Engineering Cloud Applications 3
Scalability, Elasticity, Availability, and Resilience

Dr. Philipp Leitner

University of Zurich, Switzerland
Organisatorial

- Date for Oral Exam:

  June, 16th, 2017  (Office Prof. Gall, 2.D.02)

- Please use Doodle poll to register (first come, first served)

  http://doodle.com/poll/4s469mdwpqm5rmgt
Outline

• Basic Concepts of Scalability and Availability

• Performance and Performance Variation in Clouds

• Microservices as a Common Way to Build Cloud Apps
Foundations of Scalability and Elasticity
In order to increase the amount of resources available for an application, we may do 2 different things:

- **Variant 1:** Scale up (or: scale vertically)

- **Variant 2:** Scale out (or: scale horizontally)
Scaling up:

- Inverse: scaling down
- Advantage: easy to implement, works for many applications
- Disadvantages: often cannot be done without interruption, there are natural limits how far one can scale up (there is no infinitely fast machine)
Scaling Up and Out (2)

Scaling out:

• Inverse: scaling in

• Advantage: technical limits much higher, can be done without interruption (in well-built systems)

• Disadvantages: requires some load balancing, speedup not the same for each application
Load Balancing

• In a scaled-out system, whenever a new request comes in, it needs to be scheduled (mapped) to one of the nodes
• This request scheduling algorithm should optimally balance the load between nodes so that the average processing time for requests is minimal
Request Scheduling Algorithms

- **Generic:**
  - Round Robin
  - Random

- **With heterogeneous nodes:**
  - Weighted Random

- **Based on load monitoring:**
  - Least loaded
  - “The supermarket model”

- **Predictive**
  - Bin packing
For which systems does it make sense to scale out?

For systems that are **scalable**
Scalability

- Scalability is a measure for how much “better” a system becomes if it gets more of a given resource.

**Speedup:**

- How much faster does a system become if we add more CPU time?

- E.g., if I go from one CPU to two (identical) CPUs, and my system now takes 80% of the time to finish, the speedup is 0.4

\[
\text{Speedup} = \frac{N_{orig} \cdot t_{new}}{t_{orig} \cdot N_{new}}
\]

*e.g.*

\[
\text{Speedup} = \frac{1 \cdot 80}{100 \cdot 2} = 0.4
\]
Amdahl’s Law (1)

- Many systems have a speedup < 1
  - (the system does not scale perfectly, there are “diminishing returns” on adding more CPUs)

- Amdahl’s Law:
  \[ t_{total} = t_{serial} + \frac{t_{parallel}}{N} \]
  \[ Speedup = \frac{1}{(1 + (N - 1)a)}, \text{ with } a = \frac{t_{serial}}{t_{serial} + t_{parallel}} \]

- For a > 0, the Speedup converges against 0
Amdahl’s Law (2)

Source: https://www.cac.cornell.edu/VW/Optimization/doitfaster.aspx
Special Cases

- **Special case 1:** $a = 0$
  - The system is fully scalable, there is no serial part
  - "Embarassingly parallel" application
    \[
    \forall N \in \mathbb{N} : \text{Speedup}(N) = 1
    \]

- **Special case 2:** $a = 1$
  - The system is not parallelizable at all
  - E.g., a single-threaded application
    \[
    \forall N \in \mathbb{N} : \text{Speedup}(N) = 0
    \]
Embarassingly Parallel Problems (1)

- In practice, a large set of problems turn out to be embarassingly parallel

  - Handling Web requests (if stateless)
  - Many rendering problems
  - Rainbow table attacks on hashes
  - Optimization via genetic algorithms
  - ...

Embarassingly Parallel Problems (2)

- Characteristic of all embarassingly parallel problems:
  - Individual small work items (handling a web request, rendering a frame, etc.) can be handled **entirely independently**
  - Any node that handles a work item needs to know nothing about any other work item
  - There are no cross-references or interactions between work items

— Statelessness —
Limitation of Amdahl’s Law

• Amdahl’s law gives us a good feeling for whether a system will be scalable

• However:
  • Even an embarrassingly parallel system may have some amount of parallelization overhead
  • **Time required for data transfer between nodes**
  • (not captured at all in Amdahl’s law)
Extension of Amdahl’s Law (1)

\[ t_{total} = t_{processing} + t_{messaging} + t_{dataTransfer} \]

\[ t_{processing} = \frac{t_{parallel}}{N} + t_{serial} \]

\[ t_{messaging} = M \times \text{latency} \]

\[ t_{dataTransfer} = \frac{|M|}{\text{bandwidth}} \]

