Thousands of Threads and Blocking I/O

The old way to write Java Servers is New again

(and way better)

Paul Tyma

paul.tyma@gmail.com

paultyma.blogspot.com

SD WEST 2008

MARCH 3-7, 2008, SANTA CLARA, CA

Who am I

- Day job: Senior Engineer, Google
- Founder, Preemptive Solutions, Inc.
  - Dasho, Dotfuscator
- Founder/CEO ManyBrain, Inc.
  - Mailinator
  - empirical max rate seen ~1250 emails/sec
  - extrapolates to 108 million emails/day
- Ph.D., Computer Engineering, Syracuse University

About This Talk

- Comparison of server models
  - IO – synchronous
  - NIO – asynchronous
- Empirical results
- Debunk myths of multithreading
  - (and nio at times)
- Every server is different
  - compare 2 multithreaded server applications

An argument over server design

- Synchronous I/O, single connection per thread model
  - high concurrency
  - many threads (thousands)
- Asynchronous I/O, single thread per server!
  - in practice, more threads often come into play
  - Application handles context switching between clients

Evolution

- We started with simple threading modeled servers
  - one thread per connection
  - synchronization issues came to light quickly
  - Turned out synchronization was hard
    - Pretty much, everyone got it wrong
    - Resulting in nice, intermittent, untraceable, unreproducible, occasional server crashes
    - Turns out whether we admit it or not, “occasional” is workable

Evolution

- The bigger problem was that scaling was limited
- After a few hundred threads things started to break down
  - i.e., few hundred clients
- In comes java NIO
  - a pre-existing model elsewhere (C++, linux, etc.)
- Asynchronous I/O
  - I/O becomes “event based”

Evolution - nio

- The server goes about its business and is “notified” when some I/O event is ready to be processed
- We must keep track of where each client is within a i/o transaction
  - telephone: “For client XYZ, I just picked up the phone and said 'Hello'. I'm now waiting for a response.”
  - In other words, we must explicitly save the state of each client

Evolution - nio

- In theory, one thread for the entire server
  - no synchronization
  - no given task can monopolize anything
- Rarely, if ever works that way in practice
  - small pools of threads handle several stages
  - multiple back-end communication threads
  - worker threads
  - DB threads

Evolution

- SEDA
  - Matt Welsh's Ph.D. thesis
  - http://www.eecs.harvard.edu/~mdw/proj/seda
  - Gold standard in server design
    - dynamic thread pool sizing at different tiers
    - somewhat of an evolutionary algorithm – try another thread, see what happens
    - Build on Java NBIO (other NIO lib)

Evolution

- Asynchronous I/O
  - Limited by CPU, bandwidth, File descriptors (not threads)
- Common knowledge that NIO >> IO
  - java.nio
  - java.io
- Somewhere along the line, someone got “scalable” and “fast” mixed up – and it stuck

NIO vs. IO
(asynchronous vs. synchronous io)

An attempt to examine both paradigms

- If you were writing a server today, which would you choose?
  - why?

Reasons I've heard to choose NIO
(note: all of these are up to debate)

- Asynchronous I/O is faster
- Thread context switching is slow
- Threads take up too much memory
- Synchronization among threads will kill you
- Thread-per-connection does not scale

Reasons to choose thread-per-connection IO
(again, maybe, maybe not, we'll see)

- Synchronous I/O is faster
- Coding is much simpler
  - You code as if you only have one client at a time
- Make better use of multi-core machines

All the reasons
(for reference)

- *nio: Asynchronous I/O is faster*
- *io: Synchronous I/O is faster*
- *nio: Thread context switching is slow*
- *io: Coding is much simpler*
- *nio: Threads take up too much memory*
- *io: Make better use of multi-cores*
- *nio: Synchronization among threads will kill you*
- *nio: Thread-per-connection does not scale*

NIO vs. IO
which is faster

- Forget threading a moment
    - single sender, single receiver
- For a tangential purpose I was benchmarking NIO and IO
  - simple “blast data” benchmark
- I could only get NIO to transfer data up to about 75% of IO's speed
  - asynchronous (blocking NIO was just as fast)
  - I blamed myself because we all know asynchronous is faster than synchronous right?

NIO vs. IO

- Started doing some emailing and googling looking for benchmarks, experiential reports
  - Yes, everyone knew NIO was faster
  - No, no one had actually personally tested it
- Started formalizing my benchmark then found
- http://www.theserverside.com/discussions/thread.tss?thread_
- http://www.realityinteractive.com/rgrzywinski/archives/00009

Excerpts

- Blocking model was consistently 25-35% faster than using NIO selectors. Lot of techniques suggested by EmberIO folks were employed - using multiple selectors, doing multiple (2) reads if the first read returned EAGAIN equivalent in Java. Yet we couldn't beat the plain thread per connection model with Linux NPTL.
- To work around not so performant/scalable poll() implementation on Linux's we tried using epoll with Blackwidow JVM on a 2.6.5 kernel. while epoll improved the over scalability, the performance still remained 25% below the vanilla thread per connection model. With epoll we needed lot fewer threads to get to the best performance mark that we could get out of NIO.

Rahul Bhargava, CTO Rascal Systems

Asynchronous
(simplified)

1)Make system call to selector

2)if nothing to do, goto 1

3)loop through tasks

   a)if its an OP_ACCEPT, system call to accept the connection, save the key

   b)if its an OP_READ, find the key, system call to read the data

   c)if more tasks goto 3

4)goto 1

Synchronous
(simplified)

1)Make system call to accept connection

(thread blocks there until we have one)

