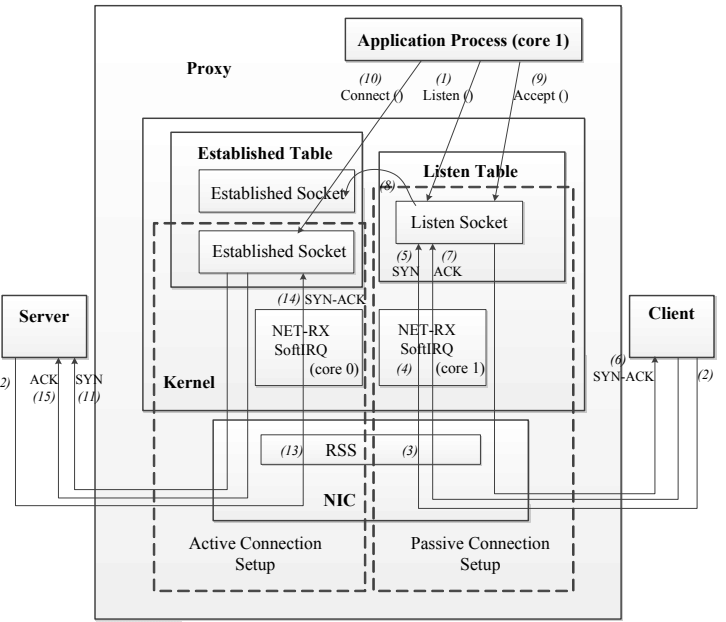


(9) When the application notices that there is new connection to serve, it calls accept() system call to pull the new connection out of accept queue of the listen socket.

Legacy Linux kernels uses **a single lock** to synchronize operations on data structures of the listen socket, such as the accept queue.

work flow for a proxy on a TCP connection

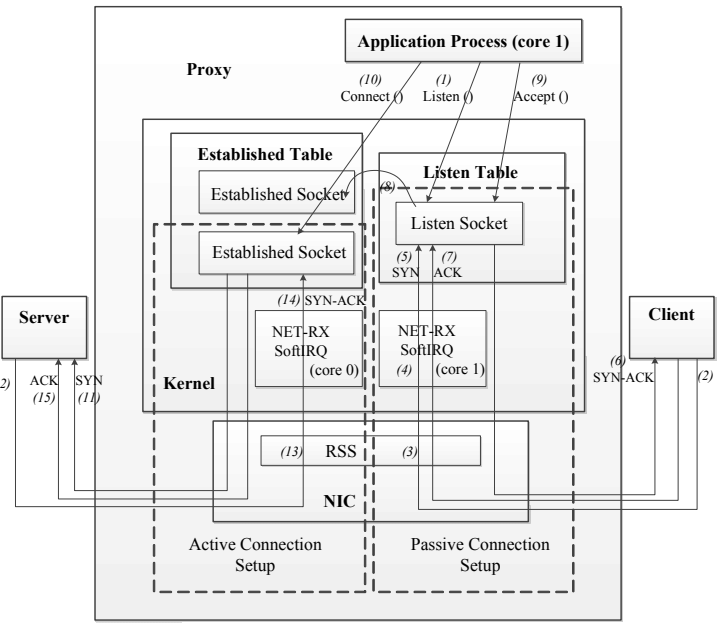
Active Connections



(10) As a proxy server, it needs to actively establish connections with the backend servers. In the example, suppose the proxy process is running on CPU core 1. On the active connection setup, the proxy creates a new socket and call connect() system call to start a connection request to the backend server.

work flow for a proxy on a TCP connection

Active Connections

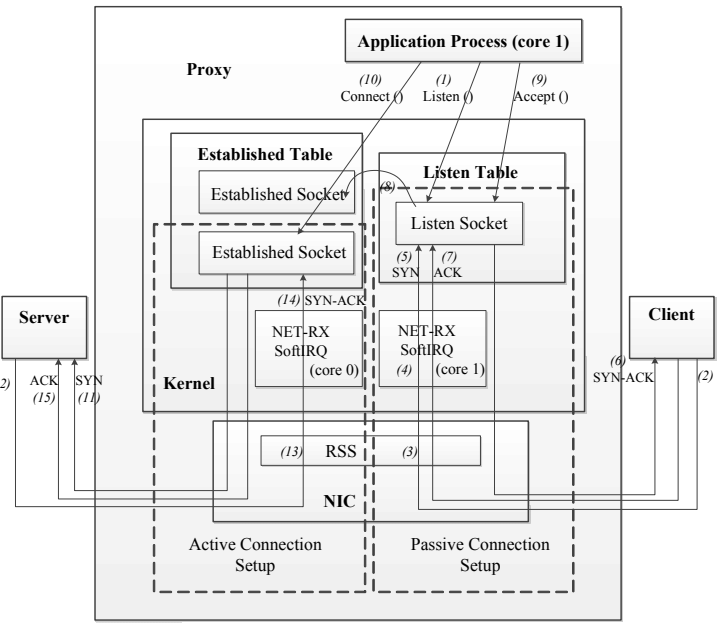


(11) The new socket is inserted into the global established table and a SYN pakcket is sent to the backend server.

(12/ 13) When the packet goes down to the NIC though the network stack, all the work is done on CPU core 1 where the application runs. The server responses with a SYN ACK packet and again proxy NIC RSS would randomly select a CPU core for this SYN ACK packet based on packet hash. Likely, it is not the same core that transmits the outgoing packets of the same connection. In this example, CPU core 0 is selected.

