

Top tien Enterprise Java performance problemen en hun oplossingen

Vincent Partington
Xebia

- Xebia doet in Enterprise Java:
 - Ontwikkeling
 - Performance audits
 - e.a.
- Veel ervaring met performance problemen.
- Top tien:
 - Input van alle Xebia consultants, NL en FR.
 - Verzameld en gecategoriseerd.

Opbouw presentatie

- Van 10 naar 1.
- Van ieder issue:
 - Voorbeeld uit de praktijk.
 - Beschrijving van het probleem.
 - Mogelijke maatregelen om te voorkomen en/of op te lossen.

#10 – Overvloedige logging – Voorbeeld

- webMethods Glue heeft eigen logging framework.
- Ontwikkelaars wilde toch gebruik maken van Log4j
- Eigen Glue logging -> Log4j bridge geschreven.
- Maar deze logging bridge gaf Log4j log levels niet door aan Glue logging framework.
- Resultaat: in productie werd veel log info opgebouwd om uiteindelijk niet gelogd te worden.

#10 – Overvloedige logging – Probleem

- Opbouwen van informatie die niet gelogd wordt.
 - veel String manipulaties -> duur.
- Te veel logging
 - Stacktraces
 - Debug level
- Kost tijd om weg te schrijven.

#10 – Overvloedige logging – Maatregelen

- Gebruik een logging framework:
 - Log4j
 - JDK logging
 - Apache Commons Logging als je niet wilt kiezen. 😊
- Gebruik idioom om String manipulaties te voorkomen:

```
if ( isDebugEnabled() ) log.debug ( )
```
- Maak aparte configuraties voor test en productie.

#9 – Verkeerde app. server config. – Voorbeeld

- JBoss 4.0.2 op Solaris.
- 25 requests per seconde.
- Max. heap op 64MB (default).
- Bijna de hele tijd GC's -> CPU usage naar 100%.
- Heap op 256MB -> CPU usage onder 50%.

#9 – Verkeerde app. server config. – Probleem

- Niet goed configureren c.q. tunen van applicatie server.
- Aanname dat houden aan contract *automatisch* goede performance oplevert.

- JVM settings:
 - -Xms (=Xmx), -Xmx,
 - No `-verbose` and `-verbosegc`
 - `-server` to select server VM
- Pools:
 - Connecties: JDBC, JMS, etc.
 - Threads: HTTP, etc.
- Prepared statement cache.

#8 – Foutief gebruik van J2EE – Voorbeeld

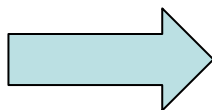
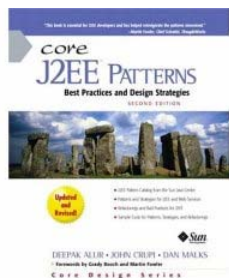
- Applicatie waar de business en presentatie layer strict van elkaar gescheiden waren.
- De EJBs boden geen functionaliteit om een bepaalde subset aan data op te halen.
- JSPs deden daarom query en haalde één voor één de EJBs binnen.
- 1+n queries: $470 \times 25\text{ms} = 11750\text{ms}$.
- Na fix: 100ms.

#8 – Foutief gebruik van J2EE – Probleem

- J2EE architecturen "volgens het boekje".
 - *Alle* features: transacties, security, distributie.
 - Toepassing van verouderde J2EE patterns van Sun:
 - Service Locator -> DI
 - Business Delegate -> i.d.
 - Fast Lane Reader -> echte persistency framework
 - Transfer Object -> i.d.
 - Intercepting Filter -> AOP
 - etc.
- Geen frameworks om het werk te verlichten -> veel nadruk "plumbing".

#8 – Foutief gebruik van J2EE – Maatregelen

- Gebruik alleen die onderdelen van J2EE die nodig zijn:



- Gebruik lightweight frameworks zoals Spring.

#7 – Onnodig gebruik van XML – Voorbeeld

- Xalan interpreter i.p.v. XSLTC voor runnen van XSLT sheets.
 - Default in JDK 1.4.
 - Slechte performance.

