Instant Distribution of Updates to Hundreds of Millions of Users

Zbyněk Šlajchrt, slajchrt@avast.com
Lukáš Karas, karas@avast.com

Agenda

- Introduction
- Architecture & Design
- Optimizations
- Conclusion
- Q & A

User Base & Requirements

- ~200M installed clients
- ~35M online
- Update delivery time ASAP (< 5min)
- Update packet size 1-10kB
- Update frequency ~200 per day

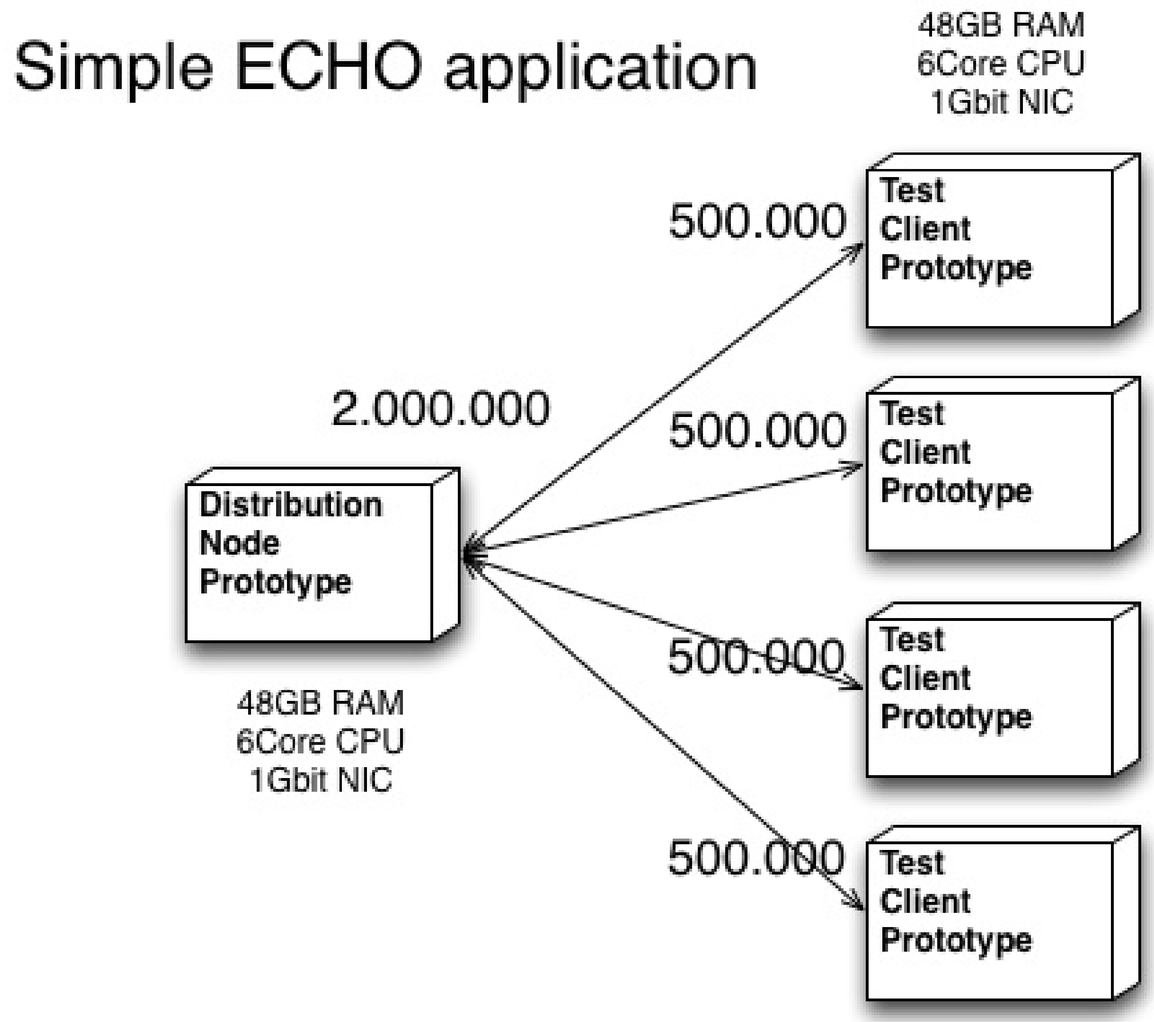
Pull vs. Push

- Pull
  - PROS: Existing distribution, simple
  - CONS: Danger of SYN flood,
    minimal control over connected clients
- Push
  - PROS: Better control over clients,
    kept-alive connections => less SYNs
  - CONS: Unproven technology,
    lack of experience, complex

Decision Criteria

- How many connections can we keep on one server?
  - Less than 1M => Pull
  - Otherwise Push

Prototype



-Xmx20g -XX:MaxDirectMemorySize=20g

System Configuration

- CentOS 6
- Max open file descriptors (startApp.sh)

```
ulimit -n 3000000
```

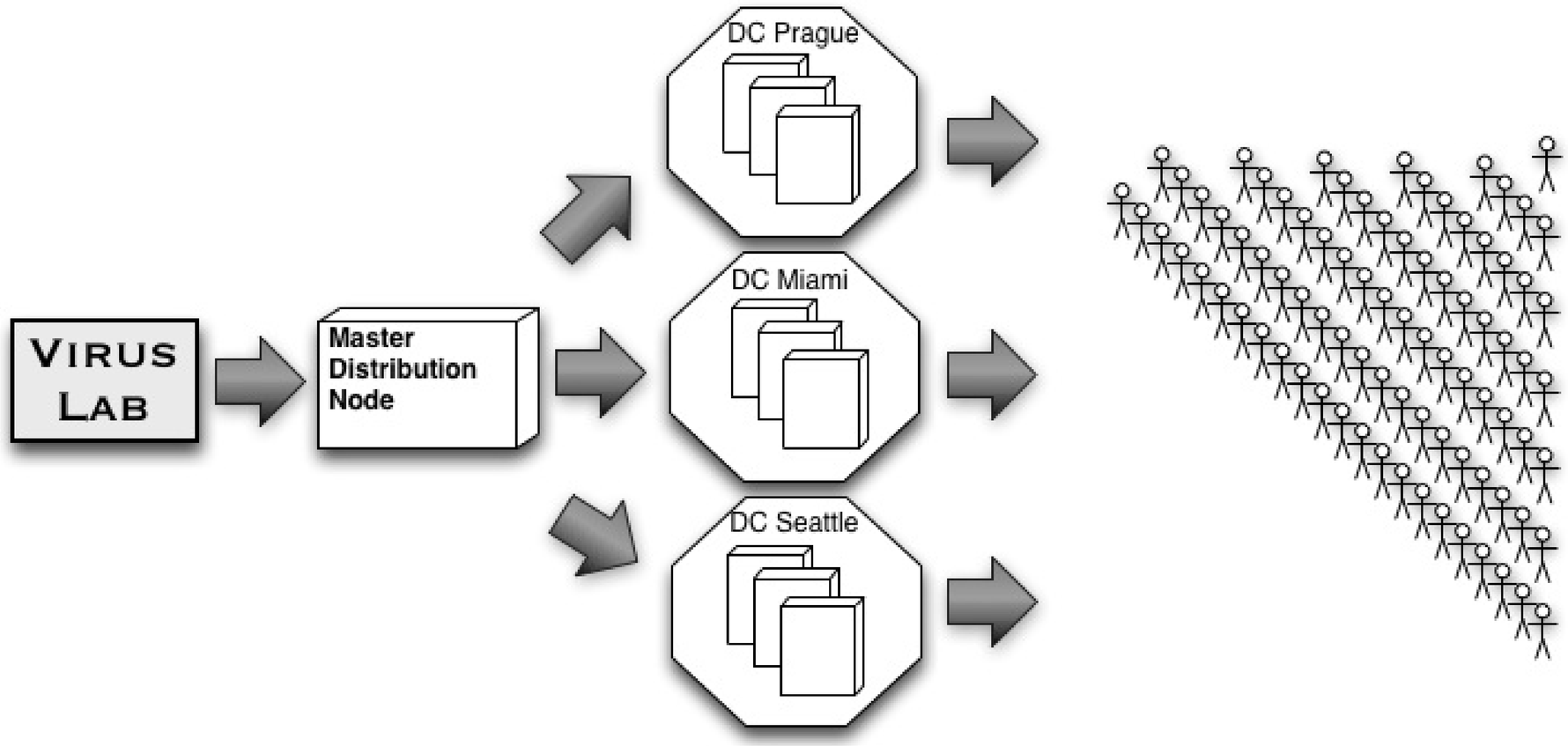
- System wide settings for file descriptors (/etc/sysctl.conf)

```
fs.nr_open = 6291456
fs.file-max = 6291456
```