work flow for a proxy on a TCP connection

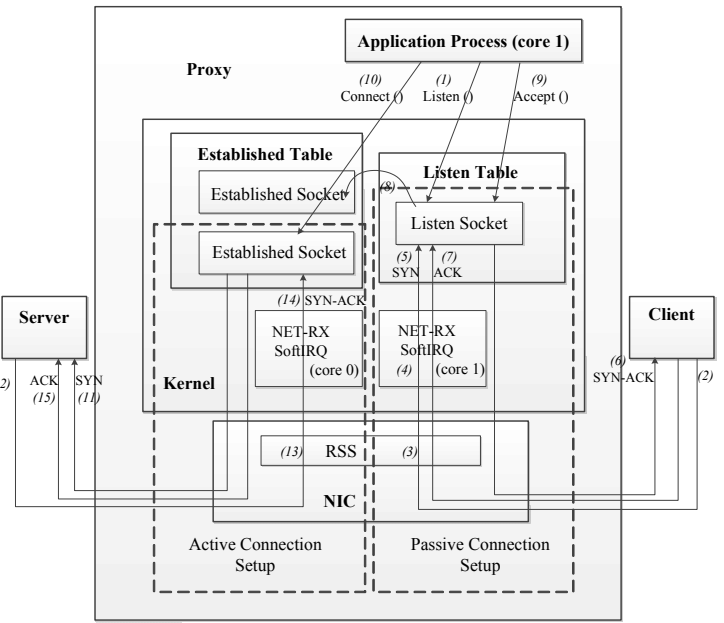
Active Connections



(14/ 15) The SYN ACK packet will finally marks the socket in the established table as established connection (14) and proxy responses the server with a ACK packet to finish the connection setup of the backend server.

work flow for a proxy on a TCP connection

Active Connections



(14/ 15) The SYN ACK packet will finally marks the socket in the established table as established connection (14) and proxy responses the server with a ACK packet to finish the connection setup of the backend server.

Notice that in (12/ 14), CPU core 0 is used to process NET RX SoftIRQ for reply packets from backend server. However, CPU core 1 is used to send out request packet and process the reply data in the application. The problem results in cache bouncing and degrade scalability.

Kernel Bottleneck

- Non Local Process of Connections
- Global TCP Control Block (TCB) Management
- Synchronization Overhead from VFS

Non Local Process of Connections

A given connection is processed in two phases:

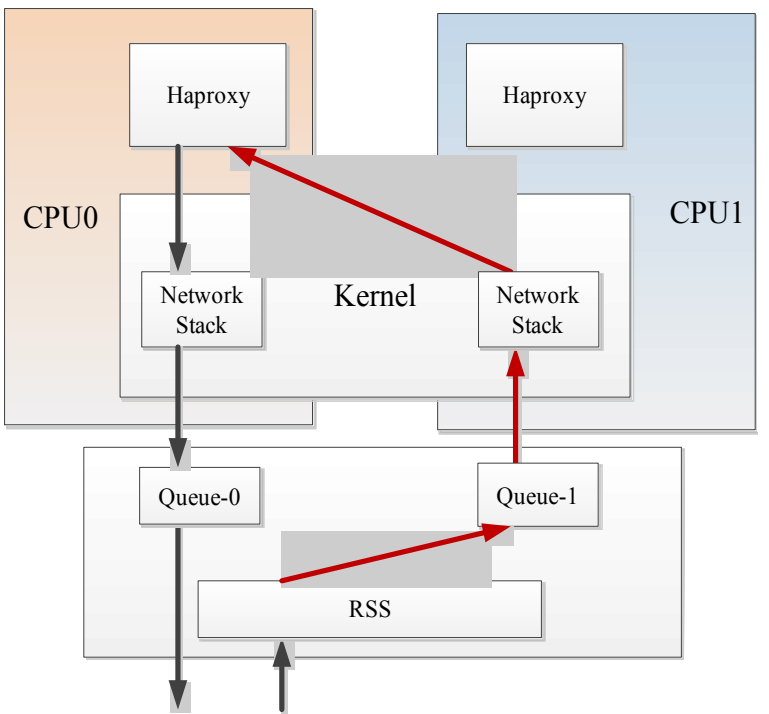
- Net-RX SoftIRQ in **interrupt context**
- Application and System Call in **process context**.

Two phases are often handles by different CPU cores:

- Introduces lock contentions.
- Causes CPU cache bouncing.

Non Local Process of Connections

Scenario that server actively connects out:



Global TCB Management

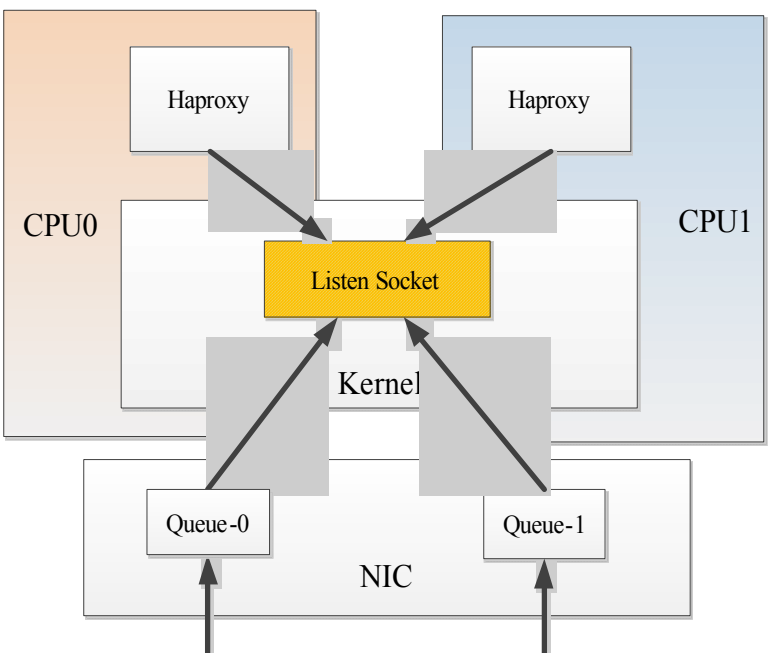
TCB is represented as socket in Linux kernel:

- Listen Socket:
  - A single listen socket is used for connection setup

- Established Socket:
  - A global hash table is used for connection management

Global TCB Management

Single listen socket HOTSPOT:



VFS Overhead

- Socket is abstracted under VFS
- Intensive synchronization for *Inode* and *Dentry* in VFS
- These overhead are inherited by socket

