Top Ten Enterprise Java performance problems and their solutions

Vincent Partington

Xebia

Introduction

- Xebia is into Enterprise Java:
  - Development
  - Performance audits
  - a.o.
- Lots of experience with performance problems.
- Top Ten:
  - Input from all Xebia consultants, NL and FR.
  - Gathered and categorized.

Agenda

- From 10 to 1.
- For every issue:
  - Example taking from our experiences.
  - Problem analysis.
  - Possible solutions and/or measures to prevent the problem.

#10 – Excessive logging – Example

- webMethods Glue has its own logging framework.
- Developers wanted Log4j and wrote Glue -> Log4J bridge.
- Bridge did not return Log4j log levels to Glue
- Result: lots of logging info was constructed but not used.

#10 – Excessive logging – Analysis

- Info gathered that is not logged.
  - Lots of String manipulations -> expensive.
- Too much logging
  - Stacktraces
  - Debug level
- I/O latency.

#10 – Excessive logging – Measures

- Use a logging framework:
  - Log4j
  - JDK logging
  - Apache Commons Logging if you don't want to choose. ☺
- Use idiom to prevent unnecessary String manipulations:

```
if(isDebugEnabled()) log.debug()
```

- Separate configurations for development and production.

#9 – Incorrect app. server config. – Example

- JBoss 4.0.2 on Solaris.
- 25 requests per second.
- Max. heap size was 64MB (default).
- Back-to-back GC's -> CPU usage to 100%.
- Max. heap size set to 256MB -> CPU usage below 50%.

#9 – Incorrect app. server config. – Analysis

- Incorrect configuration (tuning) of application server.
- Assumption that keeping to the J2EE contract will automatically get you good performance.

#9 – Incorrect app. server config. – Measures

- 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 – Incorrect usage of J2EE – Example

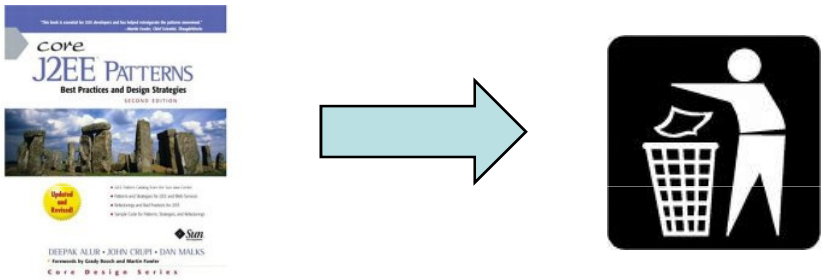
- Business and presentation layer of application were strictly separated.
- EJBs did not expose any functionality to retrieve subset of data.
- JSPs worked around that by performing query and then retrieving the required objects one by one.
- 1+n queries: 470 x 25ms = 11750ms.
- After fix: 100ms.

#8 – Incorrect usage of J2EE – Analysis

- J2EE architectures "by the book":
  - *All* features: transactions, security, distribution.
  - Usage of outdated Sun J2EE patterns:
    - Service Locator -> DI
    - Business Delegate -> i.d.
    - Fast Lane Reader -> real persistency framework
    - Transfer Object -> i.d.
    - Intercepting Filter -> AOP
    - etc.
- No frameworks to lessen the burden -> lots of "plumbing" code.

#8 – Incorrect usage of J2EE – Measures

- Use only parts of J2EE that apply:



- Use lightweight frameworks such as Spring.

#7 – Unnecessary use of XML – Example

- Xalan interpreter instead of XSLTC to execute XSLT sheets.
  - Default in JDK 1.4.
  - Bad performance.

#7 – Unnecessary use of XML – Analysis

- XML processing is *slow*:
  - Parsing:
    - DOM vs. SAX. vs. StAX.
      - BEA's implementation of STaX API.
    - Validation.
  - Transformation:
    - Interpreted vs. compiled stylesheets.
  - Generation:
    - DOM serialization.

#7 – Unnecessary use of XML – Measures

- Only use XML when you have to.
  - *"Java is the verb, XML is the noun."*
    Yeah, right.
- Especially when remoting.
- Use proper API and good implementation of that API.

#6 – Improper caching – Example

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

- Retrieval from backend takes 100ms.
- Cache hit ratio < 10%.

#6 – Improper caching – Analysis

- Caching when it's not needed:
  - Low cache hit ratio.
  - Memory usage.
  - Lock contention.
  - Hard to test.
- Not caching when it *is* needed:
  - Repeated retrieval of (nearly) identical information from backend.
  - Database connecties, AxisEngine, Spring BeanFactory, Lazy loading.