- System wide limit for open files per user (/etc/security/limits.conf)

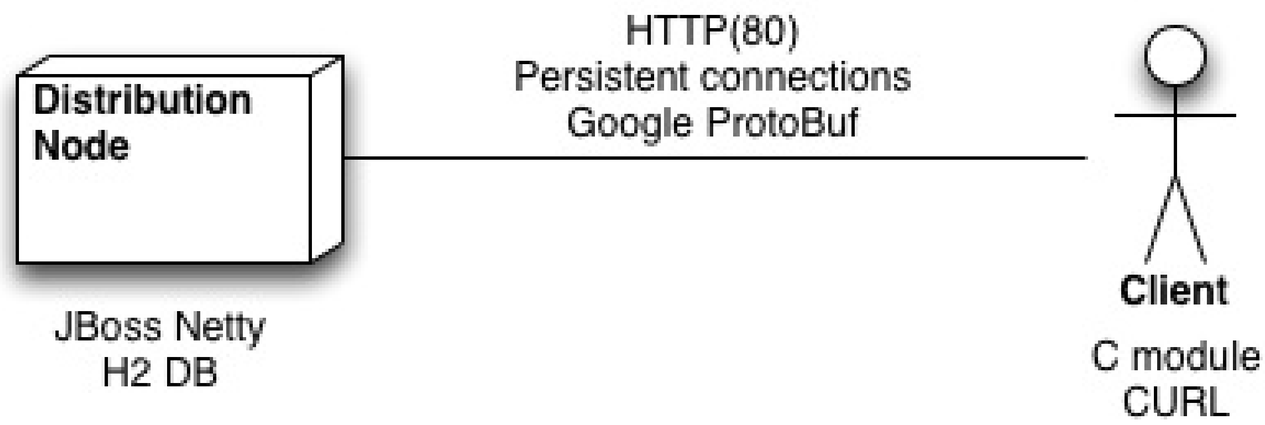
```
hard nofile  6291456
```

Distribution model



De-facto a client of the master

Client/Server Communication



3 DataCenters: Prague, Miami, Seattle
Starting with 3 servers @ DC
serving 15-20M online clients

Commands

UPDATE – to carry the update packet to clients

NOP – Heartbeat command

REJECT – to balance the load

SHUTUP – to silence “misbehaving” or obsolete clients

ECHO – to diagnose the connections to clients

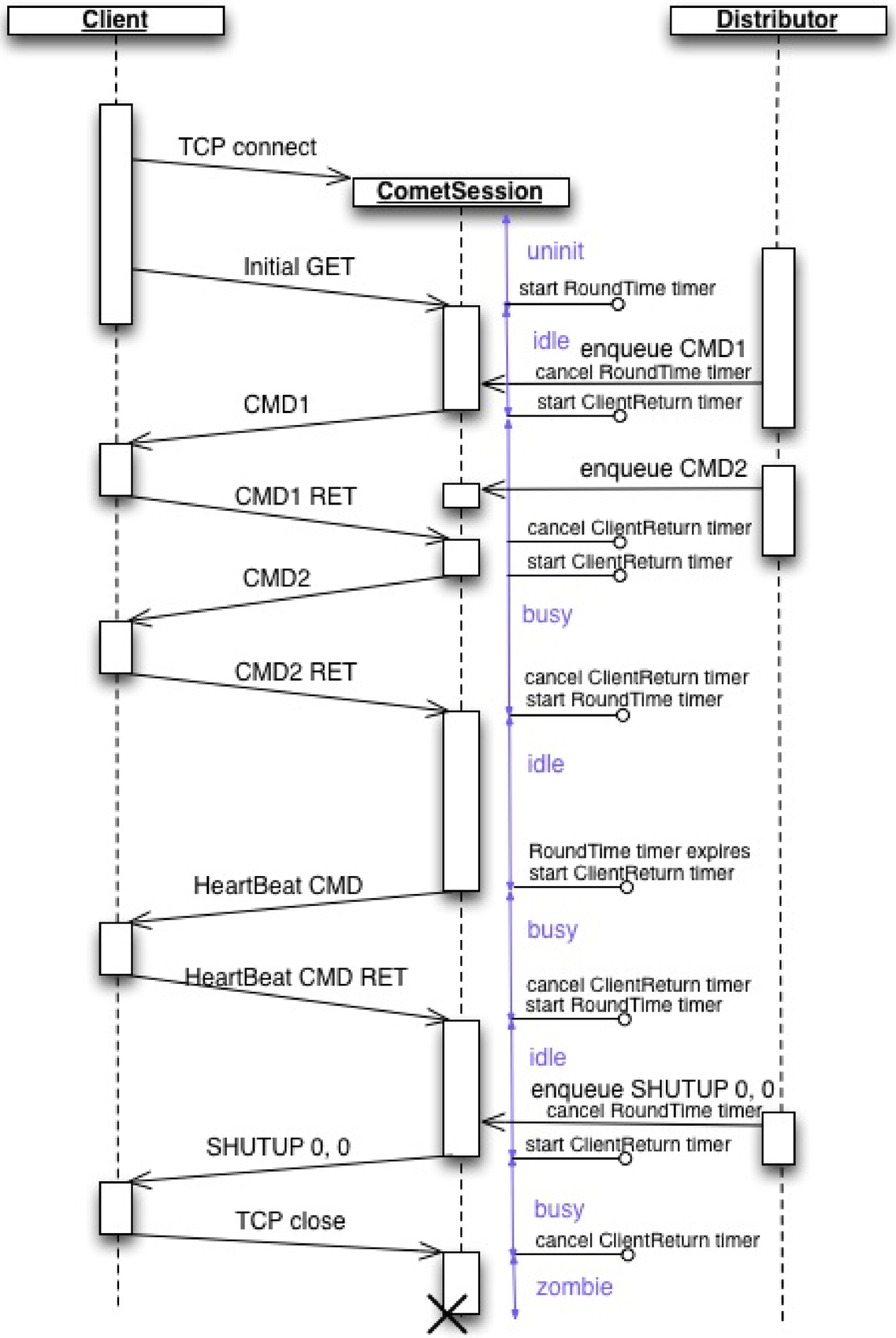
WAKEUP – similar to Google cloud messaging for Android

