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

Gopal Kakivaya*, Lu Xun*, Richard Hasha*, Shegufta Bakht Ahsan†, Todd Pfleiger*, Rishi Sinha*, Anurag Gupta*, Mihail Tarta*, Mark Fussell*, Vipul Modi*, Mansoor Mohsin*, Ray Kong*, Anmol Ahuja*, Oana Platon*, Alex Wun*, Matthew Snider*, Chacko Daniel*, Dan Mastrian*, Yang Li*, Aprameya Rao*, Vaishnav Kidambi*, Randy Wang*, Abhishek Ram*, Sumukh Shivaprakash*, Rajeet Nair*, Alan Warwick*, Bharat S. Narasimman*, Meng Lin*, Jeffrey Chen*, Abhay Balkrishna Mhatre*, Preetha Subbarayalu*, Mert Coskun*, Indranil Gupta†

†: University of Illinois at Urbana Champaign. *: Microsoft Azure

ABSTRACT

We describe Service Fabric (SF), Microsoft's distributed platform for building, running, and maintaining microservice applications in the cloud. SF has been running in production for 10+ years, powering many critical services at Microsoft. This paper outlines key design philosophies in SF. We then adopt a bottom-up approach to describe low-level components in its architecture, focusing on modular use and support for strong semantics like fault-tolerance and consistency within each component of SF. We discuss lessons learned, and present experimental results from production data.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**; *Distributed architectures*; *Cloud computing*;

KEYWORDS

Microservices, Distributed Systems, Production Systems, Failure Detection, Scheduling

ACM Reference Format:

Gopal Kakivaya*, Lu Xun*, Richard Hasha*, Shegufta Bakht Ahsan†, Todd Pfleiger*, Rishi Sinha*, Anurag Gupta*, Mihail Tarta*, Mark Fussell*, Vipul Modi*, Mansoor Mohsin*, Ray Kong*, Anmol Ahuja*, Oana Platon*, Alex Wun*, Matthew Snider*, Chacko Daniel*, Dan Mastrian*, Yang Li*, Aprameya Rao*, Vaishnav Kidambi*, Randy Wang*, Abhishek Ram*, Sumukh Shivaprakash*, Rajeet Nair*, Alan Warwick*, Bharat S. Narasimman*, Meng Lin*, Jeffrey Chen*, Abhay Balkrishna Mhatre*, Preetha Subbarayalu*, Mert Coskun*, Indranil Gupta† †: University of Illinois at Urbana Champaign. *: Microsoft Azure . 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190546

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '18, April 23–26, 2018, Porto, Portugal

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190546

1 INTRODUCTION

Cloud applications need to operate at scale across geographical regions, and offer fast content delivery as well as high resource utilization at low cost. The *monolithic* design approach for building such cloud services makes them hard to build, to update, and to scale. As a result modern cloud applications are increasingly being built using a *microservices* architecture. This philosophy involves building smaller and modular components (the microservices), connected via clean APIs. The components may be written in different languages, and as the business need evolves and grows, new components can be added and removed seamlessly, thus making application lifecycle management both agile and scalable.

The loose coupling inherent in a microservice-based cloud application also helps to isolate the effect of a failure to only individual components, and enables the developer to reason about fault-tolerance of each microservice. A monolithic cloud application may have disparate parts affected by a server failure or rack outage, often in unpredictable ways, making fault-tolerance analysis quite complex. Table 1 summarizes these and other advantages of microservices.

	Monolithic design	**Microservice-based design**
Application complexity	Complex	Modular
Fault-tolerance	Complex	Modular
Agile development	No	Yes
Communication between components	NA	RPCs
Easily scalable	No	Yes
Easy app lifecycle management	No	Yes
Cloud ready	No	Yes

Table 1: Monolithic Vs. Microservice Applications.

In this paper we describe Service Fabric, Microsoft's platform to support microservice applications in cloud settings. Service Fabric (henceforth denoted as 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.

Today's SF system is a culmination of over a decade and a half of design and development. SF's design started in the early 2000's, and over the last decade (since 2007), many critical production systems inside Microsoft have been running atop SF. These include Microsoft Azure SQL DB [15], Azure Cosmos DB [11], Microsoft Skype [77], Microsoft Azure Event Hub [12], Microsoft Intune [60], Microsoft Azure IoT Suite [14], Microsoft Cortana [59] and others. Today, Microsoft Azure SQL DB running on SF hosts 1.82 Million DBs containing 3.48 PB of data, and runs on over 100 K machines

across multiple geo-distributed datacenters. Azure Cosmos DB runs on over 2 million cores and 100 K machines. The cloud telemetry engine on SF processes 3 Trillion events/week. Overall, SF runs 24×7 in multiple clusters (each with 100s to many 1000s of machines), totaling over 160 K machines with over 2.5 Million cores.

Driven by our production use cases, the architecture of SF follows five major design principles:

- **Modular and Layered Design** of its individual components, with clean APIs.
- **Self-* Properties** including self-healing and self-adjusting properties to enable automated failure recovery, scale out, and scale in. Self-sufficiency, meaning no external dependencies on external systems or storage.
- **Fully decentralized operation** avoids single points of contention and failure, and accommodates microservice applications from small groups to very large groups of VMs/containers.
- **Strong Consistency** both within and across components, to prevent cascades of inconsistency.
- **Support for Stateful Services** such as higher-level data-structures (e.g., dictionaries, queues) that are reliable, persistent, efficient, and transactional.

Service Fabric is the only microservice system that meets all the above principles. Existing systems provide varying levels of support for microservices, the most prominent being Nirmata [64], Akka [3], Bluemix [21], Kubernetes [51], Mesos [44], and AWS Lambda [7]. SF is more powerful: it is the only data-aware 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. Section 9 expands on further differences between SF and related systems.

Service Fabric was built over 16 years, by many (over 100 core) engineers. It is a vast system containing several interconnected and integrated subsystems. It is infeasible to compress this effort into one paper. Therefore, instead of a top-down architectural story, this paper performs a deep dive on selected critical subsystems of SF, illustrating via a bottom-up strategy how our principles drove the design of key low-level building blocks in SF.