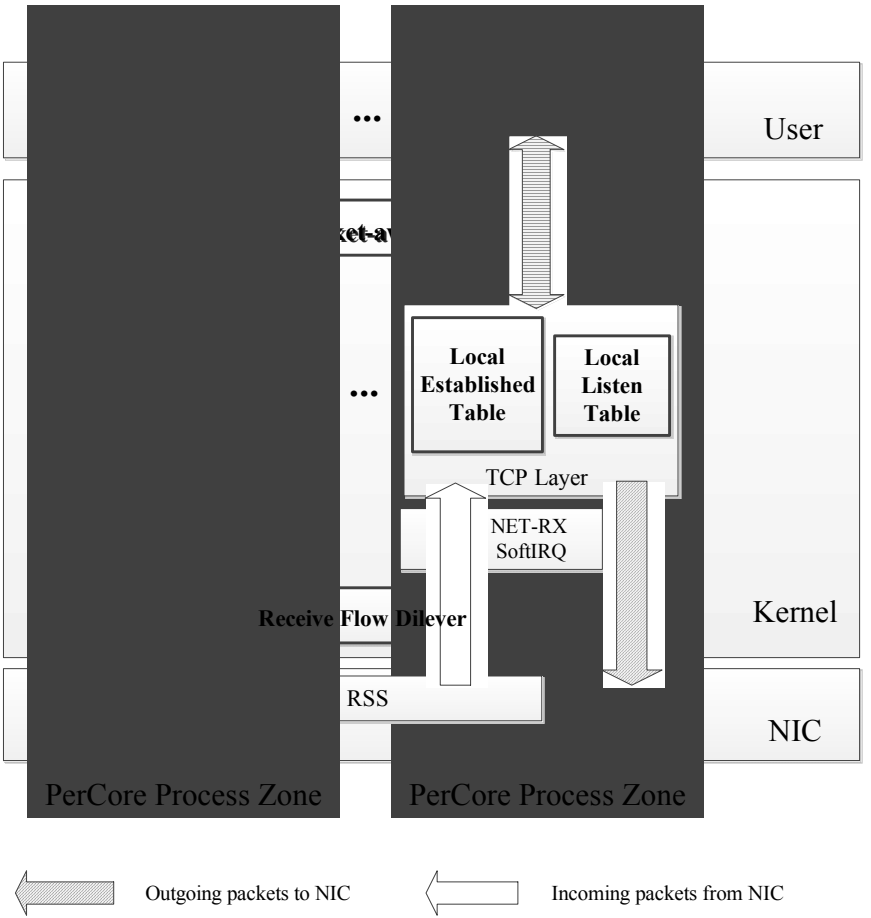
Methodology

- Resources Partition
- Local Processing

Design Component

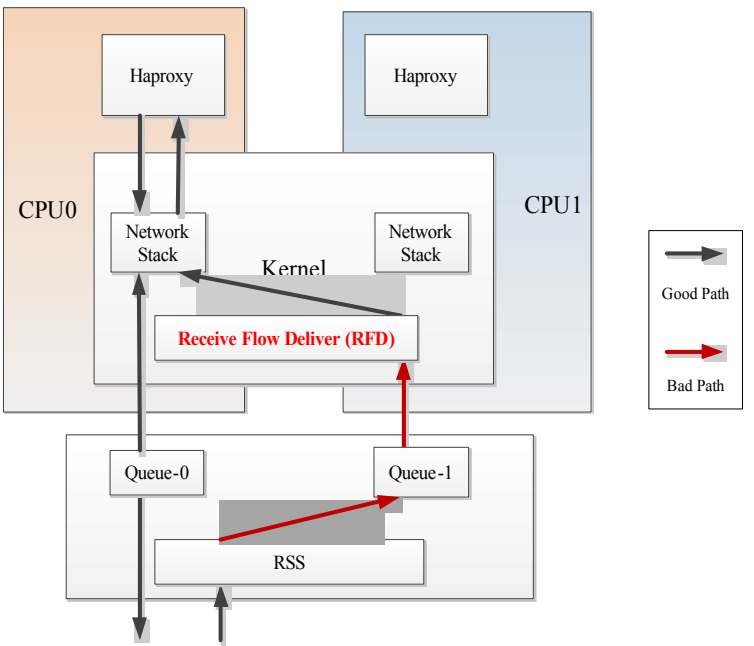
- Receive Flow Deliver
- Local Listen Table & Local Established Table
- Fastsocket-aware VFS

Fastsocket Architecture



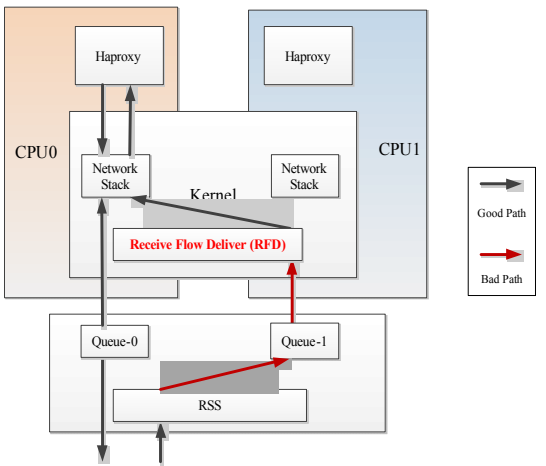
Receive Flow Deliver (RFD)

RFD delivers packets to the CPU core where application will further process them.



RFD can leverage advanced NIC features (Flow-Director)

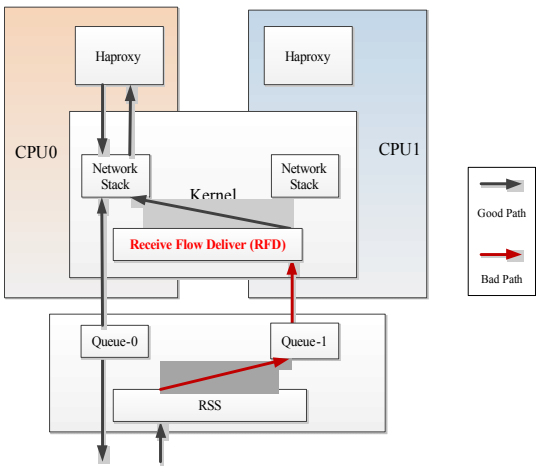
Receive Flow Deliver (RFD)



Build Active Connection Locality

The key insight is that kernel can encode current CPU core id into the source port when making an active connection request.