– addresed to specific client

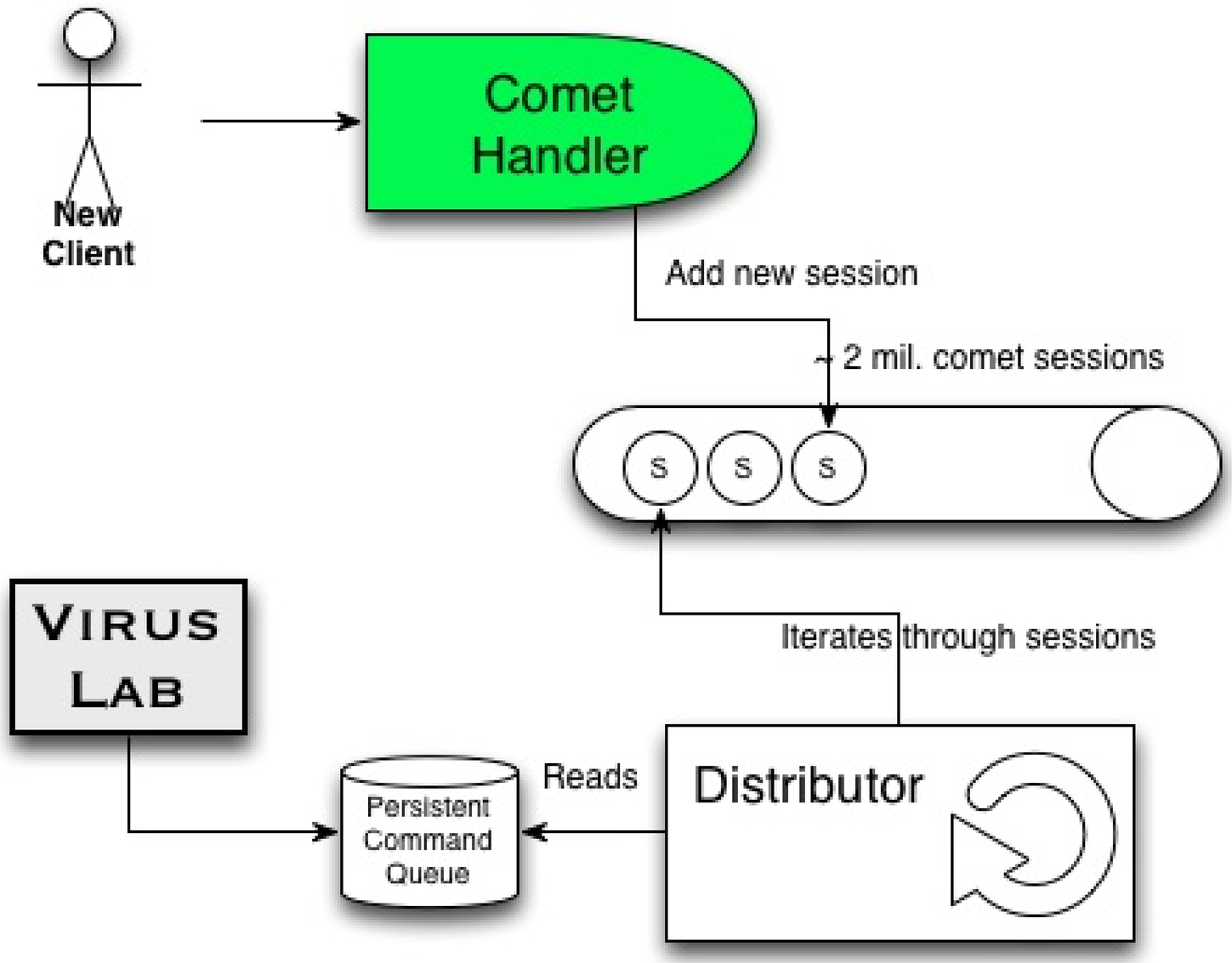
– helps decrease load on other

Communication

Long-held sessions on servers (comet, PUSH)



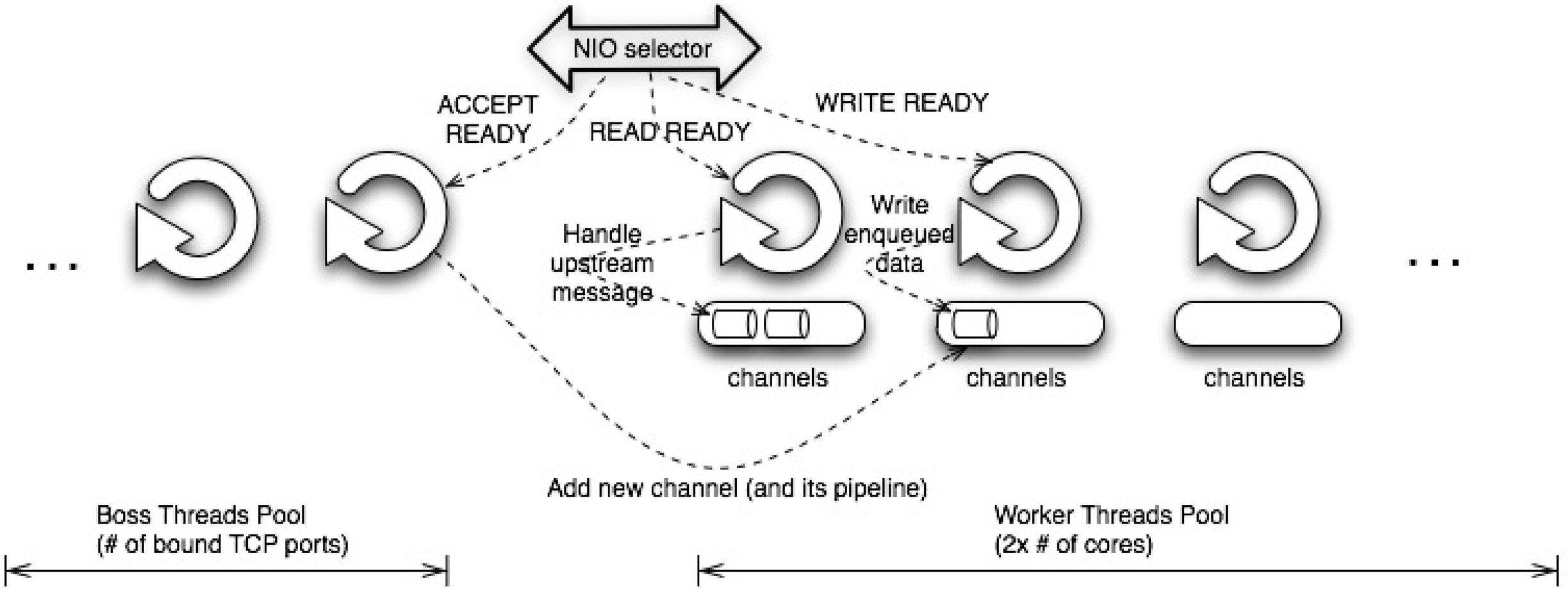
Distributor



Under the Hood – Netty

- Asynchronous event-driven network application framework written in Java
- Very good performance thanks to NIO and underlying **epoll** or **kqueue** kernel functions
- Easy programming model

Netty's Non-Blocking I/O

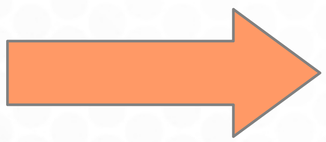


Concurrency optimizations

- `synchronized` is safe but expensive

```
class CommandQueue{
    private final NavigableSet<Command> queue =
        new TreeSet<Command>();

    synchronized void push(Command command) {
        queue.add(command)
    }
    synchronized Command pop(){
        return queue.pollFirst();
    }
    synchronized int size(){
        return queue.size();
    }
}
```



```
class CommandQueue{
    private final NavigableSet<Command> queue =
        new TreeSet<Command>();
    private final AtomicInteger size =
        new AtomicInteger(0);

    synchronized void push(Command command) {
        if (queue.add(command))
            size.incrementAndGet();
    }
    synchronized Command pop(){
        if (size() > 0){
            size.decrementAndGet();
            return s.pollFirst();
        }
        return null;
    }
    int size(){
        return size.get();
    }
}
```