The contributions of this paper include:

- We describe design goals, and SF components that: detect failures, route virtually among nodes, elect leaders, perform failover, balance load, and manage replicas.
- We touch on higher-level abstractions for stateful services (Reliable Collections).
- We discuss lessons learnt over 10+ years.
- We present experimental results from real datasets that we collected from SF production clusters.

It is common in industry to integrate disparate systems into a whole. We believe there is a desperate need in the systems community for a paper that reveals insight into how different subsystems can be successfully built and integrated under cohesive design principles.

2 MICROSERVICE APPROACH

The concepts underlying microservices have been around for many years, from object-oriented languages, to Service Oriented Architectures (SOA). Many companies (besides Microsoft), rely on a microservice-based approach. Netflix has used a fine-grained SOA [85] for a long time to withstand nearly two billion edge API requests per day [82].

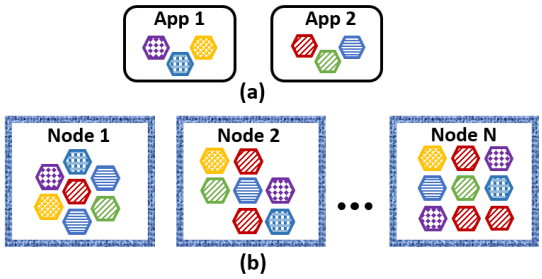

Figure 1: A Microservice-based Application. a) *Each colored/tiled hexagon type represents a microservice, and* b) *Its instances can be deployed flexibly across VMs.*

SF provides first-class support for full Application Lifecycle Management (ALM) of cloud applications, from development, deployment, daily management, to eventual decommissioning. It provides system services to deploy, upgrade, detect, and restart failed services; discover service location; manage state; and monitor health. SF clusters are today created in a variety of environments, in private and public clouds, and on Linux and Windows Server containers.

If such microservices were small in number, it may be possible to have a small team of developers managing them. In production environments, however, there are hundreds of thousands of such microservices running in an unpredictable cloud environment [17, 18, 35, 36, 87]. SF is an automated system that provides support for the complex task of managing these microservices.

Building cloud applications atop SF (via microservices) affords several advantages:

(1) **Modular Design and Development:** By isolating the functionality and via clean APIs, services have well-defined inputs and outputs, which make unit testing, load testing, and integration testing easier.
(2) **Agility:** Individual teams that own services can independently build, deploy, test, and manage them based on the team's expertise or what is most appropriate for the problem to be solved. This makes the development process more agile and lends itself to assigning each microservice to small nimble teams.
SF provides rolling upgrades, granular versioning, packaging, and deployment to achieve faster delivery cycles, and maintain up-time during upgrades. Build and deployment automation along with fault injection allows for continuous integration and deployment.
(3) **Scalability:** A monolithic application can be scaled only by deploying the entire application logic on new nodes (VMs/containers). As Fig. 1 shows, in SF only individual

microservices that need to scale can be added to new nodes, without impacting other services.
This approach allows an application to scale as the number of users, devices and content grows, by scaling the cluster on demand. Incremental deployment is done in a controlled way: one at a time, or in groups, or all at once, depending on the deployment stage (integration testing, canary deployments, and production deployments).

(4) **Resource Management:** SF manages multiple applications running on shared nodes, scaling themselves continuously, because the workloads change dynamically all the time. The components of SF that this paper fleshes out help keep nodes' load balanced, route messages efficiently, detect failures quickly and without confusion, and react to failures quickly and transparently.
(5) **Support for State:** SF provides useful abstractions for stateful services, namely Reliable Collections, a data-structure that is distributed, fault-tolerant, and scalable.

2.1 Microservice Application Model in Service Fabric

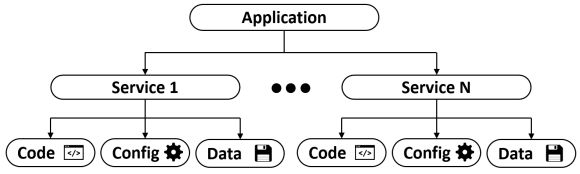

Figure 2: Service Fabric Application Model. *An application consists of N services, each of them with their own Code, Config. and Data.*

In Service Fabric, an application is a collection of constituent microservices (stateful or stateless), each of which performs a complete and standalone function and is composed of code, configuration and data. This is depicted in Fig. 2. The code consists of the executable binaries, the configurations consist of service settings that can be loaded at run time, and the data consists of arbitrary static data to be consumed by the microservice. A powerful feature of SF is that each component in the hierarchical application model can be versioned and upgraded independently.

2.2 Service Fabric and Its Goals

As mentioned earlier, Service Fabric (SF) provides first-class support for full Application Lifecycle Management (ALM) of microservice-based cloud applications, from development to deployment, daily management, and eventual decommissioning. The two most unique goals of SF are:

i) Support for Strong Consistency: A guiding principle is that SF's components must *each* offer strong consistency behaviors. Consistency means different things in different contexts: strong consistent failure detection in the membership module vs. ACID in Reliable Collections.

We considered two prevalent philosophies for building consistent applications: build them atop inconsistent components [2, 88, 89], or use consistent components from the ground up. The end to end principle [76] dictates that if the performance is worth the cost for a functionality then it can be built into the middle. Based on our use case studies we found that a majority of teams needing SF had strong consistency requirements, e.g., Microsoft Azure SQL DB, Microsoft Business Analytics Tools etc., all rely on SF while executing transactions. If consistency were instead to only be built at the application layer, each distinct application will have to hire distributed systems developers, spend development resources, and take longer to reach production quality.

Supporting consistency at each layer: a) allows higher layer design to focus on their relevant notion of consistency (e.g., ACID at Reliable Collections layer), and b) allows both weakly consistent applications (key-value stores such as Azure Cosmos DB) and strongly consistent applications (DBs) to be built atop SF–this is easier than building consistent applications over an inconsistent substrate. With clear responsibilities in each component, we have found it easier to diagnose livesite issues (e.g., outages) by zeroing in on the component that is misbehaving, and isolating failures and root causes between platform and application layers.

ii) Support for Stateful Microservices: Besides the stateless microservices (e.g., protocol gateways, web proxies, etc.), SF supports stateful microservices that maintain a mutable, authoritative state beyond the service request and its response, e.g., for user accounts, databases, shopping carts etc. Two reasons to have stateful microservices along with stateless ones are: a) The ability to build high-throughput, low-latency, failure-tolerant online transaction processing (OLTP) services by keeping code and data close on the same machine, and b) To simplify the application design by removing the need for additional queues and caches. For instance, SF's stateful microservices are used by Microsoft Skype to maintain important state such as address books, chat history, etc. In SF stateful services are implemented via Reliable Collections.