Receive Flow Deliver (RFD)



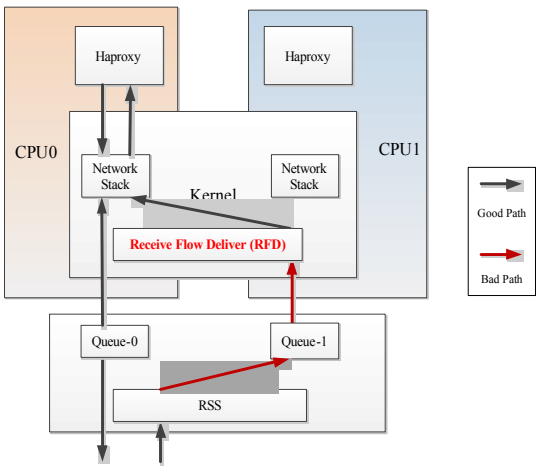
Build Active Connection Locality

The key insight is that kernel can encode current CPU core id into the source port when making an active connection request.

When the application running on CPU core c attempts to establish an active connection, RFD chooses a port p_{src} so that $c = hash(p_{src})$.

$hash(p) = p \;\&\&\; (\text{ROUND UP POWER OF } 2(n) / \; 1)$

Receive Flow Deliver (RFD)



Build Active Connection Locality

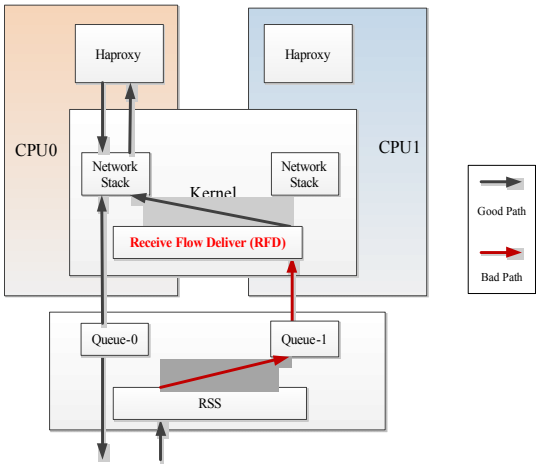
The key insight is that kernel can encode current CPU core id into the source port when making an active connection request.

When the application running on CPU core c attempts to establish an active connection, RFD chooses a port p_{src} so that $c = hash(p_{src})$.

$$hash(p) = p \;\&\&\; (\text{ROUND UP POWER OF } 2(n) / 1)$$

Upon receiving a response packet, RFD picks the destination port of the received packet p_{dst} which is the port RFD previously chosen, determines which CPU core should handle the packet by $hash(p_{dst})$

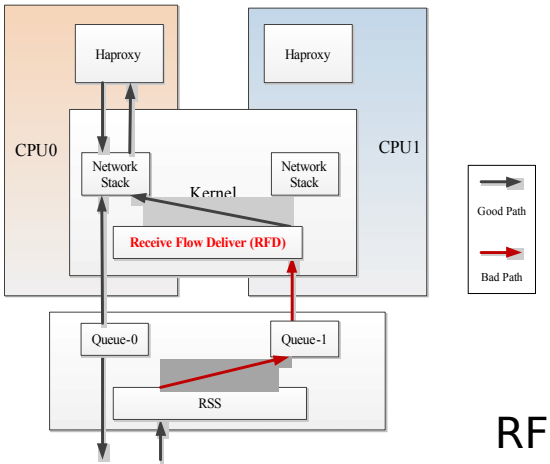
Receive Flow Deliver (RFD)



Retain Passive Connections Locality

Problem: passive connections and active connections can exist in one machine simultaneously, and incoming traffic consists of both passive incoming packets and active incoming packets.

Receive Flow Deliver (RFD)

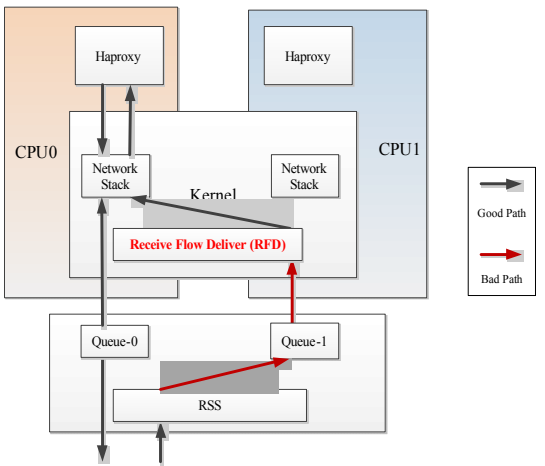


Retain Passive Connections Locality

Problem: passive connections and active connections can exist in one machine simultaneously, and incoming traffic consists of both passive incoming packets and active incoming packets.

RFD has to distinguish the two incoming packet categories since the hash function is designed for active connections and should only be applied to active incoming packets.

Receive Flow Deliver (RFD)