Thread priorities

- Instead of controlling workload programatically we played with adjusting thread priorities
- Scheduler does the job for us

```
Thread.currentThread()
            .setPriority(Thread.MAX_PRIORITY);
```

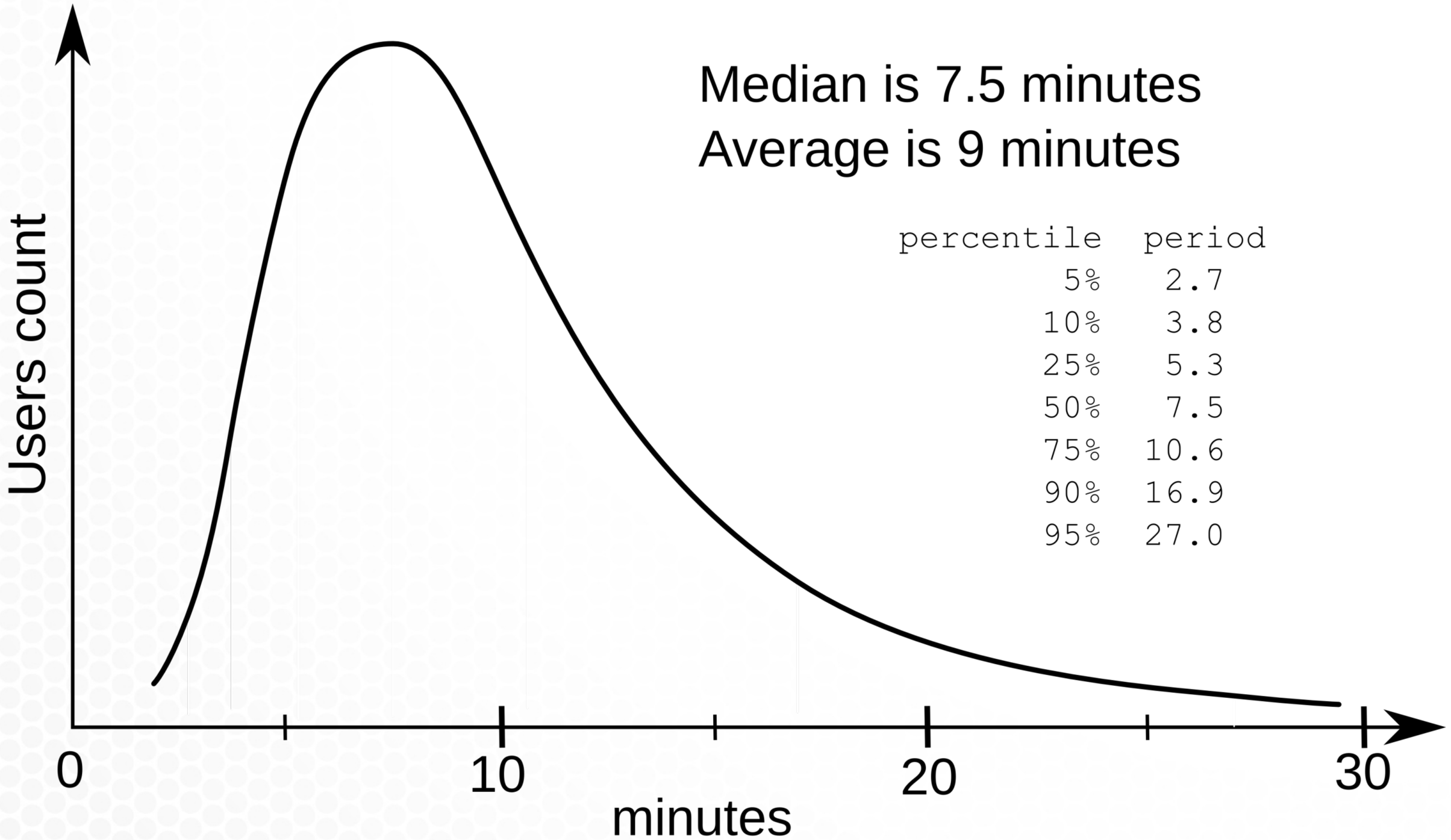
- Non-root processes can't increase thread priority...

JVM Workaround

- Workaround:
  - Bypassing Java permissions check
  - Mapping Java priorities explicitly to system „niceness“

```
-XX:+UseThreadPriorities
-XX:ThreadPriorityPolicy=42
-XX:JavaPriority10_To_OSPriority=0
-XX:JavaPriority9_To_OSPriority=1
  ...
```

Heartbeat period histogram

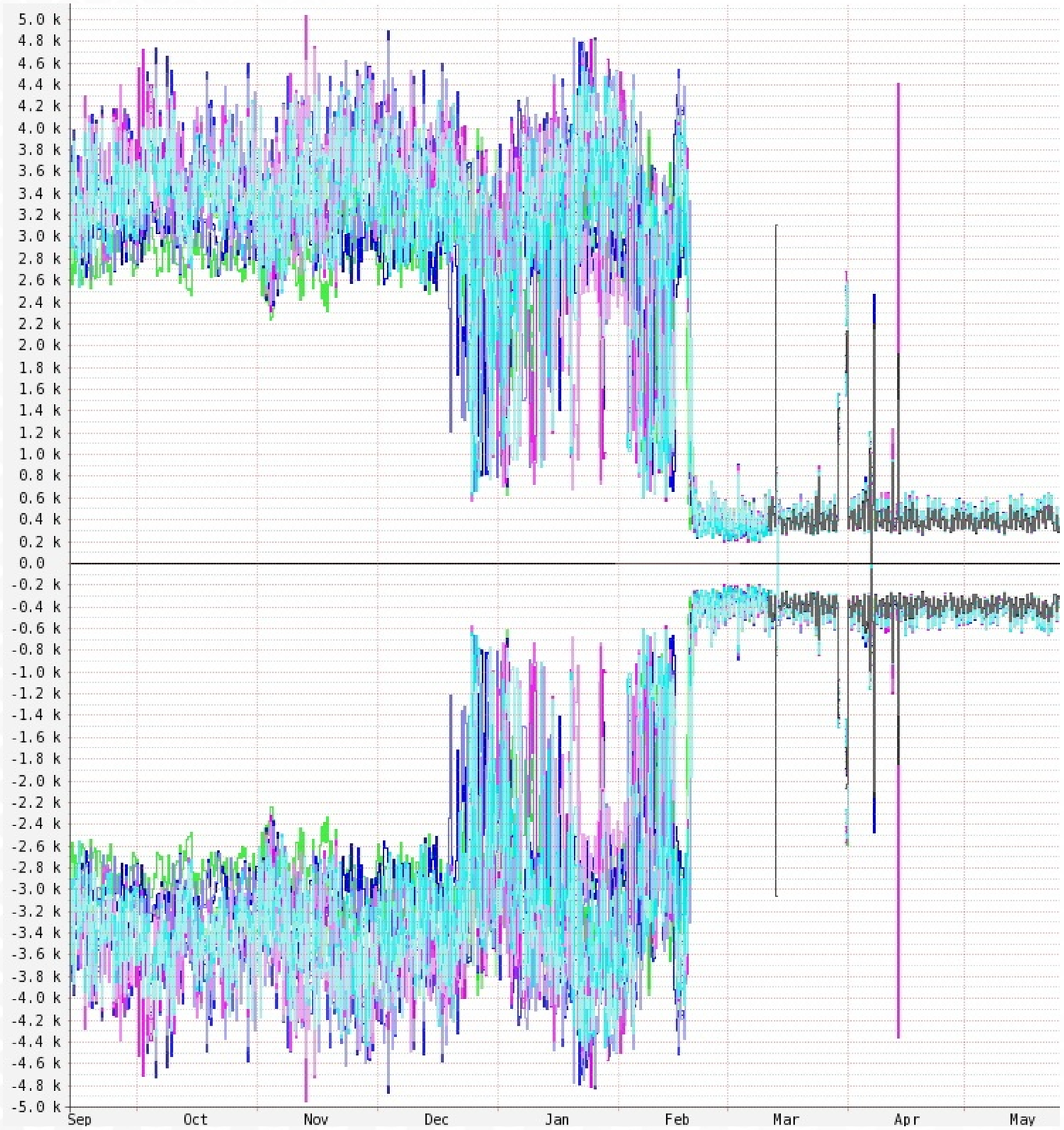


Median is 7.5 minutes

Average is 9 minutes

percentile	period
5%	2.7
10%	3.8
25%	5.3
50%	7.5
75%	10.6
90%	16.9
95%	27.0

Dynamic heartbeat



- NOP command
- Frequency evaluated for each client
- Heartbeat period varies from 2 to 30 minutes
- Reduces reconnection attempts from 5k/sec to 600/sec (8 times less)

