Do you wish you could hear the audio and read the transcription of this session?

Then come to JavaOne℠ Online where this session is available in a multimedia tool with full audio and transcription synced with the slide presentation.

JavaOne Online offers much more than just multimedia sessions. Here are just a few benefits:

- 2003 and 2002 Multimedia JavaOne conference sessions
- Monthly webinars with industry luminaries
- Exclusive web-only multimedia sessions on Java technology
- Birds-of-a-Feather sessions online
- Classified Ads: Find a new job, view upcoming events, buy or sell cool stuff and much more!
- Feature articles on industry leaders, Q&A with speakers, etc.

For only $99.95, you can become a member of JavaOne Online for one year. Join today!

Visit http://java.sun.com/javaone/online for more details!
How to Write a Scalable Server

Mark Reinhold
Senior Staff Engineer
Sun Microsystems, Inc.
Outline

- Measuring Server Performance
- Server Architectures
  - From simple to complex
- Conclusion
  - Tuning
  - NIO Bugs Discovered
  - Lessons
- References
Measuring Server Performance
Measuring Server Performance

... is really really hard

- Real web servers do lots of stuff
  - CGI, JSP, SSL, database access, authentication, logging, session tracking, caching, ...
  - Essentially a very large distributed system
  - Hard to duplicate production environment in lab
  - Hard to measure a production system without interfering with it (cf. Heisenberg)
Measuring Server Performance

... is really really hard
... so we resort to microbenchmarks

- Microbenchmarks can be deceiving
  - Measure extreme cases, not real systems
- Microbenchmarks are much more practical
  - Still useful for exploring a design space
  - This is the approach taken here
“Two tests in particular are simple, interesting, and hard.” – Dan Kegel

- How many requests/second can you serve?
  - Small files (1KB)
  - Connections/second vs. requests/second
  - Response-time distribution
- How many slow (56kb) clients can you serve?
  - Large files (1MB)
  - Bytes/second vs. number of clients
Test Environment

- 3 Sun E450s
  - 4x480 MHz UltraSparc® II processors
  - 8MB cache/processor, 4GB main memory
  - Solaris™ 9 Operating Environment + patches
  - Two client machines, one server

- Point-to-point gigabit ethernet
  - Peak bandwidth: 76MB/s
  - Latency: Difficult to measure

- No bottlenecks
  - Except for the server itself
Test Methodology

- Each sample point measured separately
- Restart client and server programs
- Warm up server
  - 1000 small requests/second for 30 seconds
- Run test for five minutes
  - Experimentation showed no significant difference with longer durations
- Java™ runtime environment
  - Latest "Mantis" (1.4.2) build
Baselines: Two Real Servers

- **thttpd 2.20c**
  - Jef Poskanzer @ Acme Laboratories
  - [http://www.acme.com/software/thttpd](http://www.acme.com/software/thttpd)
  - Small, fast, simple
  - Single-process, single-threaded, non-blocking

- **Apache 2.0.42**
  - Apache Software Foundation
  - [http://httpd.apache.org](http://httpd.apache.org)
  - Larger, more complex, but still pretty fast
  - Multi-process, multi-threaded
Small-File Test

- Test tool: httpperf
  - David Mosberger & Tai Jin @ HP Labs
  - Can generate and sustain overload
  - Can initiate requests at fixed or variable rates
  - Measures response-time distributions

- Test parameters
  - 1000 1KB files, requested randomly
  - 1000-2500 requests/second, fixed rate
Small-File Test: Apache

Fraction of requests completed

- 3749
- 3000
- 2000
- 1000

Response time (ms)

Requests/second

0.0
0.2
0.4
0.6
0.8
1.0

11
Large-File Test

- **Test tool:** http_load
  - Also by Jef Poskanzer @ Acme Laboratories
  - [http://www.acme.com/software/thttpd](http://www.acme.com/software/thttpd)
  - Can open connections at fixed rate or with fixed parallelism
  - Can throttle bandwidth to simulate slow clients

- **Test parameters**
  - 100 different 1MB files, requested randomly
  - 1000-10000 parallel connections
  - Each connection throttled to 56kb/s
Large-File Test: thttpd

![Graph showing throughput vs. parallel requests]

