Service Fabric: A Distributed Platform for Building Microservices in the Cloud


#: University of Illinois at Urbana Champaign | *: Microsoft Azure

Presenter: Shegufta Bakht Ahsan

EuroSys 2018, April 23rd-26th | Porto, Portugal

DPRG@UIUC: http://dprg.cs.uiuc.edu
Service Fabric: aka.ms/servicefabric
Microsoft Service Fabric

A **distributed platform** that enables building and management of **scalable** and **reliable** **microservice based** applications

Culmination of **over 15 years** of design and development

- **Microsoft Azure SQL DB:**
  - Hosts ~2 Million DBs | **Containing 3.5 PB of data** | Spans over 100K machines

- **Azure Cosmos DB:**
  - Utilizes 2 million cores | **Spans over 100K machines**

- **Cloud Telemetry Engine:**
  - Processes 3 Trillion events/week
Monolithic Vs. Microservice Based Approach

Classic Monolithic Approach

- Cannot scale out individual functions
- Needs to scale out everything

Microservice Based Approach

Can Scale-out individual components

Not Cloud Friendly

Cloud Friendly
# Monolithic vs. Microservice Applications

<table>
<thead>
<tr>
<th>Feature</th>
<th>Monolithic design</th>
<th>Microservice-based design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application complexity</td>
<td>Complex</td>
<td>Modular</td>
</tr>
<tr>
<td>Fault-tolerance</td>
<td>Complex</td>
<td>Modular</td>
</tr>
<tr>
<td>Agile development</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Communication between components</td>
<td>NA</td>
<td>RPCs</td>
</tr>
<tr>
<td>Easily scalable</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Easy app lifecycle management</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cloud ready</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Application Model
Service Fabric and Its Goals

➢ Support for Strong Consistency:
   • Ground Up
   • Higher layer focuses on “their” relevant notion of consistency (ACID at Reliable Collections)

➢ Fault Tolerance

➢ Support for Stateful Microservices:
   • Microservices can have their own state
Service Fabric Major Subsystems

- Reliable Collections
  - Reliability Subsystem (Failover Manager, PLB)
  - Federation Subsystem
    - (Failure Detection, Leader Election, Routing Consistency, Consistent Neighborhood)
Reliable Collection (Queue, Dictionary):
- Highly Available
- Fault Tolerant
- Persisted
- Transactional

Leader Election
Routing Consistency
Reliable Failure Detector
Routing Token

Consistency: Higher layers reuse lower layer's, implementing their own notion of consistency

Reliability Subsystem
Federation Subsystem
Federation Subsystem

- **Nodes are organized in a virtual ring (SF-Ring):**
  - Consists of $2^m$ points (e.g., $m=128$ bits)
  - Key -> owned by the closest node
  - Neighborhood set: \{ ‘n’ successors, ‘n’ predecessors \}

- **Ensures:**
  - Consistent Membership and Failure Detection
  - Consistent Routing
  - Leader Election
Consistent Membership and Failure Detection

➢ Design Principles:
   1. Membership -> Strongly Consistent
      • For each node, all its monitors agree on its up/down status
   2. Decouples Failure Detection from Failure Decision (using Arbitrator)

➢ Lease Based Monitoring:
   • Node A sends Lease Request to Node B
   • If Node A receives ACK, lease establishes

➢ Symmetric Monitoring (SM)
   • Node A and Node B monitor each other

➢ Node X (Decoupling Detection-Decision):
   • Maintains SM with all neighbors
   • If at least one Lease fails (Detection)
     • ask for Arbitration (Decision)

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Lease Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OK</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2n</td>
<td>OK</td>
</tr>
</tbody>
</table>
Arbitrator – Decouple Detection From Decision

➢ Fail to renew lease (lease timeout Tm) (Detection)
  • Ask for arbitration immediately (Decision)
    • IF don’t receive any reply within Tm, leave!
    • ELSE follow arbitrators decision!

  In Production: Multiple Arbitrators, Quorum Based approach

[1] Symmetric Monitoring Failed
[2] Hey, I think B is dead!
[3] Yes it is!
[4] Hey, I think A is dead!
[5] It’s too late! You have to leave

Arbitration Log
Log 1: Time T : Node B declared dead
Routing is Bidirectional and Symmetric (SF-Routing)

- The $i^{th}$ clockwise/anticlockwise routing table entry is the node whose ID is closest to the key $(n + - 2^i) \mod(2^m)$

SF-Routing:
- Provides more routing options
- Routes message faster

In latest design, SF-Routing is used for:
- Discovery routing when a node starts up
- After Discovery, nodes communicate directly
Consistent Routing

➢ At any given time all messages sent to key ‘K’ will be received by a unique Node. If that node crashes, a new node will take the responsibility
  • Leader Election: For entire system use K=0

➢ Each Node owns a routing token:
  • A portion of the ring whose keys it is responsible for

➢ SF-Ring ensures following consistency properties:
  • Always Safe: there is no overlap among tokens owned by nodes
  • Eventually Live: Eventually every token range will be claimed by a node

➢ Efficiently Handle: Node Join, Leave and Fail
Consistent Routing

- At any given time all messages sent to key ‘K’ will be received by a unique Node. If that node crashes, a new node will take the responsibility.

