the morning paper

a random walk through Computer Science research, by Adrian Colyer

ServiceFabric: a distributed platform for building microservices in the cloud

JUNE 5, 2018 ~ ADRIAN COLYER

ServiceFabric: a distributed platform for building microservices in the cloud Kakivaya et al., *EuroSys'18*

(*If you don't have ACM Digital Library access, the paper can be accessed either by following the link above directly from The Morning Paper blog site*).

Microsoft's Service Fabric powers many of Azure's critical services. It's been in development for around 15 years, in production for 10, and was made available for external use in 2015.

> ServiceFabric (SF) enables application lifecycle management of scalable and reliable applications composed of microservices running at very high density on a shared pool of machines, from development to deployment to management.

Some interesting systems running on top of SF include:

- Azure SQL DB (100K machines, 1.82M DBs containing 3.48PB of data)
- Azure Cosmos DB (2 million cores and 100K machines)
- Skype
- Azure Event Hub
- Intune
- Azure IoT suite

- Cortana

SF runs in multiple clusters each with 100s to many 100s of machines, totalling over 160K machines with over 2.5M cores.

Positioning & Goals

Service Fabric defies easy categorisation, but the authors describe it as "*Microsoft's platform to support microservice applications in cloud settings*." What particularly makes it stand out from the crowd is that it is built on foundations of *strong* consistency, and includes support for stateful services through *reliable collections*: reliable, persistent, efficient and transactional higher-level data structures.

> Existing systems provide varying levels of support for microservices, the most prominent being Nirmata, Akka, Bluemix, Kubernetes, Mesos, and AWS Lambda [there's a mixed bag!!]. SF is more powerful: it is the only data-ware orchestration system today for stateful microservices. In particular, our need to support state and consistency in low-level architectural components drives us to solve hard distributed computing problems related to failure detection, failover, election, consistency, scalability, and manageability. Unlike these systems, SF has no external dependencies and is a standalone framework.

Every layer in SF supports strong consistency. That doesn't mean you can't build weakly consistent services on top if you want to, but this is an easier challenge than building a strongly consistent service on top of inconsistent components. "*Based on our use case studies, we found that a majority of teams needing SF had strong consistency requirements, e.g., Microsoft Azure DB, Microsoft Business Analytics Tools, etc., all rely on SF while executing transactions*."

High level design

SF applications are collections of independently versioned and upgradeable microservices, each of which performs a standalone function and is composed of code, configuration, and data.

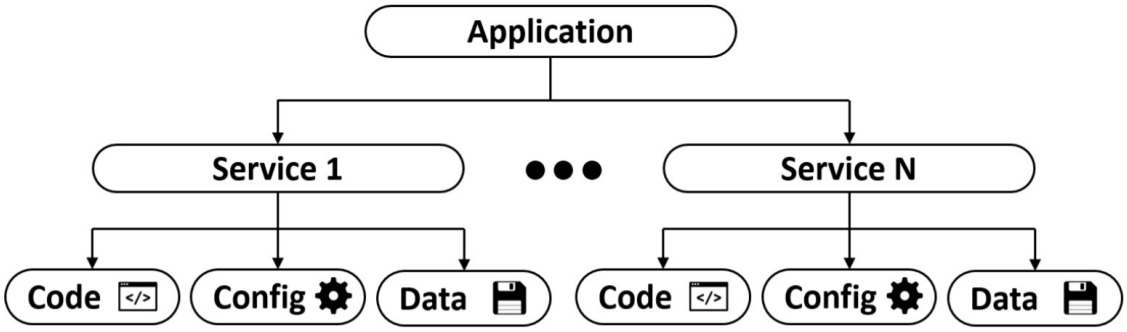


Figure 2: Service Fabric Application Model. *An application consists of N services, each of them with their own Code, Config. and Data.*

SF itself is composed of multiple subsystems, with the major ones shown in the figure below.

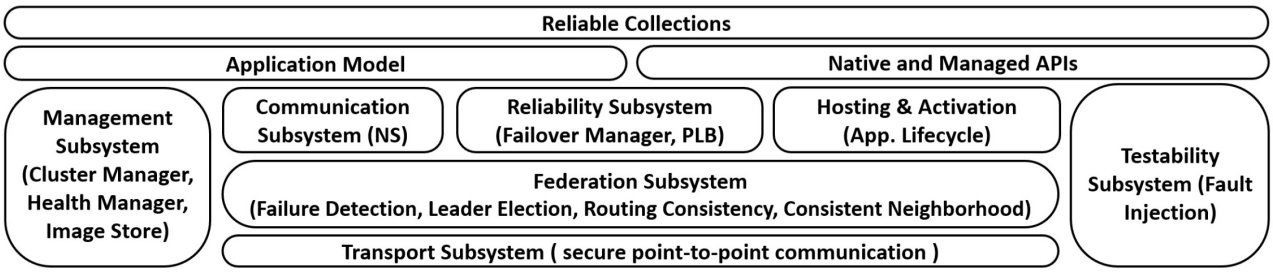


Figure 3: Major Subsystems of Service Fabric. *NS = Naming Service, PLB = Placement and Load Balancer.*

At the core of SF is the *Federation Subsytem*, which handles failure detection, routing, and leader election. Built on top of the federation subsystem is the *Reliability Subsystem* providing replication and high availability. The meat of the paper describes these two subsystems in more detail.

