Improving Scalability of NaviServer

Univ.-Prof. Dr. Gustaf Neumann
Vienna University of Economics and Business
Information Systems and New Media
Overview

- NaviServer brief history

- Scalability
  - Improve Locking
  - Improve behaviour on overload situations
  - Help to identify bottlenecks
  - Bandwidth management

- More strict input handling
  - Induced by vulnerability scanning

- Misc
  - Expose internals to Tcl
  - Unit support for API and configuration (e.g. 1s, 100µs, ... 1KB, 1.5MB, ...)
  - Crypto support (OCSP Stapling, SNI, SCRYPT, SCRAM, HMAC, 20+ digest algorithms, ...)
  - Experiments with HTTP/2 support
History of NaviServer

- **NaviServer brief history**
  - 1991 Closed-source product, developed by company “NaviSoft”, used by AOL as AOLserver
  - 1999 AOLserver open source
  - 2005 fork of AOLserver 4.10 -> NaviServer 4.99.0 (original name)
  - 2016: integration of OpenSSL support directly in NaviServer, IPv6

- **NaviServer: multi-protocol server**
  - HTTP, UDP, SMTP, LDAP, DNS, COAP, IMAP, ...

- **Code Overview**
  - 494 unique files
  - 140K lines of code (+58K lines comment + empty lines)
  - C: 73%, Tcl: 15%
  - >1900 tests in regression test
Code Age analysis for NaviServer
(based on git-blame statistics)

History: 22 years in repository
Average age per line: 10 years
Oldest lines: Copyright lines from AOL
Bottlenecks based on mutex locks: Example from OpenACS 5.9.1 (2019)

Symptom: cores are under-utilized

**Total locks: 1.83G, total requests 2.62M, avg locks per req 700, avg lock time 5.76μs, lock time per req 4.03ms**
(except: interp, jobThreadPool, ns:sched, tcjob:jobs, tcjob:ns_eval_q:production)

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Locks</th>
<th>Busy</th>
<th>Contention</th>
<th>Total Lock</th>
<th>Avg Lock</th>
<th>Total Wait</th>
<th>Max Wait</th>
<th>Locks/Req</th>
<th>Pot Locks/sec</th>
<th>Pot Req/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns:cache:util_memoize</td>
<td>1209</td>
<td>110.71M</td>
<td>237.51K</td>
<td>0.2145</td>
<td>5.14Ks</td>
<td>48.42μs</td>
<td>2.91Ks</td>
<td>274.88ms</td>
<td>42.27</td>
<td>21.54K</td>
<td>509.63</td>
</tr>
<tr>
<td>ns:cache:xo_site_nodes</td>
<td>1204</td>
<td>127.85M</td>
<td>266.04K</td>
<td>0.2081</td>
<td>647.28s</td>
<td>5.06μs</td>
<td>147.62s</td>
<td>312.55ms</td>
<td>48.81</td>
<td>197.52K</td>
<td>4.05K</td>
</tr>
<tr>
<td>ns:cache:db_cache_pool</td>
<td>1210</td>
<td>5.33M</td>
<td>6.23K</td>
<td>0.1170</td>
<td>519.08s</td>
<td>97.42μs</td>
<td>84.37s</td>
<td>99.88ms</td>
<td>2.03</td>
<td>10.26K</td>
<td>5.05K</td>
</tr>
<tr>
<td>ns:cache:xotcl_object_type_cache</td>
<td>1215</td>
<td>27.72M</td>
<td>28.04K</td>
<td>0.1011</td>
<td>507.2s</td>
<td>18.3μs</td>
<td>226.16s</td>
<td>150.96ms</td>
<td>10.58</td>
<td>54.66K</td>
<td>5.16K</td>
</tr>
<tr>
<td>nsv:138:production</td>
<td>78</td>
<td>199.99M</td>
<td>238.76K</td>
<td>0.1194</td>
<td>352.42s</td>
<td>1.76μs</td>
<td>22.08s</td>
<td>1.05s</td>
<td>76.36</td>
<td>567.48K</td>
<td>7.43K</td>
</tr>
<tr>
<td>tcjob:xocal_queue</td>
<td>1249</td>
<td>29.62M</td>
<td>0</td>
<td>0.0000</td>
<td>316.13s</td>
<td>10.67μs</td>
<td>0s</td>
<td>0s</td>
<td>11.31</td>
<td>93.69K</td>
<td>8.29K</td>
</tr>
<tr>
<td>nsv:169:production</td>
<td>47</td>
<td>237.75M</td>
<td>177.15K</td>
<td>0.0745</td>
<td>274.19s</td>
<td>1.15μs</td>
<td>3.2s</td>
<td>67.14ms</td>
<td>90.77</td>
<td>867.08K</td>
<td>9.55K</td>
</tr>
</tbody>
</table>

