www.chenshuo.com

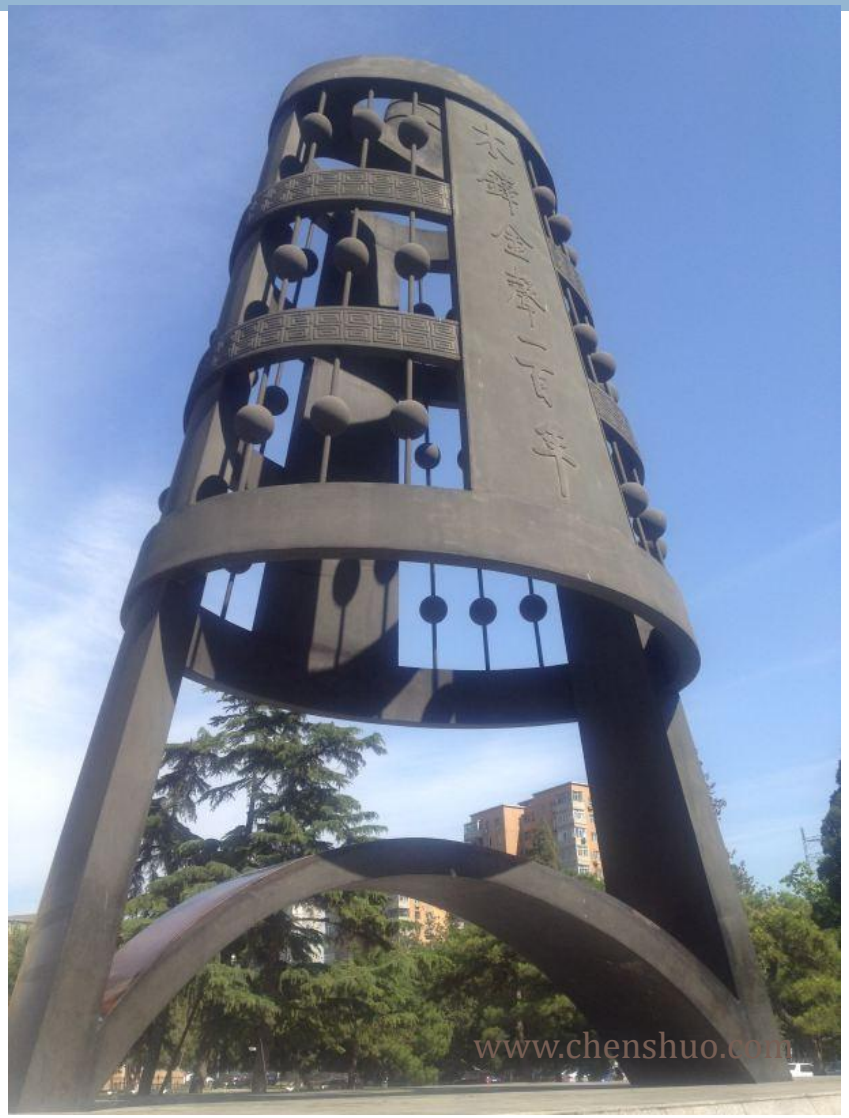
Network Programming in C++ with Muduo

2012/06

Shuo Chen

What is Muduo?

- non-blocking,
- event-driven,
- multi-core ready,
- modern (NYF's*)
- C++ network library
- for Linux
- Buzz words!!!
- BSD License of course



* Not your father's

Learn network programming in an afternoon? Let's build greeting server/client


```
import socket, time

serversocket = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM)
# set SO_REUSEADDR
serversocket.bind(('', 8888))
serversocket.listen(5)

while True:
    (clientsocket, address) = serversocket.accept()
    name = clientsocket.recv(4096)
    datetime = time.asctime()
    clientsocket.send('Hello ' + name)
    clientsocket.send('My time is ' + datetime + '\n')
    clientsocket.close()
```

```
import socket, os

sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM)
sock.connect((host, 8888))
sock.send(os.getlogin() + '\n')
message = sock.recv(4096)
print message
sock.close()
```

~10 Sockets APIs
Simple, huh ?

Sockets API might be harder than you thought

- Run on local host

```
$ ./hello-client.py localhost
Hello schen
My time is Sun May 13 12:56:44 2012
```

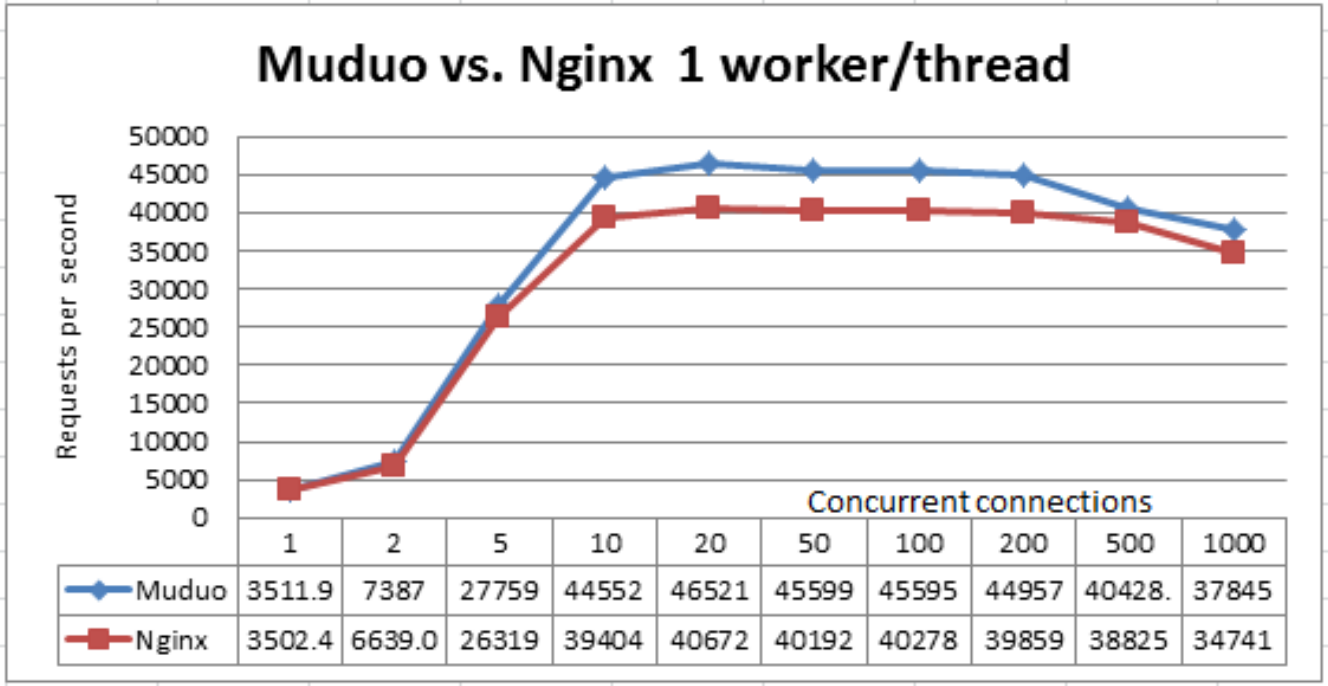
- Run on network

```
$ ./hello-client.py atom
Hello schen
```

- Incomplete response!!! Why ?
- Standard libraries (C/Java/Python) do *not* provide higher abstractions than Sockets API
  - Naive implementation is most-likely wrong
    - Sometimes hurt you after being deployed to prod env
- That's why we need good network library ☺

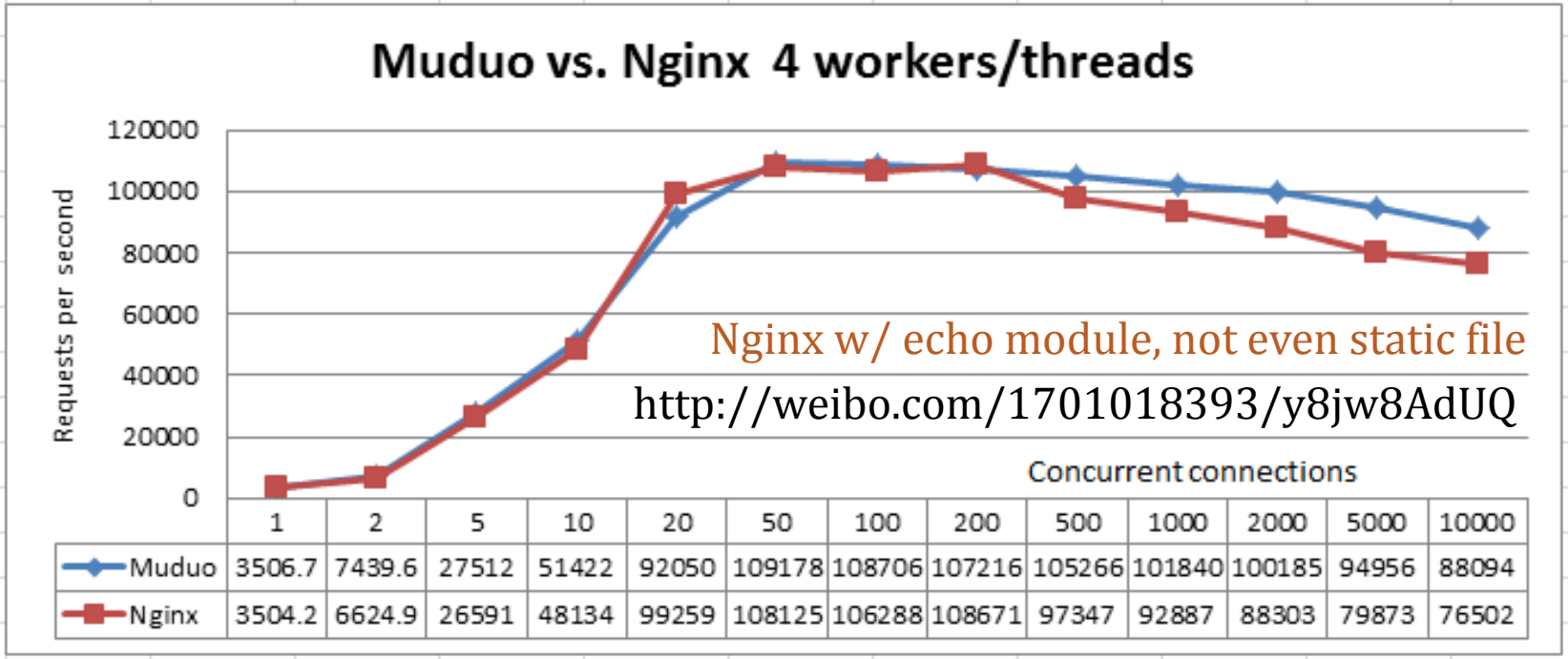
Performance goals

- High performance? Hard to define
- Satisfactory (adequate) Performance
  - Not to be a/the bottleneck of the system
- Saturate GbE bandwidth
  - Even Python can do this
- 50k concurrent connections
  - No special efforts needed on modern hardware
- *n*0k+ messages per second
  - Distribute msg to 30k clients in 0.99s (EC2 small)
    - 40k clients in 0.75s (Atom D525 1.8GHz dual core HT)



