Last- und Performancetest verteilter Systeme mit Docker & Co

Dehla Sokenou

GEBIT Solutions

Übersicht

Worum geht es in diesem Vortrag?

Testen verteilter Systeme
Unter Verwendung von Virtualisierungsumgebungen
Virtualisiert wird das zu testende System

Worum geht es nicht?

Virtualisierung der Build- & Testumgebung
Das ist ein anderes, ebenfalls spannendes Thema

Ein kurzer Einblick ins Testen

Testen – ein paar Fakten

Testen ist teuer

- Schätzung: 20% - 70% der Gesamtentwicklungs-kosten

Tests sind pflegeaufwendig

- Änderung an der Software → Änderungen an den Testfällen

Automatisierte Tests sind besonders pflegeaufwendig

- Änderung an der Software → Änderungen an den Testfällen + Änderung an der Testimplementierung

Warum testet man dann?

Testen ist teuer – aber:

- Nicht testen ist teurer

Tests sind pflegeaufwendig – aber:

- Alternativ fließt Aufwand in Fehlerbehebung

Automatisierte Tests sind besonders pflegeaufwendig – aber:

- Je größer die Testsuite, desto mehr Sparpotential im Vergleich zu manuellen Tests
- Unterstützung von Refactorings, kurzen Releasezyklen, agiler Entwicklung

Testbarkeit sicherstellen

Testbarkeit =
Steuerbarkeit + Beobachtbarkeit

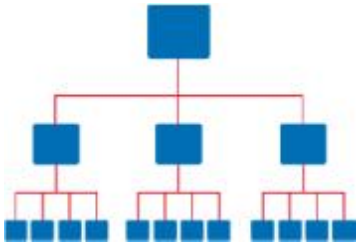
Systeme nicht unbedingt testbar → Testbarkeit muss explizit eingebaut werden

Wiederholbarkeit

- Wichtig für automatisierte Tests

Praxisbeispiel

Konfiguration-Manager



Kern:

- Serverkomponente
  - An mehreren Standorten (ggf. weltweit) verteilt
- Schnittstellen für Client-Systeme
- Web- und Rich-Client-Frontend

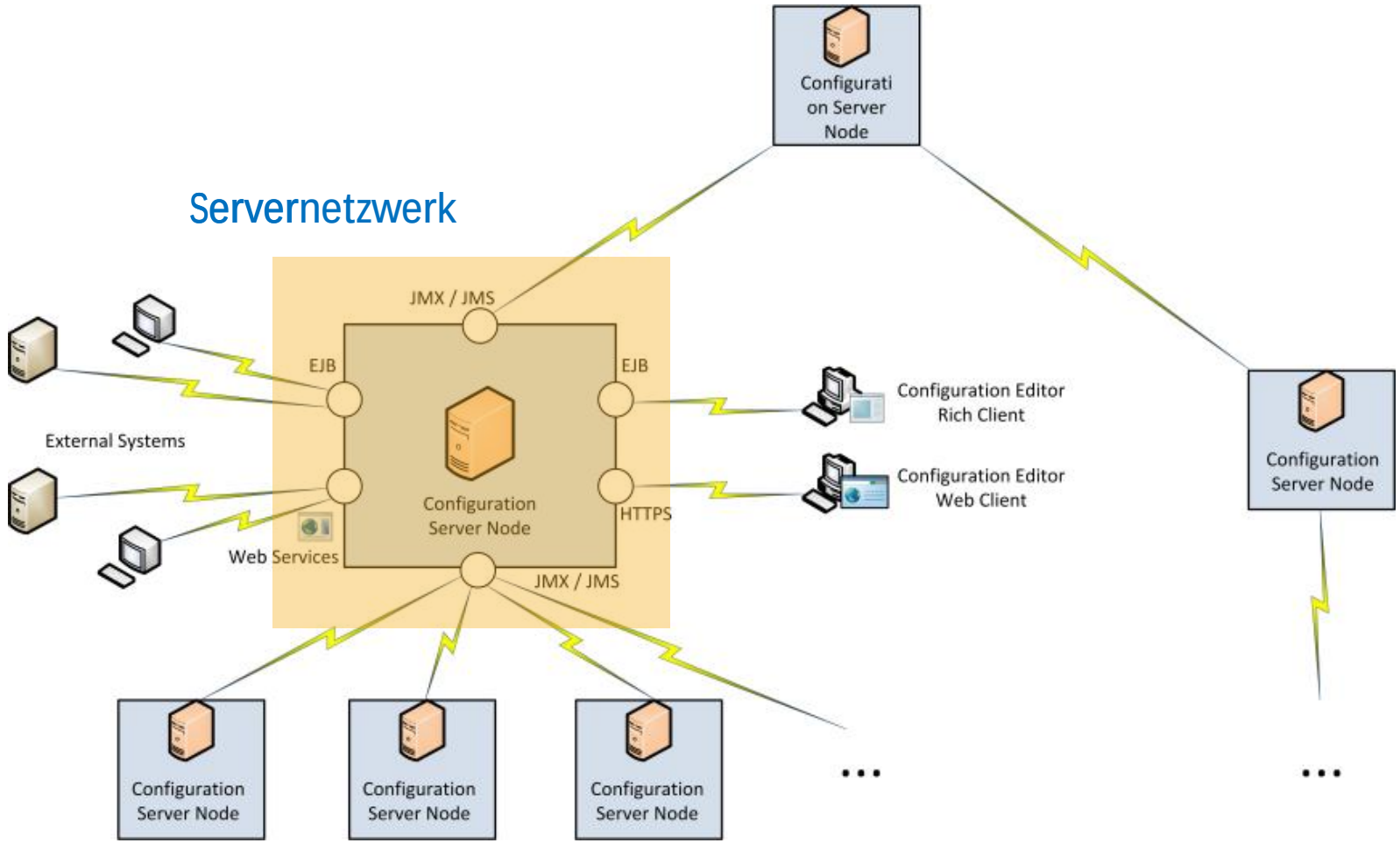
Erweiterungen:

- Kundenspezifische Addons
- Unterschiedliche Betriebskonfigurationen

Umgebung:

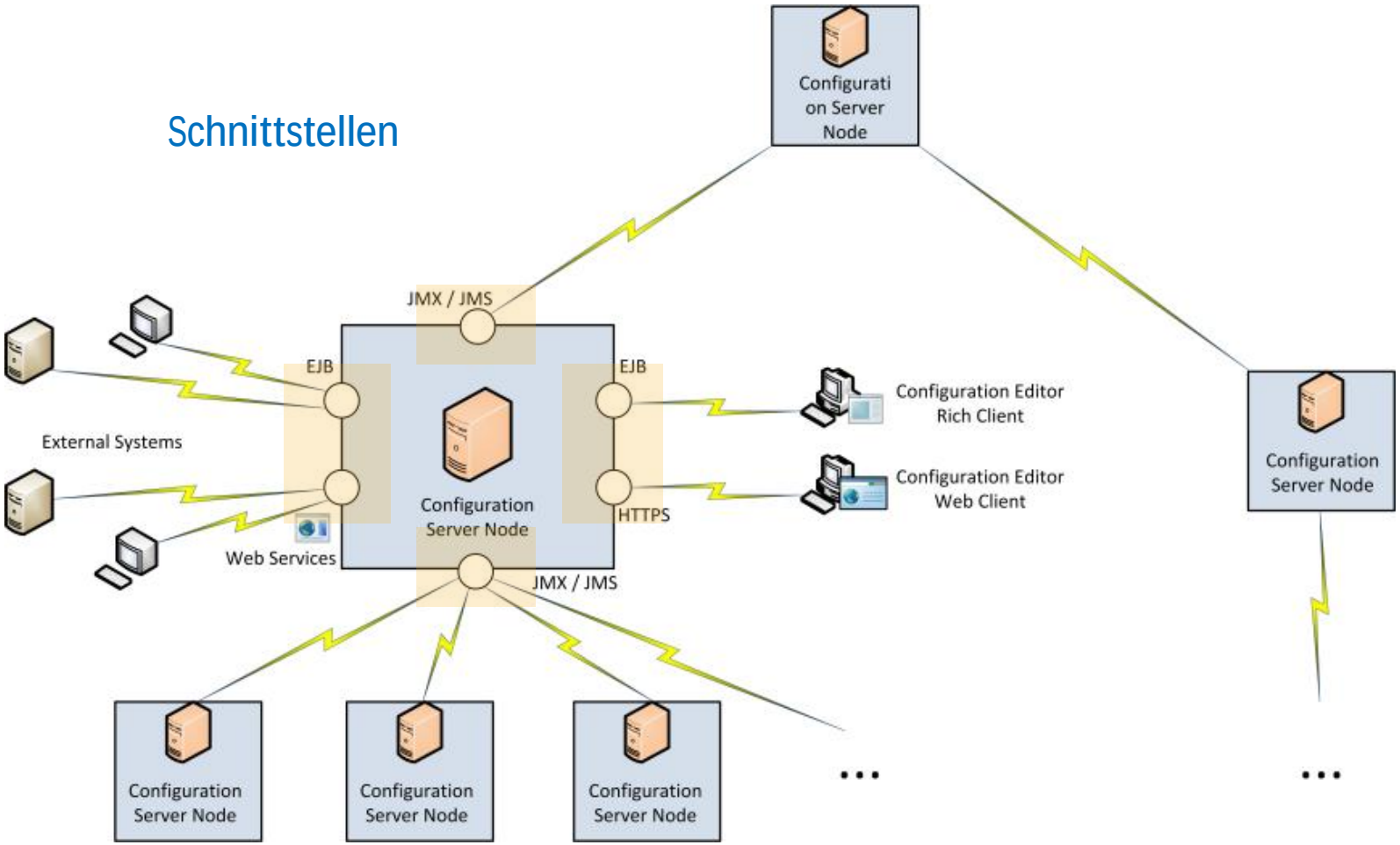
- Diverse Client-Systeme, bspw. Kassen

Konfiguration-Manager

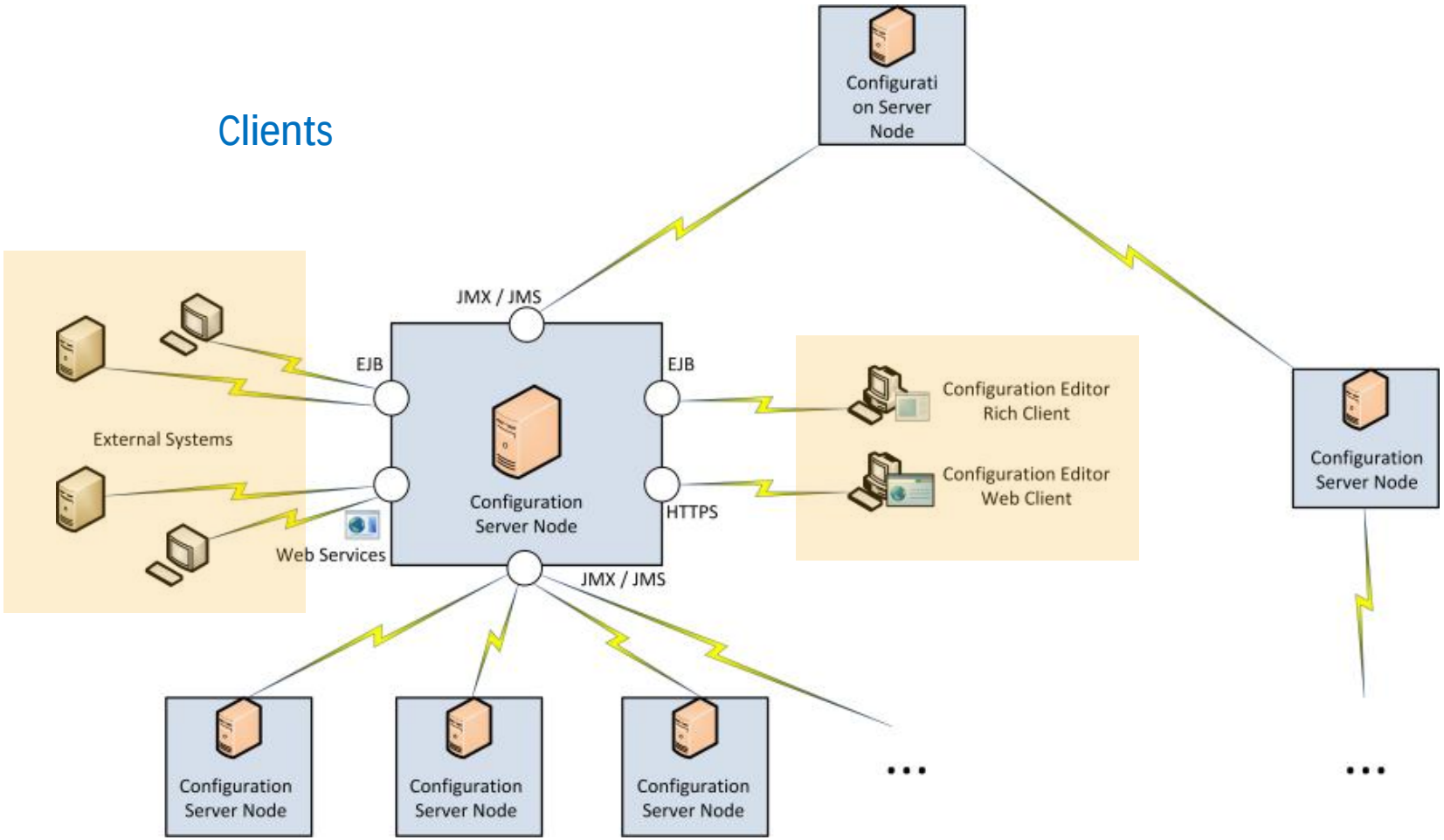


Konfiguration-Manager

Schnittstellen

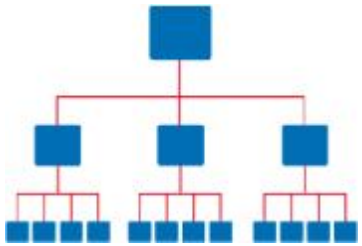


Konfiguration-Manager



Herausforderungen

- Kommunikation der Komponenten
- Simulation des weltweiten Einsatzes
- Simulation von Clients
- Berücksichtigung von Laufzeitdifferenzen
- Effiziente Verteilung und Setup der Einzel-komponenten
- Kundenspezifische Betriebskonfigurationen und Erweiterungen
  - Abgrenzung zum Produkttest