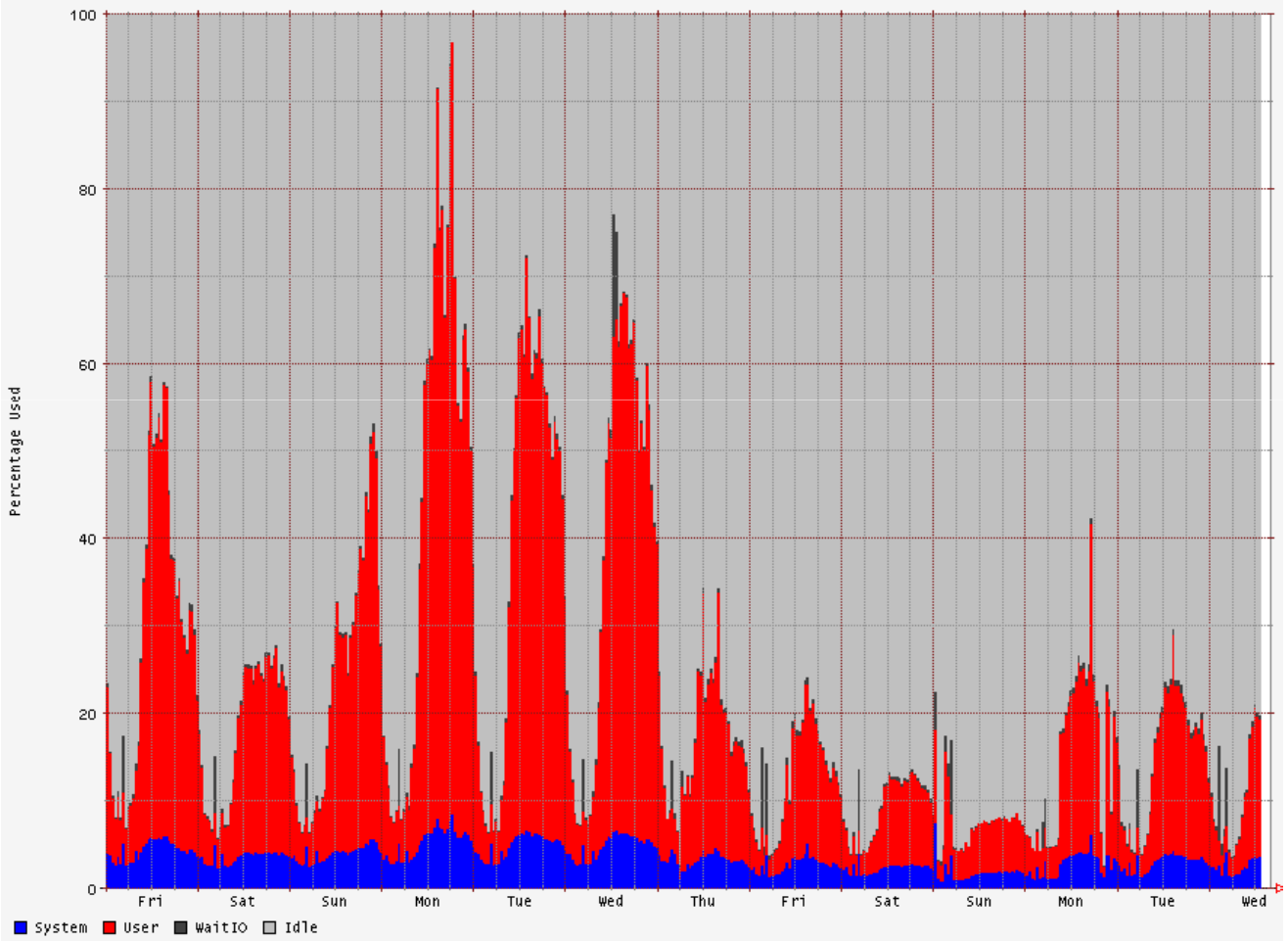
#6 – Improper caching – Measures

- Only implement caching when you need it.
- Verify the behaviour of the cache with a profiler.
- Make your cache monitorable and manageable.
  - Invalidation on change in remote resource.

#5 – Excessive memory usage – Example

- Web application with ±100 request per second, 95% CPU usage.
- Application had about 1.400.000 objects in memory:
  - 200.000 for application server
  - 200.000 for the application
  - 1.000.000 for an i18n module
- IBM JDK 1.3 GC did not perform well.
- After tuning: CPU usage back to 22%.

#5 – Excessive memory usage – Example



#5 – Excessive memory usage – Analysis

- Too many object allocations and deallocations.
  - Many garbage collection cycles.
  - But: object (de)allocation is getting cheaper and cheaper.
- Too many objects:
  - Depends on JVM; generational garbage collector since 1.4 JVM.
  - Too many classes, interned Strings, etc. fill up PermGen.