Retain Passive Connections Locality

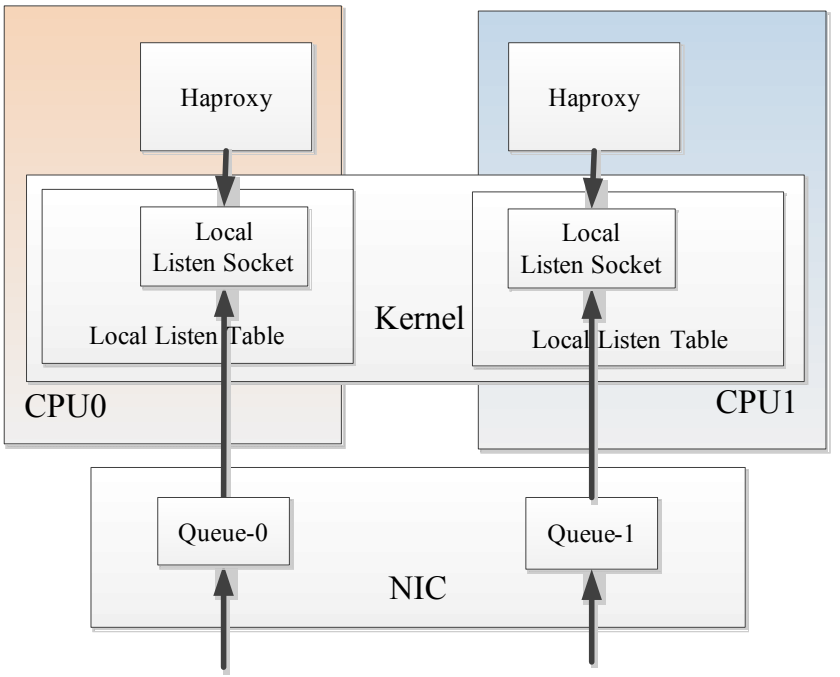
1. If the source port of a incoming packet is in the range of the well-known (less than 1024) ports, it should be a active incoming packet.

2. If the destination port is a well-known port, it is an passive incoming packet for the same reason.

3. (optional) If neither previous condition is met, we see whether the packet can match a listen socket. If so, it indicates that it is a passive incoming packet. Otherwise, the packet is an active incoming packet.

Local Listen Table

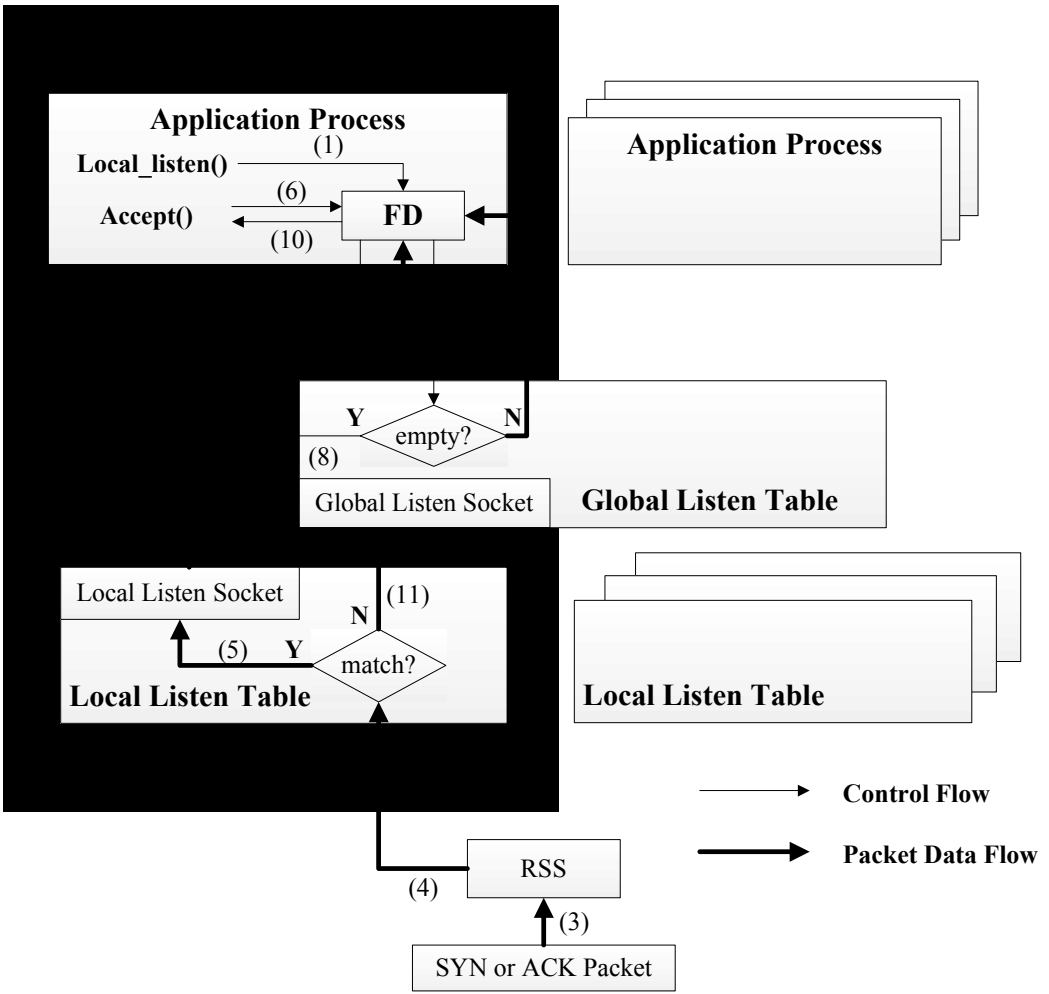
Clone the listen socket for each CPU core in LOCAL table.



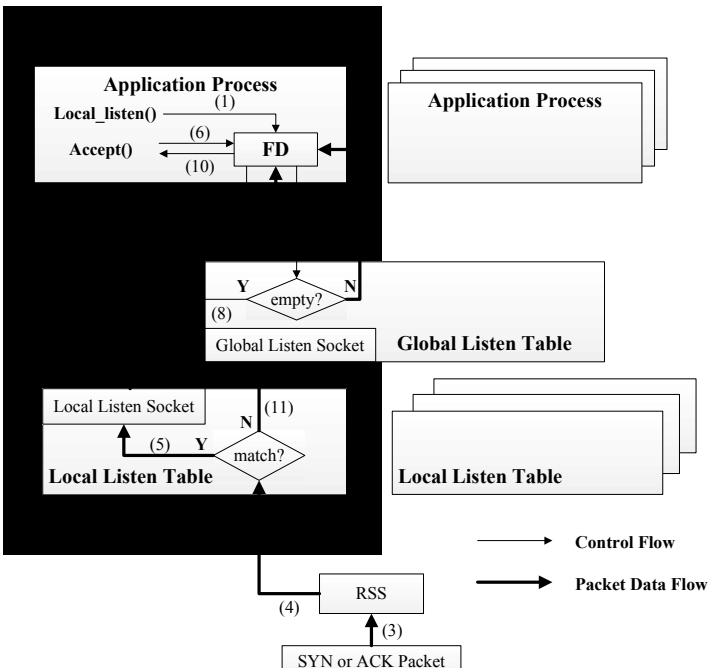
When Fastsocket is enabled, the server pre/ forks multiple processes, binds them with different CPU cores, each process invokes local listen( ) to inform kernel that the process wants to handle incoming connetion from the CPU core to which it have been set CPU affinity

Local Listen Table

Clone the listen socket for each CPU core in LOCAL table.