2.3 Use Cases: Real SF Applications

Since Service Fabric was made public in 2015 several external user organizations have built applications atop it. In order to illustrate how global-scale applications can be built using microservices, we briefly describe four of these use cases. Our use cases show: a) how real microservice applications can be built using SF; b) how the microservice approach was preferable to users than the monolithic approach; and c) how SF support for state and consistency (in particular Reliable Collections) are invaluable to developers. (This section can be skipped by the reader without loss in continuity.)

Tutorials are available to readers interested in learning how-to build microservice applications over Service Fabric–please see [62].

I. BMW is one of the largest luxury car companies in the world. Their in-vehicle app *BMW Connected* [22] is a personal mobility companion that learns a user's mobility patterns by combining machine-learned driver intents, real-time telemetry from devices, and up-to-date commute conditions such as traffic. This app relies on a cloud service that was built using SF and today runs on Microsoft Azure, supporting 6 million vehicles worldwide.

The SF application is called BMW's *Open Mobility Cloud (OMC)* [23, 29]. It needs to be continually updated with learned behaviors and from traffic commute update streams. OMC consists of several major subsystems. Among them, we will focus on the core component

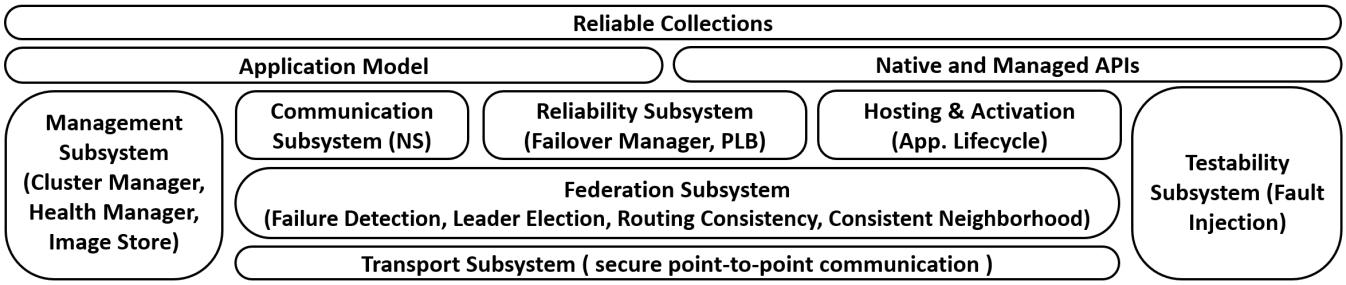

Figure 3: Major Subsystems of Service Fabric. *NS = Naming Service, PLB = Placement and Load Balancer.*

called the Context and Profile Subsystem (C&P). C&P consists of five key SF microservices:

i) **Context API Stateless Service:** Non-SF components communicate with the C&P via this service, e.g., mobile clients can create/change locations and trips.
ii) **Driver Actor Stateful Service:** This per-driver stateful service tracks the driver's profile, and generates notifications such as trip start times. It receives data from five sources: sync messages from the Context API service, a stream of current locations of the driver (from Location Consumer service), learned destinations and predicted trips (from MySense machine learning service), deleted anonymous user IDs (from User Delete service), and trip time estimates (from ETA Queue service).
iii) **Location Consumer Stateless Service:** Each mobile client sends a stream of geo-locations to the Microsoft Azure Event Hub, pulled by the Location Consumer service and fed to the appropriate driver actor.
iv) **Commute Service:** The Commute service takes geo-location and trip start and end points, and then communicates with an external service to generate drive time.
v) **ETA Queue Stateful Service:** This decouples driver actors from the Commute server and allows asynchronous communication between the two services.

The use of SF makes BMW's C&P Subsystem highly-available, fault-tolerant, agile, and scalable. For instance, when the number of active vehicles increases, the Context API service and Driver actor services are scaled out in size. When the number of moving vehicles changes, the Location Consumer and ETA Queue stateful services can be scaled in size. The remaining services remain untouched. SF helps to optimize resource usage so that incurred dollar costs of using Microsoft Azure are minimized.

II. Mesh Systems [30, 58] is an 11-year old company that provides IoT software and services for enterprise IoT customers. They started out with a monolithic application that was too complex, and were unable to accommodate the needs of their growing business. This previous system also underutilized their cluster.

Mesh Systems's SF application achieves high resource utilization, and scalability by leveraging Reliable Collections. One of their needs was to scale out the payload processing independent of notifications, and it was a good match with SF's ability to scale out individual microservices. Their SF application also leverages local state to improve performance, e.g., to minimize the load on Microsoft Azure SQL DB, they implemented an SQL broker that periodically caches the most heavily-accessed metadata tables.

III. Quorum Business Solutions [68, 69] is a SCADA company that collects and manages data from field-operations platforms on tens of thousands of wells across North America. Their implementation on SF uses actors that reliably collect and process data because they are stateful, a stateless gateway service for auto-scalability, and a stateful batch aggregator service that monitors actors themselves. They implement interactions with third parties (SQL DB, Redis) via notification and retry microservices in SF.

IV. TalkTalk TV [31, 81] is one of the largest cable TV providers in United Kingdom. It delivers the latest TV and movie content to a million monthly users, via a variety of devices and smart TVs. Their SF application is used to encode movie streams before delivery to the customer, and uses stateful services, structured in a linear sequence: record encoding requests, initiate encoding processes, and track these processes. A stateless gateway interacts with clients.