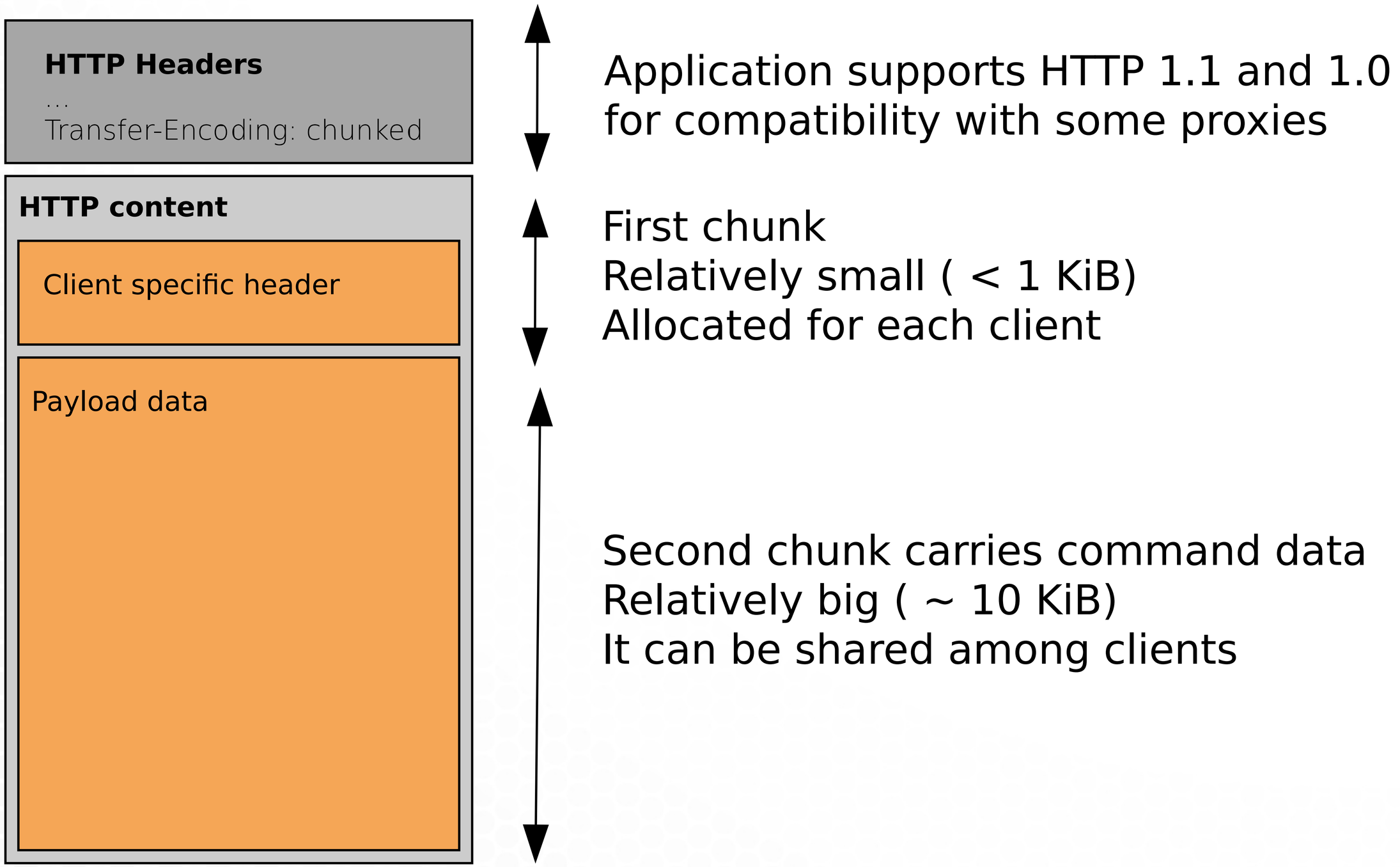
Java Heap

- Our app heap can consume up to 20 GiB
- One server handles up to 2 M clients in peak

  => we can't allocate buffer bigger than 10 KiB per client

- The solution is a chunked HTTP response with shared chunk buffer...
  - first chunk is user specific header
  - second is shared update package

Message chunks



Direct buffers

```
HttpResponse response = new DefaultHttpResponse(
        HttpVersion.HTTP_1_1, HttpResponseStatus.OK);
response.setChunked(true);
response.setHeader("Content-Type", "application/octet-stream");
// ... other http headers
response.setHeader("Transfer-Encoding", "chunked");
Channels.write(ctx, channelFuture, response); // write http header

// send first chunk with user specific GPB header
// ...

// now cached buffer with real content
ByteBuffer directBuffer = HTTP_CHUNK_CACHE.get(updateCommand);
ChannelBuffer payloadChunkBuffer =
        ChannelBuffers.wrappedBuffer(directBuffer);
Channels.write(ctx, channelFuture, payloadChunkBuffer);


java.nio.ByteBuffer.allocateDirect(capacity).order(ByteOrder.BIG_ENDIAN);
```

Kernel TCP memory

- How much memory does TCP stack use?

```
$ cat /proc/net/sockstat
sockets: used 1500997
TCP: inuse 1178915 orphan 6078 tw 729 alloc 1506876 mem 1399963
UDP: inuse 10 mem 2
UDPLITE: inuse 0
RAW: inuse 0
FRAG: inuse 1 memory 960
```

1399963 pages is ~5GiB*

- Is there some limit?