M … number of messages

|M| … total size of those messages

Source: https://www.cac.cornell.edu/VV/Optimization/estimate.aspx
Extension of Amdahl’s Law (2)

• With the previous definitions, we can assume the following model for total time incl. message transfer:

\[ t_{total} = \frac{t_{parallel}}{N} + t_{serial} + k_0 \cdot N^\alpha \cdot \text{latency} + k_1 \cdot \frac{N^\beta}{\text{bandwidth}} \]

Source: https://www.cac.cornell.edu/VV/Optimization/estimate.aspx
Application actually becomes slower at some point.
Message Queueing

- EC2 Worker Instances
- Load Balancer
- AWS System with Interactions via Java RMI

- EC2 Worker Instances
- Load Balancer
- AWS System with Interactions via Message Passing
  - AWS SQS
Advantages of Message Queueing

• When architecting a scalable system, (asynchronous) message queueing has many advantages:

  • **Loose coupling** — it is easy to add more instances to the system, as other components (e.g., the load balancer) never talk directly to instances

  • **Reduced number of open connections** — the load balancer does not need to keep connections open to all other components

  • **Fault tolerance** — individual faulty instances do not break the system
• In addition to improving performance, message passing and replication improves availability

• Availability:

  • The probability that a system is online at any point in time

\[
\text{av}(S) = 1 - \frac{\text{downtime}}{\text{uptime}}
\]
Fault Tolerance (2)

- Assume a system consisting of a frontend, an application server, and a database.

- The availability may be:
  - $av(\text{frontend}) = 0.99$
  - $av(\text{appserver}) = 0.99$
  - $av(\text{db}) = 0.99$
  - $av(\text{system}) = 0.99^3 = 0.97$

The overall system availability can be interpreted as the probability that all necessary components are online at the same time...
Fault Tolerance (3)

• Now assume we have 3 app servers (each with a bad availability of 0.9) and a cluster of 5 crappy database servers with an availability of 0.85

• The availability is now:
  • \( \text{av(frontend)} = 0.99 \)
  • \( \text{av(appserver)} = 1 - 0.1^3 = 0.999 \)
  • \( \text{av(db)} = 1 - 0.15^5 = 0.9999 \)
  • \( \text{av(system)} = 0.9889 \)

  *(this is under the assumption that one database / app server alone is able to handle a request)*
• Big cloud customers tend to be fanatic about testing for fault tolerance

• E.g., Netflix has the **Chaos Monkey**

> “Chaos Monkey is a service which runs in the Amazon Web Services (AWS) that seeks out Auto Scaling Groups (ASGs) and terminates instances (virtual machines) per group.”

http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html
Most big public clouds have data centers all around the world (regions)

(as of March 30th, 2017)
Regions and Availability Zones (2)

In each region, there are multiple (e.g., 3) availability zones. Your instances typically exist in exactly one of those.

Redundant deployments for high availability should deploy in different AZs!

Two resources in the same AZ are much more likely to fail at the same time.
Rules of Thumb for Region and AZ Selection

• Select the *region* based on:
  • Where your customers are (latency)
  • Where you want to store your data (privacy)
  • If none of those matters, choose the cheapest

• As for selecting the *AZ*:
  • If you want to reduce latency, make sure that all resources are in the same AZ
  • If you want to increase availability (e.g., have backups), make sure that they are in *different* AZs
Unpredictability in Clouds
# Coarse-Grained QoS Guarantees in Clouds

## Instance Types Matrix

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>vCPU</th>
<th>Memory (GiB)</th>
<th>Storage</th>
<th>Networking Performance</th>
<th>Physical Processor</th>
<th>Clock Speed (GHz)</th>
<th>AES-NI</th>
<th>AVX</th>
<th>Turbo</th>
<th>OPT</th>
<th>Enhanced Networking</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.micro</td>
<td>1</td>
<td>1</td>
<td>EBS Only</td>
<td>Low to Moderate</td>
<td>Intel Xeon family</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>t2.small</td>
<td>1</td>
<td>2</td>
<td>EBS Only</td>
<td>Low to Moderate</td>
<td>Intel Xeon family</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>t2.medium</td>
<td>2</td>
<td>4</td>
<td>EBS Only</td>
<td>Low to Moderate</td>
<td>Intel Xeon family</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3.75</td>
<td>1 x 4 SSD</td>
<td>Moderate</td>
<td>Intel Xeon E5-2670 v2*</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>m3.large</td>
<td>2</td>
<td>7.5</td>
<td>1 x 32 SSD</td>
<td>Moderate</td>
<td>Intel Xeon E5-2670 v2*</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>15</td>
<td>2 x 40 SSD</td>
<td>High</td>
<td>Intel Xeon E5-2670 v2*</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>8</td>
<td>30</td>
<td>2 x 80 SSD</td>
<td>High</td>
<td>Intel Xeon E5-2670 v2*</td>
<td>2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
</tbody>
</table>
Performance Unpredictability (1)