3 SERVICE FABRIC: KEY COMPONENTS

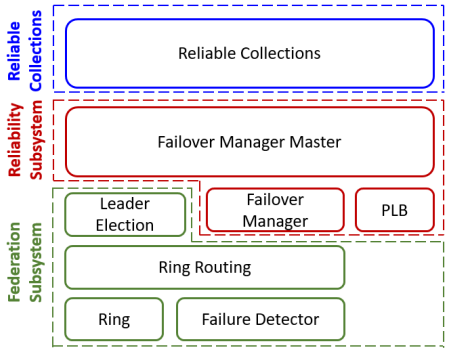

Figure 4: Federation and Reliability Subsystems: Deep-Dive.

Service Fabric (SF) is composed of multiple subsystems, relying on each other modularly via clean APIs and protocols. Fig. 3 depicts how they are stacked–upper subsystem layers leverage lower layers. Given space constraints, this paper largely focuses on SF's most unique components, shown in Fig. 4. These lie in two subsystems: Federation and Reliability.

The *Federation Subsystem* (Sec. 4) forms the heart of SF. It solves critical distributed systems problems like failure detection, a consistent ring with routing, and leader election. The *Transport Subsystem* underneath provides secure node-to-node communication.

Built atop the Federation Subsystem is the *Reliability Subsystem* (Sec. 5), which provides replication and high availability. Its components are the Failover Manager (FM), Failover Manager Master (FMM), the Placement and Load Balancer (PLB), and replication protocols. This helps create distributed abstractions named Reliable Collections (Sec. 6).

Other SF subsystems not detailed in this paper include the Management Subsystem which provides full application and cluster lifecycle management via the Cluster Manager, Health Manager, and Image Store. The Communication Subsystem allows reliable service discovery via the Naming Service. The Testability Subsystem contains a Fault Injection Service. Hosting and Activation Subsystems manage other parts of the application lifecycle.

4 FEDERATION SUBSYSTEM

We describe SF's ring, failure detection, consistent routing, and leader election.

4.1 Basic SF-Ring

Nodes in SF are organized in a virtual ring, which we call *SF-Ring*. This consists of a virtual ring with 2^m points (e.g., $m = 128$ bits). Nodes and keys are mapped on to a point in the ring. A key is owned by the node *closest* to it, with ties won by the predecessor. Each node keeps track of multiple (a given number of) its immediate successor nodes and predecessor nodes in the ring–we call this the *neighborhood* set. Neighbors are used to run SF's membership and failure detection protocol, which we describe next.

Nodes also maintain long-distance routing partners. Section 4.3 will later outline these and consistent routing.

4.2 Consistent Membership and Failure Detection

Membership and failure detection in SF relies on two key design principles:

- **Strongly Consistent Membership:** All nodes responsible for monitoring a node X must agree on whether X is up or down. When used in the SF-Ring, this entails a *consistent neighborhood*, i.e., all successors/predecessors in the neighborhood of a node X agree on X's status.
- **Decoupling Failure Detection from Failure Decision:** Failure detection protocols can lead to conflicting detections. To mitigate this, we decouple the *decision* of which nodes are failed from the *detection* itself.

4.2.1 Lease-based Heartbeating. We first describe our heartbeating protocol in general terms, and then how it is used in SF-Ring.

Monitors and Leases: Heartbeating is fully decentralized. Each node X is monitored by a subset of other nodes, which we call its *monitors*. Node X periodically sends a lease renewal request (*LR*, heartbeat message with unique sequence number) to each of its monitors. When a monitor acknowledges (LR_{ack}), node X is said to obtain a lease, and the monitor guarantees not to detect X as failed for the leasing period. The leasing period, labeled T_m, is adjusted adaptively based on round trip time and some laxity, but a typical value is 30 s. To remain healthy, node X must obtain acks (leases) from *all* of its monitors. This defines strong consistency. If node X fails to renew *any* of its leases from its monitors, it considers removing itself from the group. If a monitor misses a heartbeat from X, it considers marking X as failed. In both these cases however, the final decision needs to be confirmed by the arbitrator group (described in Sec. 4.2.2).

Lease renewal is critical, but packet drops may cause it to fail. To mitigate this, if node X does not receive LR_{ack} within a timeout (based on RTT), it re-sends the lease message *LR* until it receives LR_{ack}. Resends are iterative.

Symmetric Monitoring in SF-Ring: The monitors of a node are its neighborhood (successors and predecessors in the ring). Neighborhood monitoring relationships are purely *symmetric*. When two nodes X and Y are monitoring each other, their lease protocols are run largely independently, with a few exceptions. First, if X fails to renew its own lease within the timeout, it denies any further lease requests from Y (since X will leave the group soon anyway). Second, if X detects Y as having failed, X stops sending lease renew requests to Y. Such cases have the potential to create inconsistencies, however our use of the arbitrator group (which we describe next) keeps the membership lists consistent.

*4.2.2 **Using the Arbitrator Group to Decouple Detection from Decision.*** **Decoupling Failure Detection from Decision:** Decentralized failure detection techniques carry many subtleties involving timeouts, indirection, pinging, etc. Protocols exist that give eventual liveness properties (e.g., [32, 83]), but in order to scale, they allow inconsistent membership lists. However, our need is to maintain a strongly consistent neighborhood in the ring, and also reach decisions quickly.

To accomplish these goals, we decouple *decisions* on failures from the act of detection. Failure detection is fully decentralized using Sec. 4.2.1's lease-based heartbeating. For decisions, we use a *lightweight* arbitrator. The arbitrator does not help in detecting failures (as this would increase load), but only in affirming or denying decisions.

Arbitrator: 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.

Failure reporting to/from an arbitrator node works as follows. Suppose a node X detects Y as having failed. X sends a fail(Y) message to the arbitrator. If the arbitrator already marked X as failed, the fail(Y) message is ignored, and X is again asked to leave the group. Otherwise, if this is the first failure report for Y, it is added to a *recently-failed list* at the arbitrator. 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_o, so that X can wait for T_o time and then take actions w.r.t. Y (e.g., reclaim Y's portion of the ring).