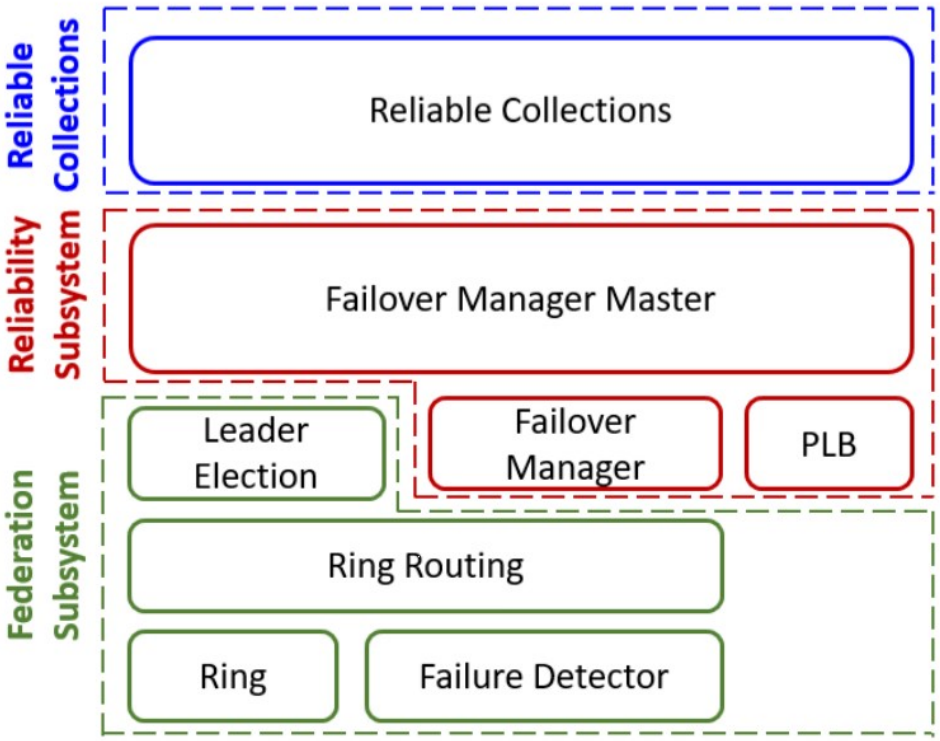
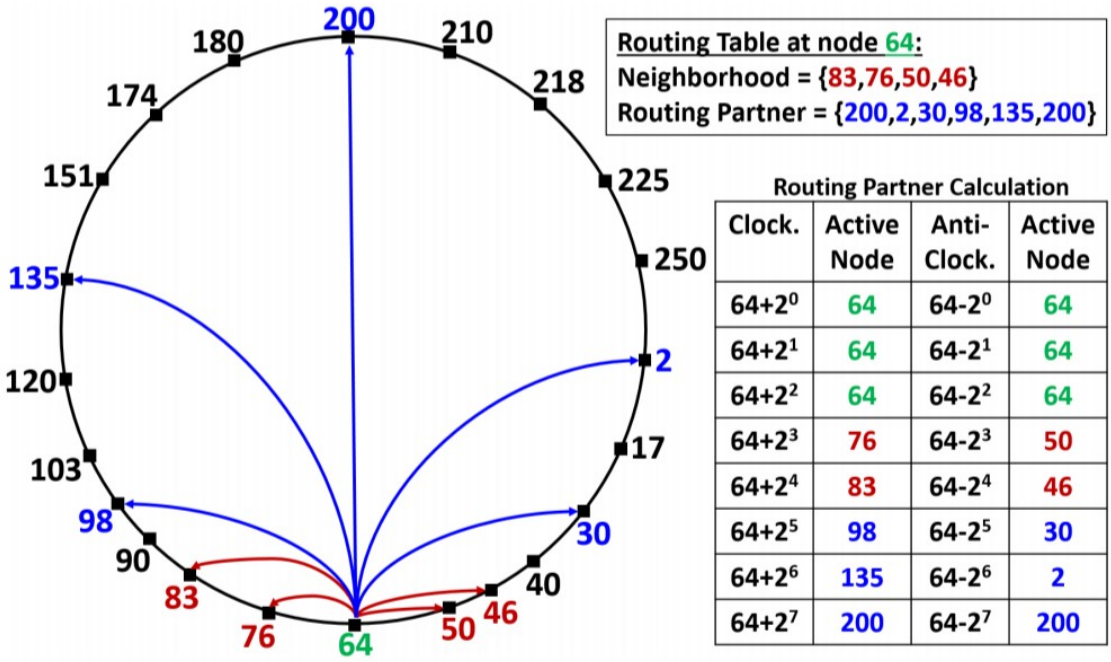


Figure 4: Federation and Reliability Subsystems: Deep-Dive.