Two identical instances - very different performance
Performance Unpredictability (2)

Same instance over time - also very different performance

![Graph showing IO Bandwidth vs Measurement Runtime for two instances.](chart.png)
Two Kinds of Predictability

- **Inter-Instance Predictability**
  
  "How similar is the performance of *multiple* identical instances?"

- **Intra-Instance Predictability**
  
  "How *self-similar* is the performance of a single instance over time?"
## Inter-Instance Predictability

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>CPU-Bound</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>MEM</td>
<td>Java</td>
<td>IO</td>
<td>OLTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC2</td>
<td>t1.micro</td>
<td>12.14</td>
<td>17.67</td>
<td>30.63</td>
<td>71.33</td>
<td>30.66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m1.small</td>
<td>3.19</td>
<td>3.77</td>
<td>3.17</td>
<td>88.49</td>
<td>13.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m3.large</td>
<td>0.13</td>
<td>2.07</td>
<td>7.22</td>
<td>35.53</td>
<td>21.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c3.large</td>
<td>0.21</td>
<td>8.60</td>
<td>6.42</td>
<td>58.88</td>
<td>21.31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i2.xlarge</td>
<td>0.12</td>
<td>11.92</td>
<td>8.44</td>
<td>20.07</td>
<td>12.28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>t1.micro</td>
<td>20.28</td>
<td>26.40</td>
<td>59.32</td>
<td>70.08</td>
<td>32.18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m1.small</td>
<td>12.81</td>
<td>26.18</td>
<td>5.34</td>
<td>94.47</td>
<td>15.68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m3.large</td>
<td>0.16</td>
<td>4.46</td>
<td>9.23</td>
<td>49.02</td>
<td>37.10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCE</td>
<td>f1.micro</td>
<td>5.28</td>
<td>8.36</td>
<td></td>
<td>3.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n1-standard-1</td>
<td>2.54</td>
<td>6.99</td>
<td></td>
<td>3.36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n1-standard-2</td>
<td>1.71</td>
<td>6.96</td>
<td></td>
<td>1.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f1.micro</td>
<td>5.13</td>
<td>7.17</td>
<td>9.47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n1-standard-1</td>
<td>2.05</td>
<td>8.31</td>
<td>10.39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n1-standard-2</td>
<td>1.16</td>
<td>9.53</td>
<td>4.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Azure</td>
<td>ExtraSmall</td>
<td>18.38</td>
<td>16.88</td>
<td></td>
<td>61.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>18.23</td>
<td>8.37</td>
<td>59.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>17.81</td>
<td>11.91</td>
<td>47.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtraSmall</td>
<td>18.13</td>
<td>15.96</td>
<td></td>
<td>49.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>19.11</td>
<td>6.62</td>
<td>44.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>18.28</td>
<td>10.96</td>
<td>48.31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL</td>
<td>1 CPU / 2048 MB</td>
<td>0.11</td>
<td>6.65</td>
<td>13.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 CPUs / 4096 MB</td>
<td>0.11</td>
<td>7.14</td>
<td>6.27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Intra-Instance Predictability

Relative Standard Deviations of Benchmark Runs Within the Same Instance

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>CPU</th>
<th>MEM</th>
<th>IO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EC2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eu</td>
<td>t1.micro</td>
<td>40.17</td>
<td>40.86</td>
<td>22.78</td>
</tr>
<tr>
<td></td>
<td>m1.small</td>
<td>1.69</td>
<td>1.66</td>
<td>27.28</td>
</tr>
<tr>
<td></td>
<td>m3.large</td>
<td>0.89</td>
<td>0.49</td>
<td>17.92</td>
</tr>
<tr>
<td><strong>GCE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eu</td>
<td>f1-micro</td>
<td>3.24</td>
<td>6.85</td>
<td>2.65</td>
</tr>
<tr>
<td></td>
<td>n1-standard-1</td>
<td>0.76</td>
<td>2.43</td>
<td>4.42</td>
</tr>
<tr>
<td></td>
<td>n1-standard-2</td>
<td>0.86</td>
<td>2.01</td>
<td>1.46</td>
</tr>
<tr>
<td><strong>Azure</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eu</td>
<td>ExtraSmall</td>
<td>2.41</td>
<td>3.39</td>
<td>27.08</td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>1.65</td>
<td>2.33</td>
<td>93.47</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>1.30</td>
<td>1.98</td>
<td>0.14</td>
</tr>
<tr>
<td><strong>SL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>1 CPU / 2048 MB</td>
<td>0.22</td>
<td>0.39</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td>2 CPU / 4096 MB</td>
<td>0.13</td>
<td>0.14</td>
<td>2.25</td>
</tr>
</tbody>
</table>
Reasons

