

Follow up on Valves

Kohsuke Kawaguchi
Jitendra Kotamraju

Agenda

- Performance Measurements
 - > Scalability and single-thread performance
- Programming Model
- ThreadLocal
- More illustrations of how it works

Single-Thread Performance

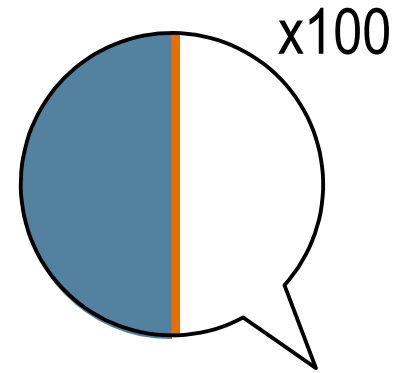
- Scenario
 - > Try to pump as many requests as possible by 1 thread
 - > How much performance hit do we get?
- Method
 - > Use wspex to determine how long one roundtrip takes
 - > Use prototype to determine how much of it is pipe overhead
 - > Use prototype to determine how long valves takes

Single-Thread Performance

- More about set up
 - > Single Opteron with 2GB mem
 - > Client in 1 VM, server in another VM on GF
 - > Client sends a simple request, server echoes it back
 - > Communication through HTTP via 127.0.0.1

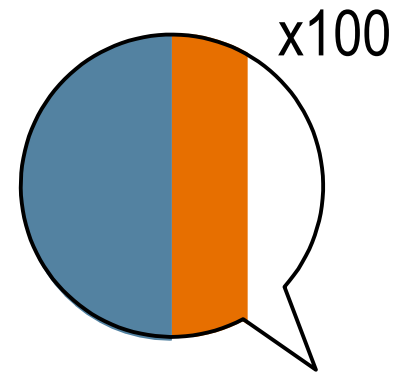
Single-Thread Performance

- Result with Pipes
 - > Pipe overhead \simeq 150ns



1roundtrip \simeq 3.5ms

- Result with Valves
 - > Valve overhead \simeq 1500ns
 - > Impact \simeq 0.05%

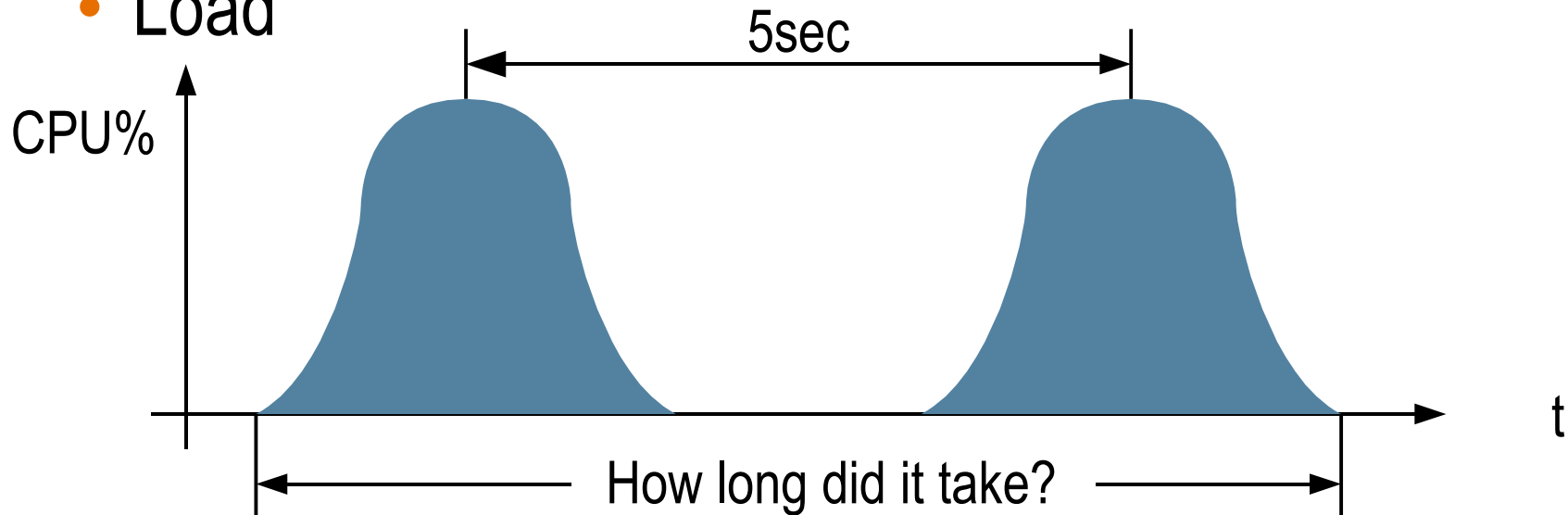


1roundtrip \simeq 3.5ms

Scalability Performance

- Scenario
 - > Simulated high-latency bussiness logic: 5 sec sleep
 - > Thread.sleep(5000) for pipes
 - > Timer trigger for valves
 - > Simulated large concurrent requests