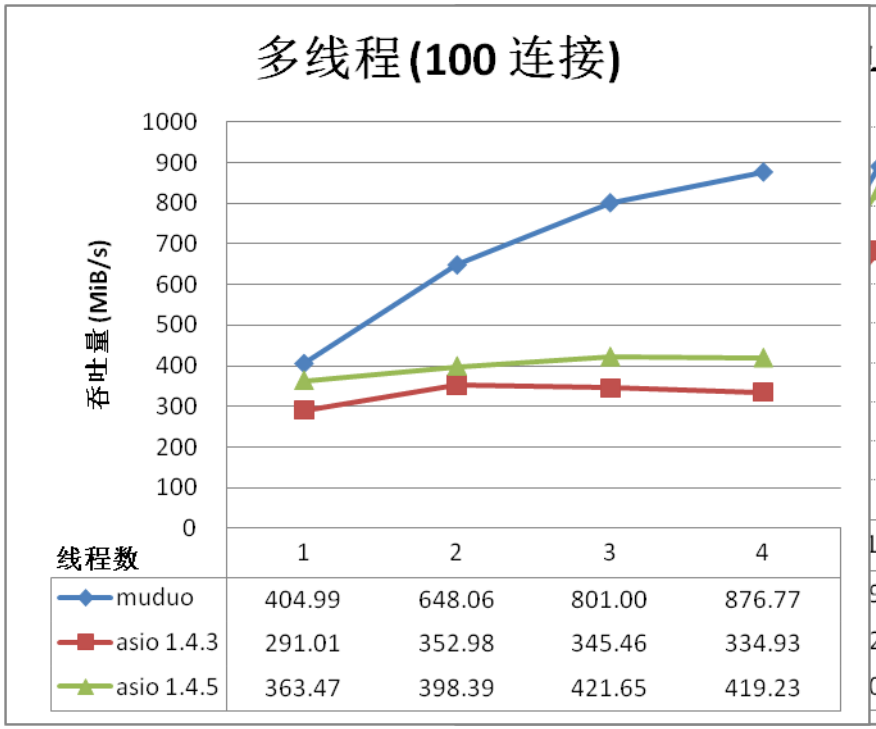
	1	2	5	10	20	50	100	200	500	1000
Muduo	3511.9	7387	27759	44552	46521	45599	45595	44957	40428.	37845
Nginx	3502.4	6639.0	26319	39404	40672	40192	40278	39859	38825	34741

Caution: Unfair Comparison

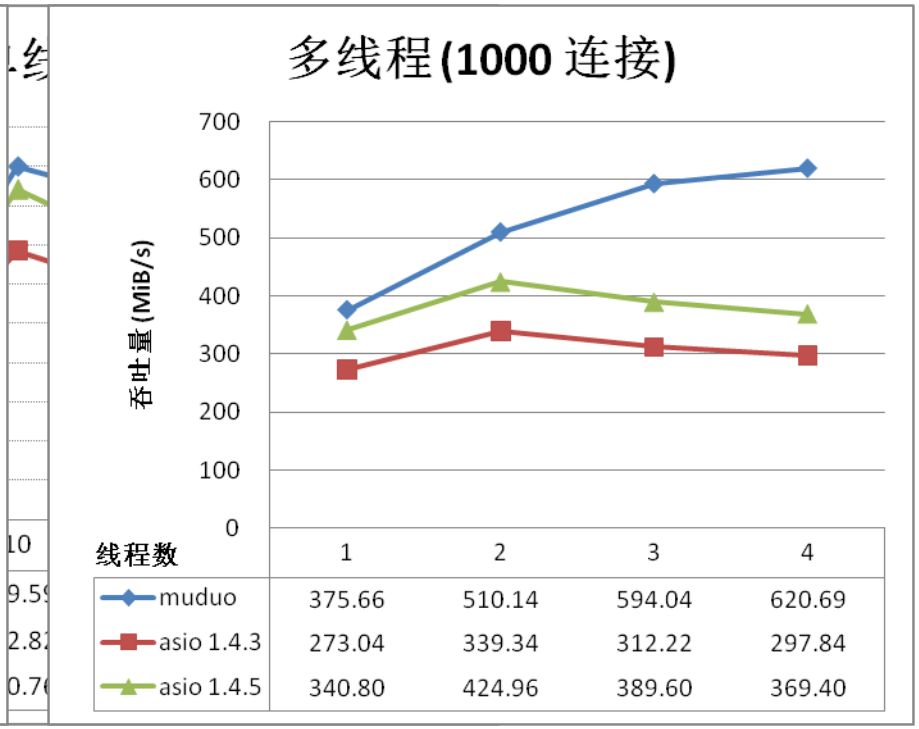


	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000
Muduo	3506.7	7439.6	27512	51422	92050	109178	108706	107216	105266	101840	100185	94956	88094
Nginx	3504.2	6624.9	26591	48134	99259	108125	106288	108671	97347	92887	88303	79873	76502

Muduo vs. Boost Asio



线程数	1	2	3	4
muduo	404.99	648.06	801.00	876.77
asio 1.4.3	291.01	352.98	345.46	334.93
asio 1.4.5	363.47	398.39	421.65	419.23

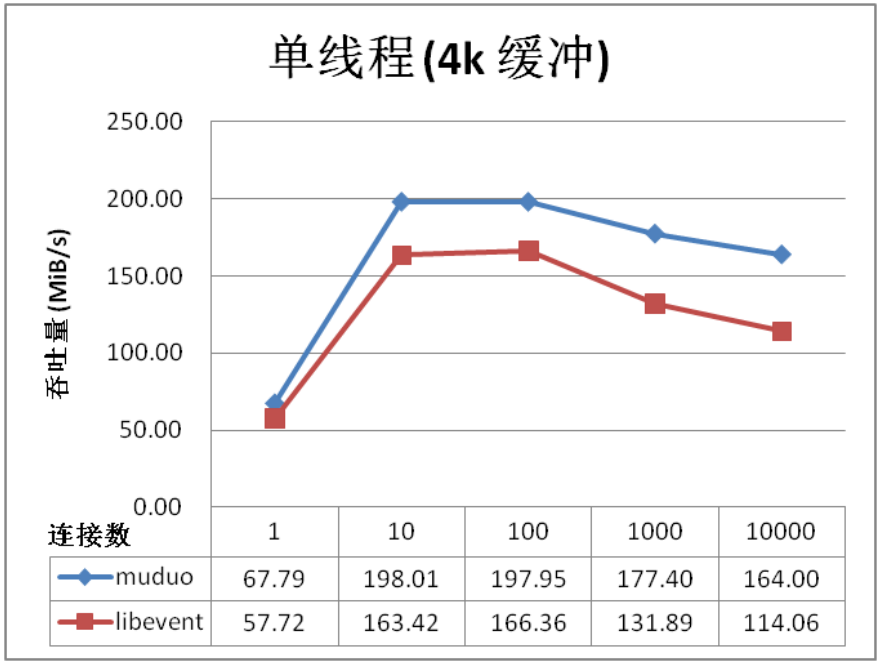


线程数	1	2	3	4
muduo	375.66	510.14	594.04	620.69
asio 1.4.3	273.04	339.34	312.22	297.84
asio 1.4.5	340.80	424.96	389.60	369.40

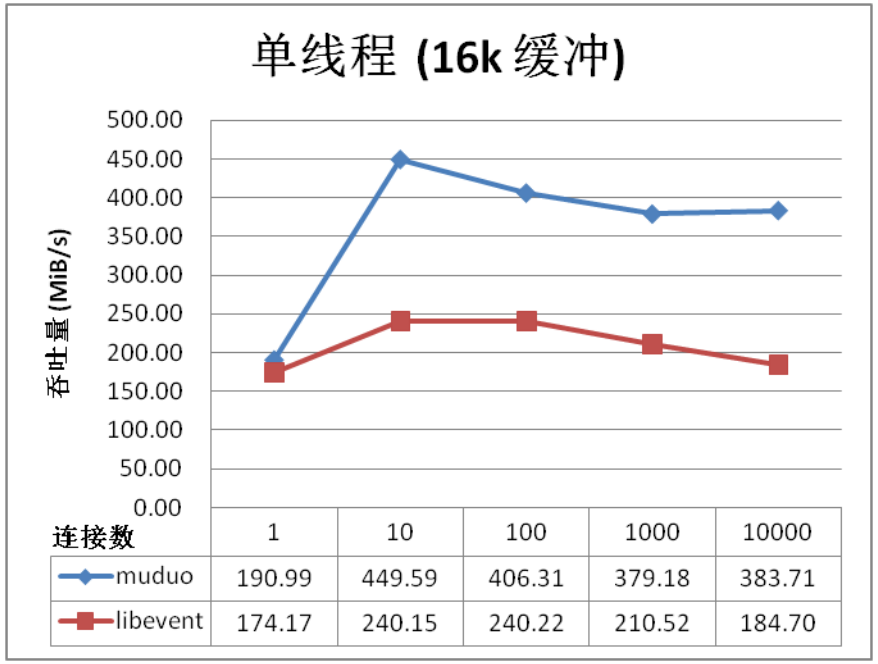
http://www.cnblogs.com/Solstice/archive/2010/09/04/muduo_vs_asio.html