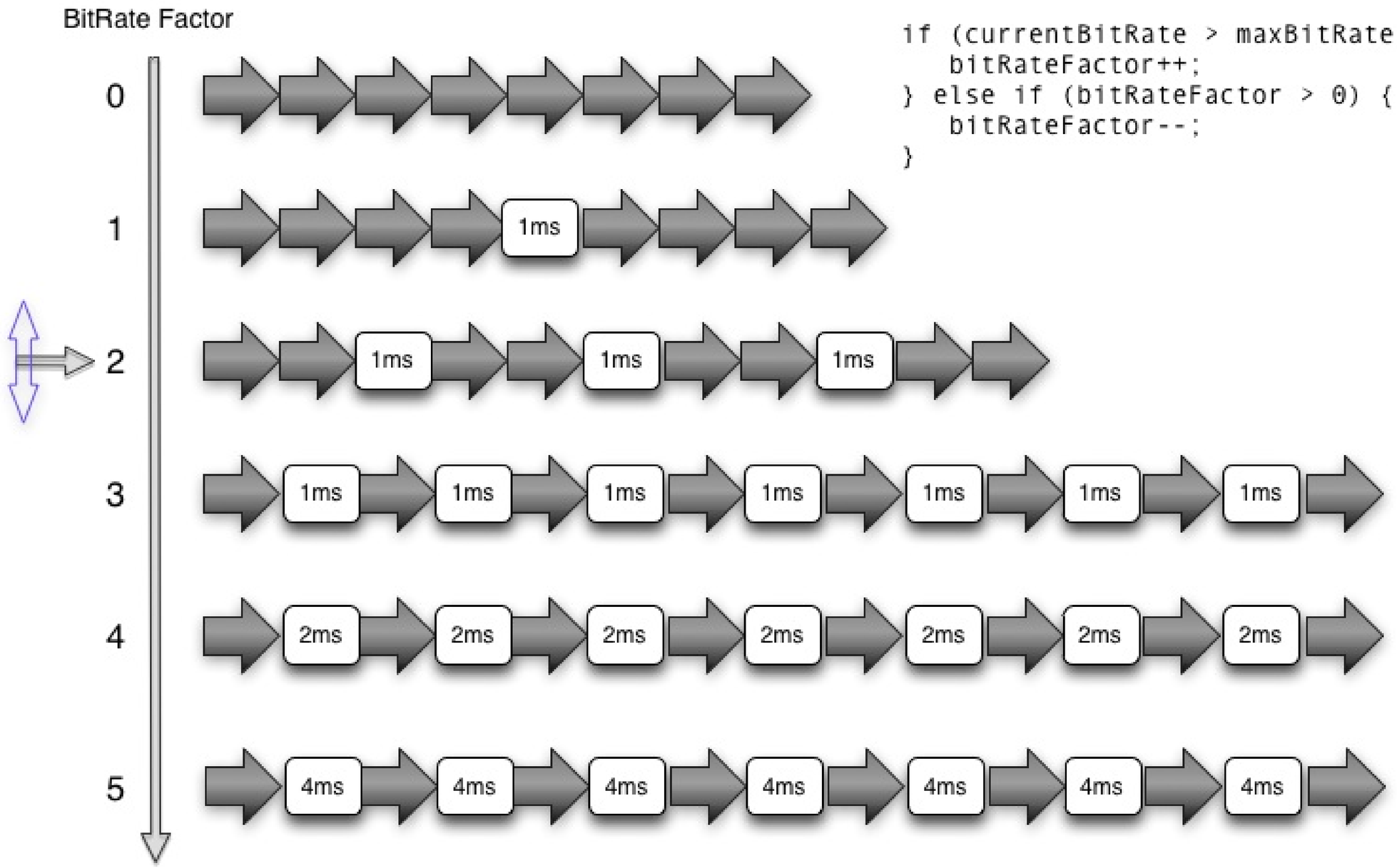
```
$ cat /proc/sys/net/ipv4/tcp_mem
Low: 500000 (2GB) Pressure: 1200000 (4.5GB) Max: 1400000 (5.3GB)
```

*Memory values are in pages (4 KiB) !

Bandwith Limiter

- Protects kernel’s socket memory from exhaustion => dropping packets by kernel
- Limiter watches the memory used by TCP stack and compares it with the pressure limit
- In case of big packets the distribution rate may exceed the bitrate contracted with the provider
- Limiter calculates the distribution rate on-the-fly and compares it with the predefined max bitrate

Bandwith Limiter




```
if (currentBitRate > maxBitRate) {
   bitRateFactor++;
} else if (bitRateFactor > 0) {
   bitRateFactor--;
}
```

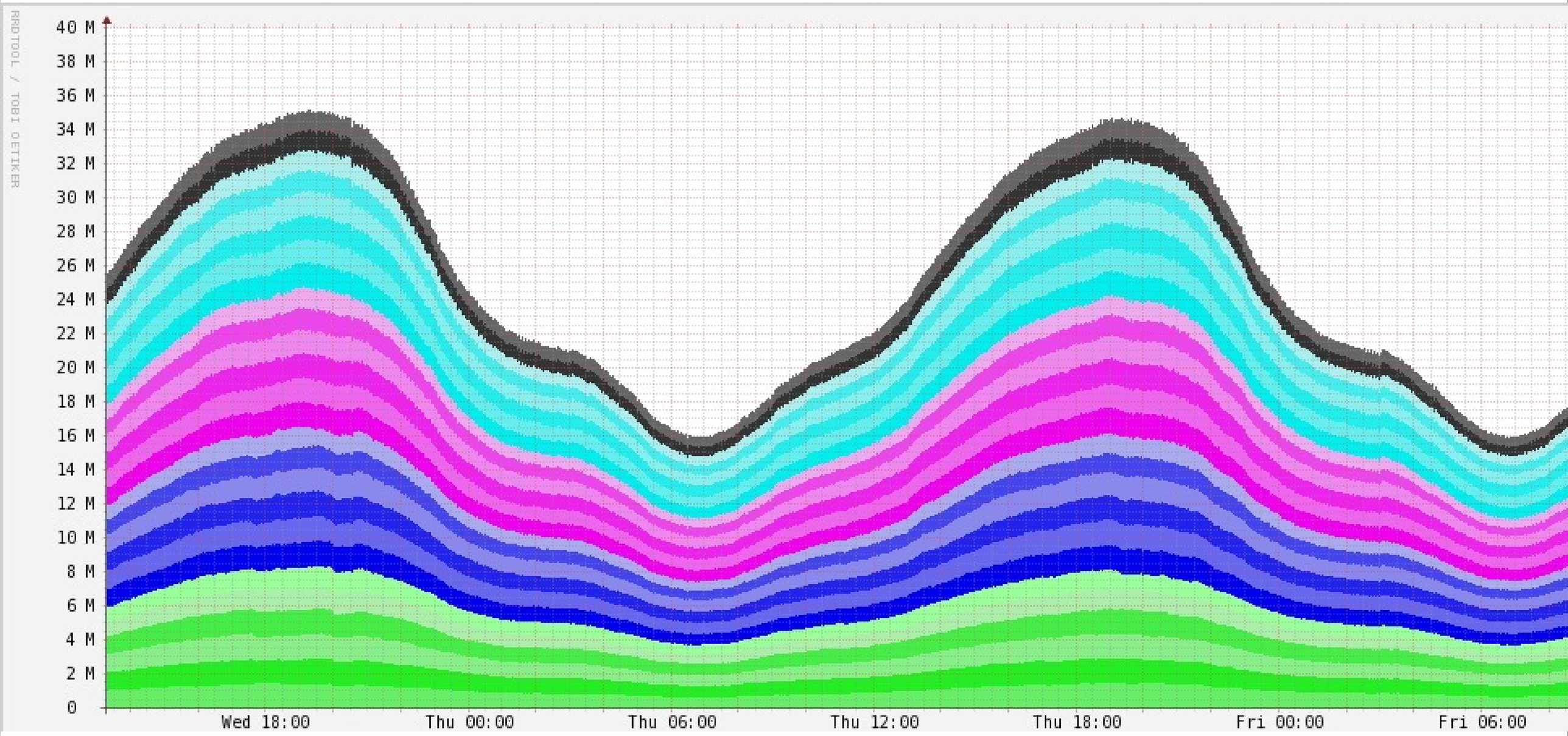
System Configuration

- Current deployment
  - 26 servers (~2M clients and 3Gbps bandwidth each)
  - Up to 40Gbps of distribution traffic
  - 15k users per second turnaround

