SPB1217LV

vmware EXPLORE

SpringOne

Preparing Spring Web Applications for Loom

Mark Thomas (he/him)

Staff Engineer, VMware

Introductions

Tomcat since 2003
Committer, PMC member

Commons (Daemon, Pool, DBCP, BCEL)
Committer, PMC member

ASF member , ASF security team, ASF infrastructure team, Director 2016 to 2019
VP, Brand Management since 2018

Java EE Expert groups for Servlet, WebSocket, Expression Language

Jakarta Servlet, Pages, WebSocket and Expression Language
Committer

“Project Loom is to intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.”

https://wiki.openjdk.org/display/loom

A Brief History of Servlet Scalability

A Brief History of Servlet Scalability

HTTP/1.0

HTTP/1.1 and keep-alive

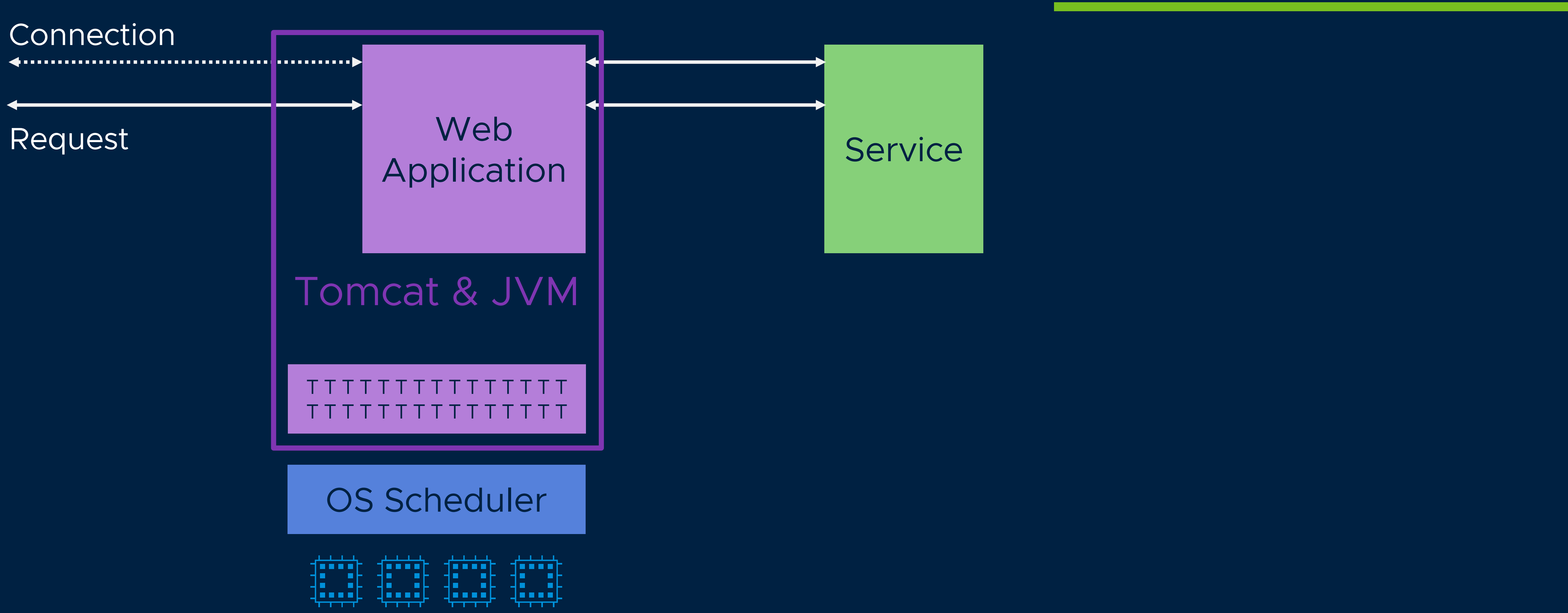
Tomcat, blocking I/O (BIO, 3.x) and thread starvation

Tomcat, non-blocking I/O (NIO, 6.0.x / NIO2, 8.0.x)

Servlet asynchronous API and non-blocking I/O (7.0.x)