Loopback device, because even Python can saturate 1GbE

Muduo vs. libevent 2.0.x



连接数	1	10	100	1000	10000
muduo	67.79	198.01	197.95	177.40	164.00
libevent	57.72	163.42	166.36	131.89	114.06

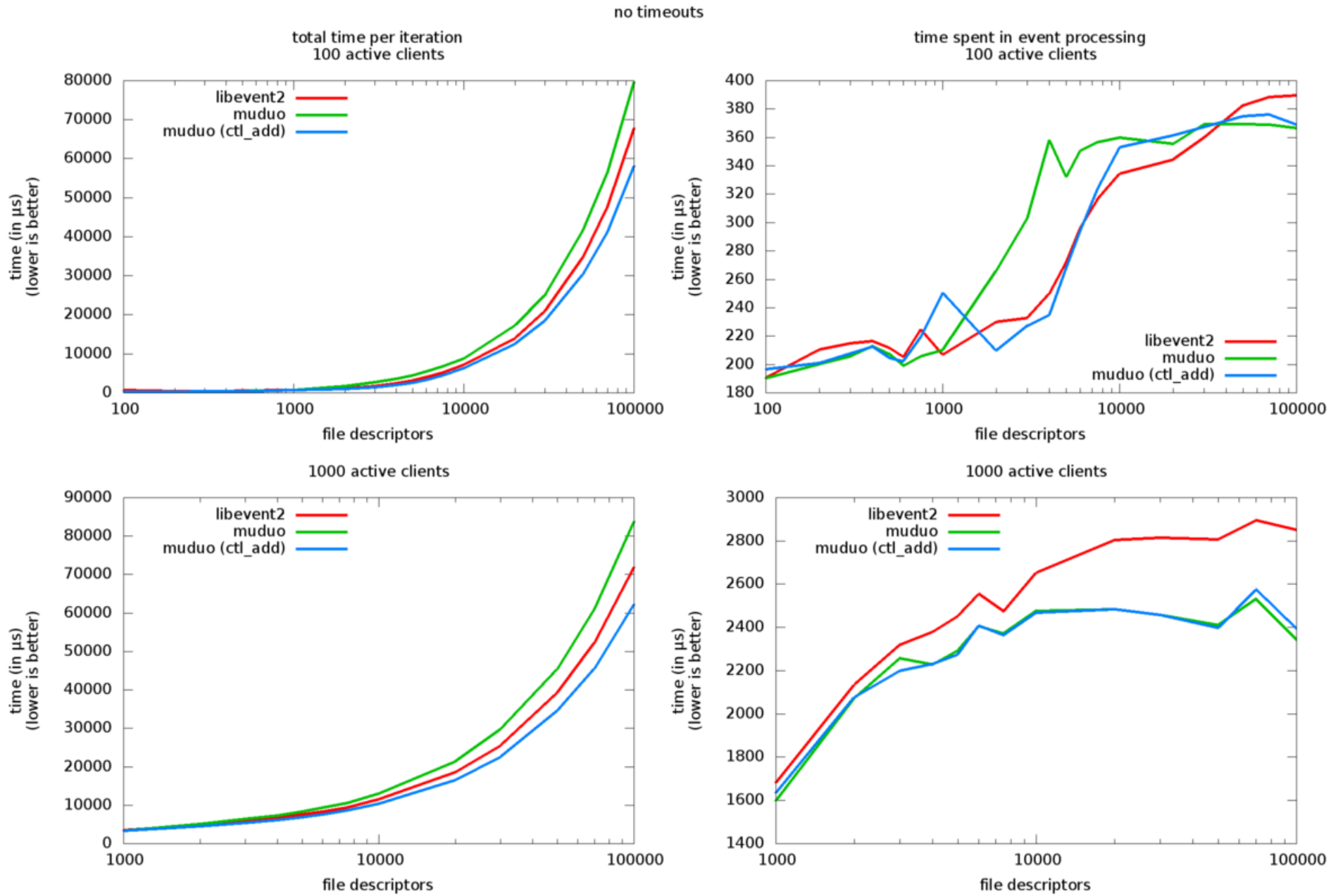


连接数	1	10	100	1000	10000
muduo	190.99	449.59	406.31	379.18	383.71
libevent	174.17	240.15	240.22	210.52	184.70

http://www.cnblogs.com/Solstice/archive/2010/09/05/muduo_vs_libevent.html

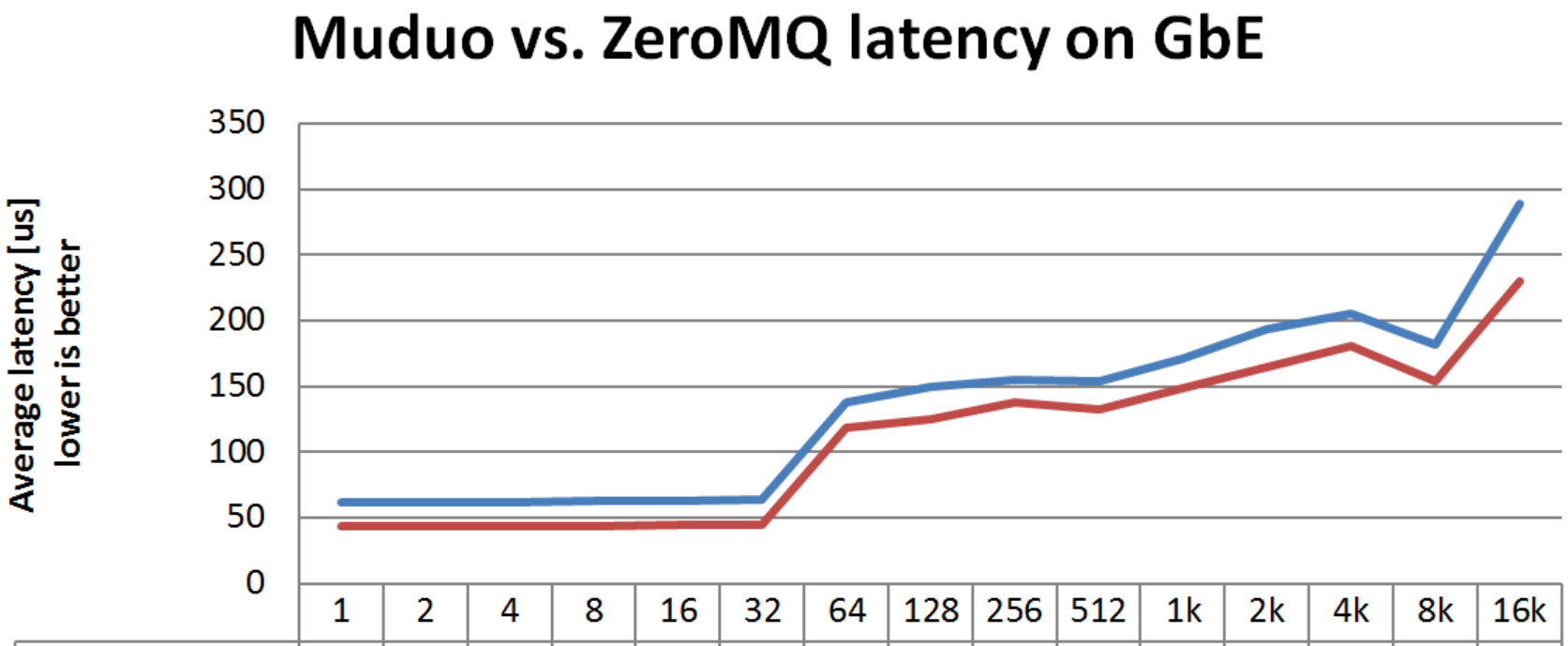
Loopback device

* Libevent 2.1.x should be better



http://www.cnblogs.com/Solstice/archive/2010/09/08/muduo_vs_libevent_bench.html

ZeroMQ local_lat, remote_lat



	1	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k
ZeroMQ 2.2	62.2	61.8	61.9	62.4	62.8	63.8	138	149	155	154	172	193	206	182	289
Muduo 0.6	43.2	43.1	43.1	43.6	44.3	44.7	118	125	138	133	149	165	181	154	230

Some performance metrics

- Use their own benchmarks
- Nginx 100k qps for in-memory reqs
- Asio higher throughput, 800MiB+/s
- Libevent ditto, same event handling speed
- pub sub deliver msg to 40k clients in 1 sec
- RPC 100k qps @ 100c,
  - 260k~515k with batching/pipelining
- At least proves “No obvious mistake made on critical path of Muduo”

Where does Muduo fit in the stack?

- General-purpose (neutral carrier) network library
  - Let you focus on business logic
  - Wraps sockets API, take care of IO complexity
  - 3.5 essential events (conn up/down, read, write complete)
- Libraries that share similar features/purposes
  - C – libevent, C++ – ACE/ASIO, Java – Netty, Mina
  - Python – twisted, Perl – POE, Ruby – EventMachine
- Not comparable to '*frameworks*'
  - ICE a RPC framework, see muduo-protorpc
  - Tomcat, Node.js built only/mainly for HTTP
  - ZeroMQ 4 messaging patterns

Two major approaches to deal with many concurrent connections

- When '*thread*' is cheap, 10k+ '*thread*'s in program
  - Create one or two *thread*s per connection, blocking IO
  - Python gevent, Go goroutine/channel, Erlang actor
- When **thread** is expensive, a handful of threads
  - Each thread serves many connections
  - Non-blocking IO with IO multiplexing (select/epoll)
    - IO multiplexing is actually thread-reusing
  - Event notification using callbacks
  - Muduo, Netty, Python twisted, Node.js, libevent, etc.
- Not all libraries can make good use of multi-cores.
  - But Muduo can ☺

Blocking IO is not always bad

- A socks proxy, TCP relay, port forwarding
  - client <-> proxy <-> server

```
def forward(source, destination):
    while True:
        data = source.recv(4096)
        if data:
            destination.sendall(data)
        else:
            destination.shutdown(socket.SHUT_WR)
            break
thread.start_new_thread(forward, (clientsocket, sock))
thread.start_new_thread(forward, (sock, clientsocket))
```

- OK to use blocking IO when interaction is simple
  - Bandwidth/throttling is done by kernel