Throughput (MB/sec)

Parallel requests

Server RSS (MB)

- 776.4
- 600.0
- 400.0
- 200.0

Throughput:

- 46.5
- 40.0
- 30.0
- 20.0
- 10.0
- 5.3

Server RSS (MB):

- 7.8
- 5.3
How to Write a Scalable Server

Server Architectures
### Server Architectures

<table>
<thead>
<tr>
<th>Code</th>
<th>Mode</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Blocking</td>
<td>Single-Threaded</td>
</tr>
<tr>
<td>BN</td>
<td>Blocking</td>
<td>Multi-Threaded</td>
</tr>
<tr>
<td>BP</td>
<td>Blocking</td>
<td>Multi-Threaded, Pooled</td>
</tr>
<tr>
<td>N1</td>
<td>Non-Blocking</td>
<td>Single-Threaded</td>
</tr>
<tr>
<td>N2</td>
<td>Non-Blocking</td>
<td>Two Threads</td>
</tr>
<tr>
<td>N4</td>
<td>Non-Blocking</td>
<td>Four Threads</td>
</tr>
</tbody>
</table>
B1: Blocking, Single-Threaded
class B1 {

    private static int PORT = 8080;
    private static int BACKLOG = 1024;

    public static void main(String[] args)
        throws IOException
    {
        ServerSocketChannel ssc
            = ServerSocketChannel.open();
        ssc.socket().setReuseAddress(true);
        ssc.socket().bind(new InetSocketAddress(PORT),
                         BACKLOG);

        for (;;)
            service(ssc);

    }
}
class B1 {
    private static int PORT = 8080;
    private static int BACKLOG = 1024;

    public static void main(String[] args)
    throws IOException
    {
        ServerSocketChannel ssc
            = ServerSocketChannel.open();
        ssc.socket().setReuseAddress(true);
        ssc.socket().bind(new InetSocketAddress(PORT),
            BACKLOG);

        for (;;)
            service(ssc);
    }
}
class B1 {
    ...
    public static void main(String[] args) {
        static void service(ServerSocketChannel ssc) throws IOException {
            SocketChannel sc = ssc.accept(); // Accept
            ByteBuffer rbb = receive(sc); // Receive
            Request rq = Request.parse(rbb); // Parse
            Reply rp = build(rq); // Build
            rp.send(sc); // Send
            sc.close(); // Close
        }
    }
}
B1: Blocking, Single-Threaded
B1: Blocking, Single-Threaded

Problem: Cannot handle more than one connection at a time

Solution: Create a new thread for each connection

→ BN: Blocking, Multi-Threaded
BN: Blocking, Multi-Threaded
class B1 {
    // Accept, receive, parse, build, send, close
    static void service(ServerSocketChannel ssc) {
        public static void main(String[] args) {
            ServerSocketChannel ssc = ... 
            for (; ;) 
                service(ssc); 
        }

        static void service(ServerSocketChannel ssc) {
            SocketChannel sc = ssc.accept();  // Accept
            ByteBuffer rbb = receive(sc);    // Receive
    
            ... 
        }
    }
}
BN: Blocking, Multi-Threaded

class BN {
    public static void main(String[] args) {
        ServerSocketChannel ssc = ...;
        for (;;) {
            SocketChannel sc = ssc.accept();
            new Thread(new Servicer(sc)).start();
        }
    }
}

class Servicer implements Runnable {
    Servicer(SocketChannel sc);
    void run(); // Receive, parse, build, send, close
}
Problem: Creating threads is expensive
Solution: Pool threads for re-use

BP: Blocking, Multi-Threaded, Pooled
BP: Blocking, Pooled
class BN {
    ...
    public static void main(String[] args) {
        ServerSocketChannel ssc = ...
        for (; ;) {
            SocketChannel sc = ssc.accept();
            new Thread(new Servicer(sc)).start();
        }
    }
}
class BP {
    ...
    public static void main(String[] args) {
        ServerSocketChannel ssc = ...
        Executor xec = new PooledExecutor();
        for (;;) {
            SocketChannel sc = ssc.accept();
            xec.execute(new Servicer(sc));
        }
    }
}
BP: Blocking, Pooled

Fraction of requests completed

Requests/second

Response time (ms)