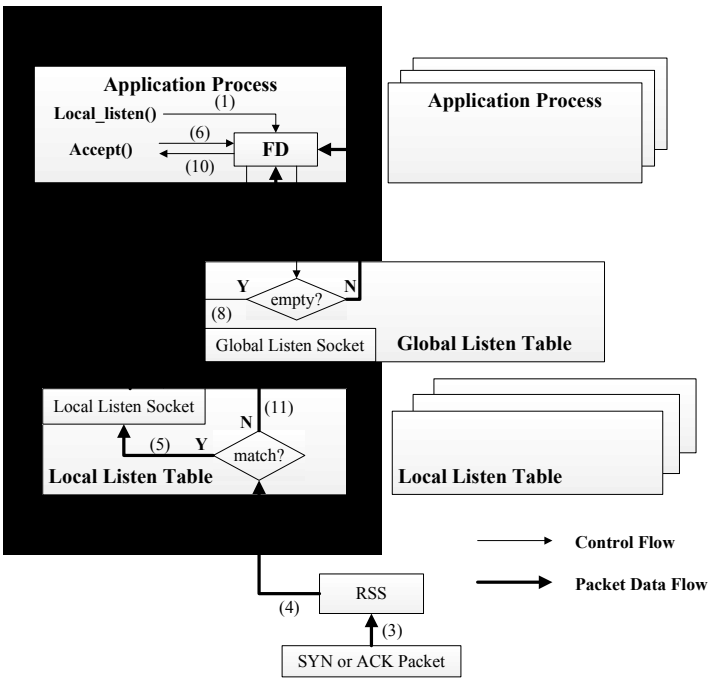
Local Listen Table



(1). As Figure shows, for the process bound on CPU core 0, a new listen socket is copied from the original listen socket and inserted into the local listen table of CPU core 0

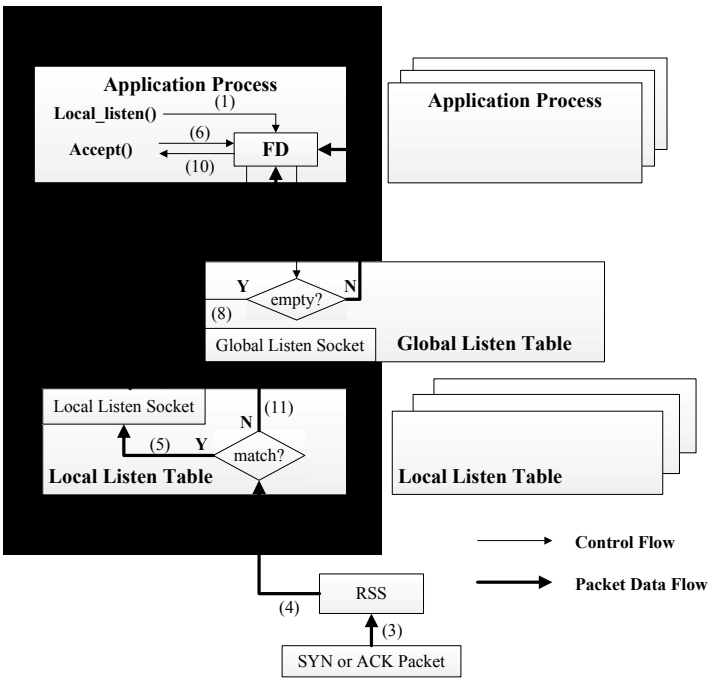
(2). We refer to the copied listen socket as the local listen socket and the original listen socket as the global listen socket.

Local Listen Table



When a SYN packet comes (3) and it is delivered to CPU core 0 by RSS, the kernel searches the local listen table of CPU core 0 to match a local listen socket (4). Without any failures, the local listen socket which is previously inserted by local listen()
will be matched (5).

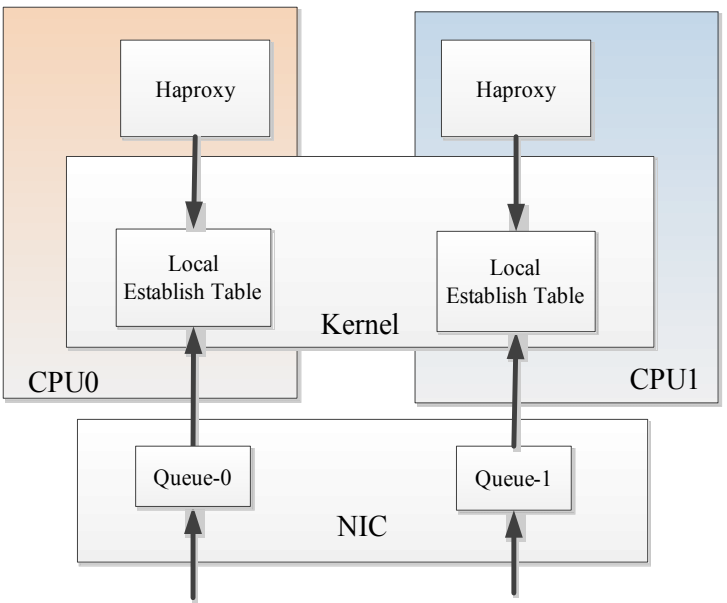
Local Listen Table



When the process bound on core 0 issues accept() for new connections (6), the kernel first checks the accept queue of the global listen socket (7). Under normal operation, the accept queue is always empty and thus this check is done without locking, as we will explain it later in this section. Then the kernel will check the local listen table of core 0 for any new connection (8). Naturally, the previously established connection can be found (9) and returned to application process (10).