Non-blocking IO

- Imagine writing a chat server with blocking IO
  - Message from one connection needs to be sent to many connections
  - Connections are up and down all the time
  - How to keep the integrity of a message being forwarded
  - How many threads do you need for N connections ?
- Try non-blocking IO instead
  - Essential of event-driven network programming in 30 lines of code
- Take a breath

```
# set up serversocket, socket()/bind()/listen(), as before
poll = select.poll() # epoll() should work the same
poll.register(serversocket.fileno(), select.POLLIN)
connections = {}
while True:    # The event loop
    events = poll.poll(10000)
    for fileno, event in events:
        if fileno == serversocket.fileno():
            (clientsocket, address) = serversocket.accept()
            # clientsocket.setblocking(0) ??
            poll.register(clientsocket.fileno(), select.POLLIN)
            connections[clientsocket.fileno()] = clientsocket
        elif event & select.POLLIN:
            clientsocket = connections[fileno]
            data = clientsocket.recv(4096)     # incomplete msg ?
            if data:
                for (fd, othersocket) in connections.iteritems():
                    if othersocket != clientsocket:
                        othersocket.send(data) # partial sent ??
            else:
                poll.unregister(fileno)
                clientsocket.close()
                del connections[fileno]
```

Demo only, not good quality
IO multiplexing only

Business logic

chat server

```
# set up serversocket, socket()/bind()/listen(), as before
poll = select.poll() # epoll() should work the same
poll.register(serversocket.fileno(), select.POLLIN)
connections = {}
while True:    # The event loop
    events = poll.poll(10000)
    for fileno, event in events:
        if fileno == serversocket.fileno():
            (clientsocket, address) = serversocket.accept()
            # clientsocket.setblocking(0) ??
            poll.register(clientsocket.fileno(), select.POLLIN)
            connections[clientsocket.fileno()] = clientsocket
        elif event & select.POLLIN:
            clientsocket = connections[fileno]
            data = clientsocket.recv(4096)
            if data:
                clientsocket.send(data)        # partial sent ??

            else:
                poll.unregister(fileno)
                clientsocket.close()
                del connections[fileno]
```

Business logic

Demo only, not good quality
IO multiplexing only

echo server

Most code are identical
Make them a library

Pitfalls of non-blocking IO

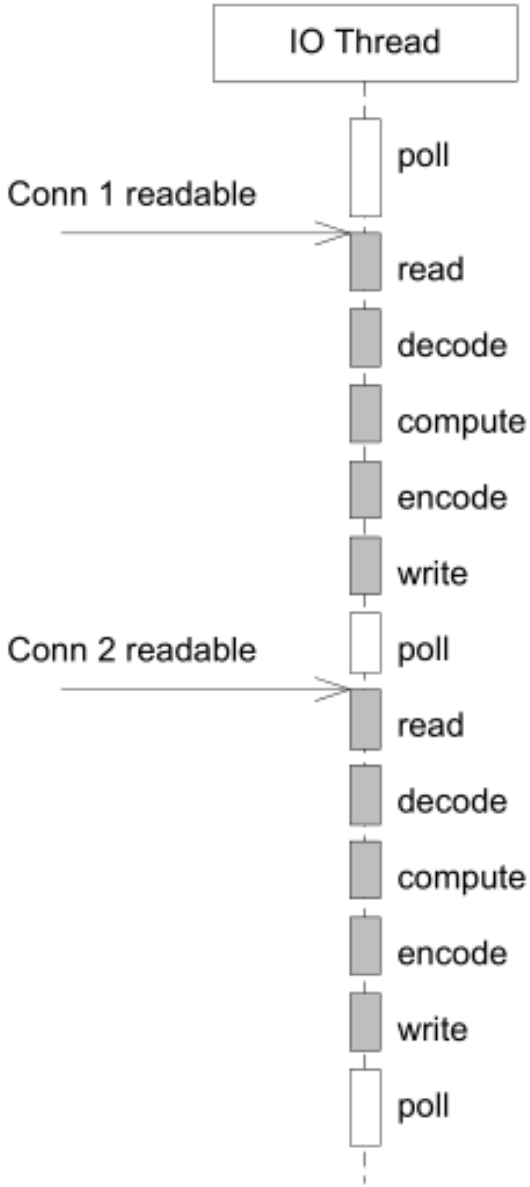
- Partial write, how to deal with remaining data?
  - You must use an output buffer per socket for next try, but when to watch POLLOUT event?
- Incomplete read, what if data arrives byte-by-byte
  - TCP is a byte stream, use an input buffer to decode
    - Alternatively, use a state machine, which is more complex
- Connection management, Socket lifetime mgmt
  - File descriptors are small integers, prone to cross talk
- Muduo is aware of and well prepared for all above!
  - Focus on your business logic and let Muduo do the rest

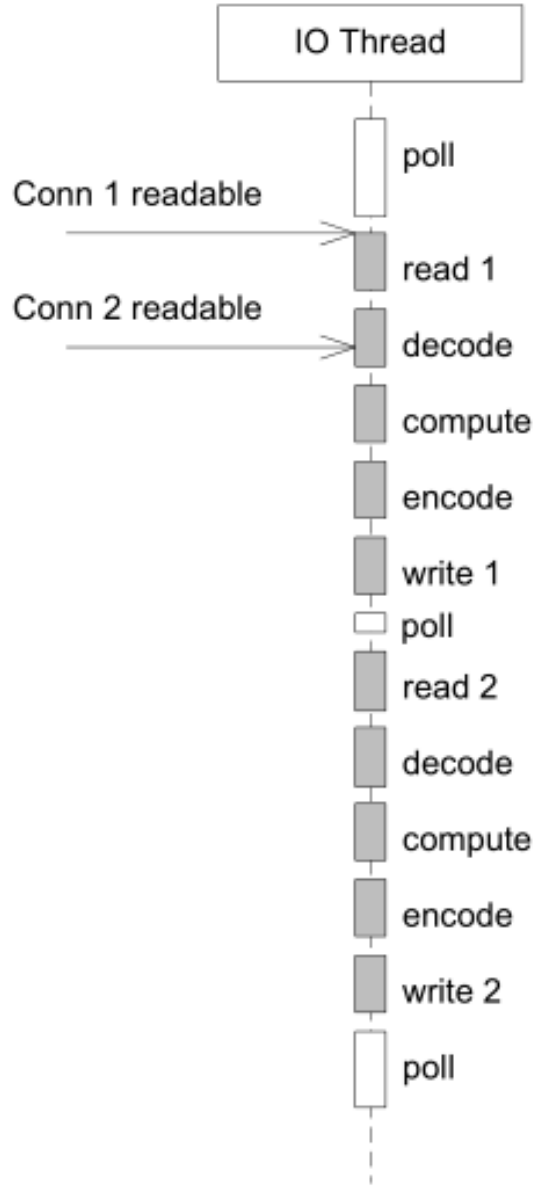
Event loop (reactor), the heart of non-blocking network programming

- Dispatches IO event to callback functions
  - Events: socket is readable, writable, error, hang up
- Message loop in Win32 programming

```
While (GetMessage(&msg, NULL, 0, 0) > 0) // epoll_wait()
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);                // here's the beef
}
```

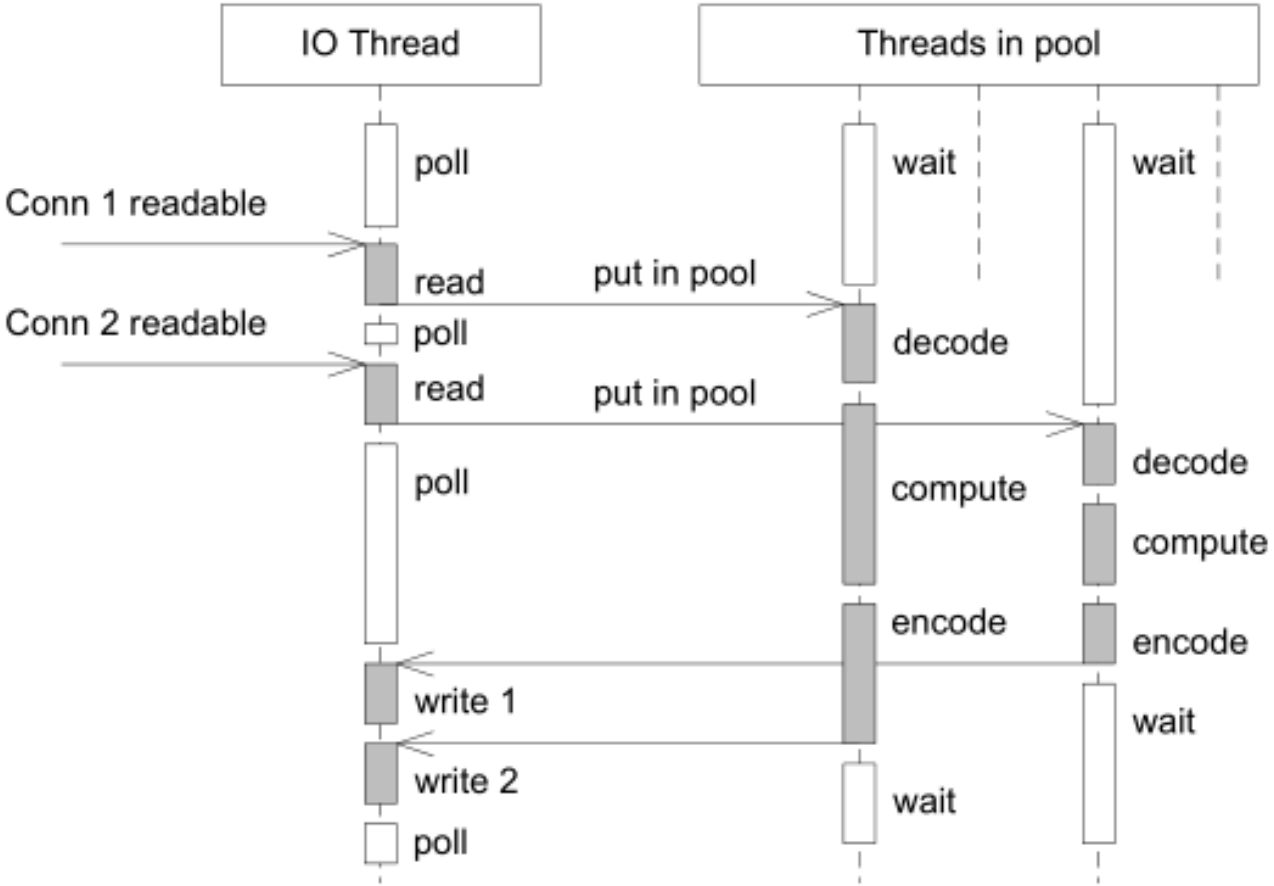
  - Cooperative multitasking, blocking is unacceptable
- Muduo unifies event loop wakeup, timer queue, signal handler all with file read/write
  - Also make it non-portable