- 3749
- 3000
- 2000
- 1000
1
BP: Blocking, Pooled

Throughput (MB/sec)

Parallel requests

Server RSS (MB)
Problem: Threads are expensive, even when pooled

Solution: Use non-blocking I/O operations

N1: Non-Blocking, Single-Threaded
N1: Non-Blocking, Single-Threaded
Digression
How to Use Selectors
How to Use Selectors

Selectable channel = A channel that can be multiplexed

Selector = A multiplexor of selectable channels

Selection key = A token representing the registration of a selectable channel with a selector
ByteBuffer bb = ByteBuffer.allocate(4096);
SocketChannel sc = SocketChannel.open(addr);

// Blocking read
return sc.read(bb);
ByteBuffer bb = ByteBuffer.allocate(4096);
SocketChannel sc = SocketChannel.open(addr);

// Blocking read
return sc.read(bb);

// Blocking read built from non-blocking ops
Selector sel = Selector.open();
sc.configureBlocking(false);
SelectionKey sk = sc.register(sel, SelectionKey.OP_READ);
sel.select();
return sc.read(bb);
Selectors: Hints Can Be Wrong

ByteBuffer bb = ByteBuffer.allocate(4096);
SocketChannel sc = SocketChannel.open(addr);

// Blocking read
return sc.read(bb);

// Blocking read built from non-blocking ops
Selector sel = Selector.open();
sc.configureBlocking(false);
SelectionKey sk = sc.register(sel, SelectionKey.OP_READ);
for (;;) {
    sel.select();
    int n = sc.read(bb);
    if (n > 0) return n;
}
SelectionKey sk1
    = scl.register(sel, SelectionKey.OP_READ);
SelectionKey sk2
    = sc2.register(sel, SelectionKey.OP_READ);
for (;;) {
    sel.select();
    if (sk1.readyOps() & SelectionKey.OP_READ != 0) {
        int n = scl.read(bb);
        if (n > 0) return n;
    }
    if (sk2.readyOps() & SelectionKey.OP_READ != 0) {
        int n = sc2.read(bb);
        if (n > 0) return n;
    }
}
sc1.register(sel, SelectionKey.OP_READ);
sc2.register(sel, SelectionKey.OP_READ);
for (; ;) {
    sel.select();
    for ( Iterator i = sel.selectedKeys().iterator();
        i.hasNext(); ) {
        SelectionKey sk = (SelectionKey)i.next();
        i.remove();
        SocketChannel sc = (SocketChannel)sk.channel();
        int n = sc.read(bb);
        if (n > 0) return n;
    }
}
sc1.register(sel, SelectionKey.OP_READ, ByteBuffer.allocate(4096));
sc2.register(sel, SelectionKey.OP_READ, ByteBuffer.allocate(4096));

for (;;) {
    sel.select();
    for (Iterator i = sel.selectedKeys().iterator();
         i.hasNext();)
        {
        SelectionKey sk = (SelectionKey)i.next();
        i.remove();
        SocketChannel sc = (SocketChannel)sk.channel();
        ByteBuffer bb = (ByteBuffer)sk.attachment();
        int n = sc.read(bb);
        if (n > 0) return sk;
    }
}
interface Handler {
    void handle(SelectionKey sk);
}

class DataHandler implements Handler {
    DataHandler(SocketChannel sc);
    void handle(SelectionKey sk);
}

class LogHandler implements Handler {
    LogHandler(SocketChannel sc);
    void handle(SelectionKey sk);
}
Selectors: Handler Attachments

SelectionKey sk1
   = scl.register(sel, SelectionKey.OP_READ,
      new DataHandler(scl));
SelectionKey sk1
   = sc2.register(sel, SelectionKey.OP_WRITE,
      new LogHandler(sc2));