#5 – Excessive memory usage – Measures

- Don't just profile time behaviour of your application but also memory behaviour:
  - Dynamic: # of (de)allocations per request.
  - Static: # of objects.

#4 – Badly performing libraries – Example

- JAMon: JSP uses lots of `DateFormat`s en `NumberFormat`s.
  - Created again and again because of threadunsafeness.
- `java.net.URLEncoder` vs. Commons Codec.
- BEA implementation of StAX API.

#4 – Badly performing libraries – Analysis

- Bad/simplistic usage of libraries.
- Badly performing libraries.
- Homebrewn libraries where more mature alternatives are available.

#4 – Badly performing libraries – Measures

- Read the documentation of the libraries you use.
  - Scan for performance features.
  - Write a small benchmark.
- Use a well known, well supported framework.
  - Performance issues may have been found and solved by others.
  - Less bugs.
  - More documentation.

#3 – Incorrectly impl. concurrency – Example

- J2EE server app. with J2ME clients.
- Communication using JINI and JavaSpaces.
- To handle requests serially (!) a lock was retrieved from JavaSpaces.
- Handling a request took 100ms, allowing only 10 requests per second.
  - Remote calls -> network latency.
  - Slow hardware.
- Solution: shorter critical section.

#3 – Incorrectly impl. concurrency – Analysis

- Bad use of synchronization features:
  - Long synchronized bloks or transactions, causing lock contention.
  - Long transactions in database, causing lock escalation (esp. with high isolation levels).
- Many JDK classes are thread-safe: `Vector`, `StringBuffer`, I/O Streams and Readers/Writers.
  - Alternatives: `List`, `StringBuilder`, NIO.

#3 – Incorrectly impl. concurrency – Measures

- Minimize shared data (points of contention).
- Make critical section as short as possible.
- Avoid writing synchronization code:
  - Use optimistic locking where possible.
  - Use a framework.

#2 – Unnecessary remoting – Example

- Application spread over two projects:
  - Web services layer providing access to main frame.
  - Web interface using web services layer.
- Other interfaces were never developed, but architecture was kept.
- Very bad performance due to SOAP overhead.

#2 – Unnecessary remoting – Analysis

- Use of remoting where it is not necessary.
  - Remote call when local call can be used (EJB).
  - Use of external app. for simple functionality:

```
Runtime.getRuntime().exec(
   "d:\\path\\to\\bin\\sleep.exe " +
   nbSecond);
```

- Incorrect usage of remoting technologies:
  - Complex remoting protocols such as SOAP.
    - SOAP : RMI : Local = 2000 : 100 : 1
  - Fine grained method calls.

#2 – Unnecessary remoting – Measures

- Only use remoting when necessary.
  - Write a benchmark.
- Tip: If business functionality needs to be accessible in multiple ways; implement as POJO's and write/configure wrapped (Spring's `RemoteExporter`s).

#1 – Incorrect usage of databases – Example

- Front-office application reads data via views.
  - FO user only has read rights on selected views and data.
  - No indexes on views and for queries executed by application.

#1 – Incorrect usage of databases – Analysis

- Bad database design.
  - Missing/incorrect database indexes.
  - Not tuned to usage by application.
  - Database design with key/value pairs.
- Bad usage of DB by application.
- Incorrect database server settings.

#1 – Incorrect usage of databases – Measures

- Don't assume the database "just works".
  - Database and application need to be tuned to work together.
  - Database is more than just a "dumb bit bucket", use its features.
- Collaborate with the DBA.
  - Let DBA use tools during performance tests.

Summary

- #10 – Excessive logging
- #9 – Incorrect application server configuration
- #8 – Incorrect usage of J2EE
- #7 – Unnecessary use of XML
- #6 – Improper caching
- #5 – Excessive memory usage
- #4 – Badly performing libraries
- #3 – Incorrectly implemented concurrency
- #2 – Unnecessary remoting
- #1 – Incorrect usage of databases

Conclusion

- To measure is to know.
  - No assumptions.
  - Tools: JProbe, Eclipse Profiler, OptimizeIt, Performasure, JMeter, Mercury LoadRunner, etc.
  - `http://www.javaperformancetuning.com/tools/`
- Representative performance tests:
  - Use: all use cases, correctly distributed.
  - Test data: enough data, random enough to account for caching, disc effects, etc.
  - Systems: same sizing as production systems.

And now?

- Read the blogs:
  `http://blog.xebia.com/2007/04/30/ejapp-top-10-countdown-wrap-up/`

- For questions contact:
  - Vincent Partington vpartington@xebia.com