IO responses are instant, one CPU used
Events happen in sequence

One event loop with thread pool



Computational task is heavy, IO is light

Any library function that accesses file or network can be blocking

- The whole C/Posix library is blocking/synchronous
  - Disk IO is blocking, use threads to make it cooperating
- '*harmless*' functions could block current thread
  - gethostbyname() could read /etc/hosts or query DNS
  - getpwuid() could read /etc/passwd or query NIS*
  - localtime()/ctime() could read /etc/localtime
  - Files could be on network mapped file system!
- What if this happens in a busy network IO thread?
  - Server is responseless for seconds, may cause trashing

 * /var/db/passwd.db or LDAP

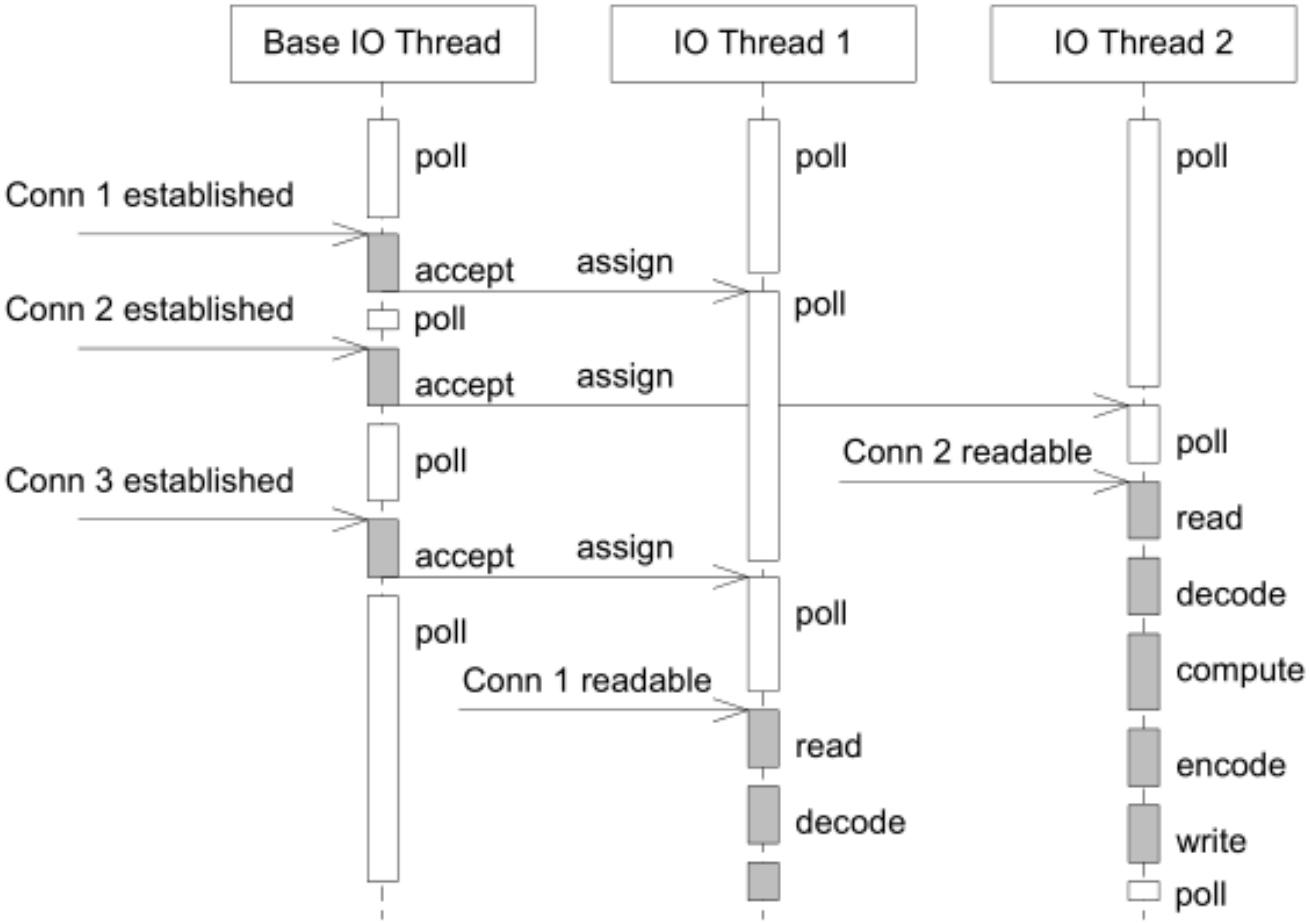
Non-blocking is a paradigm shift

- Have to pay the cost if you want to write high performance network application in traditional languages like C/C++/Java
  - It's a mature technique for nearly 20 years
- Drivers/Adaptors needed for all operations
  - Non-blocking DNS resolving, UDNS or c-ares
  - Non-blocking HTTP client/server, curl and microhttpd
    - Examples provided in muduo and muduo-udns
  - Non-blocking database query, libpq or libdrizzle
    - Need drivers to make them work in muduo
  - Non-blocking logging, in muduo 0.5.0

Event loop in multi-core era

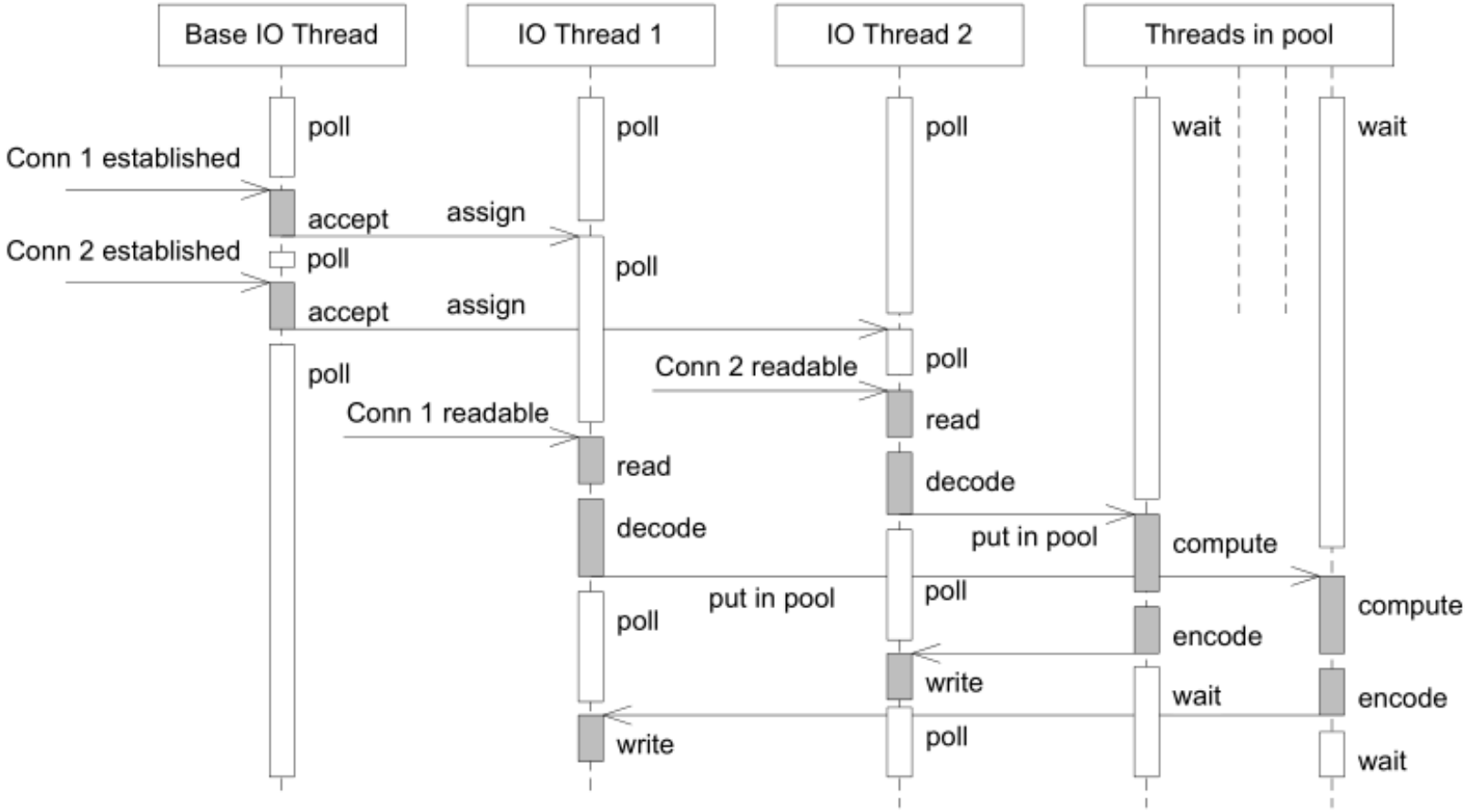
- One loop per thread is usually a good model
  - Before you try any other fancy 'pattern'
- Muduo supports both single/multi-thread usage
  - Just assign TcpConnection to any EventLoop, all IO happens in that EventLoop thread
  - The thread is predictable, EventLoop::runInLoop()
- Many other 'event-driven' libraries can't make use of multi-cores, you have to run multiple processes