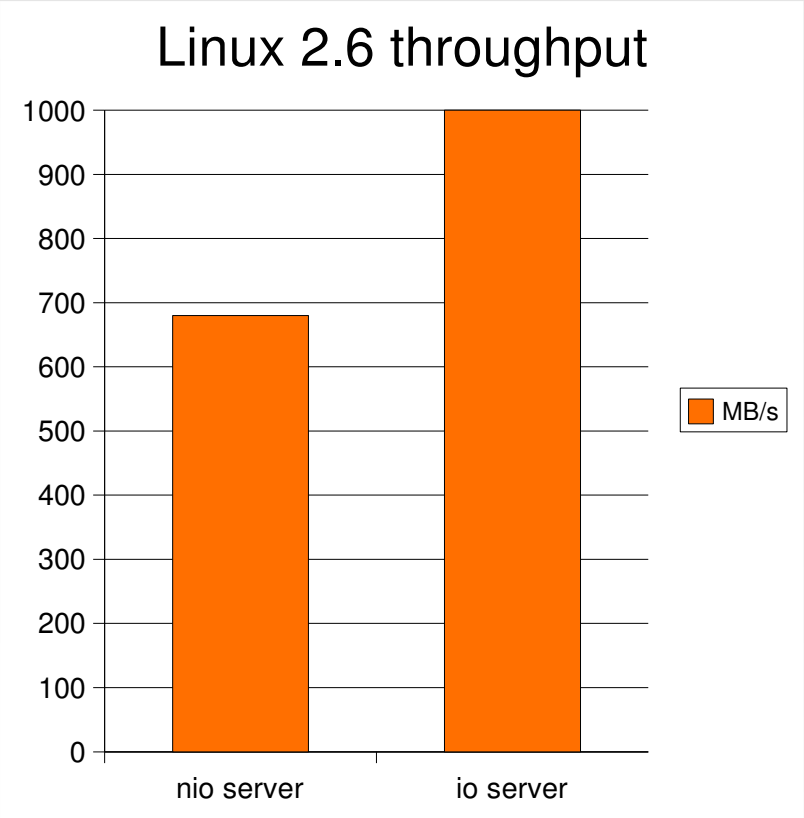
2)Make system call to read data

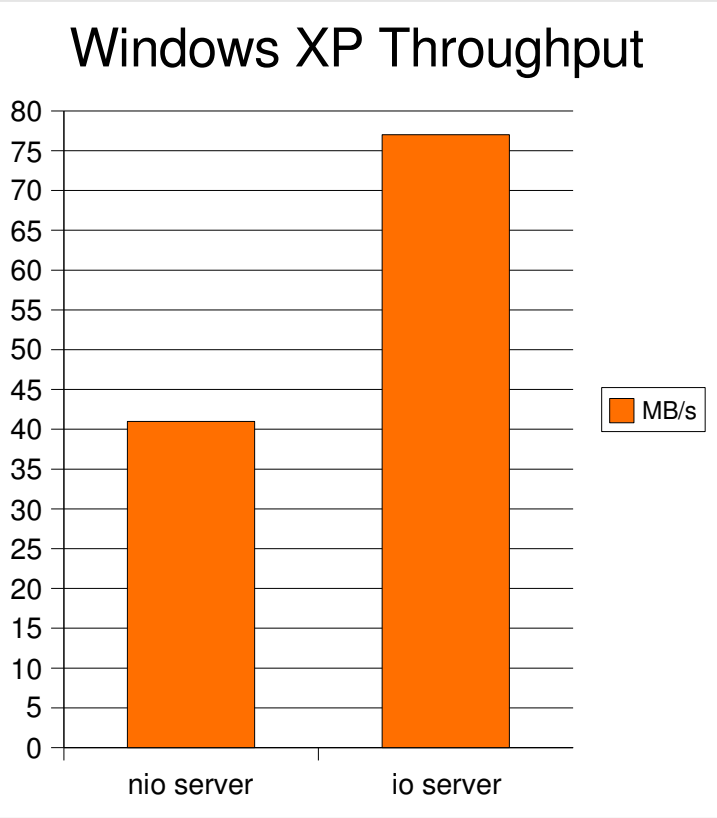
(thread blocks there until it gets some)

3)goto 2

Straight Throughput

*do not compare charts against each other





All the reasons

- ~~nio: Asynchronous I/O is faster~~
- io: Synchronous I/O is faster
- *nio: Thread context switching is slow*
- *io: Coding is much simpler*
- *nio: Threads take up too much memory*
- *io: Make better use of multi-cores*
- *nio: Synchronization among threads will kill you*
- *nio: Thread-per-connection does not scale*

Multithreading

Multithreading

- Hard
  - You might be saying “nah, its not bad”
- Let me rephrase
  - Hard, because everyone thinks it isn't
- http://today.java.net/pub/a/today/2007/06/28/extending-reentrantreadwritelock.html
- http://www.ddj.com/java/199902669?pgno=3
- Much like generics however, you can't avoid it
  - good news is that the rewards are significant
  - Inherently takes advantage of multi-core systems