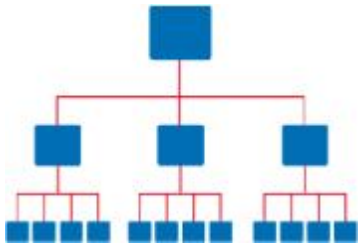


Motivation

HornetQ-Konfiguration optimieren

- Problem:
  - Viele Netzwerkausfälle aufgrund von
    - Schlechten Netzwerkverbindungen
    - Unterdimensionierter Umgebung
- Zustand vor der Optimierung:
  - Server stehen zum Teil unter Dauerlast
  - Nachrichten werden nicht verarbeitet
  - Im schlimmsten Fall: HornetQ auf dem Server hängt sich auf

Probleme treten erst auf
- Unter bestimmten Bedingungen
- Ab einer gewissen Menge Subknoten



Testen verteilter Systeme

Testplanung

Stufenweises Vorgehen:

- Unit
- Komponenten
  - *Speziell: Benutzerschnittstellen*
- Integration
  - Server-zu-Server-Kommunikation
  - Client-Server-Kommunikation
- Gesamtsystem
- Nichtfunktionale Anforderungen
- Abnahmetest (in der Regel durch den Kunden)

Tests am Reifegrad orientieren

Abgrenzung

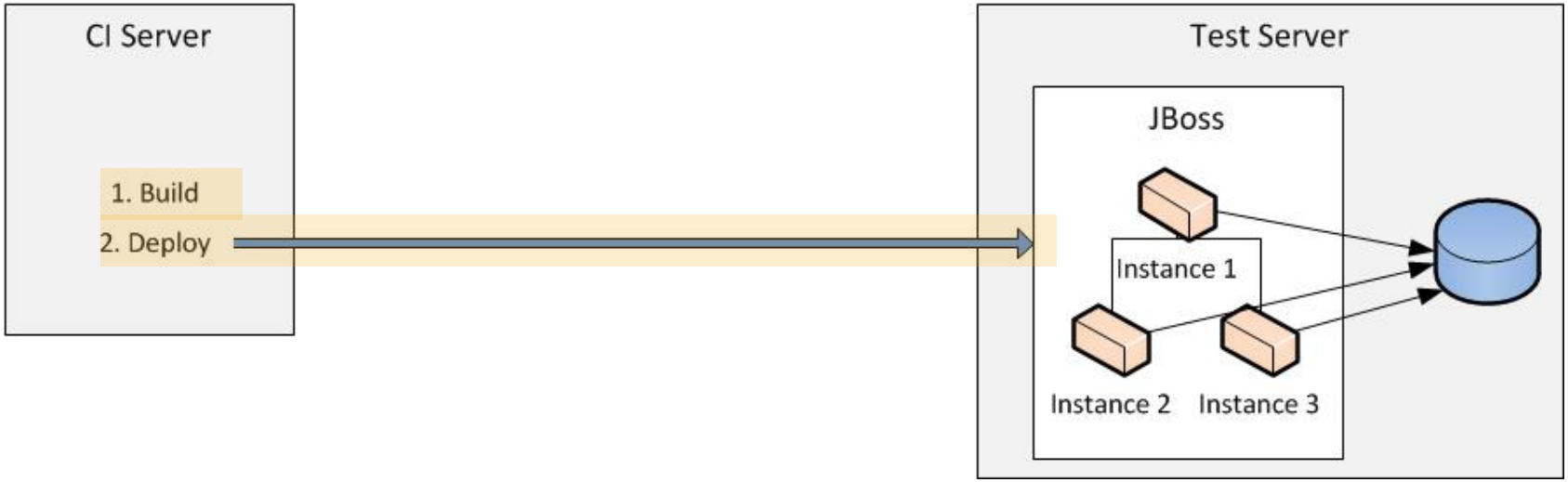
Unit / Komponenten

- Schwerpunkte
  - Test einzelner Funktionen
  - Möglichst vollständiger Test der Schnittstellen
  - Test der Oberflächen
- Testumgebung
  - Ausführung lokal
  - Services im lokalen Modus

Integration / System

- Schwerpunkte
  - Test der Kommunikation im Gesamtsystem
  - Test typischer Szenarien
- Testumgebung
  - Ausführung im Netzwerk inklusive
    - einiger weniger Serverinstanzen
    - Client-System-Repräsentant
  - Services im EJB-Modus