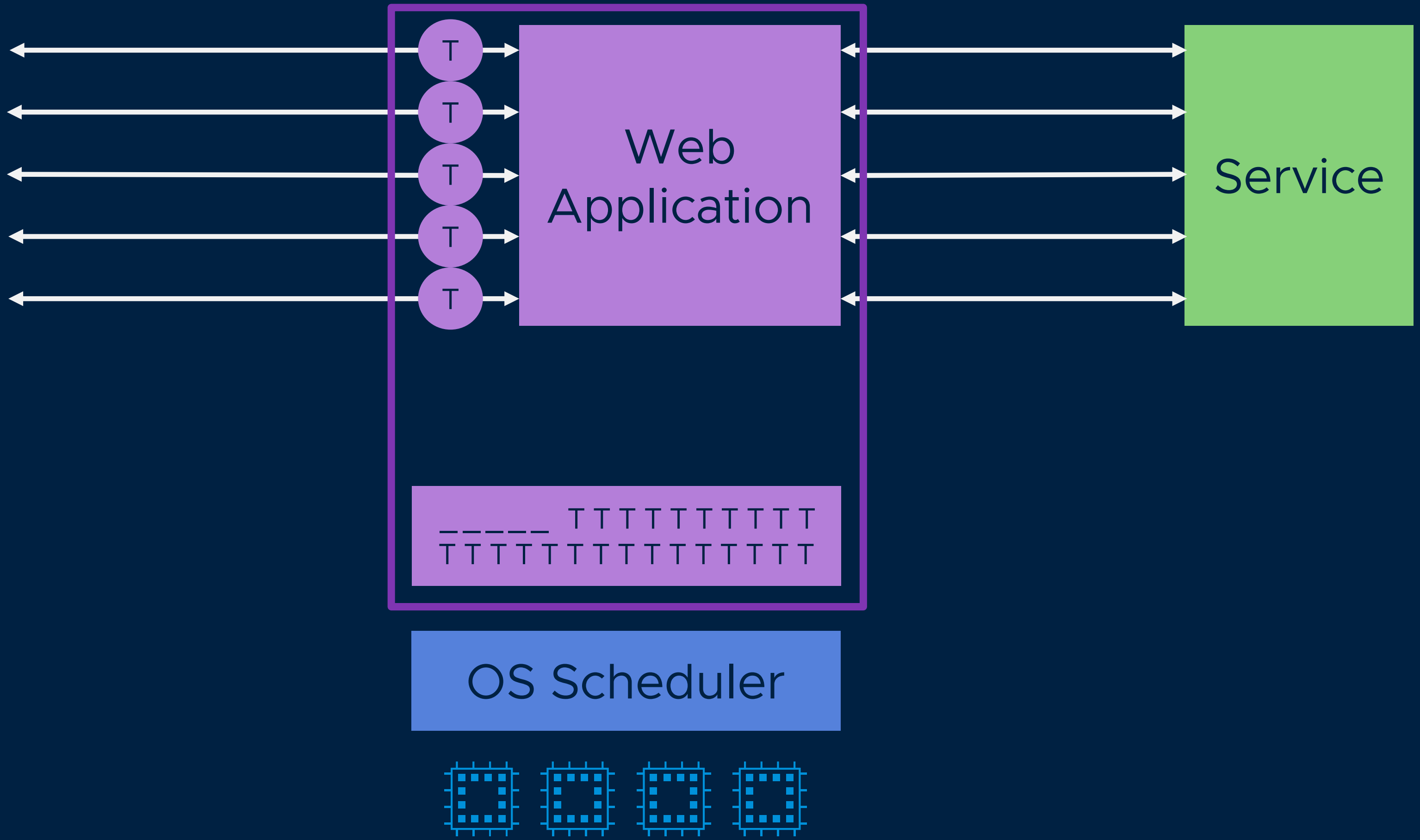
A Brief History of Servlet Scalability

Key



A Brief History of Servlet Scalability

HTTP/1.0



Connect, make request, close

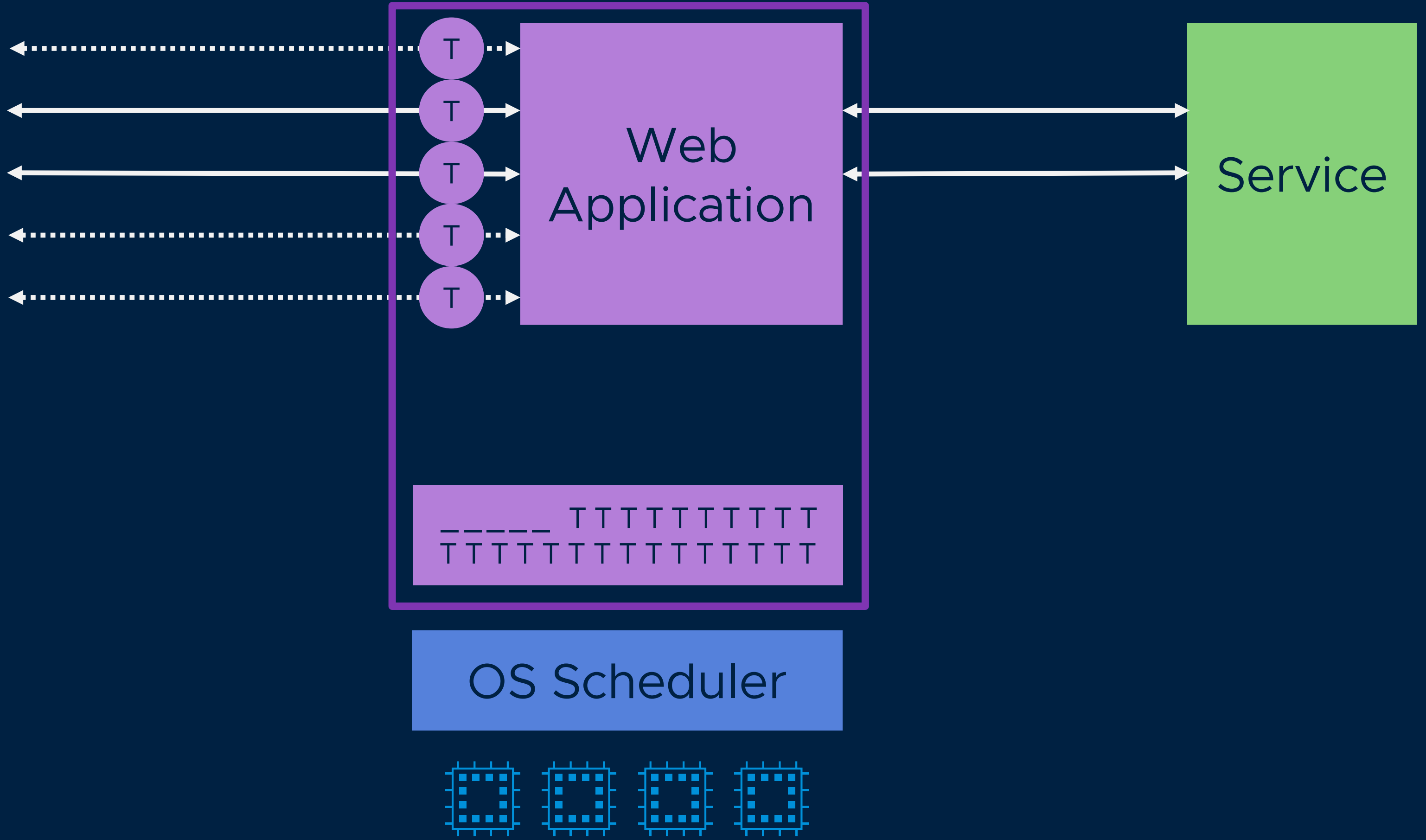
One thread per connection

Maximum connections
==
Maximum concurrent requests
==
Thread pool size

Creating connections is (relatively) expensive

A Brief History of Servlet Scalability

