OpenACS and EuroTcl 2023

NaviServer 5.0

Univ.-Prof. Dr. Gustaf Neumann
Vienna University of Economics and Business
Information Systems and New Media
What’s new?
- New Licence
- Tcl9 compatibility
- Significant code changes
- New Major version number

Reforms
- Improved memory locality for ns_set
- Persistent connections for ns_http
- Removed usage of double-checking lock pattern
- Clustering

Want’s next?
- HTTP/2, HTTP/3?
- More protocols
Why a new major number?

- **NaviServer Releases:**
  - 4.99.0 ... 4.99.26
  - "Running out of fingers and toes"
    (Citation of Linus Torvalds, when Linux stepped up to 3.20)

- **New License:**
  - Upgrade from Mozilla Public License Version 1.1 + GPL
  - to Mozilla Public License (MPL) 2.0

- **Tcl 9:**
  - Lifting various restrictions (32-bit signed integers -> 64-bit)
  - Substantial code changes in NaviServer necessary to make use of new capabilities
  - Release of NaviServer 5 will be after the release of Tcl 9

- **New Features**
  - A few set of changes cherry-picked on the next slides
  - Improved crypto functionality: E.g. support for Argon2 (winner of the 2015 Password Hashing Competition, defined by RFC 9106)
  - NaviServer 5.0 works with Tcl 8.6 and Tcl 9 (regression test with GitHub workflows)

- **EOL NaviServer 4.99.*?**
  - No, bugfixes still in the 4.99 branches, leading to 4.99.27 etc
  - Many NaviServer / OpenACS user are conservative
What is an ns_set:
- NaviServer data structure for the Tcl programmer
- Like a Tcl dict, supporting duplicate keys, having names
- Predates Tcl dict significantly (before 2000)

Used for:
- HTTP header fields
- Configuration values
- SQL tuples
- ...

Example:
- SQL query, returning 20 attributes, 1000 Tuples,
  e.g.: "select * from acs_objects limit 1000"
- 43,000 malloc/free operations (1000*(3 + 20*2))
- This is for OpenACS installations a small query, many return 100K tuples or more
ns_set reform (2/3)

Improved memory locality
- Based on Tcl_DStrings
- More CPU-cache hits, improved performance
- Less memory consumption
- Less mutex locks

New:

```
name
```

```
field 1 field 2 field 3 field 4
```

```
length ... fields[]
```

**ns_set** with N elements: 3 or 4 mallocs (memory regions)
N == 4 -> 3

CPU Cache management
- Changes in pages require refetch
- Multi-threading: refetch per thread
- Especially expensive with NUMA architectures
- Memory access might differ by a factor of 5 or more
Quick test:

- Running sample query (1000 tuples a 20 attributes) in
- 1..30 threads
- Xeon Gold 6226R CPU @ 2.90GHz, 32 cores, hyper-threading enabled

Before (classical ns_set with many mallocs):

- threads 1 total 4606.787 ms avg 3285.25 ms
- threads 5 total 4595.358 ms avg 3493.07 ms
- threads 10 total 4804.193 ms avg 3755.93 ms
- threads 20 total 6279.524 ms avg 4569.16 ms
- threads 30 total 8966.427 ms avg 6618.58 ms

After reform (using one Tcl_DString per tuple):

- threads 1 total 4524.645 ms avg 3242.54 ms
- threads 5 total 4251.266 ms avg 3450.09 ms
- threads 10 total 4656.795 ms avg 3665.31 ms
- threads 20 total 5934.105 ms avg 4671.38 ms
- threads 30 total 7384.591 ms avg 5642.76 ms

E.g. with 30 threads, the total time improved by 17%.... with a smaller RSS.
What is ns_http:
- Webservers perform as a web client requests from other servers
- Cloud services, authentication, ...
- REST interfaces
- Based on low-level server streaming infrastructure
- Significantly faster than curl (esp. for high number of requests)
- HTTP client request log (similar to access.log)

What is new in NaviServer 5:
- Persistent connections
- Managing pool of connections, sharing across threads

Challenges:
- Requires strict error and parsing implementation (request pipelining)
- Handling of streaming HTML (no content length provided)
- Handling of incorrect replies
- Handling of “100 continue”
- ...
ns_http reform (2/3)
Data visualized by NaviServer nsstats module

Often significant usage (up to several 100K client requests per day)

Here: bulk synchronization via ns_http with other systems mostly over night
## Statistics per server