#7 – Onnodig gebruik van XML – Probleem

- XML processing is traag:
 - Parsen:
 - DOM vs. SAX. vs. StAX.
 - BEA implementatie van STaX API.
 - Validatie.
 - Transformeren:
 - Geïnterpreteerde vs. gecompileerde stylesheets.
 - Genereren.
 - DOM serialisatie.

#7 – Onnodig gebruik van XML – Maatregelen

- Voorkom gebruik van XML.
 - *"Java is the verb, XML is the noun."*
Yeah, right.
- Zeker niet voor remoting.
- Gebruik goede API en goede implementatie van die API.

#6 – Verkeerde caching – Voorbeeld

```
public Object getFromCache(String key) {  
    synchronized(map) {  
        if(!map.containsKey(key)) {  
            ... get data from backend ...  
        }  
        return map.get(key);  
    }  
}
```

- Ophalen uit backend kostte 100ms.
- Cache hit ratio < 10%.

#6 – Verkeerde caching – Probleem

- Cachen wanneer niet nodig:
 - Lage cache hit ratio.
 - Geheugen gebruik.
 - Lock contention.
 - Moeilijker testen.
- Niet cachen:
 - Steeds (bijna) zelfde informatie uit backend.
 - Database connecties, AxisEngine, Spring BeanFactory, Lazy loading.