**Statistics from nsstats.tcl**

- **Avg Locks/request:** 700 (OpenACS 5.10: openacs.org: 144)
- **Max reqs/sec:** 248 (OpenACS 5.10: openacs.org: 4.8K)
- **For single cache:**
  - **Max Wait:** 275ms, Locks/req: 42, max req/s: 509
Cache statistics
from OpenACS 5.9.1 (2019)

- **Hot caches:** up to 47 cache hits per request

<table>
<thead>
<tr>
<th>Cache</th>
<th>Max</th>
<th>Current</th>
<th>Utilization</th>
<th>Entries</th>
<th>Avg Size</th>
<th>Flushes</th>
<th>Hits</th>
<th>Hits/Req</th>
<th>Reuse</th>
<th>Misses</th>
<th>Hit Rate</th>
<th>Expired</th>
<th>Pruned</th>
<th>Commit</th>
<th>Rollback</th>
<th>Saved/Hit</th>
<th>Saved/Req</th>
</tr>
</thead>
<tbody>
<tr>
<td>xo_site_nodes</td>
<td>2000000</td>
<td>1999970</td>
<td>100.00%</td>
<td>20242</td>
<td>98</td>
<td>592857</td>
<td>6657880</td>
<td>3289</td>
<td>1124789</td>
<td>98.00%</td>
<td>0</td>
<td>489245</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>181.62μs</td>
<td>8.56ms</td>
</tr>
<tr>
<td>util_memoize</td>
<td>3000000</td>
<td>2721870</td>
<td>90.73%</td>
<td>59290</td>
<td>45</td>
<td>277667</td>
<td>51432766</td>
<td>36.4024</td>
<td>867</td>
<td>5263317</td>
<td>90.00%</td>
<td>4201504</td>
<td>287099</td>
<td>0</td>
<td>0</td>
<td>308.44μs</td>
<td>11.23ms</td>
</tr>
<tr>
<td>xotol_object_type_cache</td>
<td>6000000</td>
<td>4558006</td>
<td>75.97%</td>
<td>338123</td>
<td>13</td>
<td>12665</td>
<td>11909302</td>
<td>8.4290</td>
<td>35</td>
<td>1578033</td>
<td>88.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>977.21μs</td>
<td>8.24ms</td>
</tr>
<tr>
<td>xotol_object_cache</td>
<td>50000000</td>
<td>39477515</td>
<td>78.96%</td>
<td>197606</td>
<td>1997</td>
<td>132730</td>
<td>8736833</td>
<td>6.1836</td>
<td>44</td>
<td>505598</td>
<td>94.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12.37ms</td>
<td>76.49ms</td>
</tr>
<tr>
<td>xocal_sess</td>
<td>20000000</td>
<td>24789342</td>
<td>12.39%</td>
<td>64692</td>
<td>383</td>
<td>0</td>
<td>4322867</td>
<td>3.0596</td>
<td>67</td>
<td>1928218</td>
<td>89.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.45μs</td>
<td>4.43μs</td>
</tr>
<tr>
<td>db_cache_pool</td>
<td>40000000</td>
<td>687172</td>
<td>1.72%</td>
<td>73086</td>
<td>0</td>
<td>8150</td>
<td>206867</td>
<td>10583</td>
<td>37</td>
<td>93538</td>
<td>96.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.02R³μs</td>
<td>1R³μs</td>
</tr>
</tbody>
</table>