Multithreading

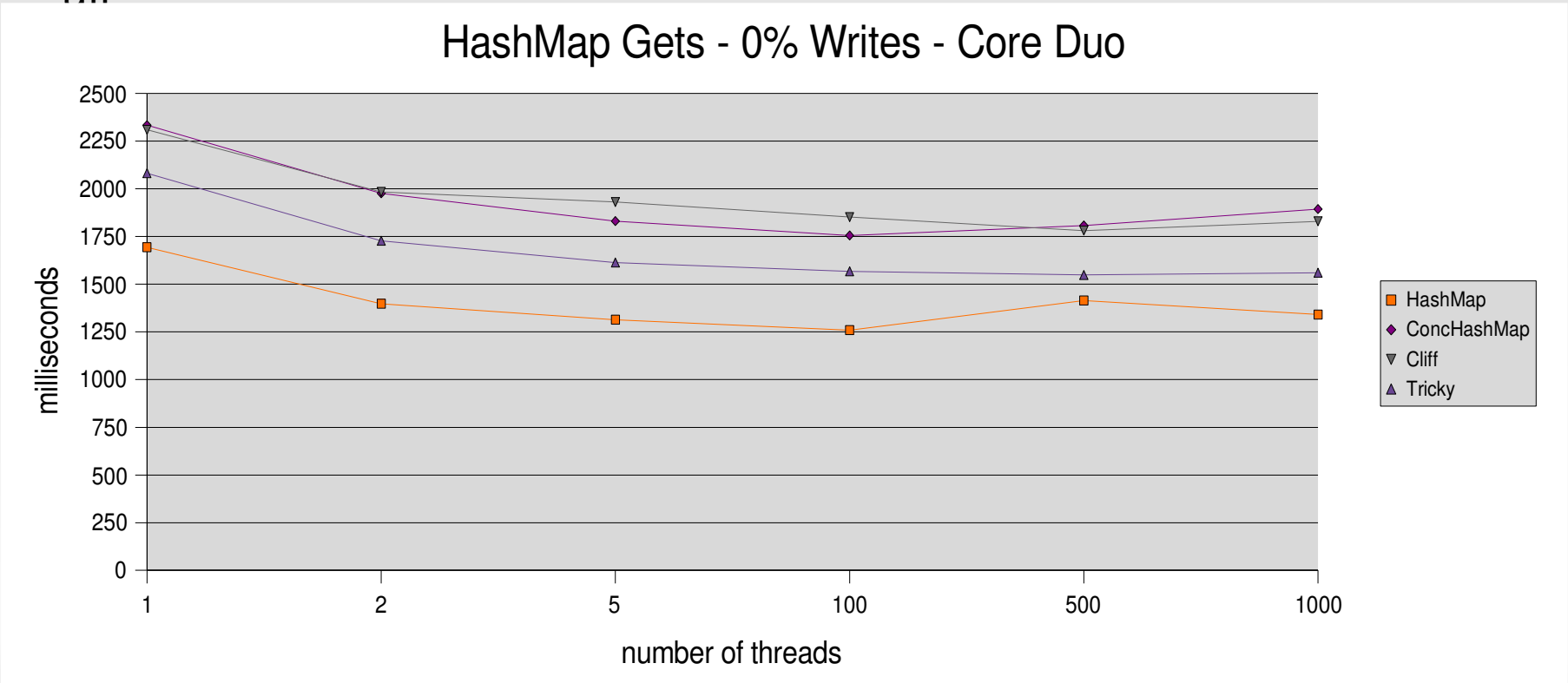
- Linux 2.4 and before
  - Threading was quite abysmal
  - few hundred threads max
- Windows actually quite good
- up to 16000 threads
  - jvm limitations

In walks NPTL

- Threading library standard in Linux 2.6
  - linux 2.4 by option
- Idle thread cost is near zero
- context-switching is much much faster
- Possible to run many (many) threads
- http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

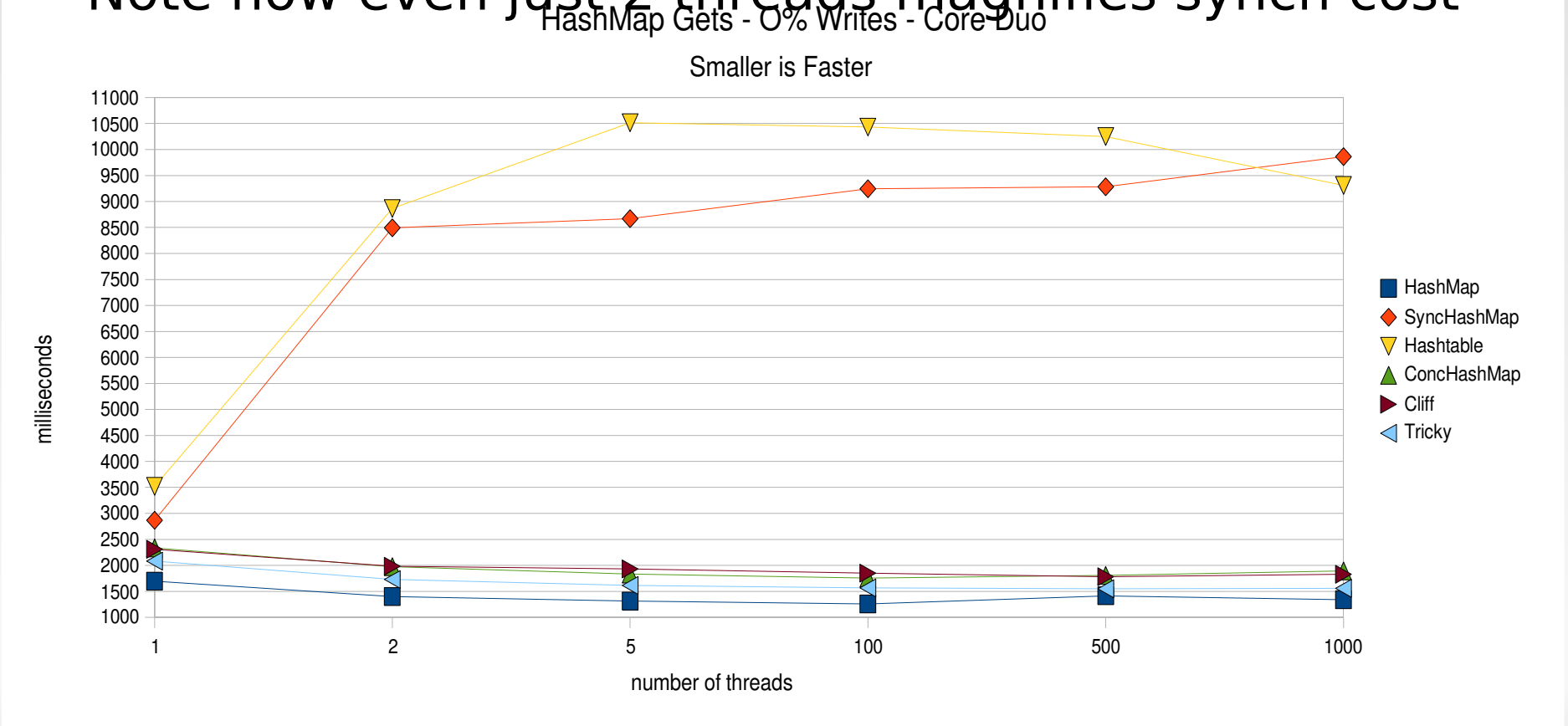
Thread context-switching is expensive

- Lower lines are faster – JDK1.6, Core duo
- Blue line represents up-to-1000 threads competing for the CPUs (core duo)
- notice behavior between 1 and 2 threads
- 1 or 1000 all fighting for the CPU, context switching doesn't cost much at all