HTTP/1.1 keep-alive



HTTP/1.0 had keep-alive with issues with interoperability

HTTP/1.1 fixed the issues

Better (lower) latency

Worse scalability

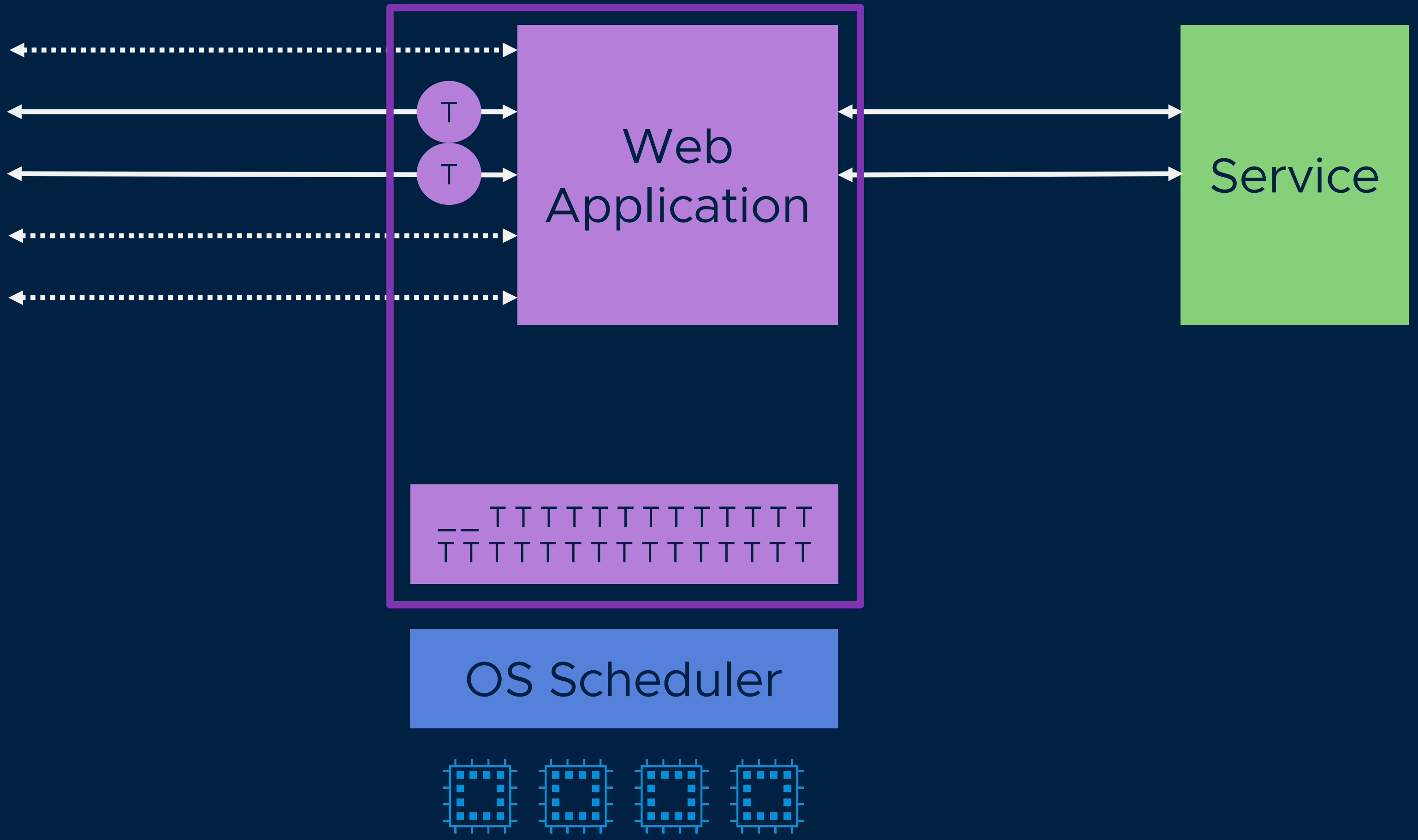
Typically uses more threads than there are concurrent requests

Thread starvation

Tomcat BIO connector disabled HTTP keep-alive for the last 25% of threads in the thread pool