- **Most important caches:**
  Avg savings per request per cache up to 76ms

<table>
<thead>
<tr>
<th>Cache</th>
<th>Max</th>
<th>Current</th>
<th>Utilization</th>
<th>Entries</th>
<th>Avg Size</th>
<th>Flushes</th>
<th>Hits</th>
<th>Hits/Req</th>
<th>Reuse</th>
<th>Misses</th>
<th>Hit Rate</th>
<th>Expired</th>
<th>Pruned</th>
<th>Commit</th>
<th>Rollback</th>
<th>Saved/Hit</th>
<th>Saved/Req</th>
</tr>
</thead>
<tbody>
<tr>
<td>xotol_object_cache</td>
<td>50000000</td>
<td>394410806</td>
<td>78.88%</td>
<td>197408</td>
<td>1997</td>
<td>132589</td>
<td>8724970</td>
<td>6.1878</td>
<td>44</td>
<td>505091</td>
<td>94.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12.37ms</td>
<td>76.51ms</td>
</tr>
<tr>
<td>util_memoize</td>
<td>3000000</td>
<td>2633448</td>
<td>87.78%</td>
<td>60746</td>
<td>43</td>
<td>276082</td>
<td>51367972</td>
<td>36.4306</td>
<td>846</td>
<td>5249226</td>
<td>90.00%</td>
<td>4188745</td>
<td>287099</td>
<td>0</td>
<td>0</td>
<td>307.4μs</td>
<td>11.2ms</td>
</tr>
<tr>
<td>xo_site_nodes</td>
<td>2000000</td>
<td>1999867</td>
<td>99.99%</td>
<td>18645</td>
<td>107</td>
<td>592708</td>
<td>66450023</td>
<td>47.1270</td>
<td>3564</td>
<td>1121385</td>
<td>98.00%</td>
<td>0</td>
<td>487697</td>
<td>0</td>
<td>0</td>
<td>181.03μs</td>
<td>8.53ms</td>
</tr>
<tr>
<td>xotol_object_type_cache</td>
<td>6000000</td>
<td>4552430</td>
<td>75.87%</td>
<td>337714</td>
<td>13</td>
<td>12656</td>
<td>11891251</td>
<td>8.4334</td>
<td>35</td>
<td>1575414</td>
<td>88.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>976.93μs</td>
<td>8.24ms</td>
</tr>
<tr>
<td>gb_schema</td>
<td>2000000</td>
<td>1134670</td>
<td>56.73%</td>
<td>10816</td>
<td>104</td>
<td>399</td>
<td>2383674</td>
<td>1.6905</td>
<td>220</td>
<td>11801</td>
<td>99.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.99μs</td>
<td>3.37ms</td>
</tr>
<tr>
<td>db_cache_pool</td>
<td>4000000</td>
<td>686642</td>
<td>1.72%</td>
<td>71951</td>
<td>9</td>
<td>8130</td>
<td>2691642</td>
<td>1.9089</td>
<td>37</td>
<td>93352</td>
<td>96.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>962.67μs</td>
<td>1.84ms</td>
</tr>
<tr>
<td>lms_favorite</td>
<td>2000000</td>
<td>102124</td>
<td>5.11%</td>
<td>12774</td>
<td>7</td>
<td>42597</td>
<td>360125</td>
<td>0.2554</td>
<td>28</td>
<td>55396</td>
<td>86.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.87μs</td>
<td>732.13μs</td>
</tr>
</tbody>
</table>
Hottest Cache entries from OpenACS 5.9.1

- Hot cache entries in `util_memorize` cache:

<table>
<thead>
<tr>
<th>Key</th>
<th>Size</th>
<th>Hits</th>
<th>Expire</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lang::system::get_locales_not_cached</code></td>
<td>17</td>
<td>6039005</td>
<td>-1</td>
</tr>
<tr>
<td><code>apm_package_installed_p_not_cached ref-timezones</code></td>
<td>1</td>
<td>1291383</td>
<td>-1</td>
</tr>
<tr>
<td><code>lc_time_fmt_compile {%a, %d. %B %Y %H:%M} de DE</code></td>
<td>241</td>
<td>1009904</td>
<td>-1</td>
</tr>
<tr>
<td><code>ad_acs_version_no_cache</code></td>
<td>5</td>
<td>841339</td>
<td>-1</td>
</tr>
<tr>
<td><code>acs_lookup_magic_object_no_cache security_context_root</code></td>
<td>2</td>
<td>746132</td>
<td>-1</td>
</tr>
<tr>
<td><code>plpgsql_utility::get_function_args content_item__get_content_type</code></td>
<td>12</td>
<td>466429</td>
<td>-1</td>
</tr>
<tr>
<td><code>package_plsql_args -object_name get_content_type content_item</code></td>
<td>7</td>
<td>466429</td>
<td>-1</td>
</tr>
<tr>
<td><code>package_function_p -object_name get_content_type content_item</code></td>
<td>1</td>
<td>466429</td>
<td>-1</td>
</tr>
</tbody>
</table>