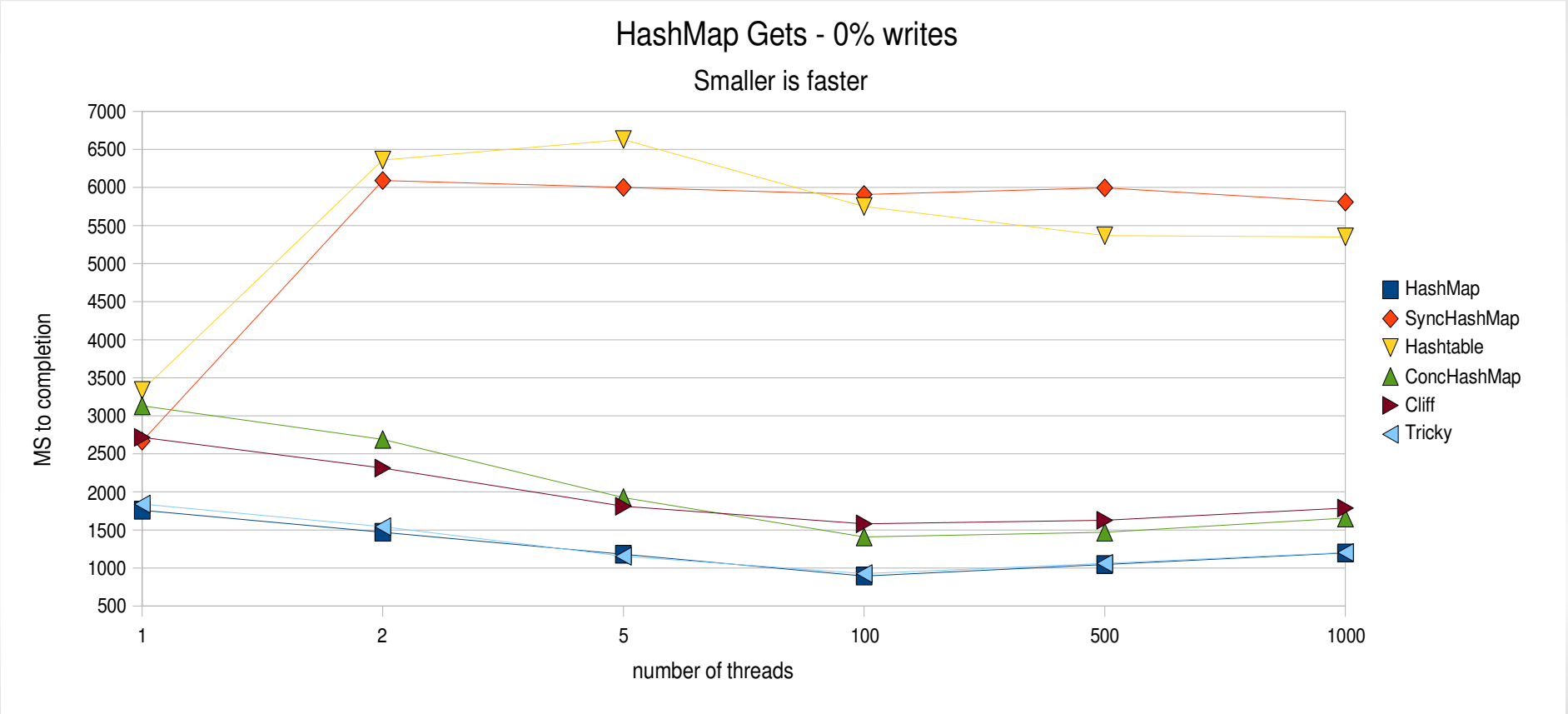
Synchronization is Expensive

- With only one thread, uncontended synchronization is cheap
- Note how even just 2 threads magnifies synch cost



