How we built a highly scalable Machine Learning platform using Apache Mesos

Daniel Sârbe
Development Manager, BigData and Cloud Machine Translation @ SDL
Co-founder of BigData/DataScience Meetup Cluj, Romania

October 2017
MesosCon Europe
Agenda

1. Some Machine Translation context
2. Why we needed a new platform?
3. Architecture overview of our current SaaS solution
4. New platform – using micro-services
5. Lesson Learned
6. Demo – Scaling micro-services to handle traffic increase
7. Q&A
@danielsarbe

- Software development background (13+ years)
- Passionate about people and technology
- Interest in anything that is related to Scalability, BigData, Machine Learning
- Currently leading the BigData and Cloud Machine Translation group at SDL Cluj, Romania
- Co-founder of BigData/DataScience Meetup Cluj
  - 1200+ members
  - 25+ events organized
    - meetups with more 100+ participants
    - workshops with 30 people
Machine Translation quality improvements

Technological Progress

2000  2005  2015
Machine Translation quality improvements

Technological Progress

2000  2005  2015

Customization

Generic  Industry Vertical  Custom Trained
On average, customized engines handled industry terminology 24% better than standard genetic MT.
Adaptive Machine Translation idea

Machine translation

Machine learns during the statistical training process

Machine does not learn or improve during the translation process

Machine translation + AdaptiveMT

Machine learns during the statistical training process

Machine learns from user feedback during the translation process
Learning from Post-Edits

Update operation

Adaptive MT Model

Source
MT Output
Post-Edit

Translator
Request Translation
Machine Translation
MT Post-edits
An update of one of the statistical MT models, the translation model

**Source**
No further requirements are needed

**MT**
Pas d'autres exigences sont requises

**PE**
Aucune exigence supplémentaire n’est nécessaire
Update Example: Translation Model Adaptation

No further requirements are needed
Pas d'autres exigences sont requises
Aucune exigence supplémentaire n’est nécessaire

<table>
<thead>
<tr>
<th>Good New Translations</th>
<th>Bad New Translations</th>
</tr>
</thead>
<tbody>
<tr>
<td>needed -&gt; <em>requises</em> nécessaire</td>
<td>are -&gt; n’est</td>
</tr>
<tr>
<td>further -&gt; <em>autres</em> supplémentaire</td>
<td>requirements -&gt; exigence</td>
</tr>
<tr>
<td>no further requirements are -&gt; aucune</td>
<td>no -&gt; aucune</td>
</tr>
<tr>
<td>exigence supplémentaire n’est</td>
<td></td>
</tr>
</tbody>
</table>

- Statistical features help choose good rules, and decide when to use them
To PE 300 segments with AdaptiveMT, we need 250 fewer edits than without AdaptiveMT.
Second use-case – Neural MT

- **Rule-based**
  - define and build the model by hand

- **Traditional SMT**
  - define the model by hand then statistically learn it from data

- **Neural MT**
  - design architecture to automatically discover, define, and learn the models from data
Neural MT

- Uses a deep learning architecture capable of learning the meaning of the text
  - fluent and naturally sounding translation output
- Neural MT shows significant translation quality improvement over SMT
  - captures both local and global dependencies and can handle longrange word reordering
  - e.g. we observe an impressive 30% improvement on English-German
Neural MT in the Cloud

To accommodate NMT in the Cloud we need:

- new hardware: GPUs
- flexible infrastructure (new&old engines)
- break the old implementation (independent services)
- new modern API (for new clients onboarding)
How can we do this?
What we had before?

Our Legacy SaaS solution

Mature, Iteratively-developed platform

>15 billion words translated in average month
>200 million translation request/month

No P1/P2 Bugs in last 24 months
Availability: 99.9x

The only large-scale, commercial-grade MT solution other than Google and Microsoft
Translation engines are not modular and is difficult to add new functionality

Redundant flows and services based on outdated requirements

Scaling up-down requires manual intervention and allocation of new VMs

Overall, monolithic design that is hard to adopt for new use-cases
A new platform
What do we want to achieve? – Key Concepts

• Scalability
• Latency
• Independent (micro-)services
• Elasticity (auto-scaling)
• Fault-tolerance & robustness
• Infrastructure automation
• Reliable monitoring and alerts
Architecture evolution

• eBay
  • 5th generation today
  • Monolithic Perl → Monolithic C++ → Java → microservices
• Twitter
  • 3rd generation today
  • Monolithic Rails → JS / Rails / Scala → microservices
• Amazon
  • Nth generation today
  • Monolithic C++ → Java / Scala → microservices
• SDL MT
  • 3rd generation today
  • Monolithic Rails -> Monolithic Java -> microservices
## Our Technology Stack – New microservices platform