A Brief History of Servlet Scalability

Non-blocking I/O part 1 – between requests



Tomcat NIO / NIO2 connectors

Use non-blocking I/O while waiting for a new request

Only use a thread for connections where there is a request to be processed

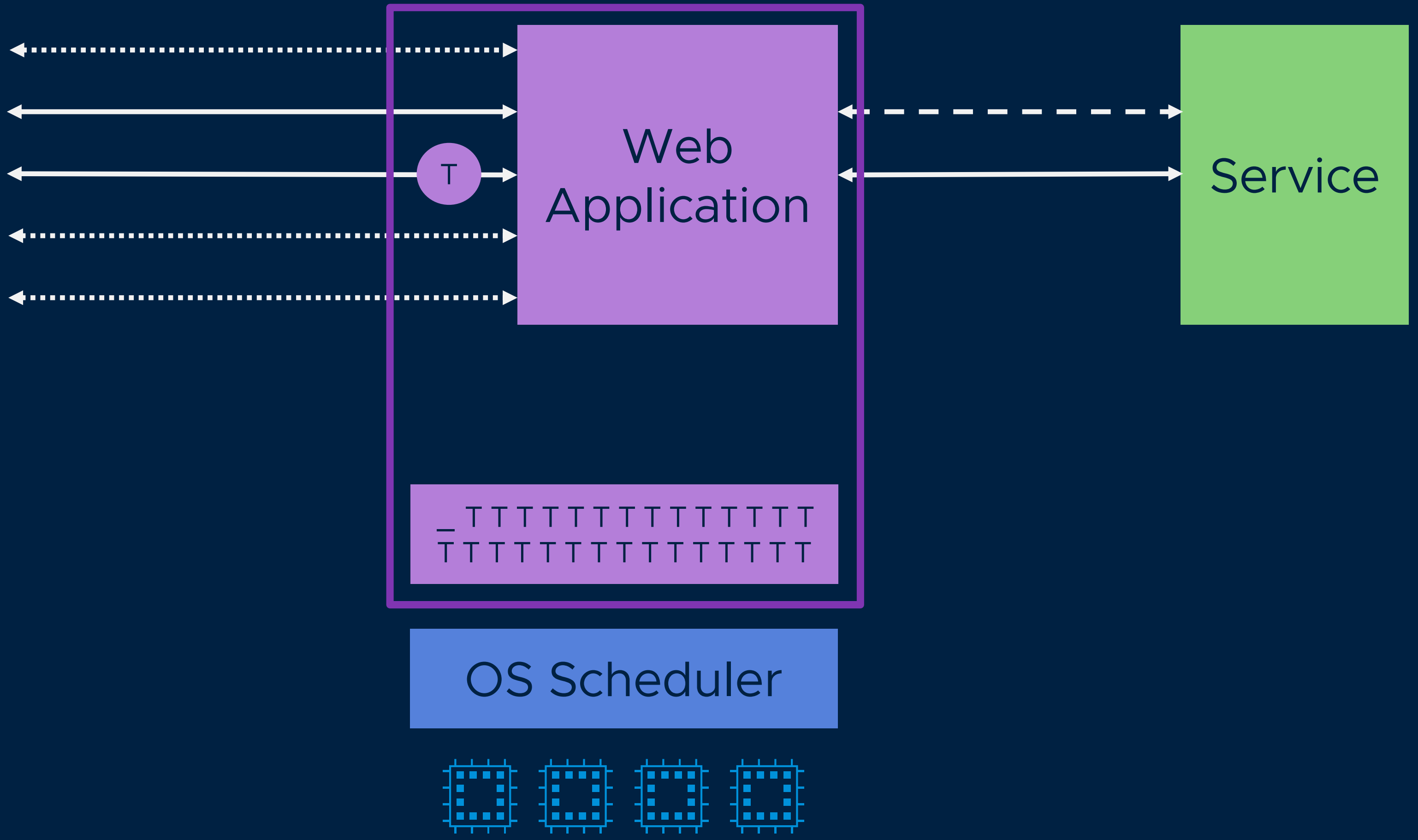
Maximum connections
>>
Maximum concurrent requests
==
Thread pool size

HTTP keep-alive latency benefits

Improved scalability

A Brief History of Servlet Scalability

Non-blocking I/O part 2 – Servlet asynchronous API


T
Web Application
Service
_ T T T T T T T T T T T T T T
T T T T T T T T T T T T T T T T
OS Scheduler

