Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice

Daniel Richter, Marcus Konrad, Katharina Utecht, and Andreas Polze

Operating Systems & Middleware Group
Hasso Plattner Institute at University of Potsdam, Germany
Motivation

▪ EPA – the legacy system
  ▪ reserve and book train seats operated by Deutsche Bahn (German railway)
  ▪ 1 mio seat requests & 300,000 bookings
  ▪ first version: 1980s
  ▪ set of Pathway Services as part of HP NonStop system
  ▪ especially fault-tolerant and highly-available
Motivation

but: **difficult to adapt to new, unknown needs**

- technological constraints
  - programming languages: C, C++, Cobol, Java
  - DBMS: Enscribe, SQL/MPm, SQL/MX
- specialized hardware
  - tied to *HP NonStop* system
- long update cycles
  - possibly multiple months

*Highly-Available Applications on Unreliable Infrastructure...*
Motivation
Motivation

...Microservices in Practice

- small, independent, autonomous services
- small, specific range of features
- encapsulates all its functions and data
- cooperation with other microservices (usually ReST & message queues)
- DevOps
Motivation

**Aim**: evaluate general properties of a microservice and its dependability compared to the legacy system

1. **Benefits & Drawbacks of MSAs**
2. **Implementing a Seat Reservation System based on Microservices**
   - Requirements, Definition of Domains
3. **Operation of Microservice Architectures**
   - Containerization with Docker, Message-Driven Communication Middleware
4. **Evaluation: Dependability & Fault Tolerance**
Benefits and Drawbacks of Microservice Architectures

introduction of self-contained services that deliver, combined, the same functionality as the original system
Benefits and Drawbacks of Microservice Architectures

Advantages

- small and independent services
  - classification of domains
  - decoupling & explicit separation of features
- free choice of technology
  - use the technology that fits the needs best
  - functionality and data
- scalability
  - designed for horizontal scaling – multiple instances
  - requires stateless services
- hardware independence
  - usually self-contained virtual machines
Benefits and Drawbacks of Microservice Architectures

Advantages

- replaceability & versioning
  - loose coupling among microservices
  - independent testing & deployment
  - redundancy: multiple versions at the same time

- automation
  - many steps for operation only differ in some minor configuration options

- DevOps
  - one single team involved in development (design, implementation, testing, deployment, maintenance) and architectural layers (frontend, backend, database)
Disadvantages

- **complexity**
  - from implementation to execution environment
  - provisioning & orchestration of many services

- **monitoring**
  - service vs. container vs. infrastructure

- **testing**
  - single service vs. combined services, communication

- **communication overhead**
  - inter-process & remote

- **consistency**
  - shared data across service boundaries
Implementing a Seat Reservation System based on Microservices

modularization into self-contained subsystems with free choice of technology
Implementing a Seat Reservation System with Microservices

Requirements

- **functional:**
  - display available seats, book a seat reservation, overview of existing bookings

- **non-functional**
  - consistency, scalability & efficiency, load balancing, portability, deployment & maintainability, changeability, replacement & versioning, interfaces

- **fault tolerance**
  - tolerate failure of several service instances, virtual machines, or infrastructure components
  - asynchronous communication between services
partitioning into functionally connected domains, each domain contains self-contained services with limited scope of operation

- Seat Management Domain
- Seat Overview Domain
- Booking Domain
- Customer Management Domain
- Price Computation Domain
- Front-end
Implementing a Seat Reservation System with Microservices

Definition of Domains

**Booking Domain [blue]**
- **Booking Database**
  - bookings
  - partial bookings
- **Key-Value Store**
  - started bookings

**Booking Service**
- check, submission, cancellation and change of bookings
- list bookings

**Price Calculation Domain [green]**
- **Seat Database**
  - current train utilization
- **Price Calculation Service**
  - calculation of dynamic price

**Price Calculation Service**
- Amount of Seats

**Seat Domain [orange]**
- **Seat Database**
  - seats, wagons, trains, train connections, route map, cancelled train connections
- **Seat Management Service**
  - data management, insert booked seats, insert cancelled train connections
- **Seat Overview Service**
  - proposal and selection of train connections and seats

**Customer Domain [yellow]**
- **Customer Database**
  - customer data
  - account data
- **Customer Mgmt Service**
  - management of customer data, notification in case of train connection cancellation
  - customer IDs
- **Seat, Train, Time**