<table>
<thead>
<tr>
<th><strong>Mesos</strong></th>
<th>Cluster manager</th>
<th><strong>Scaling</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Marathon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>NoSQL, column-oriented db</td>
<td></td>
</tr>
<tr>
<td>Hadoop</td>
<td>Storage</td>
<td></td>
</tr>
<tr>
<td><strong>Kafka</strong></td>
<td>Messaging system</td>
<td><strong>Latency, Fault-tolerance</strong></td>
</tr>
<tr>
<td>Zookeeper</td>
<td>Centralized coordination service</td>
<td></td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>Serializing structured data – developed by Google</td>
<td></td>
</tr>
<tr>
<td>Ansible</td>
<td>IT automation</td>
<td></td>
</tr>
<tr>
<td>AWS</td>
<td>Cloud Computing Services</td>
<td></td>
</tr>
</tbody>
</table>

**ELK stack**
- Log aggregation, search server (indexing & querying logs)

**Grafana**
- Beautiful metric & analytic dashboards

**OpenTSDB**
- Real-time metrics

**Zabbix**
- Monitoring solution

**Docker**
- Containerization platform

**DropWizard**
- REST application bootstrap framework

**SpringBoot**
- Programming language

**Java8**
- Programming language

**Ansible**
- IT automation

**AWS**
- Cloud Computing Services
Lessons Learned

“No regrets in life. Just lesson learned.”
1. Cost efficient

• Dev/QA/Clone clusters – ~40% cost
  – issues found only in aws-qa
  – prod clone has the same IPs/confs as prod

• AWS
  – periodical cleanup
  – email alerts on no of instances running
  – r3.4xlarge -> r4.4xlarge ($1.33/h -> $1.06/h -no ephemeral 320SSD)
  – ElasticBlockStore(EBS) vs ElasticFileSystem(EFS)
  – reserved instances
2. Security

- ssh via a single aws bastion machine
- gpg encryption of confs
  - no clear passwords in git
  - restrict access to specific envs
- secure Marathon/Kibana/HAPProxyUI
- AWS termination protection
3. Platform high availability

- **Infrastructure allocation**
  - +1 node/cluster
  - one instance decommissioned/stopped (AWS EC2/human error)
- **Microservices**
  - 2 instances/micoservice
  - unique constraints should be set
- **Test with 5x-10x more traffic**
- **Early monitoring on all fronts**
  - infrastructure - Zabbix
  - app metrics - OpenTSDB
  - usage stats - ELK
  - external – Pingdom + PagerDuty
4. Resource allocation

• Memory limitations for containers
  – Marathon memory settings were not enforced on container level
  – reported container memory = host memory
  – enforce Xmx, Xms (OOM)
  – crash dumps (mount partitions to have a crash dump)

• CPU weights (Marathon)
  – reduced 1 to 0.1 – overprovision
5. Releases are not as easy as expected

- **No downtime releases**
  - simulate 2-4 times the releases in prod-clone
  - scripts to monitor downtime during deployment
  - connection draining (killed by default)
  - messages compatibility (using protoBuff)

- **Ansible-ize the manual steps**
  - prod-clone commands run on prod (gpg to fix)
  - 0 to cluster (in x min)
6. Investigations become more complex

- Logs
  - file based logs vs centralized logs
  - aggregated logs into ELK(requestId)
  - using stdout -> no disk space on mesos-slaves (disable)
  - under high load the gelf appender caused slowdowns
    - move to log4j-kafka

- Metrics in OpenTSDB
  - application-specific-metrics

- Correlation between various sources
7. Independent microservices

• Keep microservices as independent and as small as possible
  – 30+ microservices
  – a challenge, especially for legacy code (unit tests)
  – continuous refactoring for all microservices
8. Periodically reevaluate assumptions

- Follow user behavior over time
  - users behavior is different from what we expected
  - API flow changes (v2/v3 for some APIs)
  - speed is more important on some flows(sync)

- Scale the microservices based on usage
Future improvements
Future improvements

• Maintenance and evolution of the platform
  – Periodical upgrades of the stack
  – Improve monitoring

• Auto-scaling
  – based on usage patterns
  – use of aws-spot instances

• Move all components in Mesos(DC/OS)
  – HBase
  – ElasticSearch
  – Kafka
  – HDFS
Demo Time!

- Scaling micro-services to handle traffic increase
Questions?
MesosCon EUROPE