- Load



Scalability Performance

- Pipe
 - > # of threads = # of concurrent requests
 - > Most threads will simply sleep
- Valve
 - > 5 threads used to serve all concurrent requests
 - > 1 thread used as the timer
- Tested on SunFire v440 (jwsdp.sfbay)
 - > 4 x UltraSparc IIIi, 16GB memory

Scalability Performance

- You can only create so many threads
 - > Beyond that, things just stop working

<i>Concurrency</i>	<i>Pipe</i>	<i>Valve</i>
1000	5.02sec	5.03sec
2000	5.03sec	
3000	Error	
10000		5.04sec
100000		5.43sec

Client Programming Model

- Use existing async API defined in spec

```
Future<?> f = port.invokeXYZAsync (...);  
...  
f.get(); // now block until it's done
```

- Before

- > This invocation still blocks one thread

- > Either you create a new thread, or wait until a pool thread is available

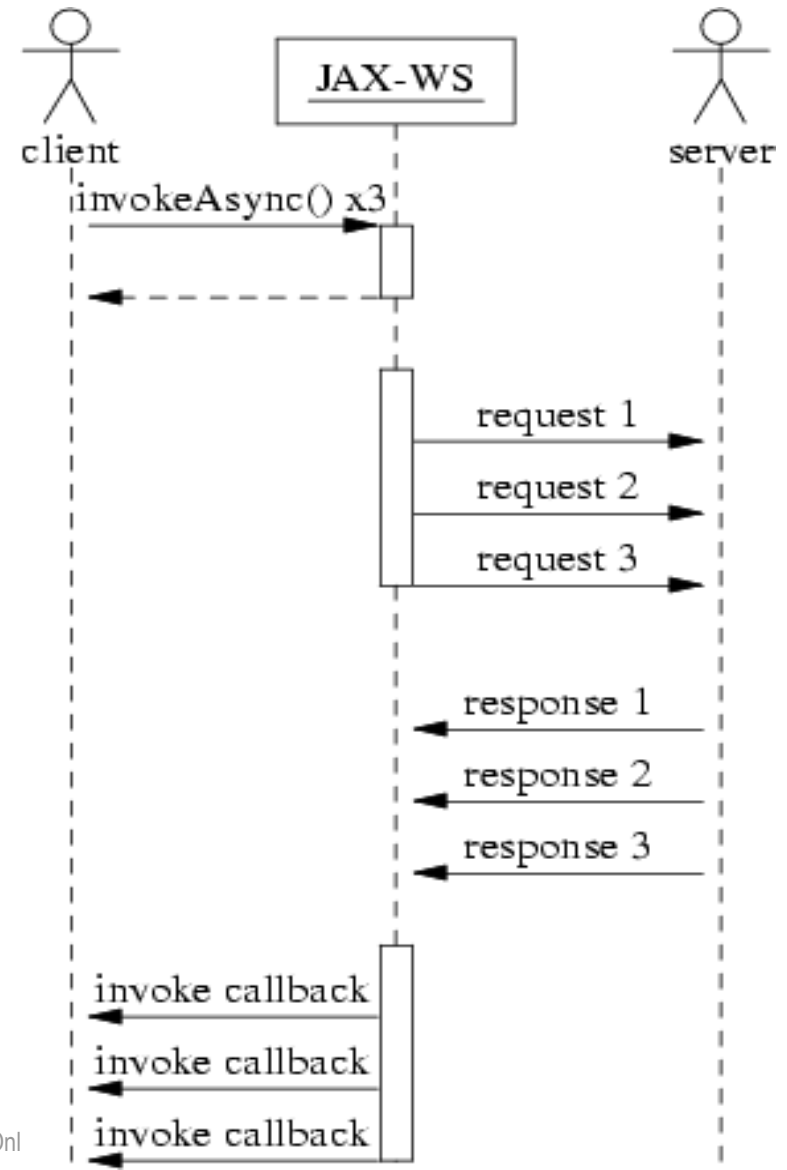
- After

- > This happens asynchronously without blocking threads

- > A large pending outgoing requests possible

Client Programming Model

- No change
 - > Underneath, we can use smaller # of threads to perform invocations