Federation subsystem

The ring

At the core of the federation subsystem you'll find a virtual ring with 2^m points, called the SF-Ring. It was internally developed at Microsoft starting in the early 2000's, and bears similarity to Chord and Kademlia. Nodes and keys are mapped to a point in the ring, with keys owned by the node *closest* to it and ties won by the predecessor. Each node keeps track of its immediate successor and predecessor nodes in the ring, which comprise its *neighborhood set*.



Clock.	Active Node	Anti-Clock.	Active Node
$64+2^0$	64	$64-2^0$	64
$64+2^1$	64	$64-2^1$	64
$64+2^2$	64	$64-2^2$	64
$64+2^3$	76	$64-2^3$	50
$64+2^4$	83	$64-2^4$	46
$64+2^5$	98	$64-2^5$	30
$64+2^6$	135	$64-2^6$	2
$64+2^7$	200	$64-2^7$	200

Figure 5: Routing Table *of node 64. The ring has* $2^{m=8}$ *points. Numbered dots represent active nodes.*

Routing table entries are *bidirectional* and *symmetric*. Routing partners are maintained at exponentially increasing distances in the ring, in both clockwise and anti-clockwise directions. Due to bidirectionality, *most* routing partners end up being symmetric. This speeds up routing, the spread of failure information, and the updating of routing tables after node churn.

When forwarding a message for a key, a node searching its routing table for the node *closest* to key in either a clockwise or anti-clockwise direction. Compared to clockwise only routing we get faster routing, more routing options in the face of stale or empty tables, better load spread across nodes, and avoidance of routing loops.

Routing tables are eventually convergent. A chatter protocol exchanges routing table information between routing partners ensuring eventual consistency for long distance neighbours.

> A key result from the SF effort is that strongly consistent applications can be supported at scale by combining strong membership in the neighbourhood

> with weakly consistent membership across the ring. Literature often equates strongly consistent membership with virtual synchrony, but this approach has scalability limits.

Nodes in the ring own *routing tokens* which represent the portion of the ring whose keys they are responsible for. The SF-Ring protocol ensures that there is never any overlap between tokens (*always safe*), and the every token range is eventually owned by at least one node (*eventually live*). When a node joins, the two immediate neighbours each split the ring segment with the new node at exactly the half-way point. When a node leaves, its successor and predecessor split the range between them halfway.

As we'll see when we look at the reliability subsystem, nodes and objects (services) are *placed* into the ring rather than simply relying on hashing. This enables preferential placement taking into account failure domains and load-balancing.

Consistent membership and failure detection

Membership and failure detection takes place within neighbourhood sets. There are two key design principles:

1. **Strongly consistent membership**: all nodes responsible for monitoring a node X must agree on whether it is up or down. In the SF-Ring, this means that all nodes in X's neighbourhood set must agree on its status.
2. **Decoupling failure detection from failure decision**: failure detection protocols (heartbeats) detect a possible failure, a separate *arbitrator group* decides on what to do about that. This helps to catch and stop cascading failure detections.

A node X periodically sends lease renewal requests to each of its neighbours (*monitors*). The leasing period is adjusted dynamically but is typically around 30 seconds. X must obtain acks (leases) from *all* of its monitors. This property defines the strong consistency. If X fails to obtain all of its leases, it considers removing itself from the group. If a monitor misses a lease renewal heartbeat from X it considers marking X as failed. In both cases, the evidence is submitted to the *arbitrator group*.

> The arbitrator acts as a referee for failure detections and for detection conflicts.

> For speed and fault-tolerance, the arbitrator is implemented as a decentralized group of nodes that operate independent of each other. When any node in the system detects a failure, before taking actions relevant to the failure, it needs to obtain confirmation from a majority (quorum) of nodes in the arbitrator group.

The arbitrator protocol details can be found in section 4.2.2 of the paper. Using lightweight arbitrator groups allows membership, and hence the ring, to scale to whole datacenters.

Leader election

Given we have a well-maintained ring, SF has a nice pragmatic solution to leader election:

> For any key *k* in the SF-Ring, there is a unique leader: the node whose token range contains *k* (this is unique due to the safety and liveness of routing tokens). Any node can contact the leader by routing to key *k*. Leader election is thus implicit and entails no extra messages. In cases where a leader is needed for the entire ring we use k=0.