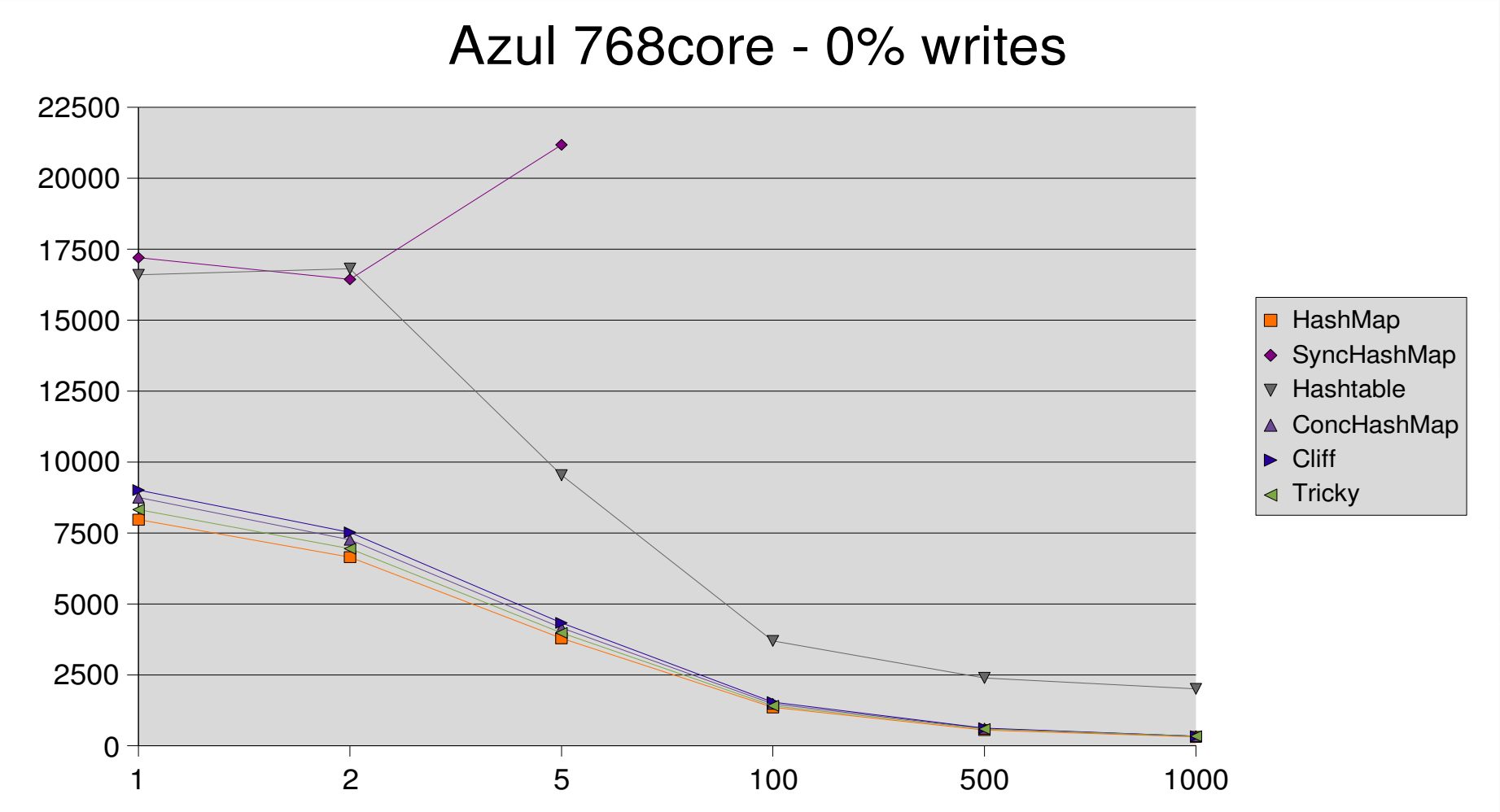
4 core Opteron

- Synchronization magnifies method call overhead
- more cores, more “sink” in line
- non-blocking data structures do very well - we don't always need explicit synchronization



How about a **lot** more cores?

guess: how many cores standard in 3 years?



Threading Summary

- Uncontended Synchronization is cheap
  - and can often be free
- Contended Synch gets more expensive
- Nonblocking datastructures scale well
- Multithreaded programming style is quite viable even on single core

All the reasons

- io: Synchronous I/O is faster
- ~~nio: Thread context switching is slow~~
- *io: Coding is much simpler*
- *nio: Threads take up too much memory*
- *io: Make better use of multi-cores*
- ~~nio: Synchronization among threads will kill you~~
- ~~nio: Thread-per-connection does not scale~~

Many possible NIO server configurations

- True single thread
  - Does selects
  - reads
  - writes
  - and backend data retrieval/writing
    - from/to db
    - from/to disk
    - from/to cache

Many possible NIO server configurations

- True single thread
  - This doesn't take advantage of multicores
  - Usually just one selector
- Usually use a threadpool to do backend work
- NIO must lock to give writes back to net thread
- Interview question:
  - What's harder, synchronizing 2 threads or synchronizing 1000 threads?

All the reasons

- io: Synchronous I/O is faster
- *io: Coding is much simpler*
- *nio: Threads take up too much memory*

CHOOSE

- *io: Make better use of multi-cores*
- ~~nio: Synchronization among threads will kill you~~

The story of Rob Van Behren

(as remembered by me from a lunch with Rob)

- Set out to write a high-performance asynchronous server system
- Found that when switching between clients, the code for saving and restoring values/state was difficult
- Took a step back and wrote a finely-tuned, organized system for saving and restoring state between clients
- When he was done, he sat back and realized he had written the foundation for a threading package

Synchronous I/O

state is kept in control flow

```
// exceptions and such left out
InputStream inputStream = socket.getInputStream();
OutputStream outputStream = socket.getOutputStream();

String command = null;
do {
 command = inputStream.readUTF();
 } while ((!command.equals("HELO") && sendError());

do {
 command = inputStream.readUTF();
} while ((!command.startsWith("MAIL FROM:") && sendError());
handleMailFrom(command);

do {
 command = inputStream.readUTF();
} while ((!command.startsWith("RCPT TO:") && sendError());
handleRcptTo(command);

do {
 command = inputStream.readUTF();
} while ((!command.equals("DATA") && sendError());
handleData();
```

Server in Action

Our 2 server designs
(synchronous multithreaded)

- One thread per connection
  - All code is written as if your server can only handle one connection
  - And all data structures can be manipulated by many other threads
  - Synchronization can be tricky
- Reads are blocking
  - Thus when waiting for a read, a thread is completely idle
  - Writing is blocking too, but is not typically significant
- Hey, what about scaling?

Synchronous Multithreaded

- Mailinator server:
  - Quad opteron, 2GB of ram, 100Mbps ethernet, linux 2.6, java 1.6
  - Runs both HTTP and SMTP servers
- Quiz:
  - How many threads can a computer run?
  - How many threads **should** a computer run?

Synchronous Multithreaded

- Mailinator server:
  - Quad opteron, 2GB of ram, 100Mbps ethernet, linux 2.6, java 1.6
- How many threads can a computer run?
  - Each thread in java x32 requires 48k of stack space
    - linux java limitation, not OS (windows = 2k?)
  - option -Xss:48k (512k by default)
  - ~41666 stack frames
  - not counting room for OS, java, other data, etc