One event loop per thread



Prioritize connections with threads

Hybrid solution, versatile



Decode/encode can be in IO thread

Object lifetime management

- Muduo classes are concrete & non-copyable
  - And have no base class or virtual destructor
- EventLoop, TcpServer, TcpClient are all long-live objects. Their ownership is clean, not shared.
- TcpConnection is vague
  - TcpServer may hold all alive connection objects
  - You may also hold some/all of them for sending data
  - It's the only class managed by std::shared_ptr
- No 'delete this', it's a joke
  - muduo will not pass raw pointer to/from client code

```
class EchoServer {  // non-copyable
 public:
  EchoServer(EventLoop* loop, const InetAddress& listenAddr)
    : server_(loop, listenAddr, "EchoServer") {
    server_.setConnectionCallback(
        boost::bind(&EchoServer::onConnection, this, _1));
    server_.setMessageCallback(
        boost::bind(&EchoServer::onMessage, this, _1, _2, _3));
    server_.setThreadNum(numThreads);
  }

 private:
  void onConnection(const TcpConnectionPtr& conn)   {
    // print, you may keep a copy of conn for further usage
  }

  void onMessage(const TcpConnectionPtr& conn,
                 Buffer* buf, Timestamp time) {
    string data(buf->retrieveAsString());
    conn->send(data);
  }

  TcpServer server_; // a member, not base class. More is possible
};
```

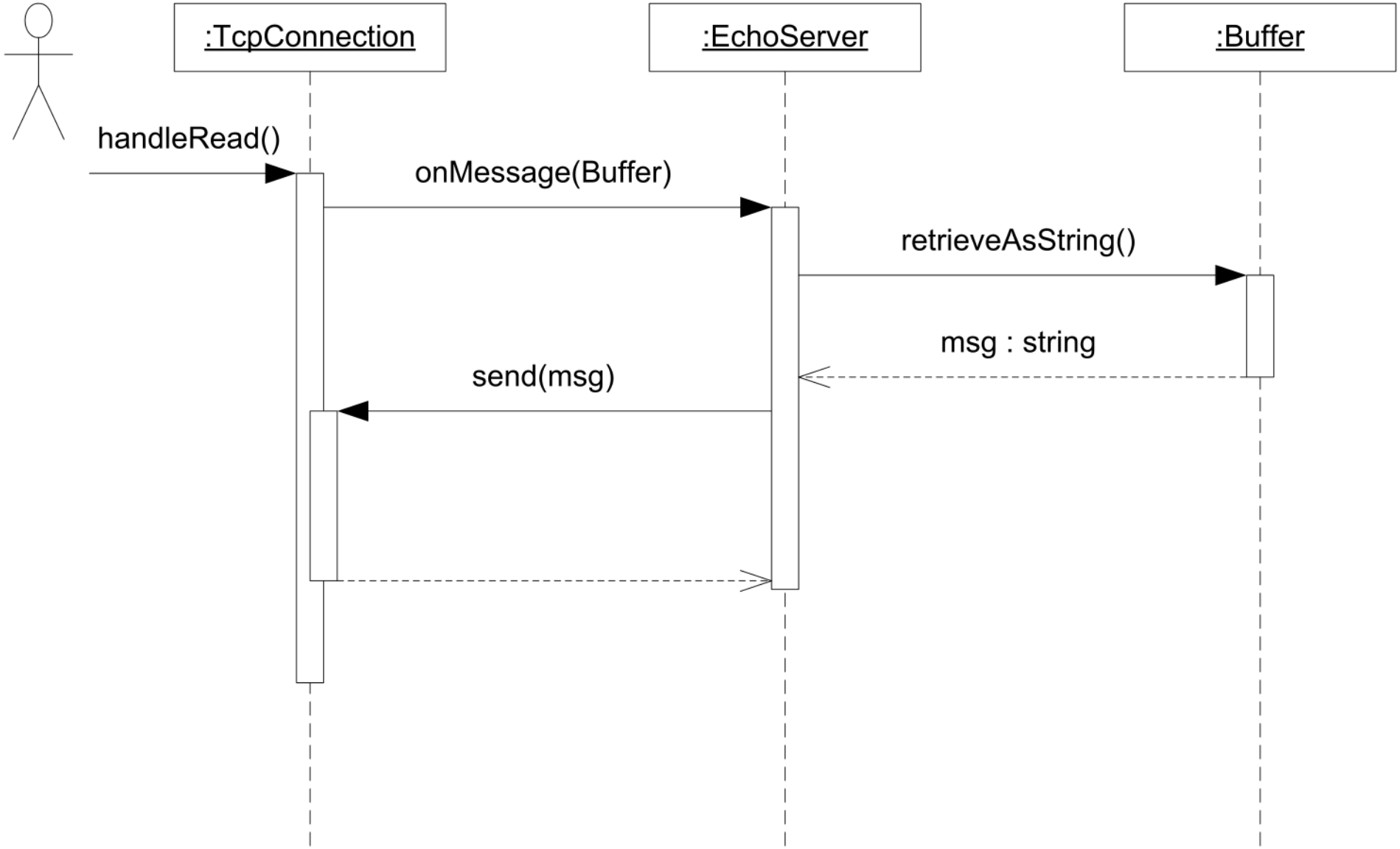
But echo is too simple to be meaningful

Muduo examples, all concurrent

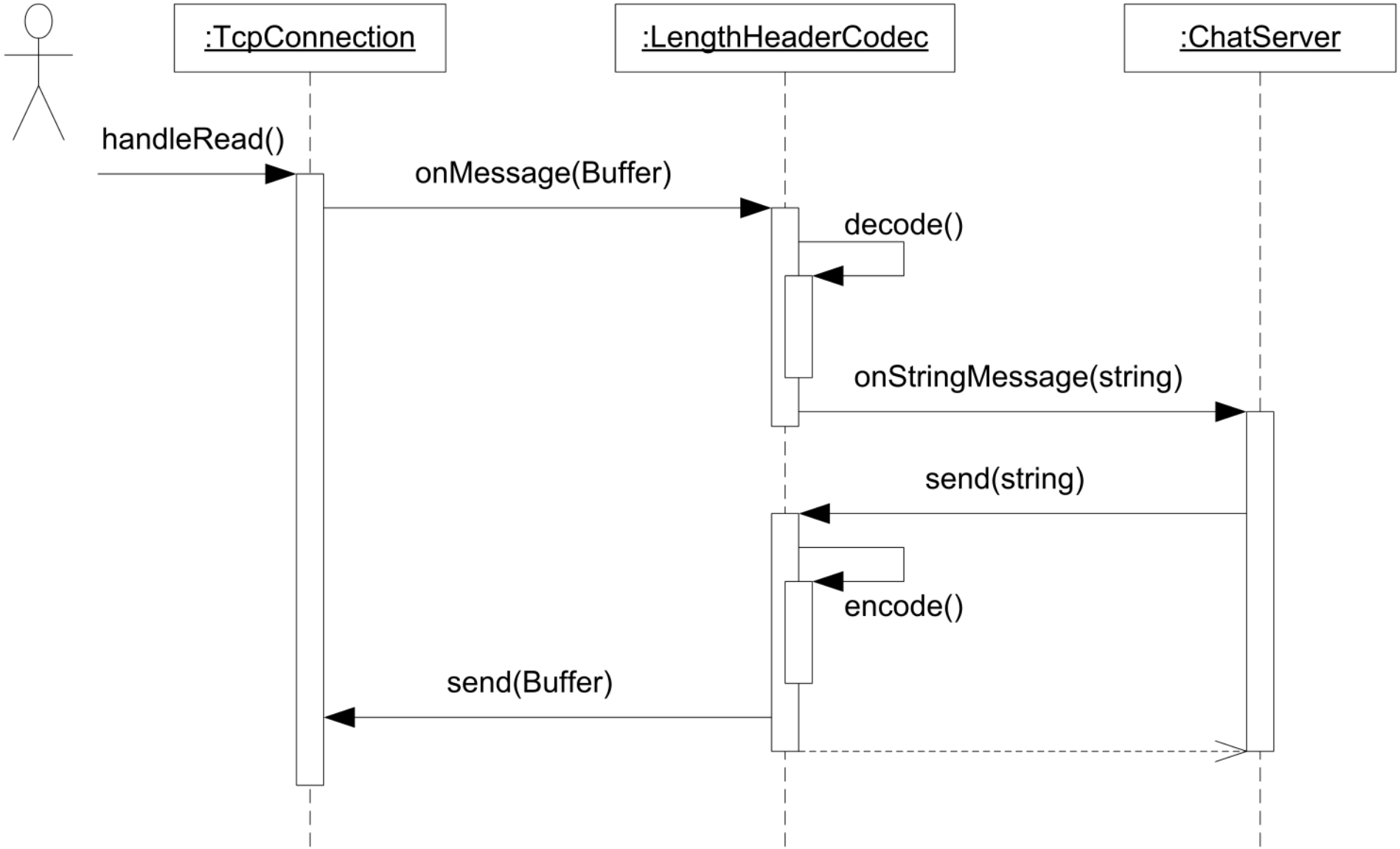
- Boost.asio chat
  - Codec , length prefix message encoder/decoder
- Google Protocol Buffers codec
- Filetransfer
- Idle connection/max connection
- Hub/Multiplexer
- Pinpong/roundtrip
- socks4a
- Many more

Business-oriented TCP network programming
Efficient multithreaded network programming

Format-less protocol, pure data



Length header fmt, ‘messages’