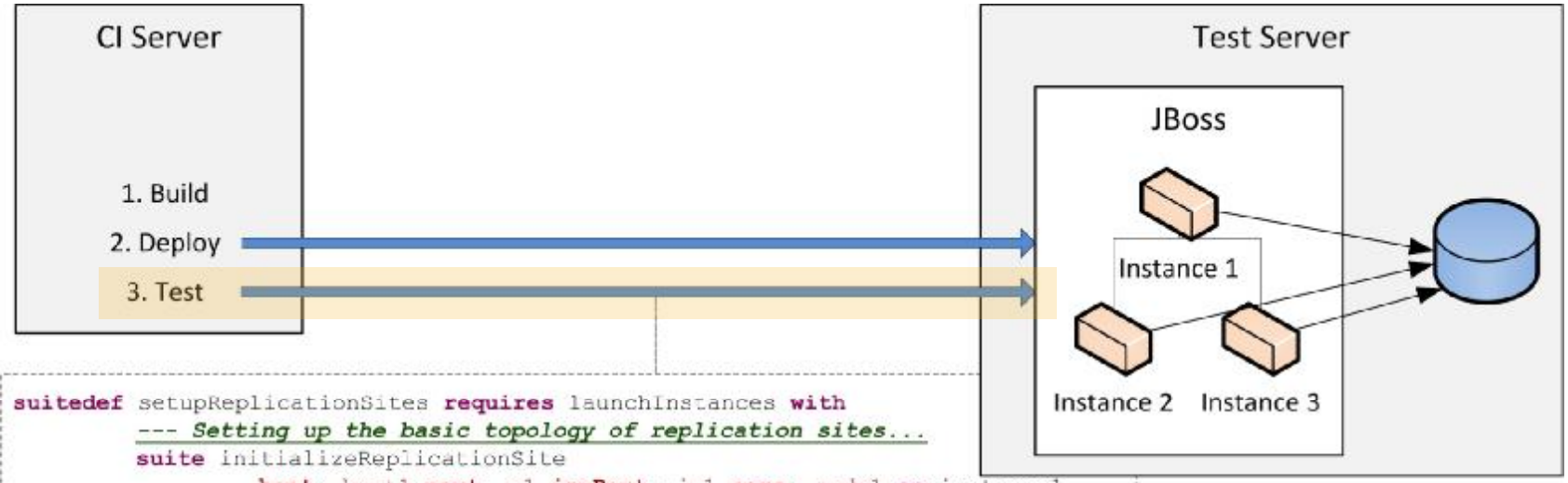
Integrationstest



Deployment

- Ein Application-Server, mehrere Configuration-Server-Instanzen (Port-Offset)
- Eine Datenbank, mehrere Schemata

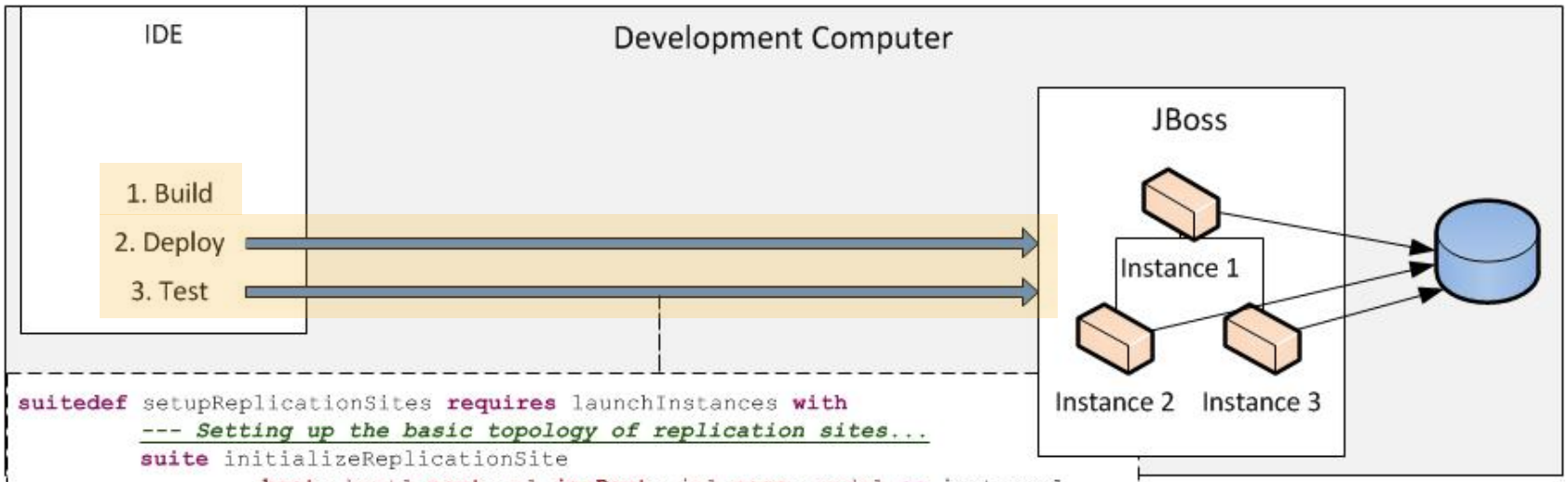
Integrationstest




```
suitedef setupReplicationSites requires launchInstances with
        --- Setting up the basic topology of replication sites...
        suite initializeReplicationSite
                host: host1 port: p1 jmxPort: jp1 name: node1 on instance1
        suite initializeReplicationSite
                host: host2 port: p2 jmxPort: jp2 name: node2 on instance2
        suite initializeReplicationSite
                host: host3 port: p3 jmxPort: jp3 name: node3 on instance3

        suite assignParentReplicationSite
                host: host1 port: p1 jmxPort: jp1 on instance2
        suite assignParentReplicationSite
                host: host2 port: p2 jmxPort: jp2 on instance3
suiteend
```