for (;;) {
   sel.select();
   for ( Iterator i = sel.selectedKeys().iterator();
      i.hasNext(); ) {
      SelectionKey sk = (SelectionKey)i.next();
      i.remove();
      Handler h = (Handler)sk.attachment();
      h.handle(sk);
   }
N1: Non-Blocking, Single-Threaded
sel.select();
for ( Iterator i = sel.selectedKeys().iterator();
     i.hasNext();)
{
    SelectionKey sk = (SelectionKey)i.next();
    i.remove();
    Handler h = (Handler)sk.attachment();
    h.handle(sk);
}
class Dispatcher {
    private Selector sel;
    void dispatch() {
        sel.select();
        for (Iterator i = sel.selectedKeys().iterator();
             i.hasNext();)
        {
            SelectionKey sk = (SelectionKey)i.next();
            i.remove();
            Handler h = (Handler)sk.attachment();
            h.handle(sk);
        }
    }
}
class Dispatcher {
    private Selector sel;
    void dispatch() {
        sel.select();
        ...
    }
    Dispatcher() {
        sel = Selector.open();
    }
}
class Dispatcher {
    private Selector sel;
    void dispatch() {
        sel.select();
        ...
    }
    Dispatcher() {
        sel = Selector.open();
    }
    void register(SelectableChannel ch, int ops, Handler h) {
        ch.register(sel, ops, h);
    }
}
class N1 {
    public static void main(String[] args) {
        ServerSocketChannel ssc
            = ServerSocketChannel.open();
        ...
        ssc.configureBlocking(false);
        Dispatcher d = new Dispatcher();
        d.register(ssc, SelectionKey.OP_ACCEPT,
            new AcceptHandler(ssc, d));
        for (; ;)
            d.dispatch();
    }
}
class AcceptHandler implements Handler {
    AcceptHandler(ServerSocketChannel ssc, Dispatcher d);
    void handle(SelectionKey sk);
}
```java
class AcceptHandler implements Handler {
    private ServerSocketChannel ssc;
    private Dispatcher d;
    AcceptHandler(ServerSocketChannel ssc, Dispatcher d)
    {
        this.ssc = ssc;
        this.d = d;
    }
    void handle(SelectionKey sk);
}
```
class AcceptHandler implements Handler {
    private ServerSocketChannel ssc;
    private Dispatcher d;
    ...
    void handle(SelectionKey sk) {
        if (sk.readyOps() & SelectionKey.OP_ACCEPT == 0)
            return;
        SocketChannel sc = ssc.accept();
        if (sc == null)
            return;
        sc.configureBlocking(false);
        d.register(sc, SelectionKey.OP_READ,
                  new RequestHandler(sc));
    }
}
class RequestHandler implements Handler {
    RequestHandler(SocketChannel sc);
    void handle(SelectionKey sk);
}
class RequestHandler implements Handler {

    private SocketChannel sc;
    RequestHandler(SocketChannel sc) {
        this.sc = sc;
    }
    void handle(SelectionKey sk);
}
class RequestHandler implements Handler {

    ... 

    private ByteBuffer rbb = ByteBuffer.allocate(4096);
    private Request request = null;
    private Reply reply = null;
    void handle(SelectionKey sk);

}
class RequestHandler implements Handler {

    ...

    private ByteBuffer rbb = ByteBuffer.allocate(4096);
    private Request request = null;
    private Reply reply = null;

    // Returns true when request is complete
    private boolean receive();

    // If okay, saves request and returns true
    private boolean parse();

    // Builds reply for request
    private void build();

    void handle(SelectionKey sk);
}
void handle(SelectionKey sk) {
    if (request == null) {
        if (!receive())
            return;
        rbb.flip();
        if (parse())
            build();
        sk.interestOps(SelectionKey.OP_WRITE);
    } else {
        if (reply.send(sc))
            sc.close();
    }
}
void handle(SelectionKey sk) {
    if (request == null) {
        if (!receive())
            return;
        rbb.flip();
        if (parse())
            build();
        if (!reply.send(sc))
            sk.interestOps(SelectionKey.OP_WRITE);
        else
            sc.close();
    } else {
        if (reply.send(sc))
            sc.close();
    }
}
void handle(SelectionKey sk) {
    try {
        if (request == null) {
            if (!receive())
                return;
            ...
        } else {
            if (reply.send(sc))
                sc.close();
        }
    } catch (IOException x) {
        sc.close();
    }
}
N1.main: Dispatcher d = new Dispatcher();
d.register(ssc, SelectionKey.OP_ACCEPT,
        new AcceptHandler(ssc, d));
for (;;)  
    d.dispatch();

