How we built a scalable micro-service application
- lessons learned & tooling -

Nikolas Herbst,
Jóakim von Kistowski, Simon Eismann, André Bauer,
Norbert Schmitt, Johannes Grohmann, Marwin Züfle,
Samuel Kounev

ScrumScale Workshop, Oslo, Norway
June 5, 2018

Slides available: descartes.tools
We are

Chair of Software Engineering (a.k.a. Descartes Research Group) at the University of Würzburg, Germany, Franconia (part of Bavaria)

- Performance Modeling and Benchmarking, Data Center Resource Management, Self-Aware Computing, Data Analytics
- New: IoT, CPS, I4.0, Block chain, Ethical hacking, …
On my research

- Started research after diploma in 2012 at Karlsruhe Institute of Technology (KIT)

- Research Interests:
  - Cloud Computing
  - Elasticity and Scalability
  - Auto-Scaler Benchmarking
  - Forecasting
  - ...
Mission Statement

- Provide a **platform for collaborative research efforts** in the area of quantitative system evaluation and analysis
- Foster interactions and **collaborations** between industry and academia
- **Scope**: computer benchmarking, performance evaluation, and experimental system analysis
- **Focus on** standard scenarios, metrics, benchmarks, analysis methodologies and tools

**Working groups:**

Cloud, DevOps Perf., Power, IDS & Security, Big Data

Find more information on: [http://research.spec.org](http://research.spec.org)
Why TeaStore? Our Motivation

Auto-Scaling and Placement

- Placement at run-time

Performance Modeling

- An approach for the auto-scaling + placement problem
- Build or extract model
- Use Model for placement decision
Requirements for a Reference Application

- Highly scalable
- Deployment flexibility at run-time
- Reproducible performance results
- Complex performance behavior
- Failover and reliable
- Online monitoring
- Load Profiles for realistic stress
- Simple setup
- Modern technology stack
The Descartes TeaStore

Micro-Service test application

- Five Services + Registry
- Uses Netflix “Ribbon” client-side load balancer
  - Swarm/Kubernetes supported, not required
- Pre-instrumented version with Kieker application monitoring
- Docker Images
  - Alternatively: manual deployment in application server (documentation available)
How we built a scalable micro-service application

Nikolas Herbst
Services II

PersistenceProvider
- Encapsulates DB
- Caching + cache coherence
- Memory

ImageProvider
- Loads images from HDD
- 6 cache implementations
- Memory + Storage

Recommender
- Recommends products based on history
- 4 different algorithms
- Memory or CPU

TraceRepository
- AMQP Server
- Collects traces from all services
# TeaStore Demo

## Open Source – Apache License v2

[https://github.com/DescartesResearch/TeaStore](https://github.com/DescartesResearch/TeaStore)

---

<table>
<thead>
<tr>
<th>Black Tea</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earl Grey (loose)</td>
<td>$75.82</td>
</tr>
<tr>
<td>Assam (loose)</td>
<td>$21.87</td>
</tr>
<tr>
<td>Darjeeling (loose)</td>
<td>$55.69</td>
</tr>
<tr>
<td>Anatolian Assam</td>
<td>$106.01</td>
</tr>
<tr>
<td>Earl Grey (20 bags)</td>
<td>$97.00</td>
</tr>
<tr>
<td>Assam (20 bags)</td>
<td>$119.49</td>
</tr>
<tr>
<td>Darjeeling (20 bags)</td>
<td>$107.86</td>
</tr>
<tr>
<td>Assam with Ginger</td>
<td>$93.98</td>
</tr>
<tr>
<td>Darjeeling (20 bags)</td>
<td>$101.85</td>
</tr>
<tr>
<td>Assam (loose), v1</td>
<td>$106.55</td>
</tr>
<tr>
<td>Darjeeling (loose)</td>
<td>$35.23</td>
</tr>
<tr>
<td>Anatolian Assam</td>
<td>$40.73</td>
</tr>
<tr>
<td>Earl Grey (20 bags)</td>
<td>$25.10</td>
</tr>
<tr>
<td>Assam (20 bags), v1</td>
<td>$61.12</td>
</tr>
<tr>
<td>Darjeeling (20 bags)</td>
<td>$41.12</td>
</tr>
</tbody>
</table>