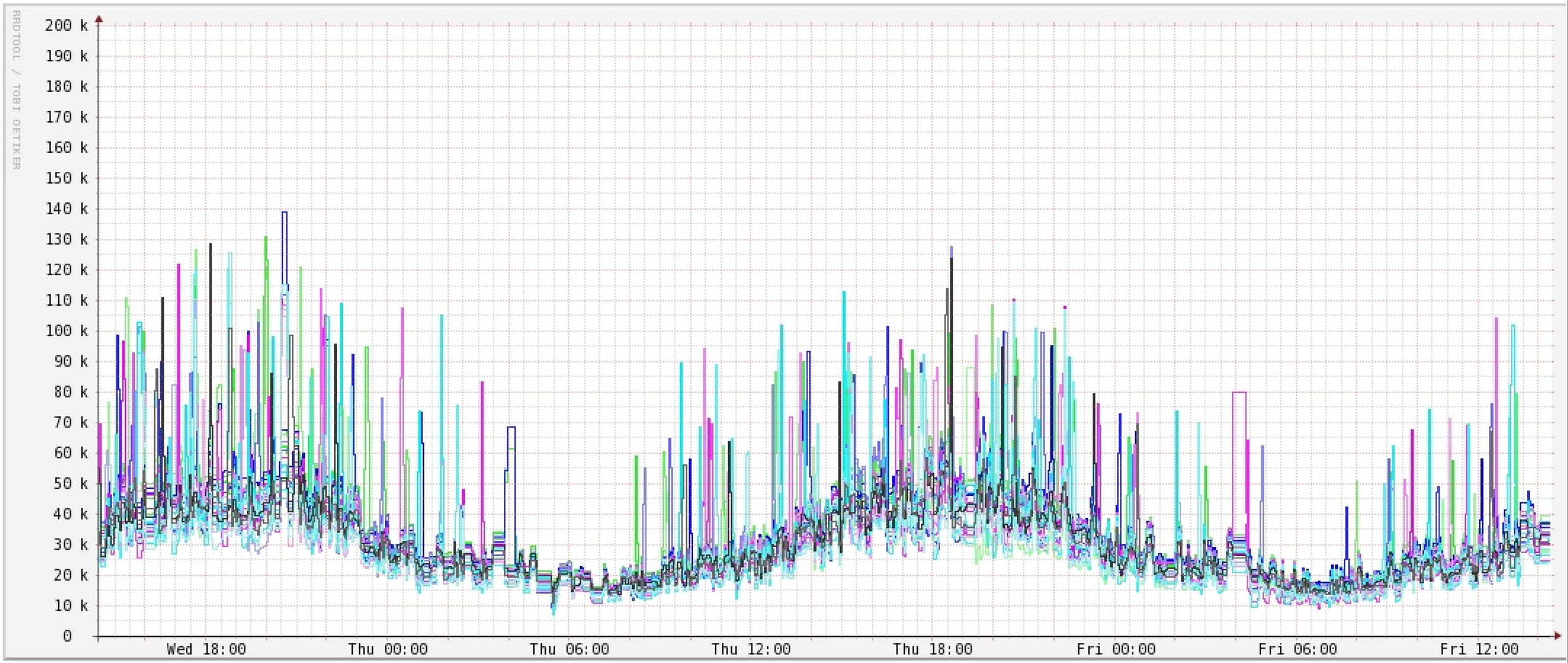
- Other usability
  - Can send any commands to the clients
  - Used for the service AccessAnywhere

- Threat of a self-induced DDoS
  - An outage can cause “DDoS” (2M/s SYN packets)

Connected clients (all nodes)

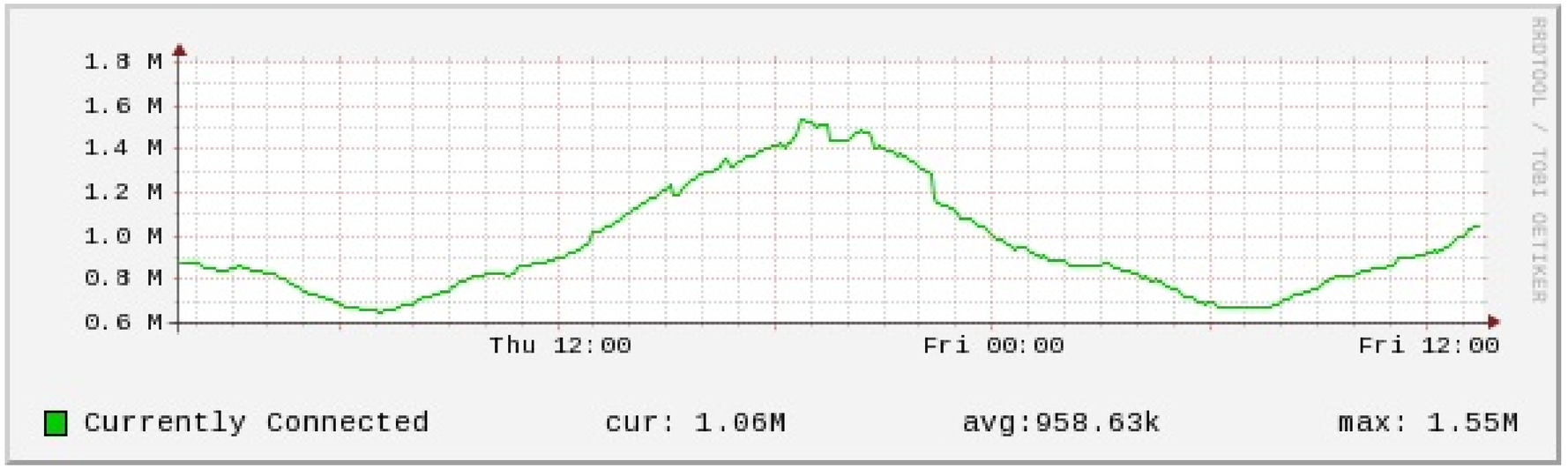


Distribution time

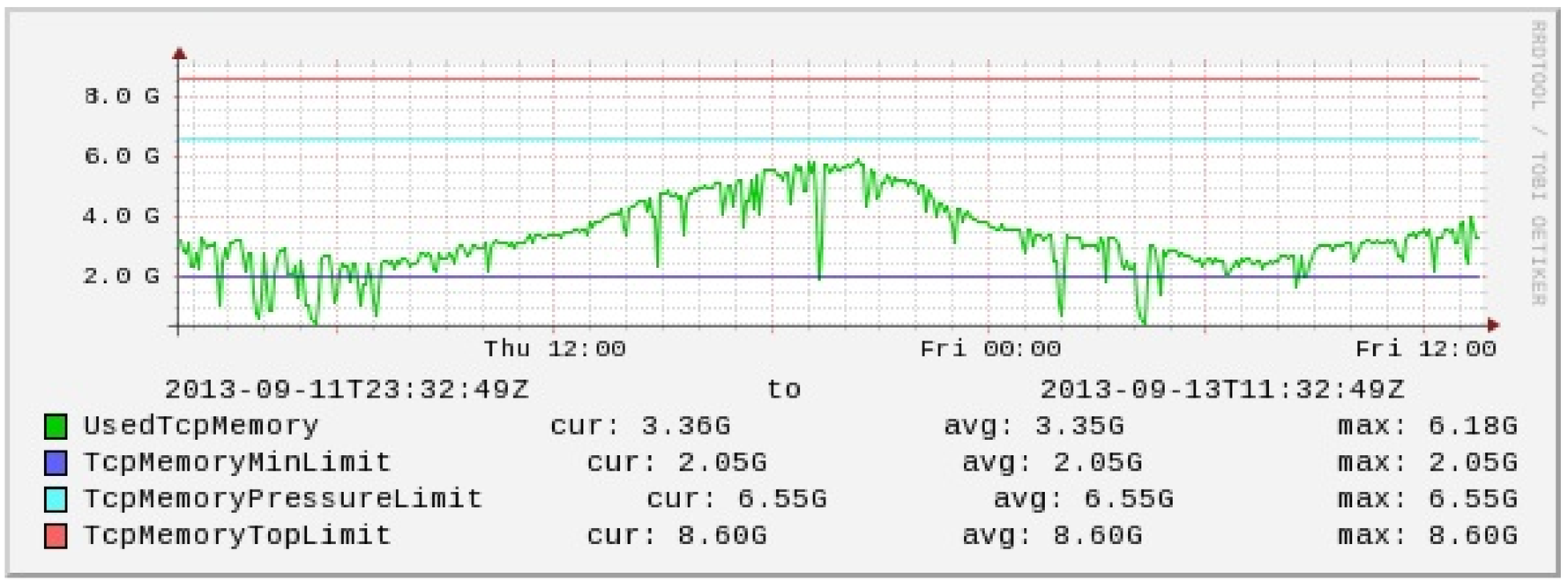


- Time is in miliseconds.
- The graph represents the delay between the reception of the update and the transfer to the outgoing TCP buffers in the kernel for all connected clients.