Local Established Table

Established sockets are managed in LOCAL table.



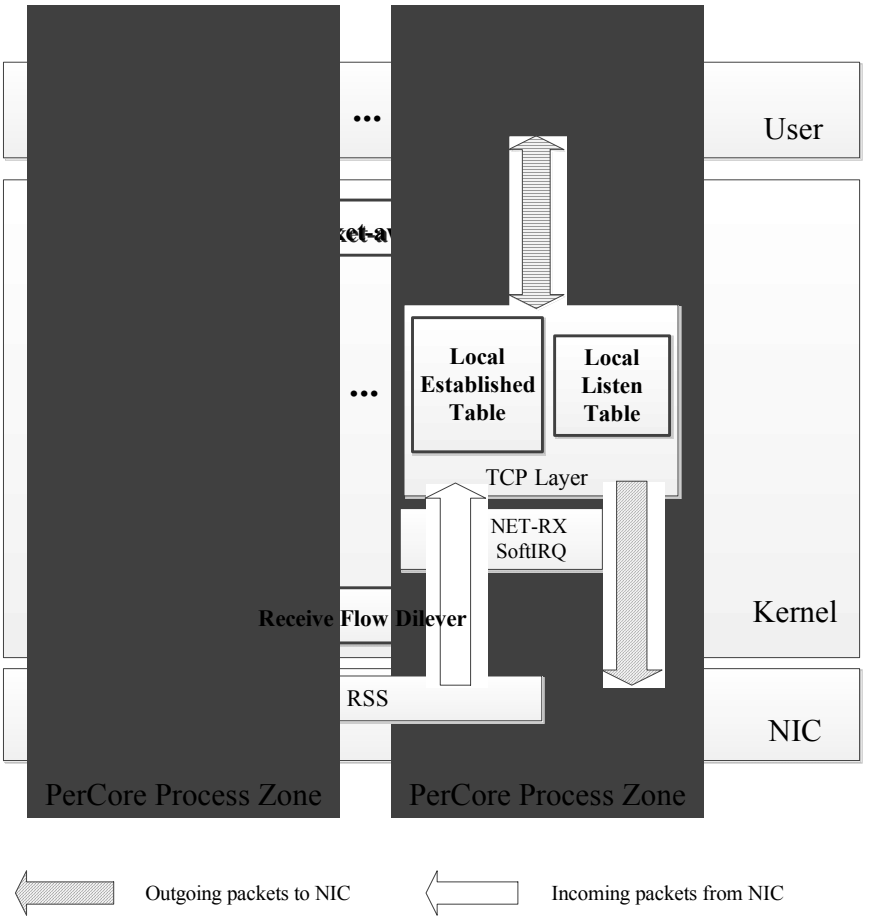
However,there is no guarantee that when processing a incoming packet in NET RX SoftIRQ , the packet is always processed in the same CPU core that has the corresponding socket for the connection in the local established table.

Fastsocket-aware VFS

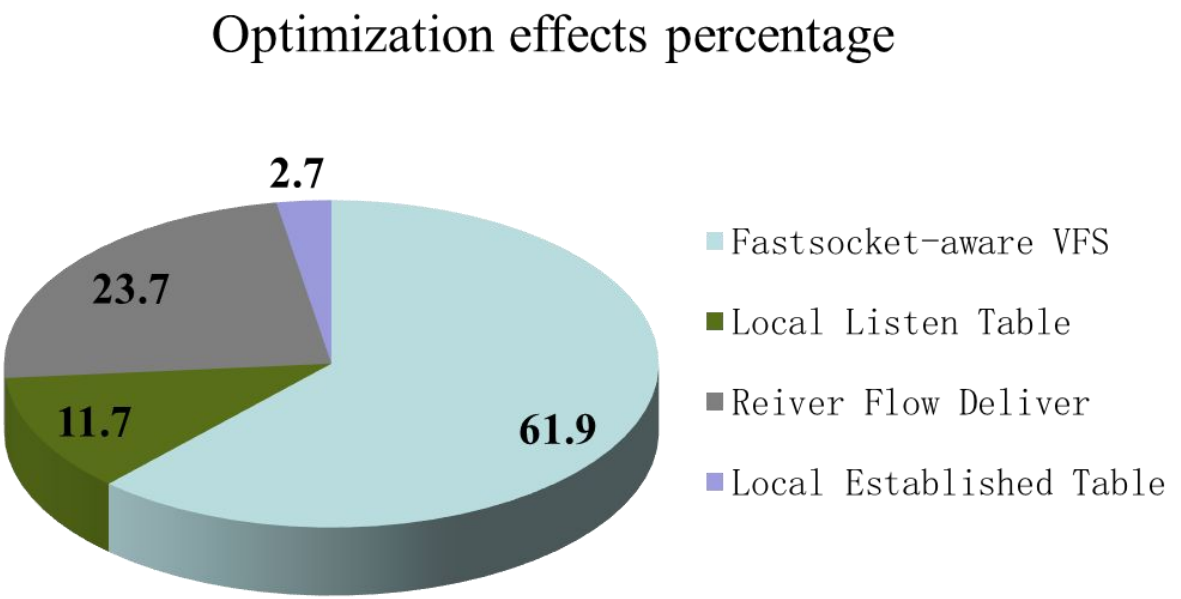
Provide a FAST path for socket in VFS:

- *Inode* and *Dentry* are useless for socket
- Bypass the unnecessary lock-intensive routines
- Retain enough to be compatible

Fastsocket Architecture



Optimization Effects



Intel Hyper-Threading

Further boot performance 20% with Intel HT.

	E5-2697-v2
Fastsocket	406632
Fastsocket-HT	476331(117.2%)

Fastsocket-HT:476.3k cps, 3.87m pps, 3.1G bps (short connection)

Fastsocket

- Scalability Performance
- Single Core Performance
- Production System Feasibility
- Future Work

Methodology

- Network Specialization
- Cross-Layer Design

Network Specialization

General service provided inside Linux kernel:

- Slab
- Epoll
- VFS
- *Timer* etc.

