MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han
in collaboration with

Scott Marshall  Byung-Gon Chun  Sylvia Ratnasamy

University of California, Berkeley  Yahoo! Research
MegaPipe is a new network programming API for message-oriented workloads to avoid the performance issues of BSD Socket API.
Two Types of Network Workloads

1. Bulk-transfer workload
   - One way, large data transfer
     - Ex: video streaming, HDFS
   - Very cheap
     - A half CPU core is enough to saturate a 10G link
Two Types of Network Workloads

1. Bulk-transfer workload
   - One way, large data transfer
     - Ex: video streaming, HDFS
   - Very cheap
     - A half CPU core is enough to saturate a 10G link

2. Message-oriented workload
   - Short connections or small messages
     - Ex: HTTP, RPC, DB, key-value stores, ...
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
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
Microbenchmark: How Bad?

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

768 Clients

new TCP connection

request (64B)

response (64B)

Teardown

10 transactions

1. Message size

2. Connection length

3. Number of cores
1. Small Messages Are Bad

- Throughput
- CPU Usage

Throughput (Gbps) vs. Message Size (B)

Low throughput: 64, 128, 256
High overhead: 2K, 4K, 8K, 16K

OSDI 2012
2. Short Connections Are Bad

Throughput (1M transactions/s)

Number of Transactions per Connection

19x lower
3. Multi-Core Will Not Help (Much)

**Graph:**
- **Y-axis:** Throughput (1M transactions/s)
- **X-axis:** # of CPU Cores
- **Legend:**
  - Ideal scaling
  - Actual scaling

**Table:**

<table>
<thead>
<tr>
<th># of CPU Cores</th>
<th>Throughput (1M transactions/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>0.3</td>
</tr>
<tr>
<td>3</td>
<td>0.6</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
</tr>
<tr>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>6</td>
<td>1.5</td>
</tr>
<tr>
<td>7</td>
<td>1.8</td>
</tr>
<tr>
<td>8</td>
<td>2.1</td>
</tr>
</tbody>
</table>
MEGAPIPE DESIGN
Design Goals

- Concurrency as a first-class citizen
- Unified interface for various I/O types
  - Network connections, disk files, pipes, signals, etc.
- Low overhead & multi-core scalability
  - Main focus of this presentation
Overview

Problem  Cause  Solution

Low per-core performance  System call overhead  System call batching

Poor multi-core scalability  Shared listening socket  Listening socket partitioning

VFS overhead  Lightweight socket
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
Sketch: How Channels Help (1/3)

User

Handles →

Channel →

Kernel

I/O Batching
Sketch: How Channels Help (2/3)

Core 1
Core 2
Core 3

Shared accept queue

accept()

New connections
Sketch: How Channels Help (2/3)

Core 1

Core 2

Core 3

Listening socket partitioning

New connections
Sketch: How Channels Help (3/3)
Sketch: How Channels Help (3/3)

Core 1

Core 2

Core 3

VFS

Lightweight socket
MegaPipe API Functions

- `mp_create() / mp_destroy()`
  - Create/close a channel

- `mp_register() / mp_unregister()`
  - Register a handle (regular FD or lwsocket) into a channel

- `mp_accept() / mp_read() / mp_write() / ...`
  - Issue an asynchronous I/O command for a given handle

- `mp_dispatch()`
  - Dispatch an I/O completion event from a channel
Completion Notification Model

- BSD Socket API
  - Wait-and-Go
    (Readiness model)

  ```
  epoll_ctl(fd1, EPOLLIN);
  epoll_ctl(fd2, EPOLLIN);
  epoll_wait(...);
  ...
  ret1 = recv(fd1, ...);
  ...
  ret2 = recv(fd2, ...);
  ...
  ```

- MegaPipe
  - Go-and-Wait
    (Completion notification)

  ```
  mp_read(handle1, ...);
  mp_read(handle2, ...);
  ...
  ev = mp_dispatch(channel);
  ...
  ev = mp_dispatch(channel);
  ...
  ```

- Batching
- Easy and intuitive
- Compatible with disk files
1. I/O Batching

- Transparent batching
  - Exploits parallelism of independent handles

---

**Application**

**MegaPipe User-Level Library**

**MegaPipe Kernel Module**

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

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

**MegaPipe API (non-batched)**

**Batched system calls**
2. Listening Socket Partitioning

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

```plaintext
mp_register()
```

OSDI 2012
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

![Diagram showing the process of accepting new connections with per-core queues]

- `mp_accept()`
- New connections
  - Listening socket
  - User
  - Kernel

OSDI 2012
3. lwsocket: Lightweight Socket

- Common-case optimization for sockets
  - Sockets are ephemeral and rarely shared
    - Bypass the VFS layer
    - Convert into a regular file descriptor only when necessary
3. lwsocket: Lightweight Socket

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

File descriptor
  ➔ dup() or fork()

File instance (states)
  ➔ dentry

File descriptor
  ➔ inode

File descriptor
  ➔ TCP socket

OSDI 2012
3. lwsocket: Lightweight Socket

- Common-case optimization for sockets
  - Sockets are ephemeral and rarely shared
    - Bypass the VFS layer
    - Convert into a regular file descriptor only when necessary
3. lwsocket: Lightweight Socket