AcceptHandler: SocketChannel sc = ssc.accept();
sc.configureBlocking(false);
d.register(sc, SelectionKey.OP_READ,
        new RequestHandler(sc));

RequestHandler: if (request == null) {
    if (!receive()) return;
    if (parse()) build();
    sk.interestOps(SelectionKey.OP_WRITE);
} else {
    if (reply.send(sc)) sc.close();
}
N1: Non-Blocking, Single-Threaded
N1: Non-Blocking, Single-Threaded
N1: Non-Blocking, Single-Threaded

Problem: Performance very poor
Solution: Use more threads

N2: Non-Blocking, Two Threads
N2: Non-Blocking, Two Threads
class Acceptor implements Runnable {
    Acceptor(ServerSocketChannel ssc, Dispatcher d);
    public void run();
}
N2: Non-Blocking, Two Threads

class Acceptor implements Runnable {
    private ServerSocketChannel ssc;
    private Dispatcher d;
    Acceptor(ServerSocketChannel ssc, Dispatcher d) {
        this.ssc = ssc;
        this.d = d;
    }
    public void run();
}
class Acceptor implements Runnable {
    private ServerSocketChannel ssc;
    private Dispatcher d;
    ...
    public void run() {
        for (;;) {
            SocketChannel sc = ssc.accept();
            sc.configureBlocking(false);
            d.register(sc, SelectionKey.OP_READ,
                        new RequestHandler(sc));
        }
    }
}
N2: Non-Blocking, Two Threads

class Acceptor implements Runnable {
    ...
    public void run() {
        for (;;) {
            try {
                SocketChannel sc = ssc.accept();
                sc.configureBlocking(false);
                d.register(sc, SelectionKey.OP_READ,
                    new RequestHandler(sc));
            } catch (IOException x) {
                x.printStackTrace();
                break;
            }
        }
    }
}
class N1 {
    public static void main(String[] args) {
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ...
        ssc.configureBlocking(false);
        Dispatcher d = new Dispatcher();
        d.register(ssc, SelectionKey.OP_ACCEPT,
                new AcceptHandler(ssc, d));

        for (; ;) {
            d.dispatch();
        }
    }
}
class N2 {
    public static void main(String[] args) {
        ServerSocketChannel ssc
            = ServerSocketChannel.open();
        ...
        // DO NOT: ssc.configureBlocking(false);
        Dispatcher d = new Dispatcher();
        Acceptor a = new Acceptor(ssc, d);
        new Thread(a).start();
        for (;;) {
            d.dispatch();
        }
    }
}
class Dispatcher {
    void dispatch() {
        sel.select();
        for (Iterator i = ...) {
            ...
        }
    }
    void register(SelectableChannel ch, int ops, Handler h) {
        ch.register(sel, ops, h);
    }
}
class Dispatcher {
    void dispatch() {
        sel.select();
        for (Iterator i = ...) { ... }
    }
    void register(SelectableChannel ch, int ops, Handler h) {
        sel.wakeup();
        ch.register(sel, ops, h);
    }
}
class Dispatcher {

    private Object gate = new Object();

    void dispatch() {
        sel.select();
        for (Iterator i = ...) {
            ... 
        }
        synchronized (gate) {
        }
    }

    void register(SelectableChannel ch, int ops, Handler h) {
        synchronized (gate) {
            sel.wakeup();
            ch.register(sel, ops, h);
        }
    }
}

N2: Non-Blocking, Two Threads
N2: Non-Blocking, Two Threads

Throughput (MB/sec)

Parallel requests

Server RSS (MB)
N2: Non-Blocking, Two Threads

Problem: Small-file performance better, but large-file performance still pretty poor