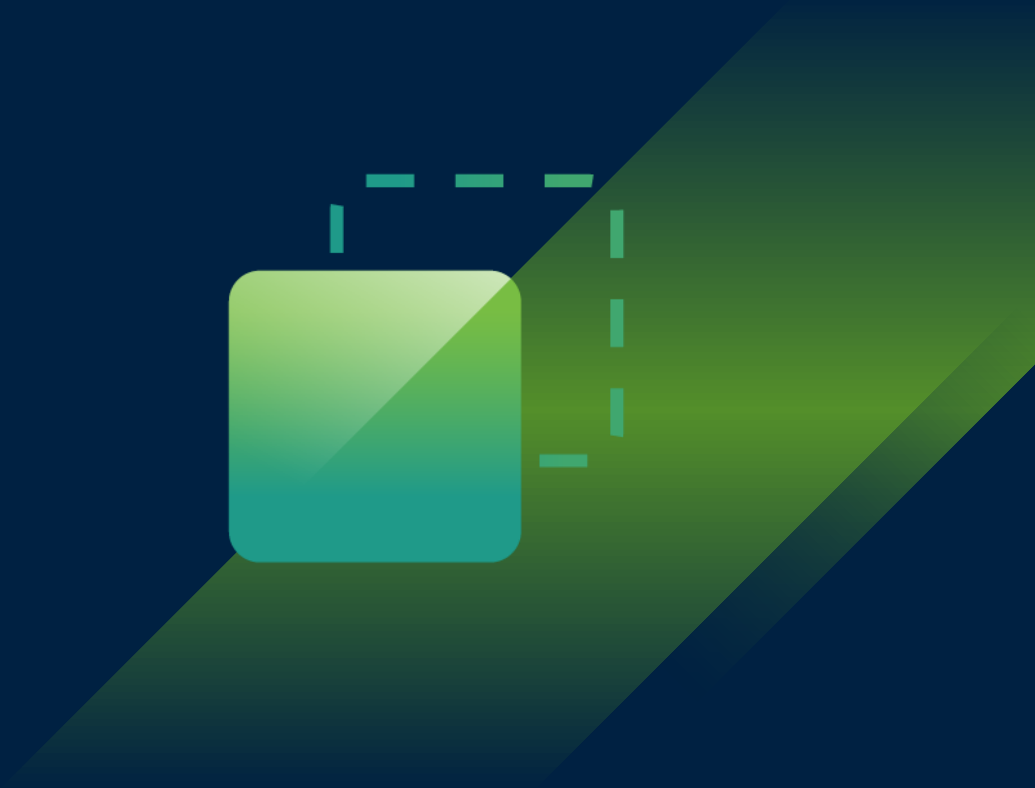
Use non-blocking I/O to communicate with services

Only use a thread for connections where there is a request actively being processed

Maximum connections
>>
Maximum concurrent requests
>
Thread pool size

Further improved scalability

Virtual Threads



Virtual Threads

Pre-Java 21 threads referred to as platform threads

Virtual threads

- Not mapped to a dedicated OS thread
- Use the heap for stack
- Created for a task and then allowed to terminate
- Do not pool virtual threads
- Have their own scheduler

Virtual thread scheduler has a pool of platform threads to do the work

- One platform thread per processor by default

Virtual Threads

Blocking operations

Platform threads

- Thread waits for operation to complete

Virtual threads

- Non-blocking operation started
- Virtual thread suspended and platform thread released
- Operations completes
- Virtual thread resumed and becomes eligible to be scheduled
- Execution continues

Virtual threads are effectively non-blocking for many blocking operations

- Increased scalability for “free”

Virtual Threads

Coding constraints

Beware of pinning

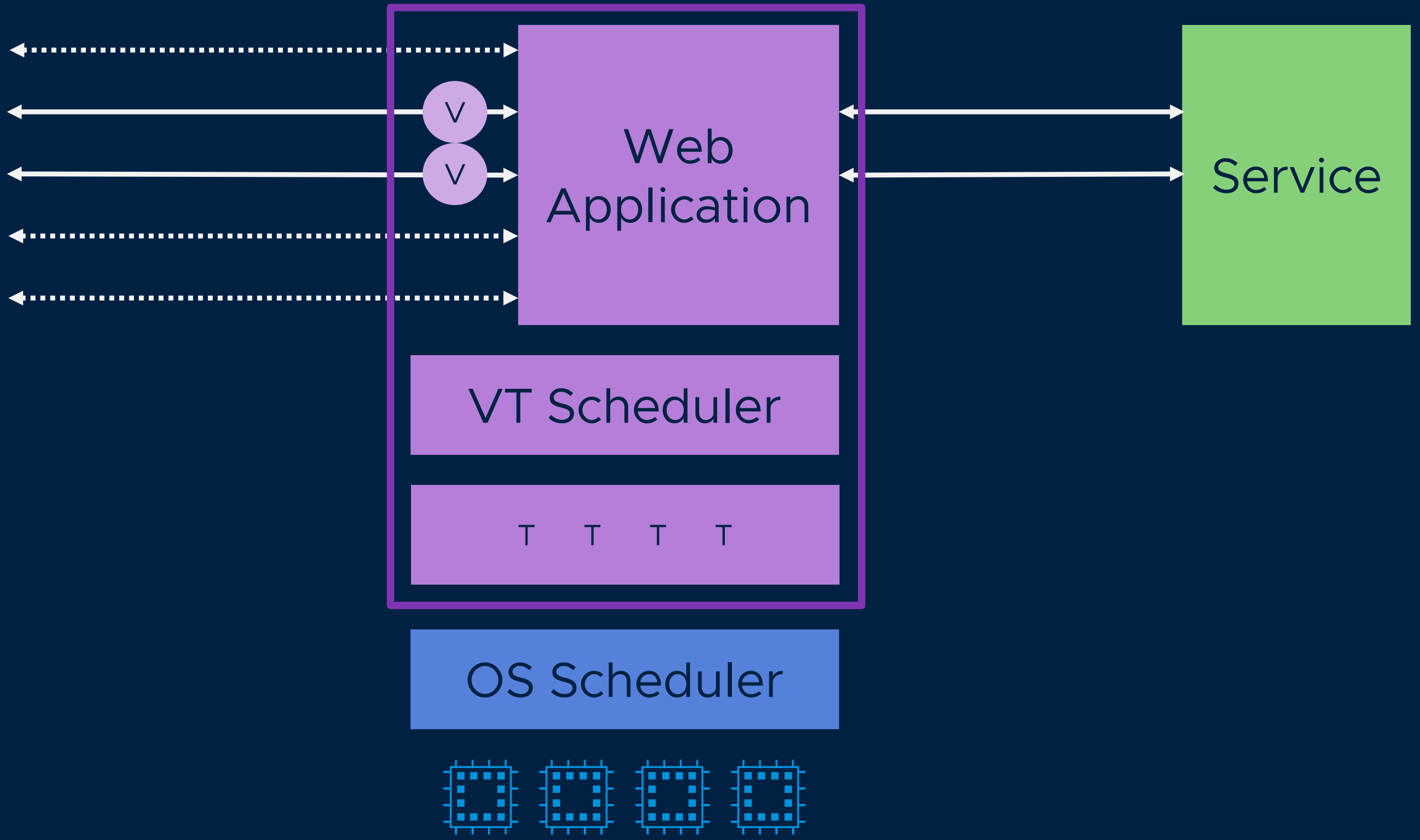
- Long lasting blocking operations are problematic
- Brief synchronized blocks are fine