Reliability subsystem

In the interests of space, I'm going to concentrate on the *placement and load balancer* (PLB) component of the reliability subsystem. It's job is place microservice instances at nodes in such a way as to ensure balanced load.

> Unlike traditional DHTs, where object IDs are hashed to the ring, the PLB explicitly assigns each service's replicas (primary and secondaries) to nodes in SF-Ring.

The placement considers available resources at nodes, outstanding requests, and the parameters of typical requests. It also continually moves services from overly exhausted nodes to under-utilised nodes. The PLB also migrates services away from a node that is about to be upgraded.

The PLB may be dealing with tens of thousands of objects in a constantly changing environment, thus decisions taken at one moment may not be optimal in the next. Thus PLB

favours making quick and nimble decisions, continuously making small improvements. *Simulated annealing* is used for this. The simulated annealing algorithm sets a timer (10s in fast mode, 120s in slow mode) and explores the state space until convergence or until the timer expires. Each state has an *energy*. The energy function is user-definable, but a common case is the average standard deviation of all metrics in the cluster (lower is better).

> Each step generates a random move, considers the energy of the new prospective state due to this move, and decides whether to jump. If the new state has lower energy the annealing process jumps with probability 1; otherwise if the new state has *d* more energy than the current and the current temperature is *T*, the jump happens with probability $e^{-d/T}$. This temperature *T* is high in initial steps (allowing jumps away from local minima) but falls linearly across iterations to allow convergence later.

Considered moves are fine-grained. For example, swapping a secondary replica to another node, or swapping primary and secondary replica.

Reliable collections

SF's reliable collections provide data structures such as dictionaries and queues that are persistent, available and fault-tolerant, efficient, and transactional. State is kept locally in the service instance while also being made highly available, so reads are local. Writes are relayed from primary to secondaries via passive replication and considered complete once a quorum has acknowledged.

Reliable collections build on the services of the federation and reliability subsystems: replicas are organised in an SF-Ring, failures are detected and a primary kept elected. PLB (in conjunction with the *failover manager*) keeps replicas fault-tolerant and load-balanced.

> SF is the only self-sufficient microservice system that can be used to build a transactional consistent database which is reliable, self-*, and upgradable.

Lessons learned

Section 7 of the paper contains an interesting discussion of lessons learned during the

development of SF. Since I'm already over my target write-up length, I will just give the headlines here and refer you to the paper for full details:

- Distributed systems are more than just nodes and a network. Grey failures are common.
- Application/platform responsibilities need to be well isolated (you can't trust developers to always do the right thing).
- Capacity planning is the application's responsibility (but developers need help)
- Different subsystems require different levels of investment

What's next?

> Much of our ongoing work addresses the problem of reducing the friction of managing the clusters. One effort towards that is to move to a service where the customer never sees individual servers… other interesting and longer term models revolve around having customers owning servers, but also being able to run microservice management as a service where those servers join in. Also in the short term we are looking at enabling different consistency levels in our Reliable Collections, automatically scaling in and out Reliable Collection partitions, and imbuing the ability to geo-distribute replica sets. Slightly longer term, we are looking at best utilizing non-volatile memory as a store for ServiceFabric's Reliable Collections. This requires tackling many interesting problems ranging from logging bytes vs. block oriented storage, efficient encryption, and transaction-aware memory allocations.

POSTED IN UNCATEGORIZED

DISTRIBUTED SYSTEMS MICROSOFT

< PREVIOUS
Hyperledger fabric: a distributed operating system for permissioned blockchains

NEXT >
Reducing DRAM footprint with NVM in Facebook

19 comments sort by **relevance**

Sign up Sign in

Start a conversation ...

Four short links: 5 June 2018 – Home Improvement Designs (guest) 2 years ago

[…] ServiceFabric: a Distributed Platform for Building Microservices in the Cloud — application lifecycle management of scalable and reliable applications composed of microservices running at very high density on a shared pool of machines, from development to deployment to management. (via Paper a Day) […]

Share Vote Reply

Four short links: 5 June 2018 – DUI Lawyer (guest) 2 years ago

[…] ServiceFabric: a Distributed Platform for Building Microservices in the Cloud — application lifecycle management of scalable and reliable applications composed of microservices running at very high density on a shared pool of machines, from development to deployment to management. (via Paper a Day) […]

Share Vote Reply

New top story on Hacker News: ServiceFabric: a distributed platform for building microservices in the cloud – JkNews (guest)

2 years ago

[…] ServiceFabric: a distributed platform for building microservices in the cloud 3 by deegles | 0 comments on Hacker News. […]

Share Vote Reply

ServiceFabric: a distributed platform for building microservices in the cloud (guest)

2 years ago

[…] ServiceFabric: a distributed platform for building microservices in the cloud 3 by deegles | 0 comments on Hacker News. […]