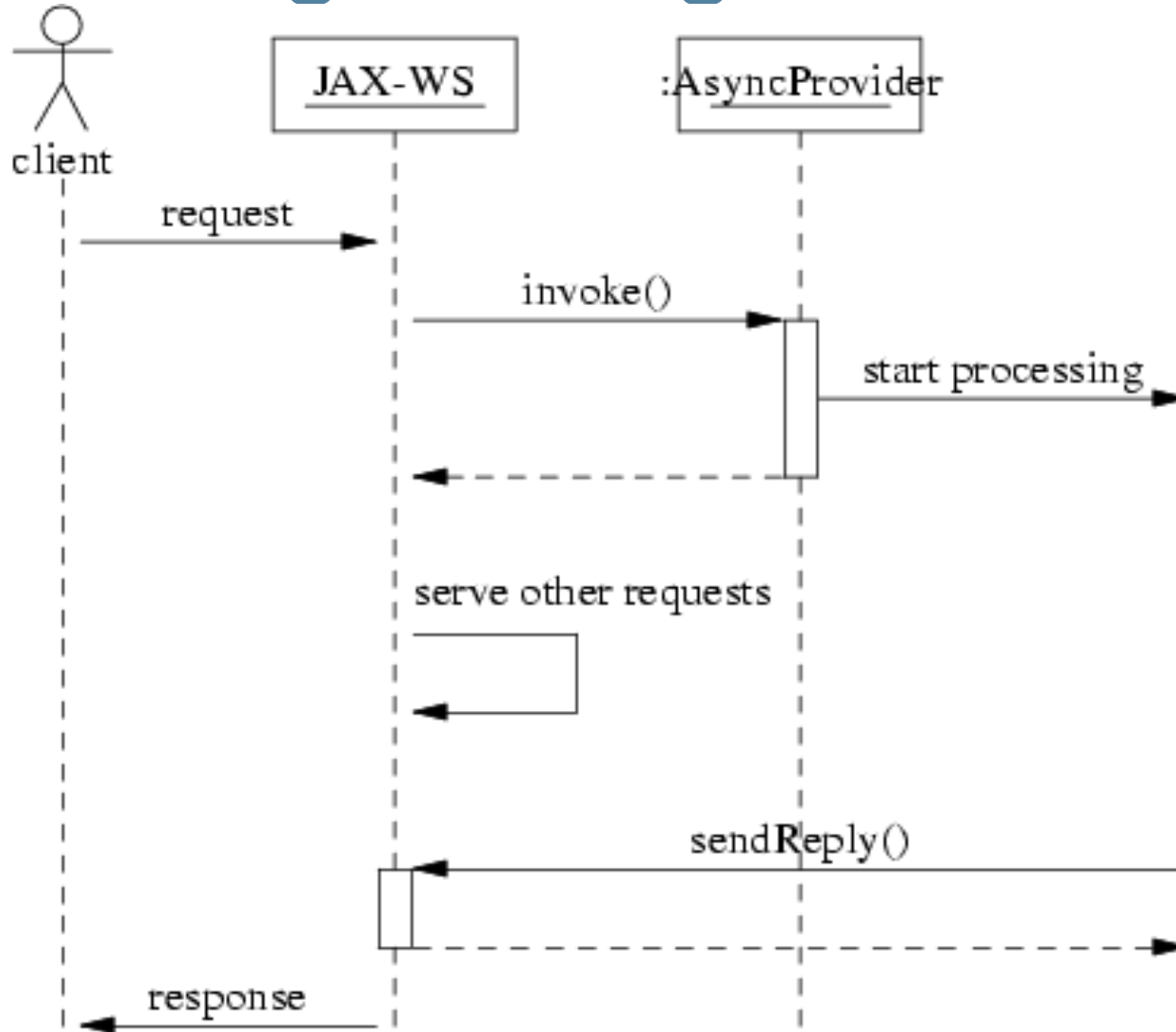
Server Programming Model

- RI specific async provider
 - > To take advantage of scalability you need to use this
 - > The invoke method may return immediately

```
interface AsyncProvider<T> {  
    void invoke(T request, Callback,  
        WebServiceContext);  
}
```

```
interface Callback<T> {  
    void sendReply(T response);  
    void sendFault(WebServiceException);  
}
```