High reuse is good and bad:
- Best savings
- Potential candidate for more scalable caching forms
Improve scalability of caching (1/2)

- Reduce locking time
  - Bad on large caches (e.g. 300K+ entries): Operations iterating over every item (e.g. wild-card operations)
    
    ```
    ns_cache names ...
    util_memoize_flush_pattern ...
    ```

  - Reducing size of cache reduces locking time, when such operations are used

- Reduce number of locking operations
  - Use per-thread or per-request caches (lock-free)
  - Use more fine-granular mutexes (split caches, cache partitioning)
Split caches and cache partitioning

- Advantages:
  - Different caches use different locks, a lock on a specialized cache does not block operations on the `util_memoize` cache
  - Less irrelevant data is processed, when wild-card operations are applied to caches

- Use different caches for different purposes
e.g. separate `permission_cache` in OpenACS 5.10*

- Cache partitioning:
  use different caches based on key values (OpenACS 5.10 supports different partitioning strategies as config options)

<table>
<thead>
<tr>
<th>xotcl_object_type_cache-1</th>
<th>30000</th>
<th>18125</th>
<th>60.42%</th>
<th>1738</th>
<th>10</th>
<th>0</th>
<th>18248</th>
</tr>
</thead>
<tbody>
<tr>
<td>xotcl_object_type_cache-0</td>
<td>30000</td>
<td>11529</td>
<td>38.43%</td>
<td>980</td>
<td>11</td>
<td>0</td>
<td>3849</td>
</tr>
</tbody>
</table>
Cache Transactions (1/2)

- **Motivation:**

  ```
  db_transaction {
      # Create service contracts
      auth::authentication::create_contract
      auth::password::create_contract
      auth::registration::create_contract
      auth::get_doc::create_contract
      # ...
  }
  ```

- **Potential problems:**
  - What happens with cached values created from API-calls, when transaction fails (e.g. in third API call)
  - Consequence: cache poisoning
  - Breaking isolation
  - Misbehavior is hard to debug
Cache Transactions (2/2)

- NaviServer API support (NaviServer 4.99.16)
  
  ```
  ns_cache_transaction_begin  
  ns_cache_transaction_commit  
  ns_cache_transaction_rollback  
  ```

- Integrated with OpenACS 5.10*:
  - Automatically performed in `db_transaction` and `xo*` counterparts
  - Rollback statistics in `nsstats`:

<table>
<thead>
<tr>
<th>Cache</th>
<th>Max</th>
<th>Current</th>
<th>Utilization</th>
<th>Entries</th>
<th>Avg Size</th>
<th>Flushes</th>
<th>Hits</th>
<th>Hits/Req</th>
<th>Reuse</th>
<th>Misses</th>
<th>Hit Rate</th>
<th>Expired</th>
<th>Pruned</th>
<th>Commit</th>
<th>Rollback</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tif_lrnpretty_link_cache-4</code></td>
<td>20000</td>
<td>17689</td>
<td>88.44%</td>
<td>271</td>
<td>65</td>
<td>2811</td>
<td>2119853</td>
<td>0.7216</td>
<td>7822</td>
<td>666165</td>
<td>76.09%</td>
<td>0</td>
<td>170475</td>
<td>161786</td>
<td>0</td>
</tr>
<tr>
<td><code>xotcli_object_cache-1</code></td>
<td>250000000</td>
<td>249982370</td>
<td>99.99%</td>
<td>40241</td>
<td>6212</td>
<td>199631</td>
<td>20666286</td>
<td>7.0351</td>
<td>514</td>
<td>1731223</td>
<td>92.27%</td>
<td>0</td>
<td>1061653</td>
<td>114045</td>
<td>0</td>
</tr>
<tr>
<td><code>xotcl_object_cache-0</code></td>
<td>250000000</td>
<td>249964500</td>
<td>99.99%</td>
<td>38296</td>
<td>6527</td>
<td>200654</td>
<td>25472052</td>
<td>8.6710</td>
<td>665</td>
<td>1735624</td>
<td>93.62%</td>
<td>0</td>
<td>1125791</td>
<td>113257</td>
<td>0</td>
</tr>
<tr>
<td><code>xotcl_object_type_cache-0</code></td>
<td>7500000</td>
<td>7499985</td>
<td>100.00%</td>
<td>525119</td>
<td>14</td>
<td>389</td>
<td>13037609</td>
<td>4.4382</td>
<td>25</td>
<td>2660310</td>
<td>83.05%</td>
<td>0</td>
<td>404031</td>
<td>4510</td>
<td>1</td>
</tr>
<tr>
<td><code>xotcl_object_type_cache-1</code></td>
<td>7500000</td>
<td>7499986</td>
<td>100.00%</td>
<td>511154</td>
<td>14</td>
<td>231</td>
<td>13054981</td>
<td>4.4441</td>
<td>26</td>
<td>2782677</td>
<td>82.43%</td>
<td>0</td>
<td>325878</td>
<td>3605</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiple Threads Accessing a Mutex-protected Resource