When Y next attempts to renew its lease with X (this occurs within T_m time units after X detects it), X either denies it or does not respond. Y sends a fail(X) message to the arbitrator. Since Y is already present in the recently-failed list at the arbitrator, Y is asked to leave the group. (If this exchange fails, Y will leave anyway as it failed to renew its lease with X.) If on the other hand, Y's lease renewal failed because X was truly failed, then the arbitrator sends an accept(fail(X)) message to Y. 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.

In SF-Ring: Inside SF-Ring, failure detections occur in the neighborhood, to maintain a consistent neighborhood. If node X suspects a neighbor (Y), it sends a fail(Y) to the arbitrator, but waits for T_o time after receiving the accept(.) message before reclaiming the portion of Y's ring. Any routing requests (Section 4.3) received meanwhile for Y will be queued, but processed only after the range has been inherited by Y's neighbors.

Arbitrator State: In the arbitrator group, each arbitrator keeps relatively small information, such as the recently-failed list containing recent failure reports and decisions. Entries in this list time out after T_m time units. When a new arbitrator joins the group, for the first T_m seconds it rejects all failure requests (this is a conservative approach). After quick initialization it moves to normal operations as described earlier. This prevents: a) decisions by a new arbitrator conflicting with those by existing arbitrators, and b) spurious nodes, i.e., failed nodes continuing to persist in membership lists (a known issue in distributed membership protocols [83]). This ensures that a detected node leaves before being forgotten.

Conflict Resolution: The arbitrator group helps decide on both simple and complex conflicts. A common simple conflict is two nodes detecting each other as failed–the first received fail(.) message (or the first one to win quorum among arbitrators) wins in this case. An alternate variant of our arbitrator pings both such nodes and if they are healthy heals their membership lists and allows them to stay.

Network congestion or partitions may result in multiple nodes detecting each other as failed. In traditional DHTs like Chord [79], Pastry [75], this causes inconsistent membership lists. In NoSQL systems like Cassandra [52], it can lead to inconsistency in the ring. SF's arbitrator group essentially automates the conflict resolution procedure.

The decoupling of detection and decision helps the arbitrators catch and nip complex *cascading detections*. For instance, consider a node X that fails to renew its lease and thus voluntarily leaves. If another node Y immediately happens to send a lease request to X (before Y has been informed about X), Y will not receive an ack and will also leave–this process can cascade and result in many healthy nodes leaving. SF's arbitrators catch the first detection, and immediately make the neighborhood consistent, thus stopping the cascade early.

Vs. Related Work: SF's leases are comparable to heartbeat-style failure detection algorithms from the past (e.g., [83]). The novel idea in SF is to use lightweight arbitrator groups to ensure membership stays consistent (in the ring neighborhood). This allows the membership, and hence the ring, to scale to whole datacenters. Without the arbitrators, distributed membership will have inconsistencies (e.g., gossip, SWIM/Serf [32]), or one needs a heavyweight central group (e.g., Zookeeper [45], Chubby [24]) which has its own issues. Stronger consistent membership like virtual synchrony [19, 20] do not scale to datacenters.

4.3 Full SF-Ring and Consistent Routing

We describe the full SF-Ring, expanding on the basic design from Section 4.1. SF-Ring is a distributed hash table (DHT). It provides a seamless way of scaling from small groups to large groups. SF-Ring was developed internally [42, 43, 47, 48] in Microsoft, in the early 2000s, concurrent with the emergence of P2P DHTs like Pastry, Chord [75, 79], and others [39, 40, 57, 70]. We describe our original design, and inline evolutionary changes that occurred over time.

SF-Ring is unique in the following five ways (I-V):

I) Routing Table entries are bidirectional and symmetric: SF-Ring maintains *Routing Partners* (in Routing Tables) at exponentially increasing distances in the ring. As shown in Fig. 5, routing partners are maintained both clockwise and anticlockwise. That is, the i^{th} clockwise routing table entry is the node whose ID is closest to the key $(n + 2^i) mod (2^m)$, while the i^{th} anticlockwise routing table entry is the node with ID closest to $(n - 2^i) mod (2^m)$.

Due to the bidirectionality, most routing partners are *symmetric*. This speeds up both spread of failure information and routing. P2P DHTs like Chord maintain exponentially far routing entries, but are unidirectional and largely not symmetric. Symmetric links lead to efficient transfer of data between nodes, fast spreading of failure information, and fast updating of routing tables after node churn [1].

II) Routing is bidirectional: When forwarding a message for a key, a node searches its routing table for the node whose ID is *closest* to the key, and forwards it. This is possible only because the routing tables are bidirectional and symmetric. This greedy routing is essentially a distributed version of binary search. This approach: i) allows the message to move both clockwise and anticlockwise, always taking the fastest path, and ii) avoid routing loops. In practice we noticed that once a message starts routing it tends to maintain its direction (clockwise or anticlockwise), until the last few hops, when directional changes may occur.

Compared to traditional DHTs like Chord which use clockwise routing, SF-Ring's bidirectional routing: i) routes messages faster, ii) provides more routing options when routing table entries are stale or empty, iii) spreads routing load more uniformly across nodes, and iv) due to the distributed binary search, avoids routing loops even under stale routing tables.

[1] Symmetry may be violated in a small fraction of cases when another node is closer to $(n - 2^i) mod (2^m)$, but our advantages still hold.

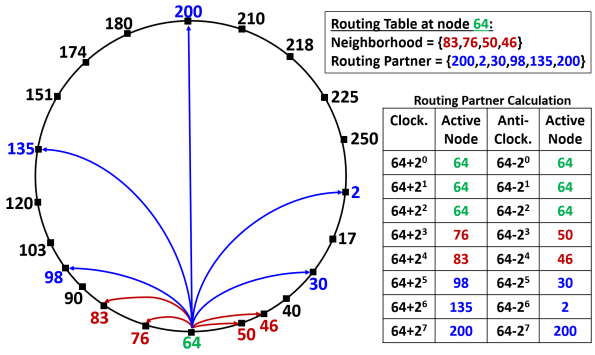

Figure 5: Routing Table *of node 64. The ring has* $2^{m=8}$ *points. Numbered dots represent active nodes.*

Changes to Routing over the Years: Once a building block is designed, its usage evolves over the years based on needs. SF-Ring's routing is no exception. Originally all messages were routed. Today, SF-Ring routing is used for: a) discovery routing when a node starts up, and b) routing to virtual addresses. After discovery when a source knows the destination's IP address, it communicates directly.