• For IO-Bound Workloads
  Multi-tenacy (noisy neighbours)

• For CPU-Bound Workloads
  Hardware Heterogeneity
# Hardware Heterogeneity

## CPU Models

*(for `ml.small` and Azure Small in North America)*

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EC2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intel E5-2650</td>
<td>1962</td>
</tr>
<tr>
<td></td>
<td>Intel E5430</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>Intel E5645</td>
<td>293</td>
</tr>
<tr>
<td></td>
<td>Intel E5507</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>Intel E5-2651</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>AMD 2218 HE</td>
<td>9</td>
</tr>
<tr>
<td><strong>Azure</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AMD 4171 HE</td>
<td>782</td>
</tr>
<tr>
<td></td>
<td>Intel E5-2673</td>
<td>595</td>
</tr>
<tr>
<td></td>
<td>Intel E5-2660</td>
<td>189</td>
</tr>
</tbody>
</table>
Basic Alleviation Strategies

• Pay
  e.g., for baremetal servers or dedicated instances
  Larger instance types also tend to be more predictable

• “Play the numbers game”
  If you scale out, individual instance performance matters less

• Benchmark-and-Discard
  Start more instances than you need, benchmark them, and keep only the ones that were the fastest for you
Microservices
Reminder: Service-Based System

Applications are built as **composition of services** (lego principle)
What Are Microservices?

- Subset / implementation strategy for service-based systems
- Sometimes called an architectural style, but has more to do with team organization than software
- Technology-agnostic
- Does *not* necessarily mean “very small services” :)}
Basic Ideas (1)

- Smart endpoints and dumb pipes
  - See also: pipes-and-filters architecture
- Conway’s Law
  - Software architecture follows organizational structure
- “Two-Pizza Rule”
  - Services can be built and operated by a team that can be fed by two large pizzas
Basic Ideas (2)

• “You build it, you run it”
  • Embrace the DevOps concept
  • Every service is run as a little product of it’s own
  • “Startup mode”

• Developer-on-call
  • Developers are on call to fix issues in production
  • There is no “hand-over” to a prod team
Conway's Law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

http://www.melconway.com/Home/Conways_Law.html
Integration Strategies

RPC services

Event-driven services

Often implemented via REST

E.g., via AWS Lambda
Basic Conceptual Service Model

Instance-Backed Microservice

Lambda-Backed Microservice

API

/ep1

/ep2

... 

Backing Instance

Data Store

API

/ep1

/ep2

... 

Lambda Function

/ep1 Lambda Function

/ep2 Lambda Function

Data Store

Elastic Loadbalancer

EC2

RDS

API Gateway

Lambda

DynamoDB
Technological Advantages (1)

• Technology-neutral
  • Every team / service chooses the stack that makes sense for them
  • “Polyglott persistence”

• Disadvantage:
  • Maintenance may be difficult
  • Knowledge transfer often hard
Technological Advantages (2)

Customer-Facing Web Interface

- Frontend:
  - /landing
  - /order
  - /login

- Customers:
  - /get
  - /save

- Order History:
  - /get
  - /save

- Recommend:
  - /generate

- Orders:
  - /get
  - /submit

- Banking Interface:
  - /process_payment

- Products:
  - /get

- Azure

- Docker

- Amazon Web Services
Technological Advantages (3)

- **Fine-Grained Elasticity**
  - Every service can scale out on its own.
  - Allows to avoid overspending

- Disadvantage:
  - Lack of global planning
  - “Friendly DoS”
Technological Advantages (4)

- Resilience and Circuit Breakers
  - Assume that any microservice may be slow or even fail at any time, and build your software accordingly.
Summary

• Scalability and elasticity is (also) a question of architecture
  • Avoid state and keep your app embarrassingly parallel
  • Redundancy helps with availability (but keep in mind that more components —> more data transfer overhead)

• Performance in cloud is unreliable
  • Inter-instance vs. intra-instance predictability

• Microservices are an architectural and organizational approach to build cloud-native applications