Synchronous Multithreaded

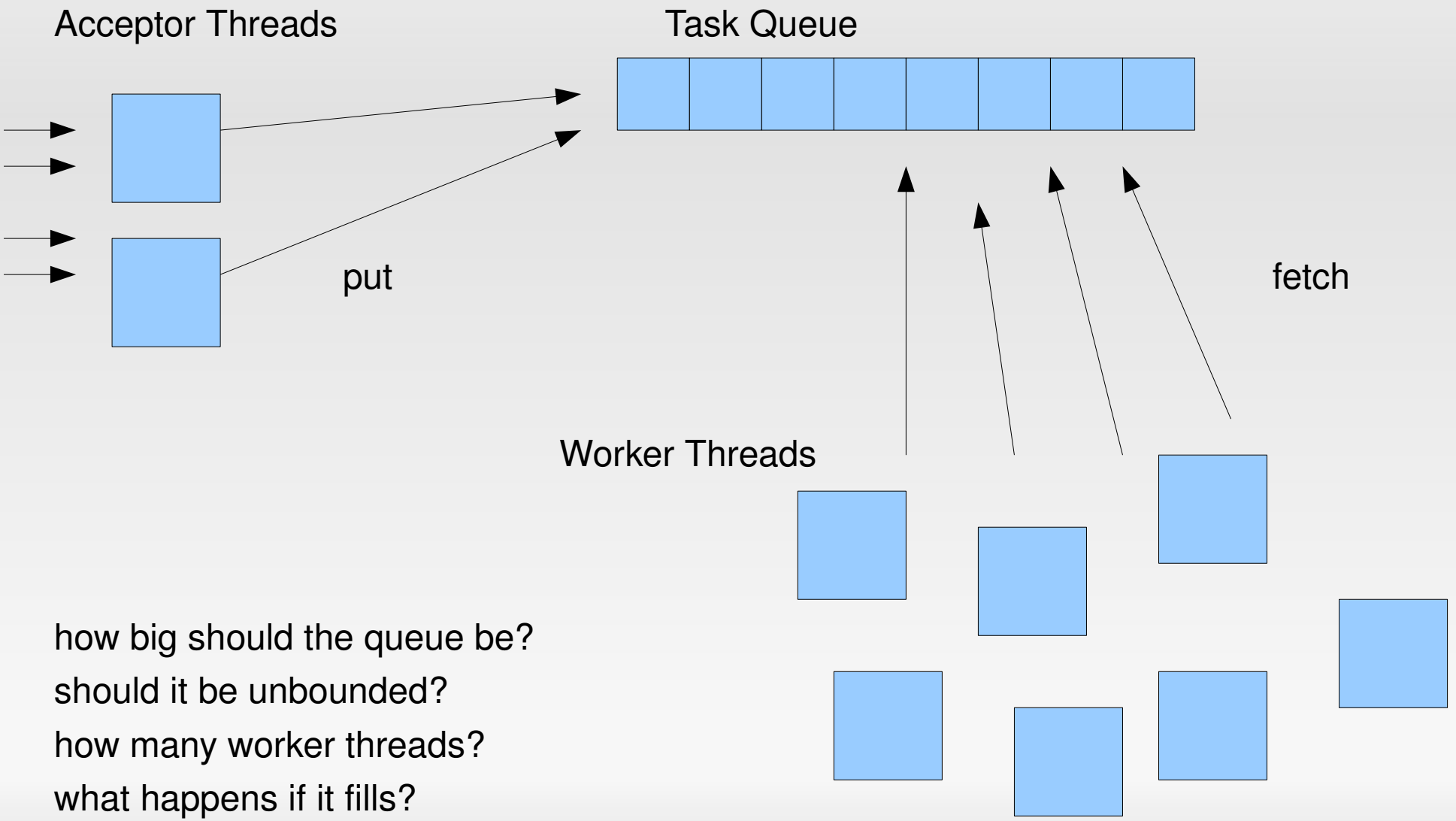
- Mailinator server:
  - Quad opteron, 2GB of ram, 100Mbps ethernet, linux 2.6, java 1.6
- How many threads **should** a computer run?
  - Similar to asking, how much should you eat for dinner?
    - usual answer: until you've had enough
    - usual answer: until you're full
    - fair answer: until you're sick
    - odd answser: until you're dead

Synchronous Multithreaded

- Mailinator server:
  - Quad opteron, 2GB of ram, 100Mbps ethernet, linux 2.6, java 1.6
- How many threads **should** a computer run?
  - “Just enough to max out your CPU(s)”
    - replace “CPU” with any other resource
  - How many threads should you run for 100% cpu bound tasks?
    - maybe #ofCores+1 or so

Synchronous Multithreaded

- Note that servers must pick a saturation point
- Thats either CPU or network or memory or something
- Typically serving a lot of users well is better than serving a lot more very poorly (or not at all)
- You often need “push back” such that too much traffic comes all the way back to the front



how big should the queue be?

should it be unbounded?

how many worker threads?

what happens if it fills?

Design Decisions

ThreadPools

- Executors.newCachedThreadPool = evil, die die die
- Tasks goto SynchronousQueue
  - i.e., direct hand off from task-giver to new thread
- unused threads eventually die (default: 60sec)
- new threads are created if all existing are busy
  - but only up to MAX_INT threads (snicker)
- Scenario: CPU is pegged with work so threads aren't finishing fast, more tasks arrive, more threads are created to handle new work
  - when CPU is pegged, more threads is not what you need

Blocking Data Structures

- BlockingQueue
  - canonical “hand-off” structure
  - embedded within Executors
  - Rarely want LinkedBlockingQueue
    - i.e. more commonly use ArrayBlockingQueue
- removals can be blocking
- insertions can wake-up sleeping threads
- In IO we can hand the worker threads the socket

NonBlocking Data Structures

- ConcurrentLinkedQueue
  - concurrent linkedlist based on CAS
  - elegance is downright fun
  - No data corruption or blocking happens regardless of the number of threads adding or deleting or iterating
  - Note that iteration is “fuzzy”

NonBlocking Data Structures

- ConcurrentHashMap
  - Not quite as concurrent
  - non-blocking reads
  - stripe-locked writes
  - Can increase parallelism in constructor
- Cliff Click's NonBlockingHashMap
  - Fully non-blocking
  - Does surprisingly well with many writes
  - Also has NonBlockingLongHashMap (thanks Cliff!)

BufferedStreams

- BufferedOutputStream
  - simply creates an 8k buffer
  - requires flushing
  - can immensely improve performance if you keep sending small sets of bytes
  - The native call to do a send of a byte wraps it in an array and then sends that
  - Look at BufferedStreams like adding a memory copy between your code and the send

BufferedStreams

- BufferedOutputStream
  - If your sends are broken up, use a buffered stream
  - if you already package your whole message into a byte array, don't buffer
    - i.e., you already did
  - Default buffer in BufferedOutputStream is 8k – what if your message is bigger?