**Train Connection**
Implementing a Seat Reservation System with Microservices

Domains + Booking Process

1. Start
   - Insert train connection data
   - Determine available train connections
   - Display train connections
   - Select actual connection and seat properties

2. Check seat availability
   - Yes: Display seat
   - No: Display seat proposal and associated price

3. Lock seat
   - Yes: Display seat
   - No: Display seat proposal and associated price

4. Insert customer data
   - Yes: Sign in
     - Yes: Update booking database
       - Yes: Finish booking
         - No: Create account
       - No: Update seat database
     - No: Display finished booking
   - No: Create account
   - No: Update seat database
   - No: Display finished booking

5. Begin booking
   - Yes: Set booking lock
   - No: Display all available seats

6. Display all available seats
   - Yes: Select seats
   - No: Display seats

7. Select seats
   - Yes: Lock seat
   - No: Display price

8. Lock seat
   - Yes: Display price
   - No: Display finished booking

9. Display finished booking
Operation of Microservice Architectures

after their implementation, the microservices, their databases, and the front-end have to be deployed into self-contained environments
Operation of Microservice Architectures

Execution Environment

**requirements:** portability, load balancing, fault tolerance, maintainability

- virtualized infrastructure
  - AWS/EC2 Ubuntu 14.4
- containerization with Docker 1.11
  - Docker Compose
  - Docker Swarm
  - Overlay Network
- message-driven communication middleware
  - RabbitMQ 3.6.2
Operation of Microservice Architectures

Execution Environment
Operation of Microservice Architectures

Execution Environment

- services for seat reservation
  - Java 8
  - Spring Boot 1.3
  - MySQL 5.7
  - Redis 3.2
  - Cassandra 3.4
Basic Set-Up of a Microservice
Evaluation

modularized software system consisting of self-contained services published as containers and executed as multiple redundant instances
Recap: Requirements

- **functional:**
  - display available seats, book a seat reservation, overview of existing bookings

- **non-functional**
  - consistency, scalability & efficiency, load balancing, portability, deployment & maintainability, changeability, replacement & versioning, interfaces
instead of relying on specialized (and expensive) highly-available infrastructure: modularize the software system into self-contained services published as containers and execution as multiple redundant instances

Redundancy

- replicas of services, containers, virtual machines
- communication middleware
- service logic and databases
Evaluation: Dependability & Fault Tolerance

Replicas of...

...services, containers, and virtual machines

- **Overlay Network**
  - uniform host name, arbitrary number of replicas
  - if service instance, RabbitMQ server, or even EC2 instance fails – redirect to another instance

- **Docker Swarm**
  - “High Availability” feature: primary manager instance + multiple replica that will take over
  - data storage (etcd, Consul) can be scaled and connected
Evaluation: Dependability & Fault Tolerance

Replicas of...

...services, containers, and virtual machines

- services
  - state-less (state is stored into domain’s database)
  - can be replaced by other instances

- messages
  - distributed among all RabbitMQ servers
  - conflict-free merging of message nodes (via master-node)
Communication Middleware

- message queue is one of the most important parts of the architecture
- tolerated faults: network failure, RabbitMQ server fault, infrastructure failure, malformed messages
- clients can connect do different RabbitMQ servers
- virtual hosts, exchanges, and message queues are synchronized between servers by default
Evaluation: Dependability & Fault Tolerance

Service Logic & Databases

- services are state-less – the critical part is the database
- use relaxed consistency guarantees (e.g. NoSQL)
  - Cassandra with multiple replicas
  - MySQL in master-slave-replication mode
Conclusion

- prototypical architecture and implementation
- freedom to choose any technology is bigger than before; several tools and frameworks for execution environment. **but:** tied to Docker
- no hardware dependency – fully virtualized infrastructure by AWS
- bring service modifications into production within minutes; architectural changes last a few days
- experience for multiple tools have to be gained; tools, libraries, and frameworks are still in development *and change quickly*
Conclusion

The results show a potential for microservice architectures and the possibility for flexible implementation, deployment, and advancement of services. In terms of non-functional requirements, there is no evidence that the new solution perform better, though.

We would like to thank Lena Feinbube, Maxi Fischer, Jonas Bounama, Nils Hennings, Timo Traulsen, Henry Hübler, Dr. Stephan Gerberding, Dr. Clements Gantert, Wolfgang Schwab, and Ingo Schwarzer for their support and assistance with this project.