Integrationstest




```
suitedef setupReplicationSites requires launchInstances with
        --- Setting up the basic topology of replication sites...
        suite initializeReplicationSite
                host: host1 port: p1 jmxPort: jp1 name: node1 on instance1
        suite initializeReplicationSite
                host: host2 port: p2 jmxPort: jp2 name: node2 on instance2
        suite initializeReplicationSite
                host: host3 port: p3 jmxPort: jp3 name: node3 on instance3

        suite assignParentReplicationSite
                host: host1 port: p1 jmxPort: jp1 on instance2
        suite assignParentReplicationSite
                host: host2 port: p2 jmxPort: jp2 on instance3
suiteend
```

Vorteile

- Einfachere Fehleranalyse
- Vermeidung von Fehlermaskierung
- Erreichbarkeit von Testkonstellationen
- Aufgabenteilung

Nichtfunktionale Tests

Einige gut automatisiert testbar

- Beispiele: Last, Performance, Stress

Einige bedingt automatisiert testbar

- Beispiel: Sicherheit

Einige nicht automatisiert testbar

- Beispiele: Bedienbarkeit, Wartbarkeit

Last- und Performancetests

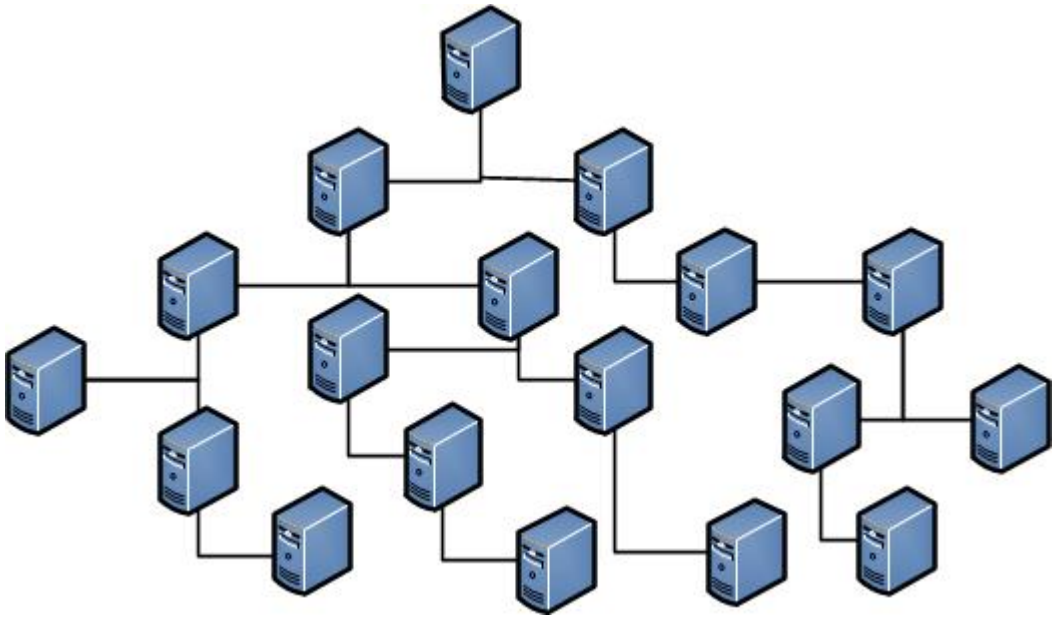
Last- und Performancetest

Definition:

- Lasttest:
  - Prüfen des Systemverhaltens innerhalb definierter Grenzen
  - Ziele: Funktionales Verhaltens unter Last, Antwortverhalten, Ressourcenverbrauch
- Performancetest:
  - Prüfen des Systems innerhalb definierter Grenzen
  - Ziel: Finden von Bottlenecks
- Stresstest:
  - Prüfen des Systemverhaltens außerhalb definierter Grenzen
  - Ziele: Maximale mögliche Last, Finden von Bottlenecks

Last und Performance

Integrationstestaufbau skaliert nicht!



Vorbereitung

1. Ziel setzen
2. Grenzen definieren
3. Anwendung testbar machen
4. Umgebung vorbereiten
5. Tests ausführen
6. Maßnahmen ergreifen
7. Weiter mit Punkt 4., bis Ziel erreicht ist

Ziel und Grenzen definieren

Im Beispiel:

- Hardware- und Netzwerkrahmenbedingungen stehen fest → in Testdefinition aufnehmen
- Maximale Anzahl der Kindknoten festlegen
  - Alternativ aus Spezifikation entnehmen

Anwendung testbar machen

Welche Funktionen können wiederverwendet werden?