Bytes

- Java programmers seem to have a fascination with Strings
  - fair enough, they are a nice human abstraction
- Can you keep your server's data solely in bytes?
- Object allocation is cheap but a few million objects are probably measurable
- String manipulation is cumbersome internally
- If you can stay with byte arrays, it won't hurt
- Careful with autoboxing too

Papers
(indirect references)

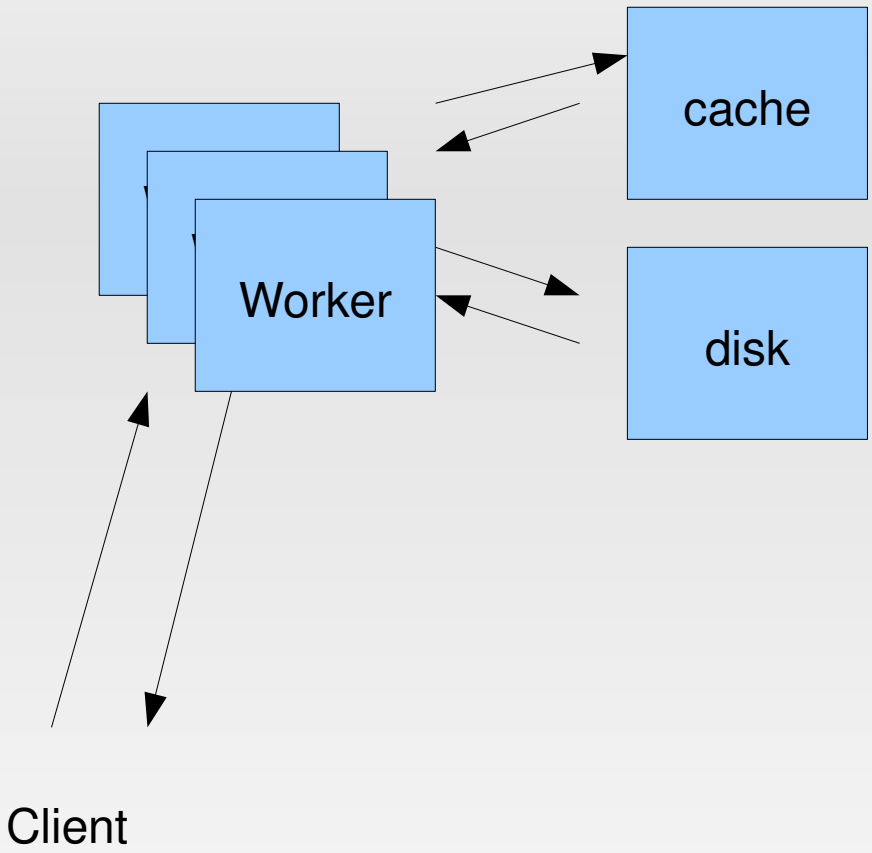
- “Why Events are a Bad Idea (For high-concurrency servers)”, Rob von Behren
- Dan Kegel's C10K problem
  - somewhat dated now but still great depth

Designing Servers

A static Web Server

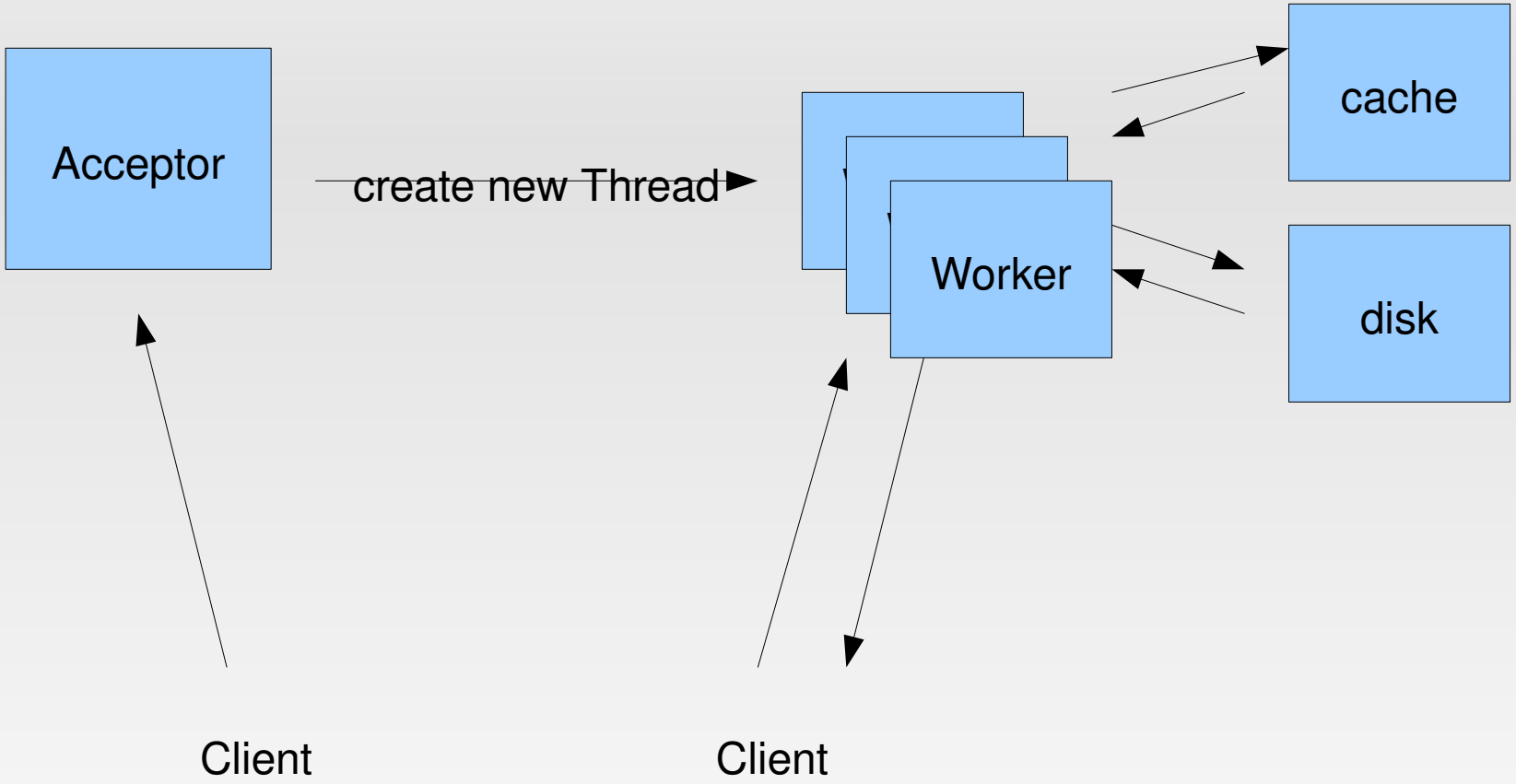
- HTTP is stateless
  - very nice, one request, one response
  - Many clients, many, short connections
  - Interestingly for normal GET operations (ajax too) we only block as they call in. After that, they are blocked waiting for our reply
  - Threads must cooperate on cache