III) Routing Tables are eventually convergent: SF nodes use a chatter protocol to continuously exchange routing table information. Due to the symmetric nature of routing relationships, failure information propagates quickly leading to fast and eventual convergence of affected routing table entries.

When a node joins the ring, it goes through a transitional phase during which it initializes its routing tables, acquires tokens (described soon), but does not yet route messages. Once it finishes transitioning it starts routing messages.

In the chatter protocol, nodes periodically send liveness messages to their routing table partners. To efficiently use bandwidth, liveness messages also allow piggybacking of application messages that are about to be sent to the partner. Liveness messages contain information about the node, instance ID (distinguishes multiple rejoins by the same node), phase (e.g., transitioning or operational), weight (reputation based on uptime, etc.), and a freshness value (which decays with time, like in ad-hoc routing protocols [46, 66]).

The chatter protocol provides eventual consistency for the long distance neighbors (routing partners). A key result from the SF effort is that strongly consistent applications can be supported at scale by combining strong membership in the neighborhood (Section 4.2) with weakly consistent membership across the ring. Literature often equates strongly consistent membership with virtual synchrony [19], but this approach has scalability limits [20].

Changes to Partial Membership over the Years: SF microservices operate in a wide range of scales, from a few nodes to many 1000s of nodes. Microservices also need to scale out and scale in during their lifetime. To support this, today SF-Ring sets a (customizable) upper bound on the number of entries in the routing table. If the number of nodes is smaller than this bound, the routing tables (eventually but quickly) capture full information about all nodes; this makes routing fast and take $O(1)$ hops. If the number of nodes is higher, routing tables come into effect, creating an $O(log(N))$ lookup cost without blowing up memory.

This design decision allows SF-Ring to move seamlessly between partial membership and full membership. In comparison, NoSQL ring-based DHTs like Cassandra [52] and Dynamo [33] rely on full $O(N)$ membership. This makes them cumbersome at large scale and under churn–the overhead of maintaining correct membership lists outweighs benefits of 1 hop routing. In Cassandra, admins need to use *nodetool* [1] to manually verify that membership lists are correct. SF-Ring automates all membership management.

IV) Decoupled Mapping of Nodes and Keys: Nodes and objects (services) are mapped onto the ring in a way that is decoupled from the ring: *i)* nearby nodes on the ring are selected preferably from different failure domains (improving fault-tolerance), and *ii)* services are mapped in a near-optimal and load balanced way (Sec. 5.3), and not hashed.

V) Consistent Routing Tokens: Each SF node owns a *routing token*, a portion of the ring whose keys it is responsible for. The SF-Ring protocol ensures two *consistency* properties: *i) always safe*: there is no overlap among tokens owned by nodes, and *ii) eventually live:* every token range is eventually owned by at least one node. When the first SF node bootstraps, it owns the entire token space. Thereafter, tokens are created as follows: two immediate neighbors split the ring segment between them at exactly the half-way point.

Upon churn, a node join/leave protocol automatically transfers tokens among nodes. NoSQL systems like Cassandra [52] also use routing tokens, but may need manual involvement to ensure correctness (via nodetool). In SF-Ring, this checking is automatic and continuous.

When a node leaves, its successor and predecessor split the range between them halfway. If a node X's immediate successor Y fails, then X and its new successor Z will split the ring segment halfway between X and Z. In the common case this splitting incurs no communication between X and Z.

If all nodes satisfy token liveness and safety conditions, SF-Ring routing will eventually succeed. If the liveness condition is not yet true (e.g., no node owns a token containing destination ID), routing messages are queued.

Vs. Related Work: SF's consistent ring was invented internally around 2002, concurrent with the first DHTs like Chord [79] and Pastry [75]. While SF was being implemented, several other DHTs came out that used bidirectional routing, e.g., Kademlia [57]. While we could conceivably go back and try replacing SF-Ring with something like Kademlia, re-integration is hard and SF-Ring has been running successfully in production for a decade (if it ain't broke, don't fix it!).

4.4 Leader Election

SF-Ring's leader election protocol builds atop the combination of the ring, routing, and consistent neighborhood just described. 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 $k = 0$ (e.g., FMM in Sec. 5.1).

5 RELIABILITY SUBSYSTEM

The Reliability Subsystem is in charge of replication, load balancing, and high availability. Objects in SF are replicated at a primary node and multiple secondary nodes. The replication subsystem's *Replicator* component uses passive replication: clients communicate with the primary, which multicasts updates to secondaries. The Reliability Subsystem contains three major components: Failover Manager (FM), Naming, and Placement and Load Balancer (PLB).

5.1 The Failover Manager (FM)

This stateful SF service maintains a global view of all replica groups. The global view includes status of all nodes in the cluster, list of current applications and services, list of replicas and their placement, etc.

The FM manages creation of services, upgrades, etc. It works closely with daemons on each node called Reconfiguration Agents (RAs), which continually collect the node's available memory, CPU utilization, disk and network access behaviors, etc. The FM coordinates with the Placement and Load Balancer (PLB) (Sec. 5.3). The FM periodically receives load reports from the RAs running on each node, aggregates, and sends it to the PLB. Newly joined nodes explicitly inform the FM, and failures are detected via the mechanisms of Sec. 4.2 and relayed from the arbitrator to the FM as they occur. The FM's main actions are:

(1) **Create a Replica:** When either: *a)* a replica is created for the first time, or *b)* a replica goes down and FM has to recreate it. In both cases FM consults with PLB which decides the placement for services/replicas, and FM initiates the placement.
(2) **Move a Replica:** When an imbalance occurs, PLB calculates a replication migration plan, and FM executes it.
(3) **Reconfiguration:** If a primary replica goes down, the FM selects a secondary replica and promotes it as primary. If the old primary comes up, it is marked as a secondary.

The failure of an entire FM (replica set), though rare, still needs fast recovery. This is handled by another stateless service called the **Failover Manager Master (FMM)**, which runs the same logic as the FM, except that it manages the FM instead of the microservices.