- When mutex (e.g. t2) tries to get a lock on an already locked resource (e.g. locked by t1), the mutex has to wait until this lock is finished.
- When multiple threads try locks: high contention, increasing wait times
- Waiting time can pile up
Read/Write Locks

- Distinction in API between “Reader” and “Writer” of a resource
- Multiple concurrent readers are allowed (t*rd) without waiting => improved scalability
- A writer request has to wait until the currently active readers are finished, behaves then like a mutex, a next reader (or writer) has to wait, until writer has finished.
- On write operations a Write-Lock of a RW-Lock is more expensive than a mutex.
- RW-locks are better, when there are substantially more reader than writer requests
Performance comparison with different read/write load patterns

Best performance increase with many concurrent read operations
Busy locks with Mutex vs. RWLocks on nsv

<table>
<thead>
<tr>
<th></th>
<th>With mutex (after 24h)</th>
<th>With rwlocks (after 24h)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>locks</td>
<td>busy</td>
</tr>
<tr>
<td>nsv:3:openacs.org</td>
<td>4.71M</td>
<td>1.3K</td>
</tr>
<tr>
<td>nsv:6:openacs.org</td>
<td>4.88M</td>
<td>1.03K</td>
</tr>
<tr>
<td>nsv:2:openacs.org</td>
<td>3.37M</td>
<td>784</td>
</tr>
<tr>
<td>nsv:7:openacs.org</td>
<td>9.11M</td>
<td>755</td>
</tr>
<tr>
<td>nsv:5:openacs.org</td>
<td>2.88M</td>
<td>460</td>
</tr>
</tbody>
</table>

Substantially reduced busy operations.
Most nsv operations on OpenACS instances are read operations

Write percentage usually very little!

RWLocks used for:
- nsv (shared variables)
- URLspace (trie for managing URLs, using path segments)
- Connection channels
Numbers every developer should know

```
114    ns time {dict get {a 1 b 2 c 3} b} 100000
156    ns set x 1; time {set x} 100000
160    ns time {set x 1} 100000
163    ns set x 1; time {info exists x} 100000
204    ns time {ns_quotehtml "hello world"} 100000
209    ns time {ns_trim {hello world}} 100000
210    ns set x 1; time {expr {$x + $x}} 100000
212    ns nsv_set foo x 1; time {nsv_get foo x} 100000
235    ns proc foo {x} {return $x}; time {foo 1} 100000
269    ns time {info commands ::db_string} 100000
288    ns time {array set x {a 1 b 2 c 3}} 100000
291    ns time {ns_cache_eval ns:memozize 1 {set x 1}} 100000
303    ns nx::Class create Foo {:public method bar {} {return 0};:create ::foo1}; time {::foo1 bar} 100000
305    ns time {nsv_set foo x 1} 100000
360    ns time {ns_shal foo} 100000
513    ns ns_urlspace set -key fool /*.adp A; time {ns_urlspace get -key fool /static/test.adp} 100000
821    ns time {nx::Object create ::o} 100000
28668   ns time {md5::md5 foo} 100000
30338   ns time {shal::shal foo} 100000
78894   ns time {xo::dc get_value dbqd..qn {select title from acs_objects where object_id=179}} 100000
86132   ns time {db_string dbqd..qn {select title from acs_objects where object_id=179}} 100000
152654  ns time {set F [open /tmp/nix w]; puts $F x; close $F} 10000
2562594  ns time {exec ls /} 1000

== 2.6 ms
```