**SF Ring**
- Is being used in production for more than 15 years
- Working successfully, hence have not had to change it

**Invented concurrent with Chord and Pastry**

**Chord/Pastry do not support Strong Consistency**

- Efficiently Handle: Node Join, Leave and Fail
Reliable Collection (Queue, Dictionary):
- Highly Available
- Fault Tolerant
- Persisted
- Transactional

Leader Election
Routing Consistency
Reliable Failure Detector
Routing Token
Federation Subsystem
Reliable Primary Selection
Consistent Replica Set
Failover Management
Replicated State Machines
Replication Subsystem

Consistency: Higher layers reuse lower layer’s, implementing their own notion of consistency
Reliability Subsystem

➢ Provides:
  • Replication
  • High Availability
  • Load Balancing

Consistency: Higher layers reuse lower layer's, implementing their own notion of consistency

Reliability Subsystem
Reliable Collection (Queue, Dictionary)

➢ Reliable Collections:
  • Fault Tolerant
  • Highly Available
  • Persisted, Replicated
  • Transactional

➢ Leverages lower layer guarantees (Failure Detection, Leader election, load balance etc.)

➢ Used in Stateful Microservices
Evaluation – SF Arbitrator vs. Fully Distributed Scheme

- **Scalable Failure Detector (SWIM): Not Strong**
- **Strong Failure Detector (Virtual Synchrony): Not Scalable**

**Arbitrator based FD:**
1. Scalable
2. Strong Failure Detection
3. Prevents Cascading Failure
4. Does not depend on #neighbors

If a node fails to maintain lease, it will gracefully leave the system. It is the fully distributed way of maintaining strong consistency.

- **Node 1** + \{1\}
  - 4 neighbors = 5
- **Node 1, 2** + \{1,2\}
  - 4 neighbors = 6
- **1,10**
- **1,2,3,4**
- **1,10,20,30**

**Crashed Node IDs**

**Total Nodes Leaving the Ring**

- **Arb. less, M=2**
- **Arb. less, M=8**

**SF arbitrator approach**
In Production: Reconfiguration Events + Reconfiguration Time

Affects Availability

<table>
<thead>
<tr>
<th>Event</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failover</td>
<td>1%</td>
</tr>
<tr>
<td>SwapPrimary</td>
<td>20%</td>
</tr>
<tr>
<td>Other</td>
<td>79%</td>
</tr>
</tbody>
</table>

Table 3: Different Reconfiguration Events. Over a 20-day trace.

Quick Control Decision (currently optimized to 100s of ms)
Evaluation (summary)

➢ Arbitrator Based Strong Failure Detector:
  • **Scalable** (Minimum Failure Detection Overhead)
  • Prevents **Cascading Failure**
  • Uses **less** stabilization messages (Compared to the Arbitrator Less Scheme)
  • Does not get affected by the **number of neighbors**

➢ Reconfiguration:
  • Control Decisions are generated **quickly** (avg 1~2 seconds)
  • SF’s current **reactive reconfiguration approach is** ensuring availability for ~10 million microservices
Evaluation (summary)

➢ Message Delay:
  • Even in the presence of higher churn, message delay remains largely unaffected (80th percentile)

➢ SF-Routing:
  • SF-Routing requires higher memory (117%) than Chord
  • Messages takes fewer hops (49.27%) than Chord and thus Routes Faster
Summary

➢ **Microsoft Service Fabric**: A **distributed platform** that enables building and management of **scalable and reliable microservice based** applications.

➢ **Service Fabric** ensures **strong consistency** and **fault-tolerance** from **lower layers**, which helps us to build state at the upper layers.

➢ **Selected Components**:  
  • Federation Subsystem, **Reliability Subsystem**, Reliable Collection (Queue, Dictionary)

Open Source: github.com/Microsoft/service-fabric

DPRG@UIUC: http://dprg.cs.uiuc.edu  
Service Fabric: aka.ms/servicefabric