- Im Beispiel:
  Autoinitialisierung
  Services zum Abrufen des internen Zustands

- Was muss neu implementiert werden?
  - Im Beispiel:
    Server-Stubs

Beispiel: Autoinitialisierung

Service-Interface: AutoInitService

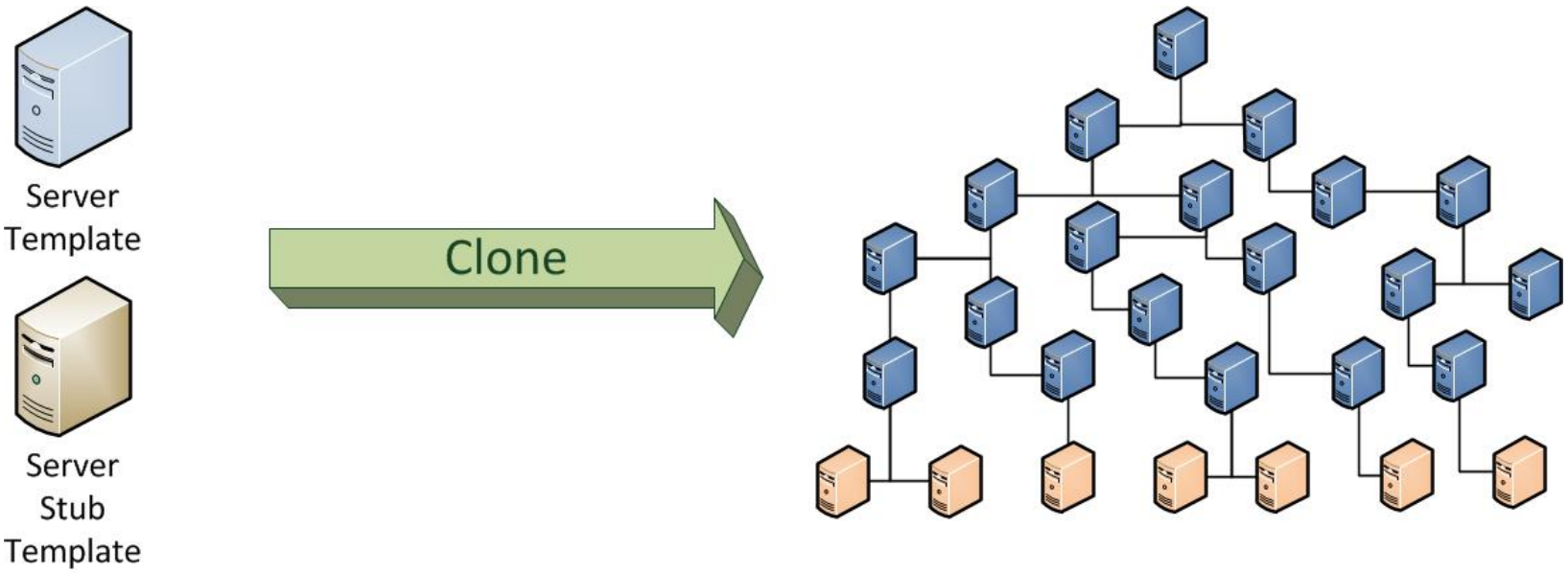
Produktiv: IPAutoInitService

```
<object class="java.util.ArrayList">
 <void method="add">..
 <void method="add">..
 <void method="add">
  <object class="de.gebit.compas.replication.core.api.vo.AutoInitImportNodeVO">
   <void property="hostname">
    <string>@{HOSTNAME}</string>
   </void>
   <void property="JMSPort">
    <int>4547</int>
   </void>
   <void property="JMXPort">
    <int>1290</int>
   </void>
   <void property="parentHostname">
    <string>compas-dock-002</string>
   </void>
   <void property="parentJMSPort">..
   <void property="parentJMXPort">..
   <void property="assignParent">
    <boolean>true</boolean>
   </void>
   <void property="modificationValues">..
   <void property="scope">..
   <void property="orgaUnit">
    <string>World.Test.@{HOSTNAME}</string>
   </void>
  </object>
 </void>
</object>
```

Testvariante 1: XMLAutoInitService

Testvariante 2: VariableXMLAutoInitService

Soll-Zielumgebung



Zielumgebung mit VMs

- ESX-Host exklusiv für Last- und Performancetests
  - Inklusive Emulator für Netzwerkprobleme (Latency, Package Loss)
- Vorbereitung einer Template-VM
- Clonen und Anpassen der Einstellungen, bspw. IPs
  - Erledigen Scripte, die sich auch in einen Build einbauen lassen
- Server setzen sich per Autoinitialisierung selbst auf

Zielumgebung mit VMs

Vorteile:

- Analogie zur Produktivumgebung
  - Im Beispiel: Windows Server
- Einfache Simulation von Netzwerkproblemen

Nachteile:

- Schwergewichtig
  - In der Regel nur wenige Instanzen auf einem Host (im Beispiel 60 VMs) zu realisieren
    - Alternativ: Space mieten
  - Aufsetzen dauert verhältnismäßig lang
    - Wird eher dauerhaft aufgesetzt und nicht nur temporär

Zielumgebung mit Docker

- Docker-Host für Last- und Performancetests
- Eigene (private) Docker-Registry
- Aufeinander aufbauende Images, bspw.
  - 1. Ebene: Betriebssystem + Java + Datenbank
  - 2. Ebene: Application-Server (JBoss oder Wildfly)
  - 3. Ebene: Deploy der Anwendung

Zielumgebung mit Docker

- Empfohlen: Verwendung von Dockerfiles
  - Flexibler bei Änderungen der Plattform
  - Vermeidet „Verschmutzen“ der Container

```
Step 9 : RUN apt-get update && apt-get -y install postgresql-${PGVERSION} postgresql-client-${PGVERSION}
 ---> Using cache
 ---> 7fd0c0260d4d
Step 10 : USER postgres
 ---> Using cache
 ---> 253bfc7b8b29
Step 11 : RUN /etc/init.d/postgresql start &&    psql --command "CREATE USER test WITH SUPERUSER PASSWORD 'test';" &&    createdb -O test test
 ---> Running in f708d2ca31a1
Starting PostgreSQL 9.4 database server: main.
CREATE ROLE
 ---> 221e15ae9f4d
Removing intermediate container f708d2ca31a1
Step 12 : RUN echo "host all  all    0.0.0.0/0  md5" >> /etc/postgresql/${PGVERSION}/main/pg_hba.conf
 ---> Running in 280966343f8f
 ---> 9d99b6240cbf
Removing intermediate container 280966343f8f
Step 13 : RUN echo "listen_addresses '*'" >> /etc/postgresql/${PGVERSION}/main/postgresql.conf
 ---> Running in 1c5367701898
 ---> ce4c41f34a8a
Removing intermediate container 1c5367701898
Step 14 : EXPOSE 5432
 ---> Running in 3b0cf5b2b657
 ---> a233038dad9e
```

Zielumgebung mit Docker

- Container auf dritter Ebene
  - Ermöglicht produktspezifischen Test
  - Container für Lasttest enthält alle notwendigen Ressourcen für den Server
    - Nicht offiziell von Docker empfohlen, aber für multiples Starten gleichartiger Container sinnvoll

Zielumgebung mit Docker

Vorteile:

- Leichtgewichtig
  - Images können aufeinander aufbauen, viele Einrichtungsarbeiten deshalb nur einmal notwendig
  - Aufsetzen geht schnell
    - Container werden temporär gestartet und nach dem Test wieder beendet
- Jeder Test startet „clean“ – kein Zurücksetzen von Testdaten notwendig

Nachteile:

- Ggf. weit weg von der Produktivumgebung
  - In unserem Fall (Container setzt auf Zielsystem CentOS auf) weit entfernt vom alternativen Zielsystem Windows Server

Integration in den Buildprozess

Viele Dockerplugins für Jenkins

- Docker als Jenkins-Slaves
- Aufruf von Dockerkommandos im Build
- Deploy in Docker-Container
- Bauen von Docker-Images auf Basis von Dockerfiles
- Docker-Integration in Pipeline

```
docker.image("MyImage").inside {
    myCommand
}
```

Fallstricke:
Zugriffsrechte (sudo...)
Zugriff auf Docker REST API

Ergebnisse

Einsatz von

- Docker
  - Für Massentests
- VMs
  - Für möglichst realistische Testszenarien
  - Für Simulation von Netzwerkproblemen

Ergebnisse

HornetQ-Probleme konnten gelöst werden

Ziel:

- In Produktiv müssen mindestens 60 Kindknoten auch unter widrigen Umständen (geringe Serverkapazität, schlechte Netzwerkverbindungen) versorgt werden

Aktueller Stand

- In Produktiv werden nach den Optimierungen deutlich mehr (> 120) Kindknoten unter den gleichen widrigen Umständen problemlos versorgt

Nebeneffekt

- Ableitung von Empfehlungen für Serverausstattung

Fazit

Fazit

- Last- und Performancetests sollten nicht vernachlässigt werden
  - Das gleiche gilt auch für bspw. Securitytests
- Moderne Virtualisierungsumgebungen vereinfachen das Aufsetzen von verteilten Systemen für den Test
- Dockercontainer ermöglichen die Beschränkung auf das Notwendige
- Test mit VMs haben aber weiterhin ihre Berechtigung
  - Außer wenn das Zielsystem ebenfalls in einem Dockercontainer läuft

Zeit für Fragen

Vielen Dank
für Ihre Aufmerksamkeit!

Besuchen Sie uns gern am Stand!

Artikel zum Thema:
Verteilte Systeme automatisiert testen
Eclipse Magazin 2.16

GEBIT Solutions