Customize these general services for Network

Fastsocket Skb-Pool

- Percore skb pool
- Combine skb header and data
- Local Pool and Recycle Pool (Flow-Director)

Fastsocket Fast-Epoll

Motivation:

- Epoll entries are managed in RB-Tree
- Search the RB-Tree each time when calling *epoll_ctl()*
- Memory access is a killer for performance

Solution:

- TCP Connection rarely shared across processes
- Cache Epoll entry in *file* structure to save the search

Cross-Layer Optimization

- What is cross-layer optimization?
- Overall optimization with Cross-Layer design
  - Direct-TCP
  - Receive-CPU-Selection

Fastsocket Direct-TCP

- Record input route information in TCP socket
- Lookup socket directly before network stack
- Read input route information from socket
- Mark the packet as routed

Fastsocket Receive-CPU-Selection

Similar to Google RFS

- Application marks current CPU id in the socket
- Lookup socket directly before network stack
- Read CPU id from socket and deliver accordingly

Lighter, more accurate and thus faster than Google RFS

Benchmark

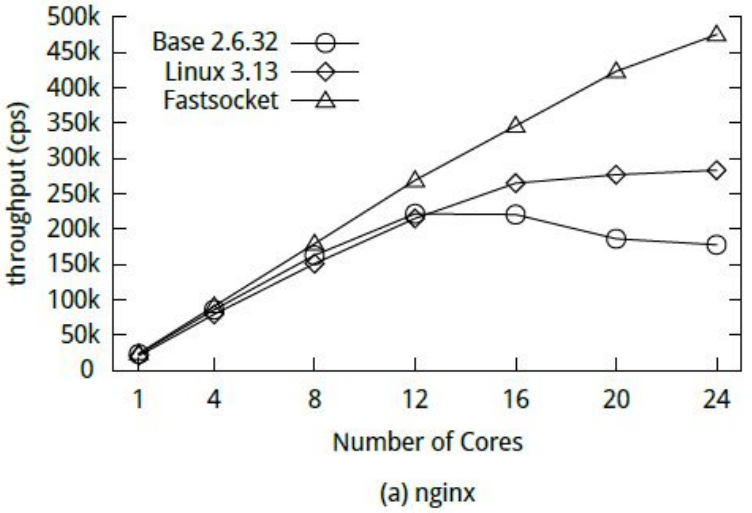
Testing Environment:

- Redis: Key-value cache and store
- CPU: Intel E5 2640 v2 (6 core) * 2
- NIC: Intel X520

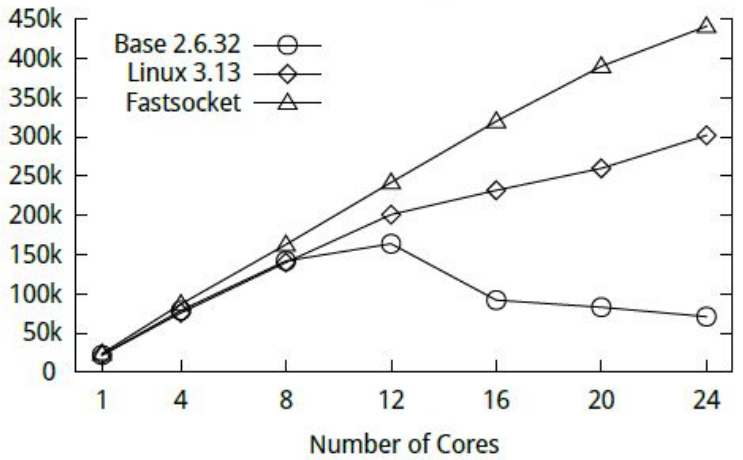
Configuration:

- Persist TCP connections
- 8 redis instances serving on difference ports
- Only 8 CPU cores are used

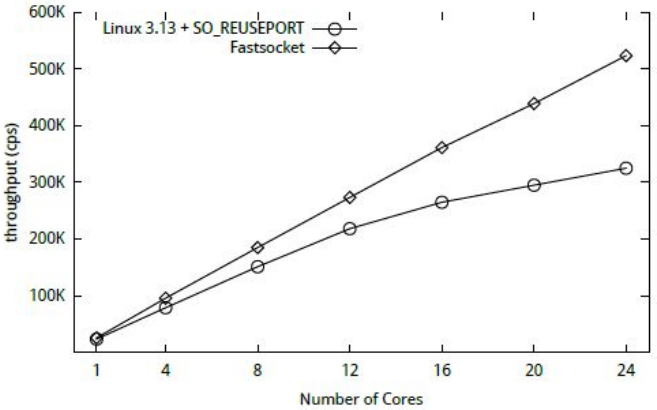
Benchmark



(a) nginx



(b) HAProxy



Redis Benchmark

Disable Flow-Director:

20% throughput increase

Enable Flow-Director:

45% throughput increase

Fastsocket

- Scalability Performance
- Single Core Performance
- Production System Feasibility
- Future Work

Compatibility & Generality

- Full BSD-Socket API compatible
- Full kernel network feature support
- Require no change of application code
- Nginx, HAProxy, Lighttpd, Redis, Memcached, etc.

Deployment

- Install RPM (Kernel, *fastsocket.ko* and *libfsocket.so*)
- Load *fastsocket.ko*
- Start application with *PRELOAD libfsocket.so*

    LD_PRELOAD=./libfsocket.so haproxy

Maintainability

- Flexible Control
  - Selectively enable Fastsocket to certain applications (nginx)
  - Compatible with regular socket (ssh)
- Quick Rollback
  - Restart the application without *libfsocket.so*
- Easy Updating
  - Most of codes are in *fastsocket.ko*
  - Updating *fastsocket.ko* and *libfsocket.so* is enough

SINA Deployment

- Adopted in HTTP load balance service (HAProxy)
- Deployed in half of the major IDCs
- Stable running for 8 months
- Fastsocket will update all HAPoxy by the end of year

Fastsocket

- Scalability Performance
- Single Core Performance
- Production System Feasibility
- Future Work

Future Work

- Make it faster
  - Improving Interrupt Mechanism
  - System-Call Batching
  - Zero Copy
  - Further Customization and Cross-Layer Optimization
- Linux Mainstream