Thread per request



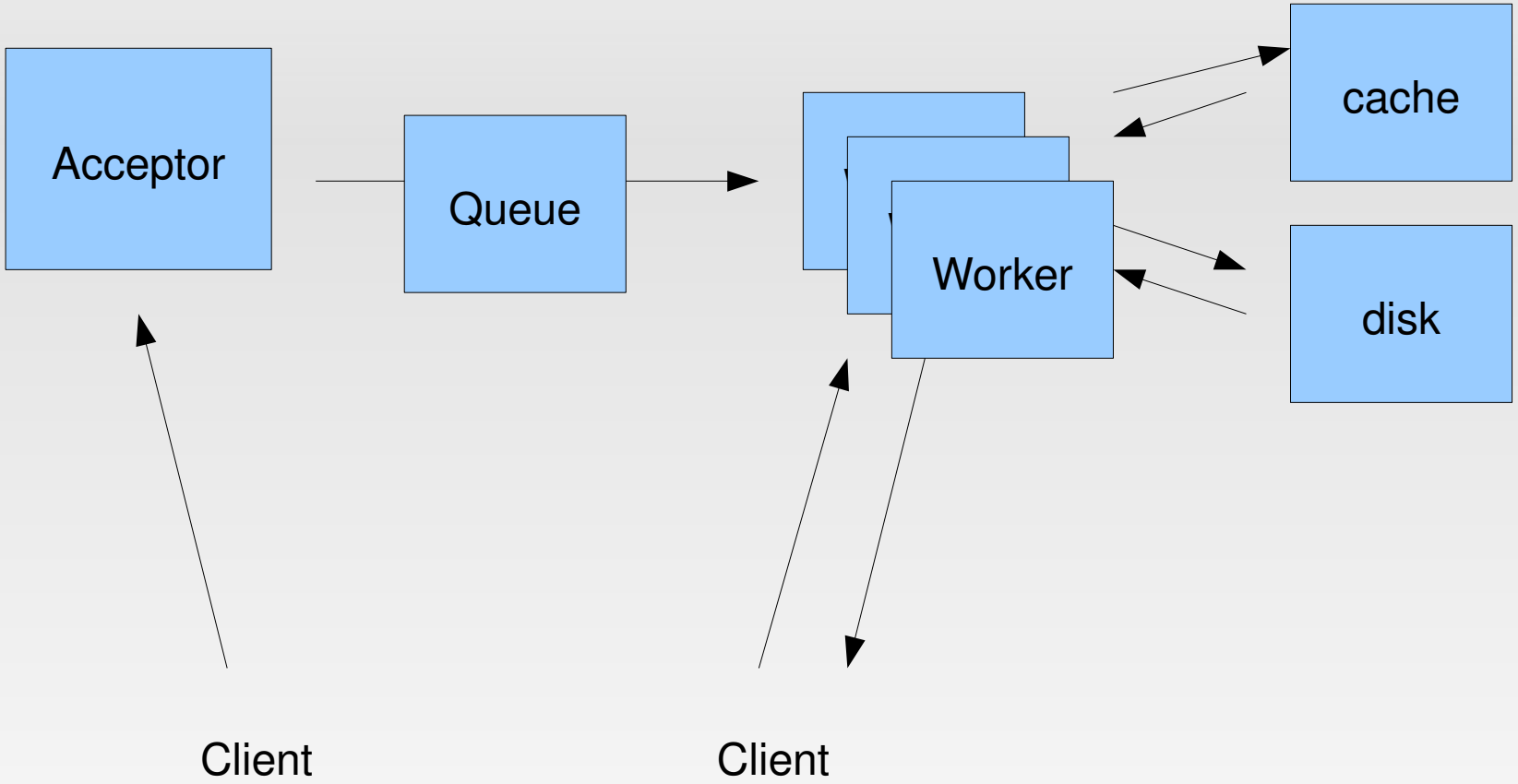
- Create a fixed number of threads in the beginning
  - (not in a thread pool)
- All threads block at accept(), then go process the client
- Watch socket timeouts
- Catch all exceptions

Thread per request



- Thread creation cost is historically expensive
- Load tempered by keeping track of the number of threads alive
  - not terribly precise

Thread per request



- Executors provide many knobs for thread tuning
- Handle dead threads
- Queue size can indicate load (to a degree)
- Acceptor threads can help out easily

A static Web Server

- Given HTTP is request-reply
  - The longer transactions take, the more threads you need (because more are waiting on the backend)
  - For a smartly cached static webserver, 2 core machine, 15-20 threads saturated the system
  - Add a database backend or other complex processing per transaction and number of threads linearly increases

SMTP Server

a much more chatty protocol

An SMTP server

- Very conversational protocol
- Server spends a **lot** of time waiting for the next command (like many milliseconds)
- Many threads simply asleep
- Few hundred threads very common
- Few thousand not uncommon
  - (JVM max allow 32k)

An SMTP server

- All threads must cooperate on storage of messages
  - (keep in mind the concurrent data structures!)
- Possible to saturate the bandwidth after a few thousand threads
  - or the disk

Coding

- Multithreaded server coding is more intuitive
  - You simply follow the flow of whats going to happen to one client
- Non-blocking data structures are very fast
- Immutable data is fast
- Sticking with bytes-in, bytes-out is nice
  - might need more utility methods

Questions?