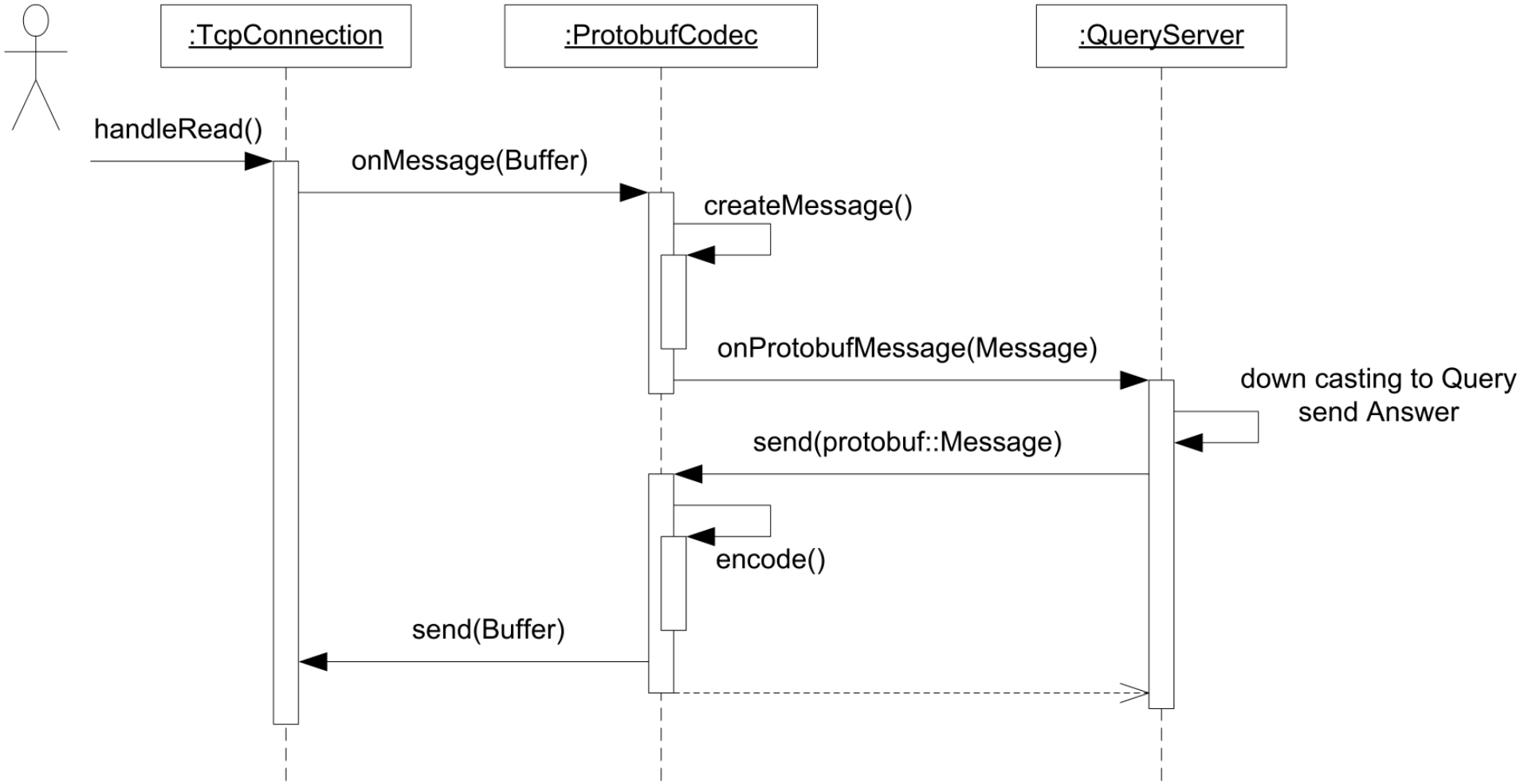


```
void onMessage(const muduo::net::TcpConnectionPtr& conn,
               muduo::net::Buffer* buf,
               muduo::Timestamp receiveTime) {
  while (buf->readableBytes() >= kHeaderLen) {  // kHeaderLen == 4
    const void* data = buf->peek();
    int32_t be32 = *static_cast<const int32_t*>(data); // FIXME
    const int32_t len = muduo::net::sockets::networkToHost32(be32);
    if (len > 65536 || len < 0) {
      LOG_ERROR << "Invalid length " << len;
      conn->shutdown();
    } else if (buf->readableBytes() >= len + kHeaderLen) {
      buf->retrieve(kHeaderLen);
      std::string message(buf->peek(), len);
      messageCallback_(conn, message, receiveTime);
      buf->retrieve(len);
    } else {
      break;
    }
  }
}
```

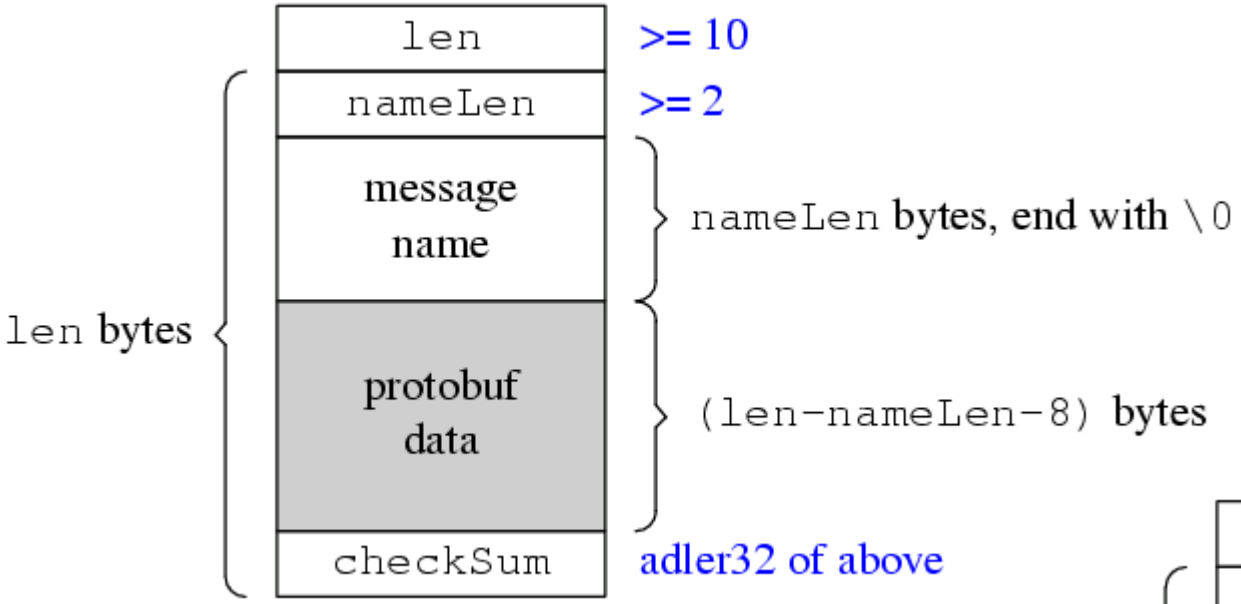
Any grouping of input data should be decoded correctly

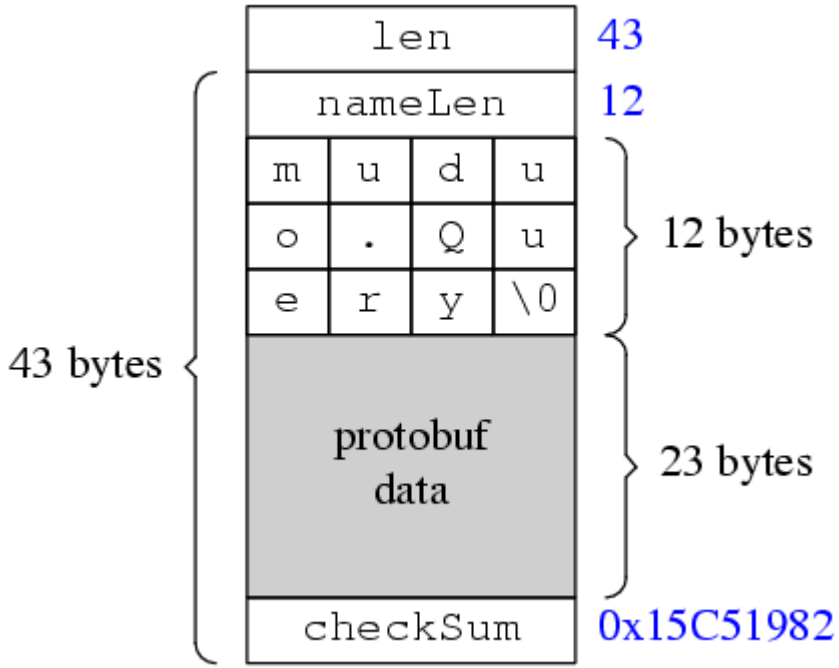
```
0x00, 0x00, 0x00, 0x05, 'h', 'e', 'l', 'l', 'o', 0x00,
0x00, 0x00, 0x08, 'c', 'h', 'e', 'n', 's', 'h', 'u', 'o'
```

Protobuf format, message objects



http://www.cnblogs.com/Solstice/archive/2011/04/13/2014362.html





http://www.cnblogs.com/Solstice/archive/2011/04/03/2004458.html

Design goals of Muduo

- Intranet, not Internet.
  - Distributed system in a global company
  - Use HTTP on internet, it's the universal protocol
- Build network application with business logic, not writing well-known network server
  - Not for building high-performance httpd, ntpd, ftpd, webproxy, bind
  - Components in distributed system
    - master/chunk-server in GFS
- TCP long connections
  - Muduo thread model is not optimized for short TCP connections, as accept(2) and IO in two loops