ThreadLocals

- Providing context across an API boundary OK
- Caching could be problematic

A Brief History of Servlet Scalability

Virtual threads



- Impact on throughput?
- Impact on scalability?
- Impact on GC?
- Impact on memory footprint?
- Impact of extra scheduler?
- Impact on code complexity?
- Impact of constraints?

Investigations

Lots of areas to explore

Areas are not independent

Try and focus on a single variable

Performance tests only ever indicative

Not meant to be representative of real applications

Java 21 is still in Early Access

This work is just a starting point

Throughput

Throughput

Aims:

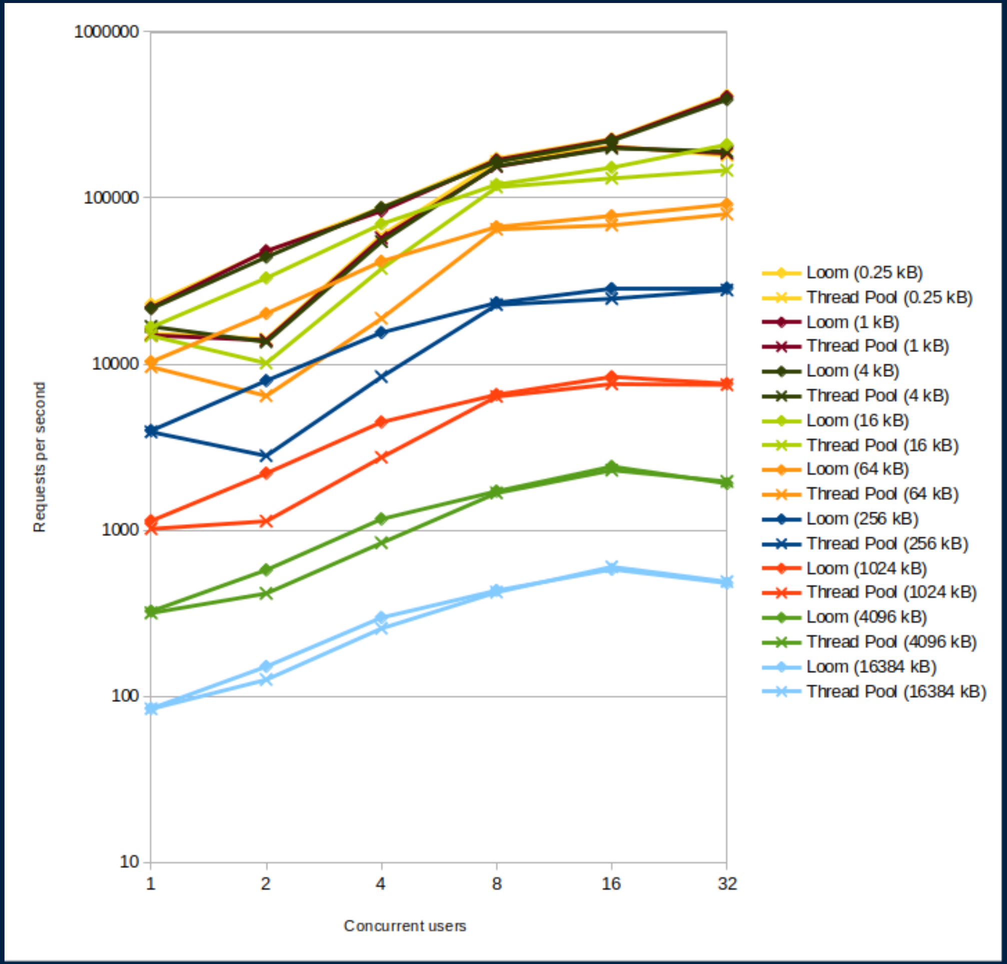
- compare virtual and platform threads in same scenario
- minimise impact of other factors
- not looking to identify maximums
- relative, rather than absolute, results were primary interest

Examined:

- Different sized requests
- Different concurrencies
- Configured to minimise Tomcat and web application processing time
- Details at https://spring.io/blog/2023/02/27/web-applications-and-project-loom

Throughput

Results



The bigger the response size, the less the difference

Platform thread performance is worse with concurrency of 2 than it is with 1

Virtual threads have higher throughput and this is more obvious with smaller response sizes

Once concurrency exceeds processor count, virtual threads show increased throughput compared to platform threads

Tomcat's thread pool uses LinkedBlockingQueue for the task queue by default.

