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


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

Presenter: Shegufta Bakht Ahsan

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

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

These slides were adapted for teaching purposes, the original set is available at https://shegufta.com/publications/
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

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

Classic Monolithic Approach

Microservice Based Approach

Can Scale-out individual components

Not Cloud Friendly

Cloud Friendly
State in Monolithic approach

State in Microservices approach
Monolithic application approach

1. App 1

2. [Diagram with multiple interconnected components]

Microservices application approach

3. App 1

4. App 2

[Diagram with various interconnected components and network connections]
# Monolithic vs. Microservice Applications

<table>
<thead>
<tr>
<th></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**
- Application Model
- Native and Managed APIs

**Management Subsystem** (Cluster Manager, Health Manager, Image Store)
- Communication Subsystem (NS)

**Reliability Subsystem** (Failover Manager, PLB)

**Federation Subsystem** (Failure Detection, Leader Election, Routing Consistency, Consistent Neighborhood)
- Hosting & Activation (App. Lifecycle)

**Transport Subsystem** (secure point-to-point communication)

**Testability Subsystem** (Fault Injection)

Reliability Subsystem
- Reliable Primary Selection
- Consistent Replica Set
- Failover Management
- Replicated State Machines

Federation Subsystem
- Leader Election
- Routing Consistency
- Reliable Failure Detector
- Routing Token

Consistency: Higher layers reuse lower layer's, implementing their own notion of consistency.
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 $T_m$) (Detection)
  • Ask for arbitration immediately (Decision)
    • IF don’t receive any reply within $T_m$, leave!
    • ELSE follow arbitrators decision!

In Production: Multiple Arbitrators, Quorum Based approach

Node A

[2] Hey, I think B is dead!

needs to obtain confirmation from a majority (quorum) of nodes in the arbitrator group.

[3] Yes it is!

An accept(fail(Y)) message is sent back to X within a timeout based on RTT (if this timeout elapses, X itself leaves the ring). The accept message also carries a timer value called $T_m$, so that X can wait for $T_m$ time and then take actions w.r.t. Y (e.g., reclaim Y’s portion of the ring).

Arbitrator

[4] Hey, I think A is dead!

needs to obtain confirmation from a majority (quorum) of nodes in the arbitrator group.

Node B

[5] It’s too late! You have to leave

We set: $T_o = T_m + laxity$ - (time since first detection). If this is the first detection, $T_o = T_m + laxity$. Here, laxity is typically 30 s, generously accounts for network latencies involved in arbitrator coordination, and independent of $T_m$. As all timeouts are large (tens of seconds), loose time synchronization suffices.
Each node Y and its monitors (X, Z) maintain Y's lease
X and Y’s LR’s are almost simultaneous and both fail: only one of them is kicked out, situation is resolved fast.

Although its lease expired, Y remains in the ring.

- Cannot be a monitor.
- Must stop serving, responsibilities will be taken over.

"Too late X - you are dead"
Y’s lease is renewed, then Y suffers temporary disconnection: Y can be kicked out, may only find out in (up to) $T_m$ time.

- Cannot be a monitor
- Should stop serving, responsibilities will be taken over
Routing is Bidirectional and Symmetric (SF-Routing)

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

- **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

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

Reliability Subsystem
- Reliable Primary Selection
- Consistent Replica Set
- Failover Management
- Replicated State Machines

Federation Subsystem
- Leader Election
- Routing Consistency
- Reliable Failure Detector
- Routing Token
Reliability Subsystem

➢ Provides:
  • Replication
  • High Availability
  • Load Balancing

Reliability Subsystem:
- Reliable Primary Selection
- Consistent Replica Set
- Failover Management
- Replicated State Machines

Federation Subsystem:
- Leader Election
- Routing Consistency
- Reliable Failure Detector
- Routing Token

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

SF arbitrator approach

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

Total neighbors

Cascading Failure

Original, M=* Arb. less, M=2 Arb. less, M=4 Arb. less, M=8

Node 1

+ \{1\}

4 neighbors = 5

Node 1, 2

\{1,2\}

4 neighbors = 6

Crashed Node IDs

Single Neighbors Non-Neighbors Neighbors Non-Neighbors

Node 1, 2, 3, 4

\{1,2,3,4\}

Node 1, 10, 20, 30

\{1,10,20,30\}

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