---

How we built a scalable micro-service application

Nikolas Herbst
Performance: Characteristics & Configurations

Two types of caches
- Black-box persistence cache
- White-box image provider cache

Different load types
- CPU
- I/O
- Network

Internal state
- Database size influences resource demands

Load independent tasks
- Periodic recommender retraining (optional)

Startup behavior
- Auth and WebUI start “instantly”
- Recommender needs training on startup
- Image Provider creates images on startup

Configuration options
- Recommender algorithms
- Recommender retraining interval
- Image Provider cache implementations
- Database size
Load and Usage Profile

HTTP load generator

Supports load intensity profiles
- Can be created manually
- Or using LIMBO (more later)

Scriptable user behavior
- Uses LUA scripting language
- e.g. “Browse” Profile on Github

Example load intensity profile:

“Browse” user profile:
Does it scale?

First stress tests:

- **Very limited scalability** due to communication overhead!
- Image provider service was network bound (no caching)
- All services: running out of ports and connections due to standard Java networking (connections, sockets)

→ Okay, let us reuse connections via connection pooling
→ Introduce image caching (service instance & client side)
Second version stress test:

- Somewhat better scalability, still not sufficient
- Performance variability
- Connection pool size configuration important, but specific for service type, platform and load

→ not a good idea to set a default in a service container image

→ Okay, think and re-implement one more time…
Third version towards scalability:

- Asynchronous communication

- Based on Java NIO APIs (multi-plexed, non-blocking I/O)
  - Leverages network card HW features
  - Managed buffers, worker and thread pools
  - Channel listener concept for Java servelets

Frameworks: Undertow (JBoss) or Grizzly NIO (Glassfish)

https://javaee.github.io/grizzly/
Does it scale? (IV)

Up to 9 Servers a 8 physical cores (16 with HT)

→ almost 7 000 req/s – linear (8\textsuperscript{th} server had old OS version)

Nikolas Herbst
Example: Energy Efficiency of Placements

Placement 1

16 cores
- Web UI
- Auth
- Recomm.
- Img
- Persist.

8 cores
- Web UI
- Auth
- Recomm.
- Img
- Persist.

<table>
<thead>
<tr>
<th>Max</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011.9 Tr/s</td>
<td></td>
</tr>
<tr>
<td>179.6 W</td>
<td></td>
</tr>
<tr>
<td>4.4 Tr/J</td>
<td></td>
</tr>
</tbody>
</table>

Placement 2

16 cores
- Web UI
- Auth
- Recomm.
- Img
- Persist.

8 cores
- Web UI
- Auth
- Recomm.
- Img
- Persist.

<table>
<thead>
<tr>
<th>Max</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1067.7 Tr/s</td>
<td></td>
</tr>
<tr>
<td>187.0 W</td>
<td></td>
</tr>
<tr>
<td>4.3 Tr/J</td>
<td></td>
</tr>
</tbody>
</table>

How we built a scalable micro-service application

Nikolas Herbst
How we built a scalable micro-service application

Nikolas Herbst
Auto-Scaling TeaStore

How we built a scalable micro-service application

Nikolas Herbst

Proactive provisioning

Few SLO violations
Load Profile Models

LIMBO

http://descartes.tools/limbo
Load Profile Description

- **Seasonal**
- **Trends & Breaks**
- **Overlaying Seasonal**
- **Burst**
- **Noise**

Workload Units vs. Time:

- + / x
- + / x
- + / x
- + / x
- + / x
- + / x
- + / x

How we built a scalable micro-service application

Nikolas Herbst
How we built a scalable micro-service application

Nikolas Herbst
How we built a scalable micro-service application

Nikolas Herbst

LIMBO toolkit

http://descartes.tools/limbo

extractors

(time series generation

EMF editor

wizard

plotter

LIMBO DEMO

example.project.exemple.dlim

exampleModel.dlim

example.dlim

Resource Set

platform/resource/example_project

Sequence example

Combinator ADD

Time Dependent Function

Constant 10.0

Selection

Parent

List

Tree

Tree Width

Property

Value

arrival rates

117.28

example

0.0; 0.1; 0.5; 1.0; 1.5; 2.0; 2.5; 3.0; 3.167; 3.333; 3.625; 3.75; 4.0; 4.1; 4.2; 4.3; 4.4; 4.5; 4.6; 4.7

0

0.0

10.0

48.0

arrival rates

http://descartes.tools/limbo
Forecasting the future workload

TELESCOPE

Released in May 2018 as R package on Github
How we built a scalable micro-service application

Nikolas Herbst
Preprocessing

Frequency Estimation:
- Periodograms for rough estimation
- List of common frequencies

Anomaly Detection:
- Generalized extreme studentized deviate test (ESD) on the remainder
- Replace anomaly by mean of non-anomaly neighbors
2 Learning Categorical Information

- Create Feature Space
- K-Means Clustering
- Calculate Characteristics
- Cluster Labels
- ANN Forecast of Cluster Labels

How we built a scalable micro-service application
Nikolas Herbst
How we built a scalable micro-service application

Nikolas Herbst
Estimating Decomposition Type

STL once on original and once on logarithmized time series

Calculate:

- Sum of squares of the auto-correlation on remainder
- Range between first and third quantile of the remainder
- Sum of squares of the remainder

Majority decision
Decomposition & Forecasting

Learning of Categorical Information

Cluster Label Forecast

Season Forecast

Trend Forecast

Final Forecast

Time Series History

STL Decomposition

How we built a scalable micro-service application

Nikolas Herbst
Example: IBM Trace

<table>
<thead>
<tr>
<th>Forecaster</th>
<th>MASE</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telescope</td>
<td>0.842</td>
<td>6.248</td>
</tr>
<tr>
<td>tBATS</td>
<td>4.547</td>
<td>33.360</td>
</tr>
<tr>
<td>SVM</td>
<td>6.557</td>
<td>2.344</td>
</tr>
<tr>
<td>XGBoost</td>
<td>7.683</td>
<td>0.172</td>
</tr>
<tr>
<td>ARIMA</td>
<td>7.828</td>
<td>87.016</td>
</tr>
<tr>
<td>ANN</td>
<td>18.678</td>
<td>10.938</td>
</tr>
<tr>
<td>ETS</td>
<td>23.389</td>
<td>0.984</td>
</tr>
</tbody>
</table>

Start of the horizon
- Original values
- Telescope forecast
- tBATS forecast

How we built a scalable micro-service application
Nikolas Herbst
Example: Airline Passengers Trace

![Graph showing passenger data over time with forecasts from different models]

<table>
<thead>
<tr>
<th>Forecaster</th>
<th>MASE</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telescope</td>
<td>0.353</td>
<td>1.671</td>
</tr>
<tr>
<td>tBATS</td>
<td>0.520</td>
<td>11.641</td>
</tr>
<tr>
<td>ARIMA</td>
<td>0.638</td>
<td>3.248</td>
</tr>
<tr>
<td>ETS</td>
<td>0.652</td>
<td>2.266</td>
</tr>
<tr>
<td>ANN</td>
<td>0.711</td>
<td>0.375</td>
</tr>
<tr>
<td>XGBoost</td>
<td>1.261</td>
<td>0.102</td>
</tr>
<tr>
<td>SVM</td>
<td>6.758</td>
<td>0.094</td>
</tr>
</tbody>
</table>

Start of the horizon
- Original values
- Telescope forecast
- tBATS forecast

How we built a scalable micro-service application

Nikolas Herbst
### Measures for 56 Time Series

- High and stable accuracy for multi-step forecasting
- Comparably short time-to-result

