Rx Processing in the kernel

NIC hardware -> rx_ring -> softirq
interrupt scheduled

IP firewall
IP routing

tcp_v4_recv

socket backlog

recv
recv_backlog

TCP process

kernel recv buffer

read

Application

Memory
Tx Processing in the kernel

1. Application writes to TCP process
2. TCP process calls send_msg
3. IP csum, IP route, IP filter
4. dev_xmit
5. DMA
6. NIC hardware
7. tx_ring
8. qdisc
9. completion queue
10. softirq to free
11. Memory
12. kernel
13. user
Various sources of performance overheads
MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han, Scott Marshal, Byung-Gon Chun, Sylvia Ratnasamy

OSDI’12

Content borrowed from Sangjin’s OSDI talk
Two Types of Network Workloads

• **Bulk Transfer**
  • Large files (HDFS)

• **Message-oriented**
  • Short connections or small messages (HTTP, RPCs, DB, key-value stores, etc)
Two Types of Network Workloads

- **Bulk Transfer**
  - Large files (HDFS)
  - A half CPU core can saturate 10Gbps link

- **Message-oriented**
  - Short connections or small messages (HTTP, RPCs, DB, key-value stores, etc)
  - CPU-intensive
BSD Socket API Performance Issues

```c
n_events = epoll_wait(...);    // wait for I/O readiness
for (...) {
...
new_fd = accept(listen_fd);   // new connection
...
bytes = recv(fd2, buf, 4096); // new data for fd2
```

- Issues with message-oriented workloads
  - System call overhead
BSD Socket API Performance Issues

```c
n_events = epoll_wait(...); // wait for I/O readiness
for (...) {
    ...
    new_fd = accept(listen_fd); // new connection
    ...
    bytes = recv(fd2, buf, 4096); // new data for fd2
}
```

- Issues with message-oriented workloads
  - System call overhead
  - Shared listening socket
BSD Socket API Performance Issues

```c
n_events = epoll_wait(...); // wait for I/O readiness
for (...) {
    ...
    new_fd = accept(listen_fd); // new connection
    ...
    bytes = recv(fd2, buf, 4096); // new data for fd2
}
```

- Issues with message-oriented workloads
  - System call overhead
  - Shared listening socket
  - File abstraction overhead
Microbenchmarks: how bad?

RPC-like test on an 8-core Linux server (with epoll)

768 Clients

1. Message size

new TCP connection

2. Connection length

request (64B)

3. Number of cores

response (64B)

10 transactions

Teardown

...
Microbenchmarks: how bad?

1. Small Messages Are Bad

- Throughput
- CPU Usage

Low throughput  Message Size (B)  High overhead
2. Short Connections Are Bad

Throughput (1M transactions/s)

Number of Transactions per Connection

19x lower
Microbenchmarks: how bad?

3. Multi-Core Will Not Help (Much)
MegaPipe Design

Focus: low-overhead and multi-core scalability.
MegaPipe: Overview

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low per-core performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poor multi-core scalability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Primitives

- **Handle**
  - Similar to file descriptor
    - But only valid within a channel
  - TCP connection, pipe, disk file, ...

- **Channel**
  - **Per-core**, bidirectional pipe between user and kernel
  - Multiplexes I/O operations of its handles
How channels help?
I. I/O Batching

- Transparent batching
  - Exploits parallelism of independent handles

![Diagram of I/O Batching]

MegaPipe User-Level Library

- Read data from handle 6
- Accept a new connection
- Read data from handle 3
- Write data to handle 5

MegaPipe Kernel Module

- New connection arrived
- Write done to handle 5
- Read done from handle 6

MegaPipe API (non-batched)

Batched system calls
How channels help?

Core 1  Core 2  Core 3

Shared accept queue

accept()

New connections
How channels help?

Core 1

Core 2

Core 3

New connections

Listening socket partitioning
2. Listening Socket Partitioning

- Per-core accept queue for each channel
  - Instead of the globally shared accept queue

```c
mp_register()
```

```
Listening socket
```

Accept queue

```
User
```

```
Kernel
```
2. Listening Socket Partitioning

- Per-core accept queue for each channel
  - Instead of the globally shared accept queue

```
mp_register()
```

```
Listening socket

Accept queue  Accept queue  Accept queue

User

Kernel
```
2. Listening Socket Partitioning

- Per-core accept queue for each channel
  - Instead of the globally shared accept queue

```plaintext
Listening socket

mp_accept()

Accept queue

New connections

User

Kernel
```
How channels help?

Core 1

Core 2

Core 3

VFS
How channels help?

Core 1
Core 2
Core 3

VFS

Lightweight socket
3. Light-weight Sockets

- Common-case optimization for sockets
  - Sockets are ephemeral and rarely shared
    - Bypass the VFS layer
    - Convert into a regular file descriptor only when necessary
Evaluation: Microbenchmarks

- Throughput improvement with various message sizes
Evaluation: Microbenchmarks

- Multi-core scalability
  - with various connection lengths (# of transactions)
Evaluation: Macrobenchmarks

- memcached
  - In-memory key-value store
  - Limited scalability
    - Object store is shared by all cores with a global lock

- nginx
  - Web server
  - Highly scalable
    - Nothing is shared by cores, except for the listening socket
Evaluation: Macrobenchmarks

- memcached
  - In-memory key-value store
  - Limited scalability
    - Object store is shared by all cores with a global lock

- nginx
  - Web server
  - Highly scalable
    - Nothing is shared by cores, except for the listening socket
Evaluation: memcached

Throughput (1k requests/s)

Number of Requests per Connection

- MegaPipe
- Baseline

Global lock bottleneck

- 3.6x
- 3.9x
- 2.4x
- 1.3x
Evaluation: memcached

A graph shows the throughput (1k requests/s) on the y-axis and the number of requests per connection on the x-axis. The graph compares different configurations:

- MegaPipe-FL
- Baseline-FL
- MegaPipe
- Baseline

The graph indicates how each configuration performs under varying numbers of requests per connection.
Evaluation: nginx
Conclusion

- **Short connections or small messages:**
  - High CPU overhead
  - Poorly scaling with multi-core CPUs

- **MegaPipe**
  - Key abstraction: per-core channel
  - Enabling three optimization opportunities:
    - Batching, partitioning, lwsocket
  - 15+% improvement for memcached, 75% for nginx
Your Opinions

Pros:
• Light-weight socket, batching, listening socket partitioning.
• Thorough evaluation of performance bottlenecks.
• Significant performance improvement (for nginx).
Your Opinions

Cons:

• Lack of backwards-compatibility.
• How much effort is required to port an application to use MegaPipe?
• Batching may impact latency.
• What do we lose out on by using lwsockets?
• Does not support (dynamic) load-balancing for partitioned sockets.
• Scaling beyond 8 cores?
• Kernel modifications may be difficult.
• Why not use MPI or RDMA?
Your Opinions

Ideas:
- Secure accept queue sharing with access control
- Is MegaPipe useful beyond network I/O?
- Beyond Linux?
- Load balancing for socket partitioning.
- Lower syscall cost.
- Combining RouteBricks with MegaPipe.
- What hardware optimizations can be applied?
- Network IO interface that is both high performance and POSIX-compliant.
- Automate application modifications to use MegaPipe.
Discuss!

What other sources of performance overhead remain?