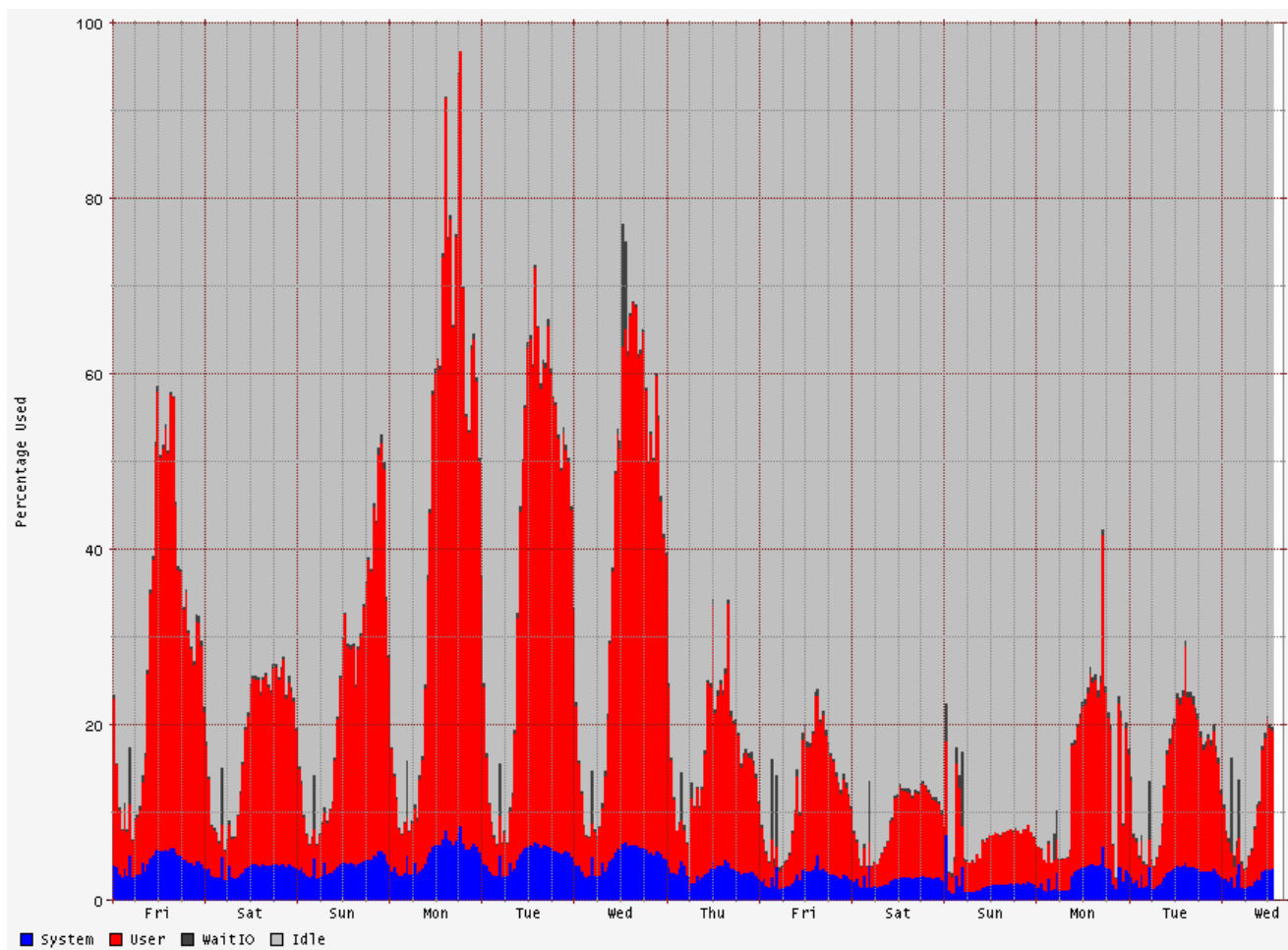
#6 – Verkeerde caching – Maatregelen

- Implementeer caching pas wanneer nodig.
- Verifieer caching gedrag met profiler.
- Maak het mogelijk caching te monitoren/beheren in productie.
 - Invalideren cache bij wijzigingen in remote resource.

#5 – Overvloedig geheugengebruik – Voorbeeld

- Een web applicatie met +/- 100 requests per seconde verbruikte soms 95% CPU.
- Applicatie had gemiddeld 1.400.000 objecten in het geheugen:
 - 200.000 voor applicatieserver.
 - 200.000 voor applicatie.
 - 1.000.000 voor i18n module.
- IBM JDK 1.3 GC had het er zwaar mee.
- Na tunen: CPU gebruik terug naar 22%.

#5 – Overvloedig geheugengebruik – Voorbeeld



#5 – Overvloedig geheugengebruik – Probleem

- Te veel object allocaties en deallocaties.
 - Veel garbage collects.
 - Máár: object allocatie wordt steeds goedkoper.
- Te veel objecten.
 - Afhankelijk van JVM: generationele garbage collector vanaf 1.4 JVM.
 - Te veel classes, ge-`intern`de Strings: PermGen vol.

#5 – Overvloedig geheugengebruik – Maatregelen

- Profile niet alleen het tijdsgedrag van je algoritme maar ook het geheugengebruik.
 - Dynamisch gedrag: aantal objectallocaties/deallocaties per request.
 - Statisch gedrag: hoeveelheid objecten.

#4 – Slecht performende libraries – Voorbeeld

- JAMon: JSP gebruikt veel `DateFormats` en `NumberFormats`.
 - Steeds opnieuw gecreëerd vanwege `threadunsafeness`.
 - Performt slecht.
- `java.net.URLEncoder` vs. `Commons Codec`.
- BEA implementatie van `StAX API`.

#4 – Slecht performende libraries – Probleem

- Verkeerd/simplistisch gebruik van libraries.
- Slecht performende libraries.
- Eigen libraries/frameworks waar bekende libraries al aanwezig zijn.
 - Bekende libraries hebben een hoop problemen al opgelost.
 - Verkeerde besteding ontwikkelenergie.

#4 – Slecht performende libraries – Maatregelen

- Lees de documentatie van de libraries die je gebruikt aandachtig.
 - Maak zonodig een kleine benchmark.
- Gebruik een bekend, goed ondersteund framework.
 - Minder bugs.
 - Meer documentatie (boeken).

#3 – Slechte concurrency – Voorbeeld

- J2EE server applicatie met J2ME clients.
- Communicatie met JINI en JavaSpaces.
- Om requests serieel af te handelen werd een lock uit JavaSpaces gehaald.
- Maar afhandelen van request kostte 100ms, waardoor niet meer dan 10 requests per seconde afgehandeld konden worden.
 - Door remote calls
 - En door traagheid hardware van J2ME client.
- Oplossing: critical section korter gemaakt.