Muduo is NOT

Muduo doesn't

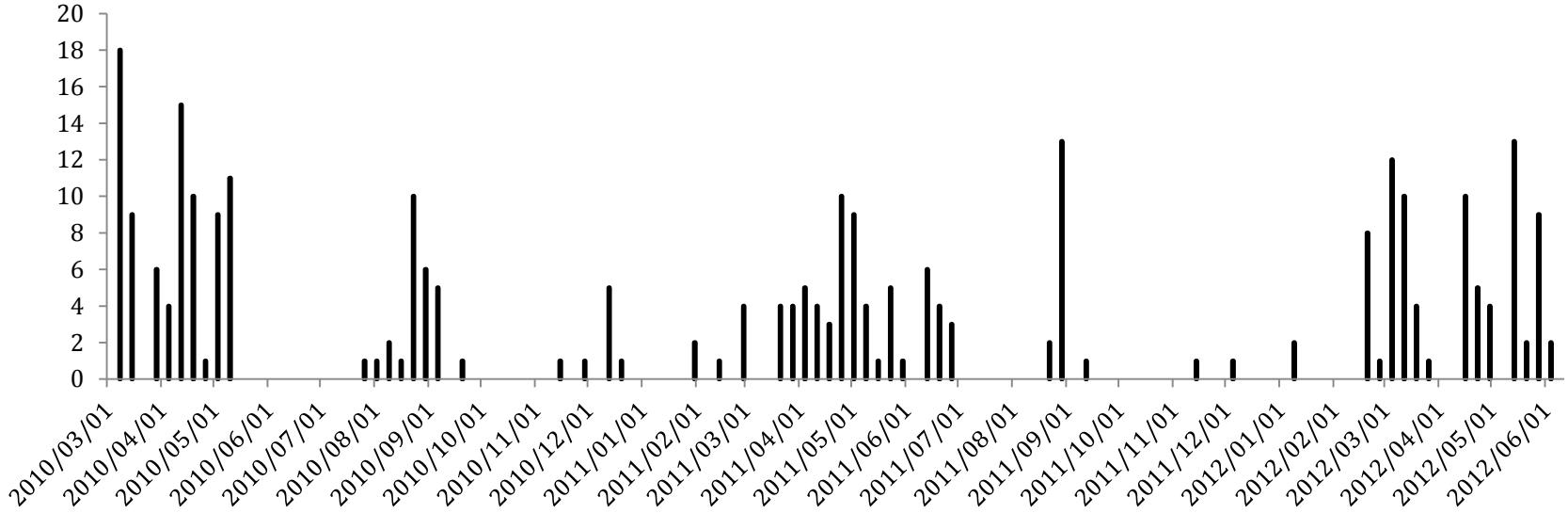
- Support transport protocols other than TCPv4
  - ~~IPv6~~, ~~UDP~~, ~~Serial port~~, ~~SNMP~~, ~~ARP~~, ~~RARP~~
  - Build your own with muduo::Channel class
    - Any thing that is 'selectable' can integrated into Muduo
  - May support SSL in future, but with low priority
    - Use https for internet service, use VPN for info security
- Support platforms other than Linux 2.6/3.x
  - Never port to Windows
  - Unlikely port to FreeBSD, Solaris
  - However, it runs on ARM9 boards, with Linux 2.6.32

List of muduo libraries

- Muduo The core library
  - base library (threading, logging, datetime)
  - network library
  - Many examples
- Muduo-udns Non-blocking DNS resolving
- Muduo-protorpc
  - Asynchronous bidirectional RPC based on Muduo
    - Also has Java bindings with Netty
  - Examples: zurg – a master/slaves service mgmt sys
  - Paxos – a consensus algorithm* (to be written)

Check-ins per week

- From 2010-03 to 2012-06



Q&A

- Thank you!

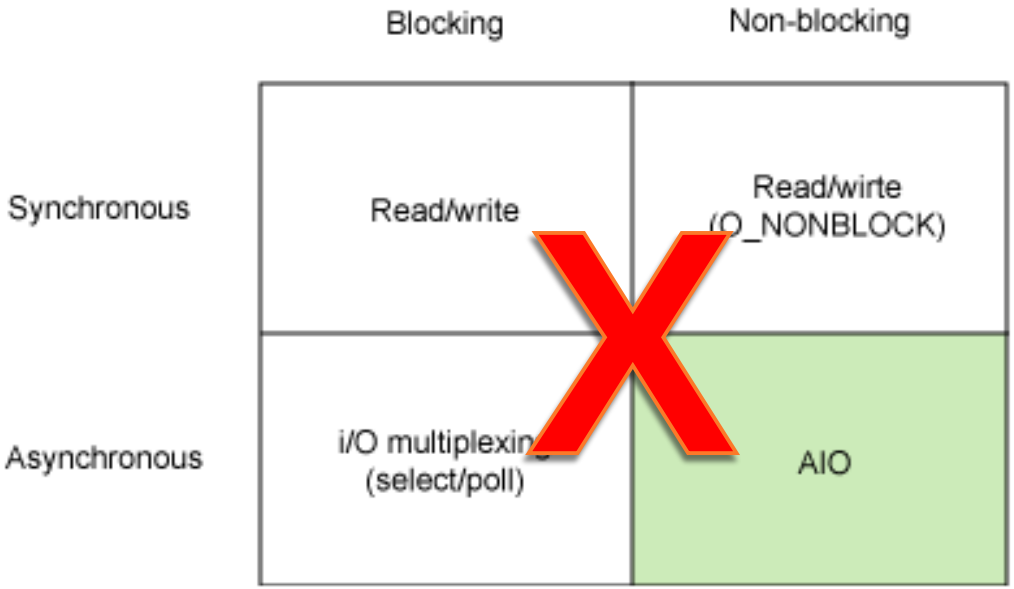
- www.chenshuo.com
- github.com/chenshuo
- weibo.com/giantchen

- github.com/downloads/chenshuo/documents/MuduoManual.pdf

Bonus Slides

- Synchronous vs. asynchronous
- Basic network performance metrics

Figure 1. Simplified matrix of basic Linux I/O models



Simply wrong and misleading

Synchronous vs. asynchronous IO

- Epoll is synchronous
  - Select/poll/epoll are O(N), but N stands differently
- Anything but aio_* are synchronous
  - Non-blocking IO is synchronous
    - you call it, it returns. It never breaks/interrupt code flow
  - The only thing that can be blocking in event-driven program are epoll_wait and pthread_cond_wait
    - pthread_mutex_lock should almost not real block anything
- Asynchronous IO is not practical in Linux
  - Either simulated with threads,
  - Or notify with signal, not good for multithreaded app

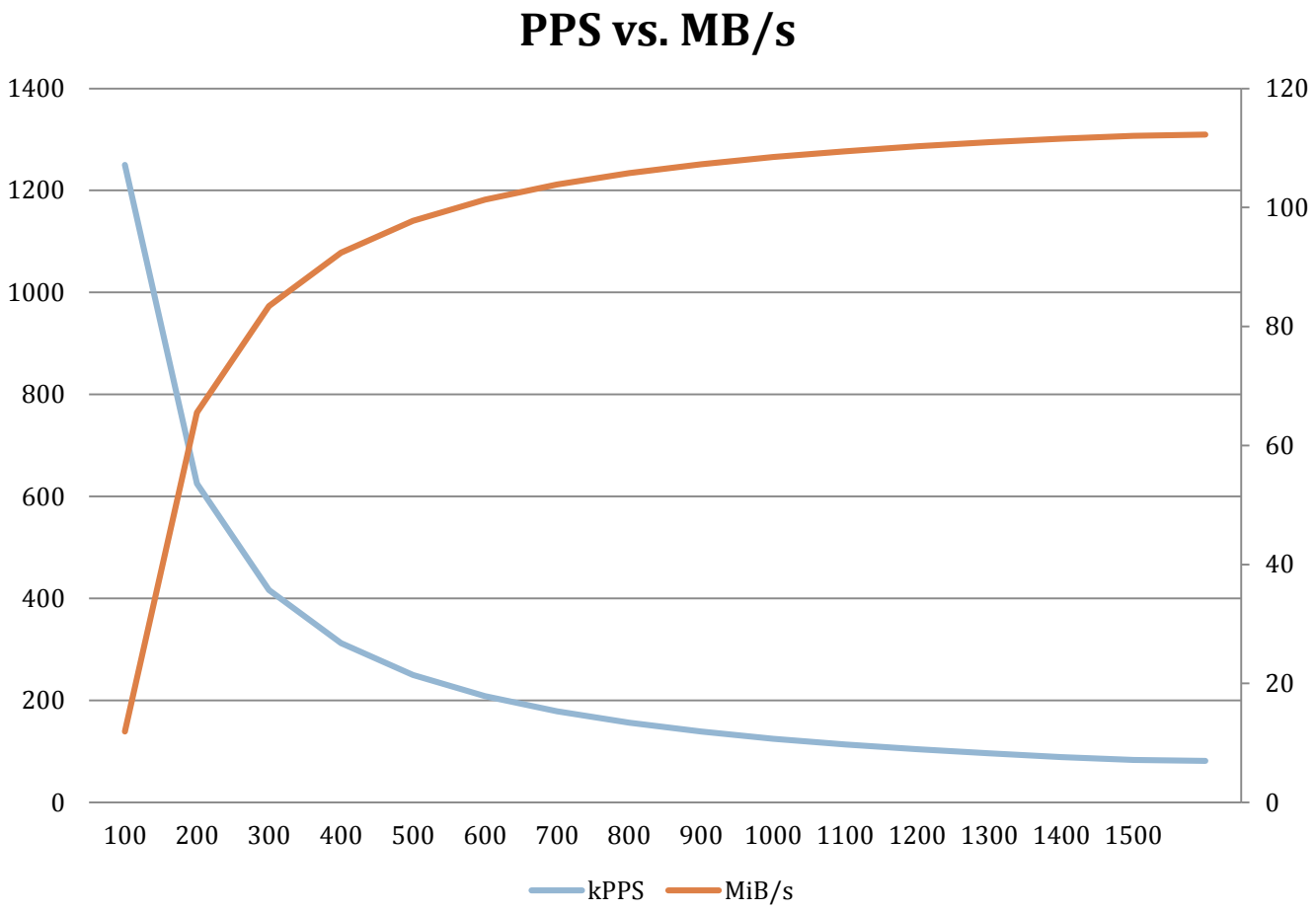
TCP/IP over 1Gb Ethernet

- Ethernet frame
- Preamble 8B
- MAC 12B
- Type 2B
- Payload 46~1500B
- CRC 4B
- Gap 12B
- Total 84~1538B

112MB/s

- Raw b/w 125MB/s
- Packet per second
  - Max 1,488,000
  - Min 81,274 (no jumbo)
- TCP/IP overhead
  - IP header 20B
  - TCP header 20B
  - TCP option 12B (TSopt)
- Max TCP throughput
  - 81274*(1500-52)

PPS vs. throughput



Back-of-the-envelope calculation

- Read 1MB from net, ~10ms
- Copy 1MB in memory, ~0.2ms on old E5150
- Copying is not a sin, CPU and memory are so fast
- Decode byte string to Message objects
  - 500MB/s decoding in IO thread, pass ptr to calc thr
  - 50MB/s copy data to calc threads, decode there
- Compress or not ? 200MB/s 2x ratio 10MB
  - 10Mb ADSL 8s vs. 4.05s
  - 1000Mb LAN 0.08s vs. 0.09s

High Performance ???

- Network application in user land
- Network service in kernel
- TCP/IP stack or network adaptor driver in kernel
- Network device (switch/router)
  - Special purpose OS for network device (firmware)
  - Special purpose chips for network device (NP)
- Control network adaptor with FPGAs
  - Coding in Verilog, hardwire logic