The virtual thread scheduler uses a work stealing queue by default.

Throughput
Bonus results

These results are from some informal testing

- Much higher concurrency than my tests (8192 concurrent users)
- Any errors are my fault
- Any credit is due to Violeta Georgieva

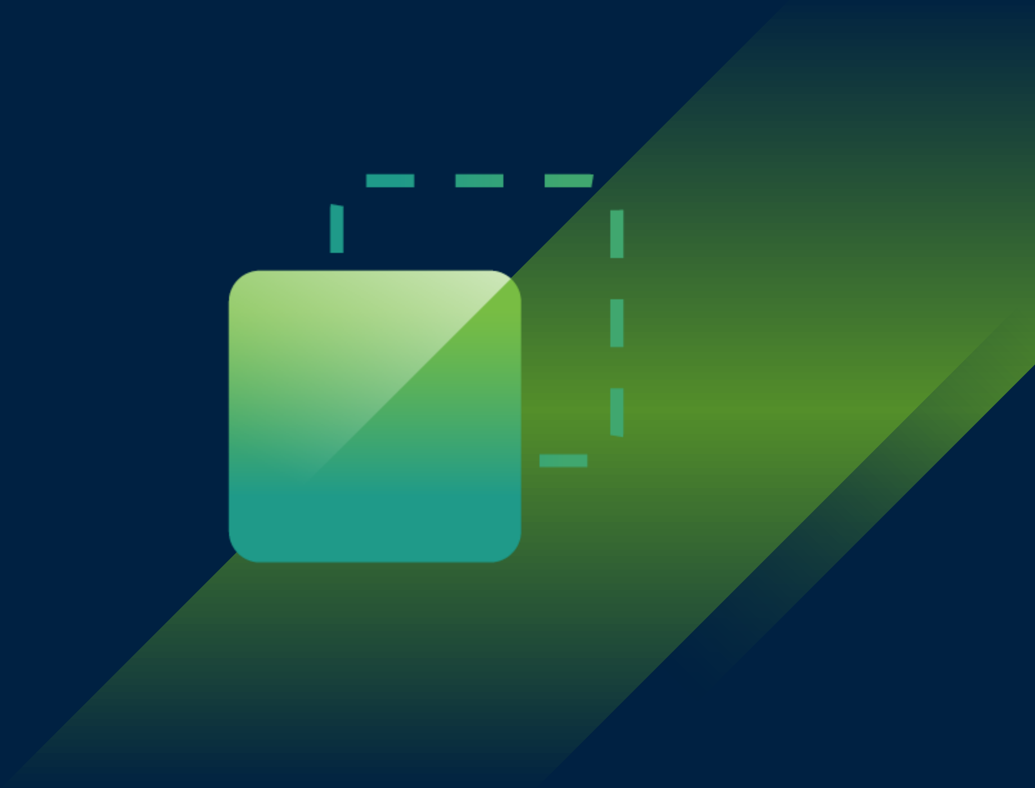
Platform threads

- 3,366,303 requests with 100% within 800ms

Virtual threads

- 3,408,798 requests with 89% complete within 800ms
- 9% complete between 800ms and 1200ms
- 2% complete in more than 1200ms

Easy to Use



Servlet Blocking IO

Counting request body bytes

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    resp.setContentType("text/plain");
    resp.setCharacterEncoding("UTF-8");
    ServletInputStream sis = req.getInputStream();
    byte[] buffer = new byte[8192];
    int read = -1;
    int totalBytesRead  = 0;
    while ((read = sis.read(buffer)) > -1) {
        if (read > 0) {
            totalBytesRead += read;
        }
    }
    ServletOutputStream sos = resp.getOutputStream();
    String msg = "Total bytes written = [" + totalBytesRead + "]";
    sos.write(msg.getBytes(StandardCharsets.UTF_8));
}
```

Servlet Non-blocking IO

Counting request body bytes

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    resp.setContentType("text/plain");
    resp.setCharacterEncoding("UTF-8");
    AsyncContext ac = req.startAsync();
    CounterListener listener =
        new CounterListener(ac, req.getInputStream(), resp.getOutputStream());
}


private static class CounterListener implements ReadListener, WriteListener {
    private final AsyncContext ac;
    private final ServletInputStream sis;
    private final ServletOutputStream sos;
    private volatile boolean readFinished = false;
    private volatile long totalBytesRead = 0;
    private byte[] buffer = new byte[8192];

    private CounterListener(AsyncContext ac, ServletInputStream sis,
            ServletOutputStream sos) {
        this.ac = ac;
        this.sis = sis;
        this.sos = sos;
        sis.setReadListener(this);
        sos.setWriteListener(this);
    }
```

```
    public void onDataAvailable() throws IOException {
        int read = 0;
        while (sis.isReady() && read > -1) {
            read = sis.read(buffer);
            if (read > 0) {
                totalBytesRead += read;
            }
        }
    }

    public void onAllDataRead() throws IOException {
        readFinished = true;
        if (sos.isReady()) {
            onWritePossible();
        }
    }

    public void onWritePossible() throws IOException {
        if (readFinished) {
            String msg = "Total bytes written = [" + totalBytesRead + "]";
            sos.write(msg.getBytes(StandardCharsets.UTF_8));
            ac.complete();
        }
    }

    public void onError(Throwable throwable) {
        ac.complete();
    }
}
```