- 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
Microbenchmark 1/2

- Throughput improvement with various message sizes

![Graph showing throughput improvement with different message sizes and number of CPU cores.](image-url)
Microbenchmark 1/2

- Throughput improvement with various message sizes

![Graph showing throughput improvement with varying message sizes and number of CPU cores. The graph indicates that throughput improves with increasing message sizes, particularly noticeable for larger message sizes like 2 KiB and 4 KiB, and shows a trend where the improvement flattens or decreases with a higher number of CPU cores.](image-url)
- Throughput improvement with various message sizes
Microbenchmark 2/2

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

![Graphs showing parallel speedup vs. number of CPU cores for Baseline and MegaPipe.](image-url)
- 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
memcachted

- memaslap with 90% GET, 10% SET, 64B keys, 1KB values
memcached

- memaslap with 90% GET, 10% SET, 64B keys, 1KB values
Based on Yahoo! HTTP traces: 6.3KiB, 2.3 trans/conn on avg.
CONCLUSION
Related Work

- **Batching** [FlexSC, OSDI’10] [libflexsc, ATC’11]
  - Exception-less system call
  - MegaPipe solves the scalability issues

- **Partitioning** [Affinity-Accept, EuroSys’12]
  - Per-core accept queue
  - MegaPipe provides explicit control over partitioning

- **VFS scalability** [Mosbench, OSDI’10]
  - MegaPipe bypasses the issues rather than mitigating
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
BACKUP SLIDES
Small Messages with MegaPipe

Throughput (1M trans/s)

# of Transactions per Connection

Baseline Throughput

MegaPipe
1. Small Messages Are Bad ➔ Why?

- # of messages matters, not the volume of traffic
  - Per-message cost >>> per-byte cost
    - 1KB msg is only 2% more expensive than 64B msg
  - 10G link with 1KB messages ➔ 1M IOPS!
    - Thus 1M+ system calls

- System calls are expensive [FlexSC, 2010]
  - Mode switching between kernel and user
  - CPU cache pollution
Short Connections with MegaPipe

Throughput (Gbps) vs. Message Size (B) for Baseline CPU Usage and MegaPipe.

- Baseline CPU Usage
- MegaPipe

Message Size (B):
- 64
- 128
- 256
- 512
- 1K
- 2K
- 4K
- 8K
- 16K

CPU Usage (%)

Throughput (Gbps)

OSDI 2012
2. Short Connections Are Bad ➔ Why?

- Connection establishment is expensive
  - Three-way handshaking / four-way teardown
    - More packets
    - More system calls: accept(), epoll_ctl(), close(), ...
  - Socket is represented as a file in UNIX
    - File overhead
    - VFS overhead
Multi-Core Scalability with MegaPipe

Baseline Per-Core Efficiency  
MegaPipe

Throughput (1M trans/s) vs. # of CPU Cores

- Baseline Per-Core Efficiency
- MegaPipe

Efficiency (%) vs. # of CPU Cores

- Baseline Per-Core Efficiency
- MegaPipe
3. Multi-Core Will Not Help → Why?

- Shared queue issues [Affinity-Accept, 2012]
  - Contention on the listening socket
  - Poor connection affinity
3. Multi-Core Will Not Help → Why?

- File/VFS multi-core scalability issues

- File instance
  - dentry
  - inode
  - VFS
  - Process file descriptor table

- Application

- TCP/IP
  - TCP socket

- Shared by threads

- Globally visible in the system
Ping-Pong Server Example

```python
ch = mp_create()
handle = mp_register(ch, listen_sd, mask=my_cpu_id)
mp_accept(handle)

while True:
    ev = mp_dispatch(ch)
    conn = ev.cookie
    if ev.cmd == ACCEPT:
        mp_accept(conn.handle)
        conn = new Connection()
        conn.handle = mp_register(ch, ev.fd, cookie=conn)
        mp_read(conn.handle, conn.buf, READSIZE)
    elif ev.cmd == READ:
        mp_write(conn.handle, conn.buf, ev.size)
    elif ev.cmd == WRITE:
        mp_read(conn.handle, conn.buf, READSIZE)
    elif ev.cmd == DISCONNECT:
        mp_unregister(ch, conn.handle)
        delete conn
```
### Contribution Breakdown

<table>
<thead>
<tr>
<th></th>
<th>Number of transactions per connection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+P</td>
<td>211.6</td>
</tr>
<tr>
<td>P +B</td>
<td>18.8</td>
</tr>
<tr>
<td>PB +L</td>
<td>352.1</td>
</tr>
<tr>
<td>Total</td>
<td>582.4</td>
</tr>
</tbody>
</table>

**Table 3:** Accumulation of throughput improvement (%) over baseline, from three contributions of MegaPipe.
memcached latency

Latency (µs)

- Baseline-FL 99%
- MegaPipe-FL 99%
- Baseline-FL 50%
- MegaPipe-FL 50%

# of Concurrent Client Connections
Clean-Slate vs. Dirty-Slate

- **MegaPipe**: a clean-slate approach with new APIs
  - Quick prototyping for various optimizations
  - Performance improvement: worthwhile!

- Can we apply the same techniques back to the BSD Socket API?
  - Each technique has its own challenges
    - Embracing all could be even harder
  - **Future Work**™

OSDI 2012