Server Programming Model



ThreadLocal

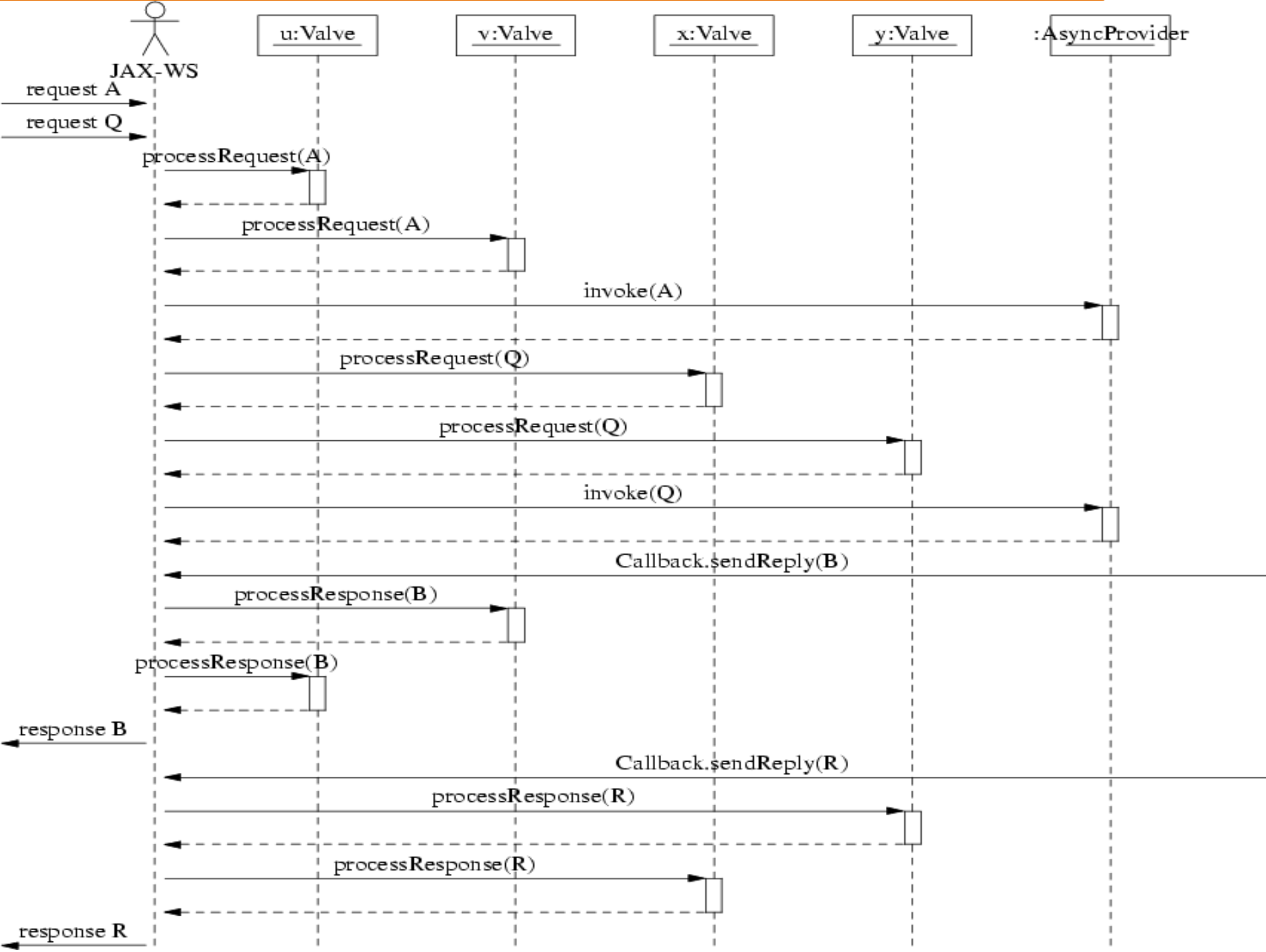
- What you DON'T need to worry about
 - > Valve instance need not be re-entrant
 - > The same instance serves both request and response
 - > Instances reused only after request/response cycle completes
 - > No synchronization necessary
- ThreadLocal works just fine if it's to avoid concurrent use of thread-unsafe resources
 - > Such as JAXB marshaller, SimpleDateFormat

ThreadLocal

- What you need to worry about
 - > In Valve, processRequest() and processResponses() may run by different threads
 - > That's the ONLY difference
- You need to know if this is going to break you
 - > If so, you need to run synchronously
 - > It's easy. It's just that you need to know

How 1 Thread Serves 2 Requests

- See picture in the next page
 - > Request A causes response B
 - > Request Q causes response R
 - > “Valveline” of two valves
 - > Two instances: (U,V) and (X,Y)
- Notice that...
 - > Valve instances are not reused
 - > Every invocations from JAX-WS may be from different threads



How To Run Synchronously?

- See picture in the next page
 - > Request A causes response B
 - > "Valveline" of 3 valves (x,y,z)
 - > Y decides that it needs to run synchronously
- Notice that...
 - > Everything before Y runs asynchronously
 - > Everything after Y runs in 1 thread
 - > No global coordination is needed
 - > Just that applications loses scalability