If an FM fails the FMM restarts it quickly with cached state. If the FMM itself fails, it reconstructs its state from scratch by querying the SF-Ring. In SF-Ring, the FMM is elected consistently using the election protocol of Sec. 4.4, i.e., as the node whose token range contains the ID 0.

5.2 Naming and Resolution

Service Fabric's Naming Service maps service instance names to the endpoint addresses they listen on. All service instances in SF have unique names represented as URIs– a typical format is `SF:/MyApplication/MyService`. The name of the service does not change over its lifetime, only the endpoint address binding can change (e.g., if the service is migrated). Full names are DNS-style hierarchical names, e.g.,
`http://mycluster.eastus.cloudapp.microsoft.com:19008/MyApp/MyService?PartitionKey=3&PartitionKind=Int64Range`. This allows DNS to resolve the prefix, and SF's Naming Service to resolve the rest. The FM (Sec. 5.1) also caches name-target mappings for fast resolution and to make fast decisions upon failures.

5.3 Placement and Load Balancer (PLB)

The Placement and Load Balancer (PLB) is a stateful SF service in charge of placing replicas/instances (of microservices) at nodes and ensuring load balance. 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. It takes into account: i) available resources at all nodes (e.g., memory, disk, CPU load, traffic, etc.), ii) conceptual resources (e.g., outstanding requests at a particular service), and iii) parameters of typical requests (e.g., request size, frequency, diurnal variation, etc.). The PLB's continuous role is to move sets of services from overly exhausted nodes to underutilized nodes. It also moves services away from a node that is about to be upgraded or is overloaded due to a long workload spike.

Large State Space: In practice the PLB needs to deal with a state space that is both huge (hundreds of different metrics and values, conflicting requirements, etc.), and occasionally quite constrained (e.g., placement of services only on certain nodes, fault-tolerance by avoiding replica colocation, etc.). A typical scenario involves tens of thousands of objects, replicated 3-ways, but spread over only a few hundred nodes. Worse, things change frequently: which resources are important, how many resources a particular workload is actually consuming, what the workload's constraints are, which nodes are failing and joining, etc., all change during the runtime of the service. This means that the decision taken currently might not be valid in future. Therefore, it is better to continuously make small improvements and re-evaluate them later. Quick and nimble decisions are preferable over algorithms that try to reach an optimal state but use up a lot of resources to explore the state space.

Simulated Annealing: In order to select a near-optimal placement of objects across nodes given the above constraints, the PLB uses simulated annealing [50]. We initially attempted to use LP/IP-based and genetic algorithms [25, 38, 63] but found they either took too long to converge or gave solution which were far from optimal. We picked simulated annealing as it bridged these worlds: it is both fast and close to optimal.

Simulated annealing calculates an energy for each state. PLB's energy function is user-definable but a common case is as the average standard deviation of all metrics in the cluster, with a lower score being more desirable. The simulated annealing algorithm sets a timer (default values later) and then explores the state space until either the timer expires or until convergence. 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 $e^{-\frac{d}{T}}$. This temperature T is high in initial steps (allowing jumps away from local minima) but falls linearly across iterations to allow convergence later.

The move chosen in each step is fine-grained. Examples include moving a secondary replica to another node, swapping primary

and secondary replica, etc. SF only considers valid moves that satisfy constraints: i) under which the PLB operates, and ii) for fault-tolerance. For instance, the PLB cannot create new nodes, nor can it move a primary replica to colocate with a secondary replica of the same partition.

SF supports two modes of annealing: fast mode (10 s timer value), and a slow mode (120 s timer) that is more likely to converge to the optimal. During initial placement we run annealing for only 0.5 s.

When the annealing ends, the energy of the system's current state is recalculated (as it may have changed), and the new state is initiated only if it actually improves the energy. Moves are compacted using transitivity rules and are sent to the FM to execute.

6 RELIABLE COLLECTIONS

Reliable Collections provide stateful services in SF. All the use cases described in Sec. 2.3 directly used Reliable Collections. Internally its biggest users are Microsoft Intune and Microsoft CRM Service.

SF's Reliable collections include Reliable Dictionary and Reliable Queue, available as classes in popular software programming frameworks. These data structures are:

- **Available and Fault-tolerant**: Via replication;
- **Persisted**: Via disk, to withstand server, rack, or datacenter outages;
- **Efficient**: Via asynchronous APIs that do not block threads on IO;
- **Transactional**: Via APIs with ACID semantics.

A key difference between storage systems built using SF APIs (e.g., Reliable Collections) and other highly-available systems (such as Microsoft Queue Storage [9], Microsoft Table Storage [10], and Redis [71]) is that the state is kept locally in the service instance while also being made highly available. Therefore, the most common operations i.e., reads, are local.

Writes are relayed from primary to secondaries via passive replication, and are considered complete when a quorum of secondaries acknowledge it. Further extension points allow an application to achieve weaker consistency by relaxing where the read can go, e.g., "always read from primary" to "read from secondary." Our users who build latency-sensitive applications find this particularly useful.

Applications can quickly failover from a failed node to a hot standby replica. Groups of applications can be migrated from one node to another during maintenance such as for patching or planned restarts.

Benefits of Reliance on Lower Layers: Reliable Collections leverage the components described previously in this paper. Replicas are organized in an SF-Ring (Sec. 4.3), failures are detected (Sec. 4.2), and a primary kept elected (Sec. 4.4). Periodically, as well as when replica changes occur (node joins, failures, leaves, etc.), FM+PLB (Sec. 5) keeps the 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, available, self-*, and upgradable. The developer only has to program with the Reliable Collections API; because lower layers assure consistency, she does not have to reason about those. Today there are 1.82 Million such transactional DBs over SF (100K machines).

7 LESSONS LEARNED

Over the past decade of running Service Fabric (SF) to support internet-scale services in production, several of our decisions stood the test of time while others had to be revisited/revised.

7.1 Decisions We Had to Revisit

Distributed systems are more than nodes and network: Applications are processes running on nodes and can fail in ways that do not always lead to total failure of the node. A common occurrence is something we have come to term as "Grey Node Failure". In these cases, the OS continued to work without the presence of a fully functional OS disk. The absence of a functional disk renders the application unhealthy. However, the SF leasing mechanism continues to run at high priority without any page faults since it does not depend on disk. We have since added an optional *disk heartbeat* in our leasing to work around this type of issue.