<table>
<thead>
<tr>
<th>Host</th>
<th>Requests</th>
<th>Avg Time</th>
<th>Sent</th>
<th>Received</th>
<th>200</th>
<th>201</th>
<th>202</th>
<th>204</th>
<th>400</th>
<th>403</th>
<th>404</th>
<th>408</th>
<th>429</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ns stats module]</td>
<td>5</td>
<td>742.46ms</td>
<td>706.82MB</td>
<td>11.06KB</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[nsstats module]</td>
<td>3974</td>
<td>128.25ms</td>
<td>1.3MB</td>
<td>3.23MB</td>
<td>2532</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1435</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>64246</td>
<td>122ms</td>
<td>19.86MB</td>
<td>294.07MB</td>
<td>64246</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>5</td>
<td>3.82s</td>
<td>1.46KB</td>
<td>3.87MB</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>8275</td>
<td>232.25ms</td>
<td>17.53MB</td>
<td>63.53MB</td>
<td>8274</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>3</td>
<td>189.84ms</td>
<td>5.37KB</td>
<td>2.2KB</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>24</td>
<td>176.96ms</td>
<td>9KB</td>
<td>58.04KB</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>1</td>
<td>594.84ms</td>
<td>375B</td>
<td>1.66KB</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>136</td>
<td>231.7ms</td>
<td>11.09MB</td>
<td>72.48MB</td>
<td>67</td>
<td>23</td>
<td>46</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>190</td>
<td>6.21ms</td>
<td>31.54KB</td>
<td>2.03MB</td>
<td>190</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[ns stats module]</td>
<td>483</td>
<td>107.89ms</td>
<td>60.86KB</td>
<td>595.25KB</td>
<td>482</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Performance**

**Amount of Data**

**Status codes**

External servers often source of sudden performance bottlenecks
Double-Checking Lock Pattern
- Goal: reduce the overhead of acquiring locks
- Testing the locking criterion before acquiring the lock.

The Problem:
- The pattern assumes a total store order (TSO), or the usage of "fences" (insert assembly)
- In some language/hardware combinations, the pattern is unsafe (RISC-V has per default a weak memory order)
- On x86: TSO, pattern is safe.

Newer architectures do aggressive optimizations, such as
1) compiler reordering instructions,
2) hardware reordering instructions,
3) cache coherency

NaviServer:
- Two major variants of the double-checking lock pattern:
  1. start-up initialization
  2. lazy initialization of heap data (actually values kept for mutexes/locks, etc.)
- Case 1: a posix/windows call can be used (pthread_once(), InitOnceExecuteOnce())
- Case 2: requires more rewriting, lazy programming style.

```c
/*
 * Core one-time server initialization to add a few Tcl_Obj * types. These calls cannot be in NsTclInit above because
 * Tcl is not fully initialized at libnss load time.
 */

if (!initialized) {
    Ns_MasterLock();
}

if (!initialized) {
    Tcl_Obj *tmpObj = Tcl_NewIntObj(0);

    NS_intTypePtr = tmpObj->typePtr;
    Tcl_DecrRefCount(tmpObj);

    NsTclInitQueueType();
    NsTclInitAddrType();
    NsTclInitTimeType();
    NsTclInitMemUnitType();
    NsTclInitKeylistType();

    initialized = NS_TRUE;
}

Ns_MasterUnlock();
```
Large Scale NaviServer Configurations

- NaviServer provides detailed statistics, such as:
  - Mutex/RWLock statistics (see conference last year)
  - Requests (per connection pool)
  - Cache (requests, hits, flushes, savings, ...)
  - Database (per DB pool, statements, performance, ...)
  - ...

- OpenACS 5.10.1 has no cluster management:
  - Up to 5.10.1: static configuration, based on IP addresses
  - Not feasible for e.g. cloud operations
  - In 5.10.1: dynamic cluster configuration:
    - Additional cluster nodes can be registered/deregistered
    - Cluster join control via cluster secret

- Various trade-offs:
  - When DB and NaviServer are on the same machine
    - Communication with DB is fast
    - Maintaining cache coherency is relatively simple (all in one NaviServer instance)
    - NaviServer is excellent in making use of a high number of cores
  - But
    - What if this reaches limits?
    - Machines with many cores are still quite expensive
    - Can the throughput be doubled?
    - What are the consequences on response times (also on idle systems)?
Performance differences:
NaviServer and DB on the same or different VMs

- Common pattern: Database Server
- For cluster setups, DB is typically on an own VM
- Performance implications depend on application (e.g. how many SQL statements/request, cost of SQL requests)
- Network latency of assign 10 ms can cause throughput decrease by a factor of 20 based on pgbench, (see: https://www.cybertec-postgresql.com/en/postgresql-network-latency-does-make-a-big-difference/)
Empirical data from 3 sample OpenACS installations