**ns_get** 25% slower than **info exists**

**nsv set** similar to **array set**

**ns_cache read** between above (0.3µs)

**ns_urlspace get** is cache *2*

... but 100x faster then DB

... REDIS cache time ~1ms

**ns_udp roundtrip**: 1ms
Motivation

```python
if {!nsv_exists ARRAY KEY} {
    #
    # Danger Zone
    #
    nsv_set ARRAY KEY DEVALUE]
}
# . . .

set oldCmds [nsv_get ARRAY KEY]
#
# Danger Zone
#
nsv_set ARRAY $newCmds
```

Race conditions!

- What happens if similar other code is executed concurrently in a different thread?
- Consequence: unreliable code
- Hard to debug
Atomic nsv operations (2/2)

- Obtain (old) value from an nsv ARRAY and set it to a new value
  (Similar to GETSET in REDIS)

  ```
  set foo [nsv_set -reset ARRAY KEY NEWVALUE]
  ```

- Obtain a value from an nsv ARRAY and unset it (no new value is provided)

  ```
  set foo [nsv_set -reset ARRAY KEY]
  ```

- Set a default value for an nsv ARRAY
  (Similar to SETNX in REDIS)

  ```
  nsv_set -default ARRAY KEY DEFAULTVALUE
  ```

- Atomic dict operations
  (Similar to “dict” in Tcl)

  ```
  nsv_dict get|set|exists|... ARRAY KEY ...
  ```
Some more selected scalability improvements

- All low level API calls in NaviServer became asynchronous
- Configurable behavior for request overruns (optional sending 503 on certain pools)
- Support of multiple driver threads (every driver thread can listen on multiple IP addresses and ports)
- Works with >1024 concurrent open connections (COVID tested)
- Logging commands with high latency

**Bandwidth management**

- Motivation: high number of bot-requests on public sites
- Bot-requests can be assigned to special request queues
- These queues can be configured
  - with max transmission rates
  - few connection threads
- Bandwidth metering of running requests
Other NaviServer improvements

- More strict input handling
  - ns_parseurl, ns_parsehostport
  - Dealing with invalid UTF-8
  - Fallback charsets

- Improve handling of external services
  - Increasingly more requests depend on external resources (e.g. cloud services)
  - Log-files for outgoing HTTP/HTTPS requests

- Misc
  - Tcl API for URLspace (with context filters 4.99.19)
  - Unit support for API and configuration (e.g. 1s, 100µs, ... 1KB, 1.5MB, ...)
  - Same collating support for Tcl as in PostgreSQL
  - Crypto support (OCSP Stapling, SNI, SCRYPT, SCRAM, key management EC, MD, HMAC, 20+ digest algorithms) through integration with OpenSSL
  - More NaviServer modules: UDP support, COAP, letsencrypt, revproxy, nsshell...

Log entry: ... invalid UTF-8: ‘xx|\xe6b|yy'
Experiment: HTTP/2 for NaviServer

Master Thesis of Philip Minić:

- Prototype version of NaviServer with HTTP/2 support
- Better performance than Apache and nginx (synthetic data, chromium)

Not done yet:
- Integration with http-client, existing API
- Pull requests for OpenSSL with HTTP/3 (QUIC) maybe included in OpenSSL 3.1
Summary

- Substantially improved concurrency
  - Cache partitioning
  - RWLocks instead of Mutex locks
  - .... Better performance means better use of existing cores

- Development bases on real-world necessities
  - Mostly OpenACS based large scale applications for us (running on year average ~100 threads, requests, background jobs, video streaming, etc. in latency sensitive applications)
  - Many development based on vulnerability scanners
  - Others have very different usage patterns
    - Zoran: windows systems, enterprise backup
    - John Buckman: coffee machine
    - ....

- Questions?
Institute for Information Systems and New Media
Welthandelsplatz 1, 1020 Vienna, Austria

UNIV.PROF. DR. Gustaf Neumann
T +43-1-313 36-4671
Gustaf.neumann@wu.ac.at
www.wu.ac.at