#3 – Slechte concurrency – Probleem

- Verkeerd gebruik van synchronisatie:
 - Te lange `synchronized` blocks/transacties: lock contention.
 - Te lange transacties in database: lock escalation (vooral bij hoge isolation levels).
- Veel JDK classes zijn thread-safe: `Vector`, `StringBuffer`, I/O Streams en `Readers/Writers`.
 - Alternatieven: `List`, `StringBuilder`, NIO.

#3 – Slechte concurrency – Maatregelen

- Minimaliseer shared data (points of contention).
- Maak critical section zo kort mogelijk.
- Schrijf niet je eigen synchronisatie:
 - Gebruik optimistic locking waar mogelijk.
 - Gebruik een framework.

#2 – Onnodige remoting – Voorbeeld

- Applicatie opgedeeld in twee projecten:
 - Webservice laag om mainframe te ontsluiten.
 - Web interface laag.
- Ontsluitingen door andere interfaces kwam niet van de grond, maar architectuur werd behouden.
- Zeer slechte performance door gebruik van SOAP.

#2 – Onnodige remoting – Probleem

- Het gebruik van remoting technologieën waar dat eigenlijk niet nodig is.
 - Remote call waar local call ook werkt (EJB).
 - Gebruik van externe app. voor simpele functionaliteit:

```
Runtime.getRuntime().exec("d:\\path\\to\\bin\\sleep.exe " + nbSecond);
```
- Verkeerd gebruik van remoting technologieën:
 - Complexe remoting protocollen zoals SOAP.
 - SOAP : RMI : Local = 2000 : 100 : 1
 - Fine grained method calls.

#2 – Onnodige remoting – Maatregelen

- Alleen wanneer nodig.
 - Doe een benchmark.
- Tip: Als business functionaliteit op meerdere manieren beschikbaar moet zijn: implementeer als POJO en schrijf/configureer wrappers (Spring's `RemoteExporters`).

#1 – Verkeerd gebruik van database – Voorbeeld

- Front-office applicatie leest data via views.
 - FO user heeft alleen rechten op views en data die gelezen mag worden.
 - Geen indexen aangemaakt voor views en de queries erop.

#1 – Verkeerd gebruik van database – Probleem

- Verkeerd database ontwerp.
 - Missende/verkeerde database indexes.
 - Niet afgestemd op gebruik door applicatie.
 - Database design met key/value paren.
- Slecht gebruik van DB door applicatie.
- Verkeerde database server settings.

#1 – Verkeerd gebruik van database – Maatregelen

- Ga er niet van uit dat de database "gewoon werkt".
 - Database en applicatie dienen op elkaar afgestemd te worden.
 - Database is meer dan "domme bit bucket", maak gebruik van de features van je database.
- Werk samen met de database beheerder.
 - Laat DBA zijn tools gebruiken tijdens performance tests.

Samenvatting

- #10 – Overvloedige logging
- #9 – Verkeerde applicatie server configuratie
- #8 – Foutief gebruik van J2EE
- #7 – Onnodig gebruik van XML
- #6 – Verkeerde caching
- #5 – Overvloedig geheugengebruik
- #4 – Slecht performende libraries
- #3 – Slechte concurrency
- #2 – Onnodige remoting
- #1 – Verkeerd gebruik van database

- Meten = weten.
 - Geen aannames.
 - Tools: JProbe, Eclipse Profiler, Optimizelt, Performasure, JMeter, Mercury LoadRunner, etc.
 - <http://www.javaperformancetuning.com/tools/>
- Representatieve performance tests:
 - Gebruik: alle use cases in de juiste verdeling.
 - Testdata: evenveel data, maar ook random genoeg i.v.m. caching, disk effecten, etc.
 - Systemen: zelfde sizing als in productie.

En nu?

- Wat de OWASP kan, kunnen wij ook!
 - Do's en don'ts voor nieuwe projecten.
 - Cheatsheet bij performance problemen.
- Wiki op JOCE.
 - Meer ervaringen.
 - Verder analyseren van problemen.
 - Andere rangorde.
- Houdt de blog in de gaten:
<http://blog.xebia.com>