<table>
<thead>
<tr>
<th>Forecaster</th>
<th>$\bar{\text{MASE}}$</th>
<th>$\sigma \text{MASE}$</th>
<th>$\bar{\text{MAPE}}$</th>
<th>$\bar{\text{Time}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telescope</td>
<td>1.503</td>
<td>1.619</td>
<td>25.217</td>
<td>9.032</td>
</tr>
<tr>
<td>tBATS</td>
<td>1.791</td>
<td>3.112</td>
<td>25.107</td>
<td>56.334</td>
</tr>
<tr>
<td>ARIMA</td>
<td>2.022</td>
<td>2.405</td>
<td>43.194</td>
<td>177.288</td>
</tr>
<tr>
<td>ANN</td>
<td>2.072</td>
<td>3.206</td>
<td>67.176</td>
<td>77.948</td>
</tr>
<tr>
<td>XGBoost</td>
<td>2.251</td>
<td>2.017</td>
<td>47.779</td>
<td>0.167</td>
</tr>
<tr>
<td>ETS</td>
<td>2.638</td>
<td>4.288</td>
<td>81.816</td>
<td>2.184</td>
</tr>
<tr>
<td>SVM</td>
<td>5.334</td>
<td>6.254</td>
<td>64.306</td>
<td>24.608</td>
</tr>
</tbody>
</table>
Estimating Resource Demands

LIBREDE

“A resource demand is the time a unit of work (e.g., request or internal action) spends obtaining service from a resource (e.g., CPU or hard disk) in a system.” S. Spinner 2015
How to quantify resource demands?

Direct Measurement

Requires specialized infrastructure to monitor low-level statistics.

Examples:
- TimerMeter [Kuperberg09] + ByCounter [Kuperberg08]
- Brunnert et al. [Brunnert13]
- Magpie [Barham04]

Statistical Estimation

Use of statistical techniques on high-level monitoring statistics.

Examples:
- Linear regression [Kraft09]
- Kalman filtering [Wang12]
- Nonlinear optimization [Kumar09]
- Maximum likelihood estimation [Kraft09]
Why should I use statistical estimation?

Direct measurements infeasible
- Only aggregate resource usage statistics available
- Unaccounted work in system or background threads

Direct measurements too expensive
- Monitoring of production system
- Heterogeneous software stacks

Coarse-grained models
- Trade-off analysis speed vs. prediction accuracy
- Usage of performance models at system runtime
Challenges

- Varying Robustness
- Computational Complexity
- Implementations not available
- Approximation Techniques
- Linear Regression
- Kalman Filter
- Nonlinear Optimization
- Maximum Likelihood Estimation
- and many more approaches...

What is the best approach for a given scenario?
LibReDE Usage

Standalone version for offline analysis

Measurement traces

or

Estimated Demands

Java library for online analysis

Monitoring tools

Custom application

Nikolas Herbst
How we built a scalable micro-service application

Estimation Process

Create estimation model
• EMF-based model
• Graphical eclipse editor

Setup estimation approaches
• Derive estimation problem(s)
• Check pre-conditions

Load monitoring data

Validation Sets
Run estimation approach(es)
Cross-Validation
Evaluate accuracy

Output results
Estimation

- 6 estimation approaches
- Extension point
- Time interval settings
- Parameters of underlying statistical techniques
If you can, build your application from micro-services with restful interfaces

- Flexibility, portability of containers
- Maintainability, reusability

Netflix offers a state of the art software stack

- Netflix Eureka service registry
- Netflix Ribbon service load-balancer with reliability features

Asynchronous communication frameworks in high demand

- E.g. Java NIO implementations: JBoss Undertow or Glassfish Grizzly
Thank You!
https://github.com/DescartesResearch/TeaStore

Contact:

nikolas.herbst@uni-wuerzburg.de
https://go.uni-wuerzburg.de/herbst