Easy to Use

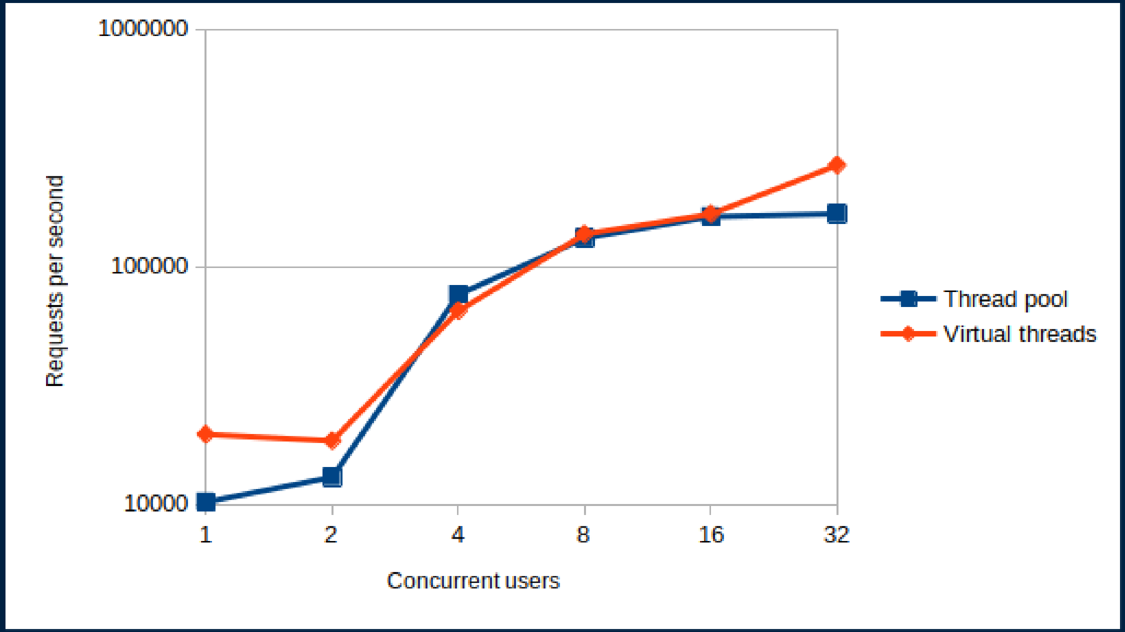
Aims:

- Compare virtual threads with blocking code to thread pool with non-blocking
- Minimise other factors

Examined

- External service that blocked and waited a preset time before continuing
- Service ‘delay’ dominated initial results

Easy to Use



Virtual threads generally a little more performant

Difference more noticeable at low concurrency and when concurrency exceeds processor cores

Performance of blocking code with virtual threads is comparable to refactoring to use non-blocking APIs

Coding constraints

Pinning

Detect it with `-Djdk.tracePinnedThreads=[full|short]`

Logs issues to `stdout` as they are detected

Tomcat experience

- Configured unit tests to run with this detection enabled
- Identified a handful of issues in HTTP/2

Can replace `synchronized` with `ReentrantLock`

- Need to be careful to ensure lock is released
- Make sure all uses of `synchronized` are replaced for a given object

Pinning

Replace

```
      synchronized (lock) {
          …
      }
```

With

```
      Lock lock = new ReentrantLock();
      …
      lock.lock();
      try {
          …
      } finally {
          lock.unlock();
      }
```

Threadlocal alternatives

Tomcat uses ThreadLocal for thread-safe caching of objects that are expensive to create

- Matchers in RewriteValve
- RequestDispatcher request mapping
- etc

Options to implement this with virtual threads

- No change, continue to use ThreadLocal
- Always create a new Object
- Cache using SynchronizedStack (or similar)

ThreadLocal alternatives

ThreadLocal

- Should be slower than new Object for virtual threads

new Object

- Lose the benefit of caching
- Is caching still required?

SynchronizedStack

- May be slower under high concurrency

ThreadLocal alternatives

Ran a series of tests

- Non-dispatching request / dispatching request
- 8, 16, 32 & 64 concurrent users (machine under test has 20 cores)
- new Object() / SynchronizedStack / ThreadLocal
- Platform threads / Virtual Threads

Ran each combination for 11 runs of 60 seconds

- Dropped the first result (warm-up)
- Took average of remaining 10

Results were inconclusive

- No clear winner or loser

Conclusions

Conclusions

Applications currently using non-blocking APIs will likely see minimal differences with virtual threads

Applications currently using blocking APIs

- will likely see minimal throughput differences with virtual threads
- will likely see measurable scalability improvements with virtual threads

Code changes may be required for:

- long lasting blocking operations
- ThreadLocals

Next Steps

Next Steps

Tomcat included virtual thread support in the June 2023 releases

Tomcat 11

- Requires a minimum of Java 21

Tomcat 8.5, 9.0 & 10.1

- No change to minimum Java versions
- Required Java 21 to use virtual threads

Future Tomcat development

- Investigate bottlenecks as they get reported

vmware EXPLORE

SpringOne

Questions...

vmware EXPLORE

SpringOne

Stay Connected

Discuss all things Tomcat @
https://tomcat.apache.org/lists.html

Visit @ https://tomcat.apache.org

https://github.com/apache/tomcat

For a discussion this week
markt@apache.org

vmware EXPLORE

SpringOne

Thank you