<table>
<thead>
<tr>
<th></th>
<th>openacs.org</th>
<th>server1</th>
<th>server2</th>
</tr>
</thead>
<tbody>
<tr>
<td>requests</td>
<td>448,029</td>
<td>7,642,282</td>
<td>4,308,318</td>
</tr>
<tr>
<td>response time/req (ms)</td>
<td>6.27</td>
<td>118.84</td>
<td>90.15</td>
</tr>
<tr>
<td>cache saving/req</td>
<td>5.3</td>
<td>69.45</td>
<td>113.72</td>
</tr>
<tr>
<td>cache flushes/req</td>
<td>0.0001</td>
<td>0.2406</td>
<td>1.3901</td>
</tr>
<tr>
<td># SQL statements/req</td>
<td>3.93</td>
<td>38.66</td>
<td>22.33</td>
</tr>
<tr>
<td>SQL time/req</td>
<td>2.05</td>
<td>28.32</td>
<td>43.07</td>
</tr>
</tbody>
</table>

- Data collected when running servers over 4 days
- “server1” and “server2” are large sites, serving per day 1 mio requests or more
- Significant database use (server1: ~38 SQL statements per request, server2: ~22)
- Very few cache invalidations per request on OpenACS.org, very high on “server2”
Difference in response time and performance when running SQL server on a different VM

<table>
<thead>
<tr>
<th></th>
<th>openacs.org</th>
<th>server1</th>
<th>server2</th>
</tr>
</thead>
<tbody>
<tr>
<td>response time</td>
<td>6.27</td>
<td>118.84</td>
<td>90.15</td>
</tr>
<tr>
<td>remote SQL</td>
<td>8.32</td>
<td>147.16</td>
<td>133.22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>openacs.org</th>
<th>server1</th>
<th>server2</th>
</tr>
</thead>
<tbody>
<tr>
<td>max throughput (reqs/s)</td>
<td>7022</td>
<td>505</td>
<td>666</td>
</tr>
<tr>
<td>remote SQL</td>
<td>5291</td>
<td>408</td>
<td>450</td>
</tr>
</tbody>
</table>

- **Assumption:**
  - remote SQL causes double latency per SQL statement (factor of 2)
  - For your applications: always best to measure, depends on local/cloud environment, etc.

- Average response time for openacs.org dropped by 30%, but still, it is fast enough, we are far from requiring max throughput.
- Drop of max throughput for “server1” and “server2” might be sometimes already an issue, but probably, still OK
Performance differences: NaviServer Cluster

- Example: 3 Nodes
- Database on a separate server

- For cache coherency:
  - Requires intra-cluster communication
  - Via HTTP/HTTPS/UDP/COAP built-in in NaviServer
  - Persistent connections handy and preferable
  - Requires updated applications, using “clusterwide” flush operations

- Alternatively:
  - Avoid caching
  - Setting parameter “cachingmode” to “none”
  - Avoids most of intra-cluster communications with its overhead
  - But base performance degrades
Performance implications for sample OpenACS installations

- **Request Latency Comparison:** comparing
  - Single server
  - Caching/no caching
  - Local SQL/remote SQL
  - Cluster nodes with 30 threads each
  - Cluster configuration with 4 nodes
  - Cluster configuration with 8 nodes

- **Observation:**
  - “server1” per-request performance drops most, when caching is deactivated (factor of 2.2)
  - Per-request performance of base configuration (DB + server on the same machine is best)
  - Caching benefits outweigh intra-cluster communication overhead
Throughput implications for sample OpenACS installations

- **Throughput Observations:**
  - With cluster “no cache” configurations, throughput of “openacs.org” and “server1” is already higher with 4 nodes.
  - Throughput can be doubled with 4 to 8 smaller cluster nodes

- **Additional benefit:**
  Higher availability in cluster configuration

- **Caveats:**
  - Is DB sufficiently scalable?
  - Statistics are collected from single VM installations
Experiment: HTTP/2 for NaviServer

Master Thesis of Philip Minić:

- Prototype version of NaviServer with HTTP/2 support
- Better performance than Apache and nginx with HTTP/2

Status
- Still experimental
- HTTP/3 (QUIC) is part of OpenSSL 3.1
- Still frequent changes in OpenSSL QUIC code base
- Little reason for HTTP/2 when HTTP/3 is available
Summary

- NaviServer 5
  - Overcomes many of the restrictions of NaviServer 4.99*
  - Strong integration with new Tcl 9 functionality
  - Many new features

- Learning from observation
  - Installations become more complex and distributed
  - Detailed monitoring eases
    - Configuration
    - Debugging

- Still much to do!

- 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