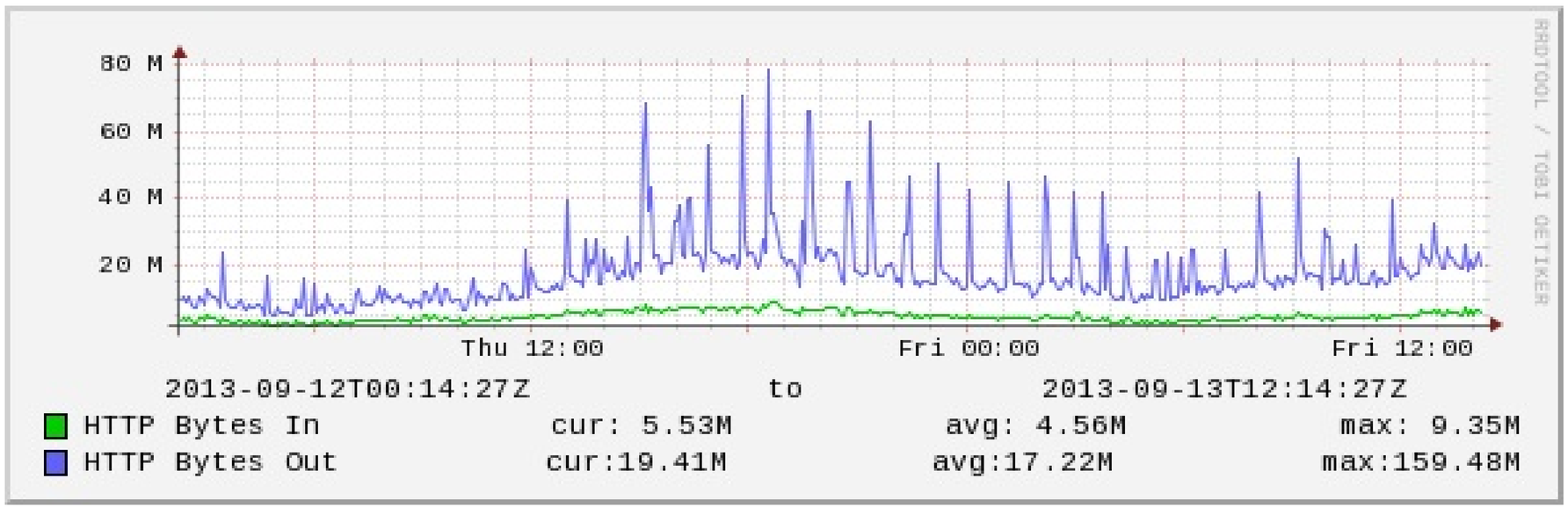
Connected clients



TCP memory



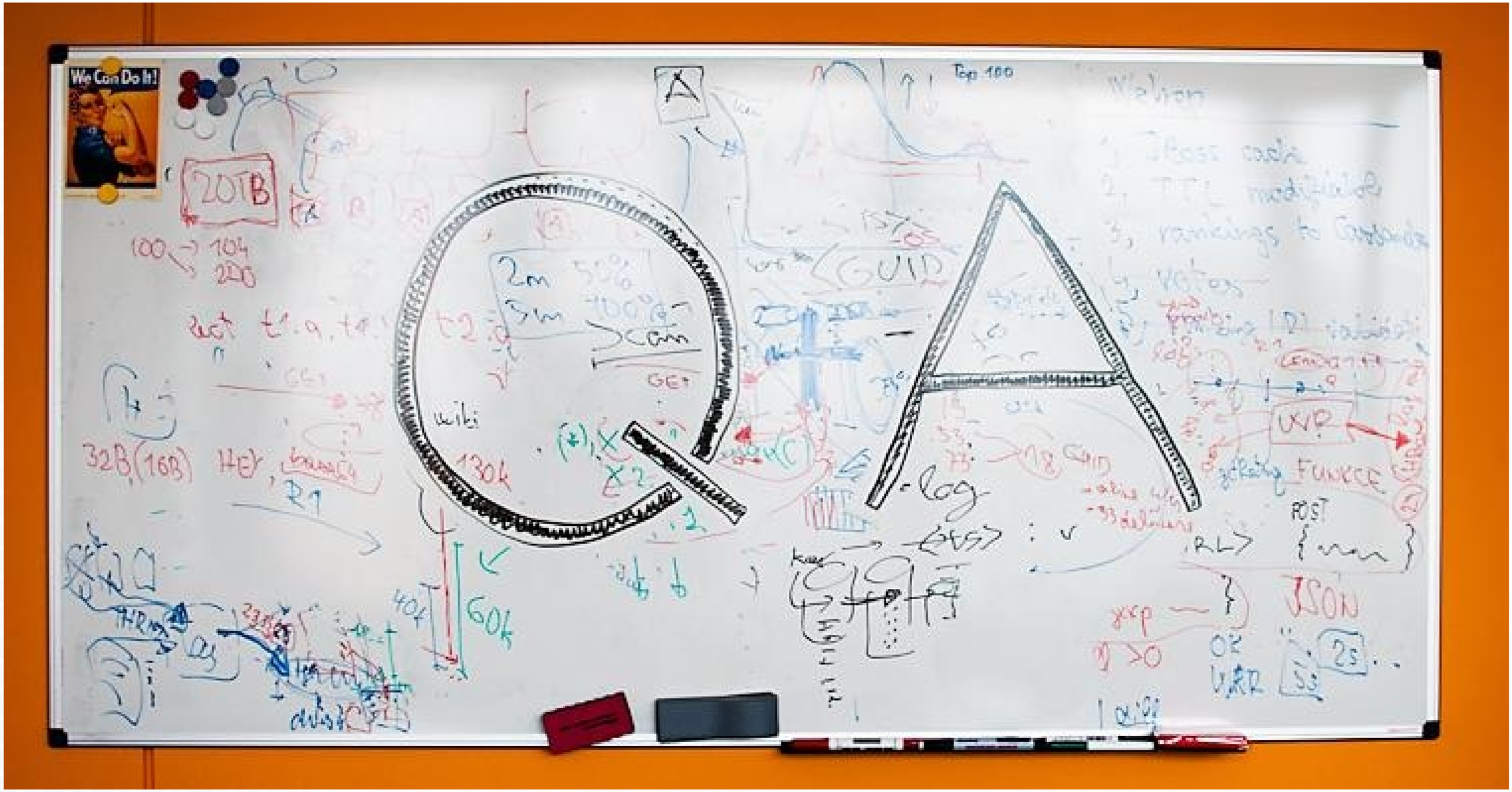
TCP throughput



Conclusion

1. Focus on synchronization and concurrency issues (lock-free algorithms, CAS)
2. Tune the JVM with respect to the underlying operating system.
3. Adaptive behavior of the application toward the clients.
4. Application must be seen in the context of the underlying system. Using as much information from the system as possible.

Questions or Answers



Links

- Avast @ Github: https://github.com/avast
- Code sample:

  https://gist.github.com/Karry/5291694