Solution: Use even more threads

→ N4: Non-Blocking, *Four* Threads
N4: Non-Blocking, Four Threads
class N2 {
    public static void main(String[] args) {
        ServerSocketChannel ssc = ...
        Dispatcher d = new Dispatcher();
        Acceptor a = new Acceptor(ssc, d);
        new Thread(a).start();
        for (;;)
            d.dispatch();
    }
}
class N4 {
    public static void main(String[] args) {
        ServerSocketChannel ssc = ...;
        Dispatcher d1 = new Dispatcher();
        Dispatcher d2 = new Dispatcher();
        Acceptor a1 = new Acceptor(ssc, d1);
        Acceptor a2 = new Acceptor(ssc, d2);
        new Thread(d1).start();
        new Thread(d2).start();
        new Thread(a1).start();
        new Thread(a2).start();
    }
}
class Dispatcher {
    void dispatch();
    void register(SelectableChannel ch, int ops, Handler h);
}
class Dispatcher implements Runnable {
    void dispatch();
    void register(SelectableChannel ch, int ops, Handler h);

    public void run() {
        for (;;) {
            try {
                dispatch();
            } catch (IOException x) {
                x.printStackTrace();
                break;
            }
        }
        ...
    }
}
N4: Non-Blocking, Four Threads

Fraction of requests completed

Requests/second

Response time (ms)

- 3749
- 3000
- 2000
- 1000
1
N4: Non-Blocking, Four Threads

Throughput (MB/sec) vs. Parallel requests
### Server Architectures

<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Non-Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Single-Threaded</td>
<td>Two Threads</td>
</tr>
<tr>
<td>BN</td>
<td>Multi-Threaded</td>
<td>Four Threads</td>
</tr>
<tr>
<td>BP</td>
<td>Multi-Threaded, Pooled</td>
<td></td>
</tr>
<tr>
<td>N1</td>
<td>Single-Threaded</td>
<td></td>
</tr>
<tr>
<td>N2</td>
<td>Two Threads</td>
<td></td>
</tr>
<tr>
<td>N4</td>
<td>Four Threads</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion
Tuning: Solaris

- **Time wait interval**  
  $ ndd -set /dev/tcp tcp_time_wait_interval 2000$

- **Connection table size**  
  /etc/system: set tcp:tcp_conn_hash_size=32768

- **File descriptor limits**  
  /etc/system: set rlim_fd_cur=65536
  set rlim_fd_max=65536
Tuning: Linux

- **Local port range**: 1024–64000
  
  ```
  $ echo 1024 64000 \
  >/proc/sys/net/ipv4/ip_local_port_range
  ```

- **File descriptor limits**: 65536
  
  ```
  /etc/security/limits.conf:
  * soft nofile 1024
  * hard nofile 65535
  $ echo 65535 >/proc/sys/fs/file-max
  $ ulimit -n unlimited
  ```
Tuning: VM

- Heap size
  -Xms512M -Xmx512M

- New ratio
  -XX:NewRatio=1
NIO Bugs Discovered

- New `FileChannel.transferTo` code broken
  - Only for some corner cases
  - Solaris docs are incomplete
- `/dev/poll` Selector code limits performance
  - Returns at most 20 events (!)
- Channel finalization is expensive
  - Removing it improves GC performance by 100x
  - Necessary to achieve final performance gains

These problems are fixed in 1.4.2
Lessons

- TCP is complicated
- VMs are complicated
- Non-blocking I/O is complicated
  - You don’t always need it
- Finalization is evil
- Small multiprocessors are good
- NIO works pretty well
- Experiment!
References: Books

- *Unix Network Programming*

- *Sun Performance and Tuning*
  Adrian Cockroft, Prentice Hall, 1998

- *JDK 1.4 Tutorial*
  Gregory M. Travis, Manning, 2002

- *Java NIO*
  Ron Hitchens, O’Reilly, 2002
References: Links

- *The C10K Problem*, Dan Kegel
  [http://kegel.com/c10k.html](http://kegel.com/c10k.html)

- *util.concurrent*, Doug Lea
  [http://gee.cs.oswego.edu/dl](http://gee.cs.oswego.edu/dl)

- Java HotSpot™ VM tuning pages

- Additional documentation & source code
  [http://java.sun.com/j2se/1.4/nio](http://java.sun.com/j2se/1.4/nio)

- Send us feedback!
  [java-io@java.sun.com](mailto:java-io@java.sun.com)
References: Additional Showtimes

- J2SE “Ask the Experts” booth
  - Thursday, noon–3:00pm

- BOFs: Thursday, Argent, Metropolitan III
  - New I/O: Scanning and Formatting (6:30pm)
  - New I/O: General Q&A (7:30pm)