Application/Platform responsibilities need to be well isolated: Early on, SF's customer base was large internal teams who had systems expertise in building internet-scale services (e.g., Azure Cosmos DB). Our initial application interaction model was designed for such teams who had close collaboration with us. They always conformed to the requirements of the SF APIs. One simple example was when the platform needed to close a replica, SF would call `close` on the application code and wait for the replica close to complete. This allowed the replica to perform proper clean up. The same model does not necessarily work with a larger set of application developers who have bugs or take a long time to cleanly shutdown. We have since made changes to SF where the `close` of the replica getting stuck does not cause availability loss and the system moves past it after a configurable close timeout.

Capacity planning is the application's responsibility (but developers need help): At the platform level we cannot completely foresee application capacity requirements and have been frequently asked to investigate increased latency issues that arise when the application is driving the machine beyond its capacity like IO or memory. We found this out the hard way when some of the core Microsoft Azure services kept having issues due to under-provisioning. Migrating to larger hardware was the only solution. We have since instituted that all applications explicitly specify their capacity requirements like CPU, Memory, IO, etc. The system now ensures application containers run on sufficiently-provisioned machines and enforces developer-specified limits on these containers, so that developers gain the insight and accountability for the capacity limits they need to specify.

Different Subsystems Require Different Investments: The Federation Subsystem was the most intellectually challenging and took up the majority of time during the overall 15+ years of development (in these early days, the SF team was small in size). It was very important to get this substrate right, as its correctness and consistency was critical in order to be able to build SF's remaining sub-systems above it. In comparison, the failover capabilities in the

Reliability subsystem required far more human-hours as they had a larger number of moving parts, and were amenable to many more optimizations. The team was also larger by this later stage of the SF project.

7.2 Decisions That Stood The Test of Time

Monitored upgrades and clean rollbacks of platform upgrades allow faster releases and give customers confidence: Customers often avoid upgrading (to the latest SF release) because of fears around consequences of bad upgrades. SF handles this via an automatic roll-back mechanism if an upgrade starts causing health issues in the cluster. Rollbacks might be needed because of a bug in the application where upgrade was not being properly handled, or a bug in the platform, or a completely different environmental reason. We have worked extremely hard to ensure auto-roll-back works smoothly, and it has given our customers immense confidence in upgrading quickly, as they know that bad upgrades will be rolled back automatically. This also allows us to quickly iterate and ship faster. Cases where rollbacks have gotten stuck are relatively rare.

Changes to the system should be staged: SF upgrades (for code or configuration) have always been staged. Some customers tried to work around SF staged deployments by performing a silent configuration deployment via external configuration stores. While this gave the perception of faster upgrades, almost all these cases eventually caused outages due to fat fingering. Today most SF customers understand the value of orchestrated upgrades and adhere to them.

Health reporting leads to better lifecycle management and easier debugging: Applications can tap into SF's Health Store to ensure that application+platform upgrades are proceeding without availability loss. The Health Store also allows for easier debugging of the cluster due to metrics it collects. This service also increases customer confidence.

"Invisible" external dependencies need care: External dependencies like DNS, Kerberos [49], Certificate Revocation Lists, need to be clearly identified. Application developers need to be cognizant of how failures of these dependencies affect the application. We had an incident where a customer was hosting a large SF cluster using machines with Fully-Qualified Domain Names (FQDNs), which meant that the SF cluster needed a DNS server for FQDN resolution. The customer did the right thing in ensuring that the DNS service is highly available by replicating it. However, DNS replication is only eventually consistent. This meant that occasionally the same FQDN was resolving to two different machines (IPs), leading to availability outages in the system. To resolve this issue, the customer switched back to directly using IP addresses instead of FQDNs.

8 EVALUATION

We evaluate the most critical aspects of Service Fabric: failure detection and membership, message delay, reconfiguration, and SF-Ring. Where available, we present results from production data (Sections 8.3, 8.4, 8.5). We use simulations in cases where we need to compare to alternative designs in a fair way (Sections 8.1, 8.6), or measure algorithmic overhead (Sections 8.1, 8.2).

We have made Service Fabric binaries available [62]. We are looking into sharing datasets, however we are limited by compliance reasons and proprietary issues. We are working on open-sourcing the code for SF.

8.1 Benefits of the Arbitrator

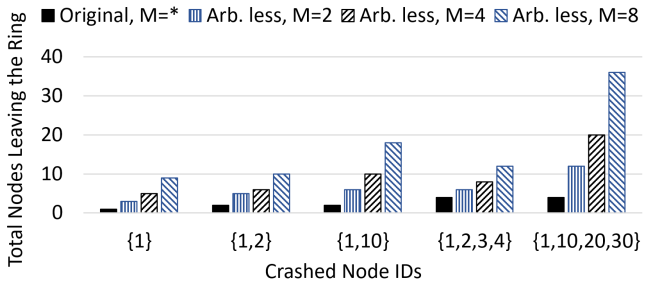

Figure 6: Comparison: Arbitrator Vs. Arbitrator less scheme. *Arbitrator handles cascading failures and reduces the number of nodes leaving the system. M = number of monitor per node.*

To show that SF's arbitrator mechanism (Sec. 4.2.2) efficiently helps maintain consistent membership, we compare it to an arbitrator-less mechanism we designed. In the latter approach, when a node fails to renew its lease, instead of contacting the arbitrators, it coordinates with its neighbors and then gracefully leaves the system. Neighbors communicate amongst each other to keep membership consistent.

Due to timeouts, both mechanisms may force good nodes to leave. Fig. 6 shows, for various failure scenarios, the total number of such false positives. We observe that SF's original arbitrator approach incurs far fewer false positives than the arbitrator-less scheme. In fact, the number of false positives under an arbitrator based scheme grows much slower (with number of failure) than under the arbitrator-less scheme. This is because of cascading failure detections (Section 4.2.2), while SF's arbitrator prevents such cascades.

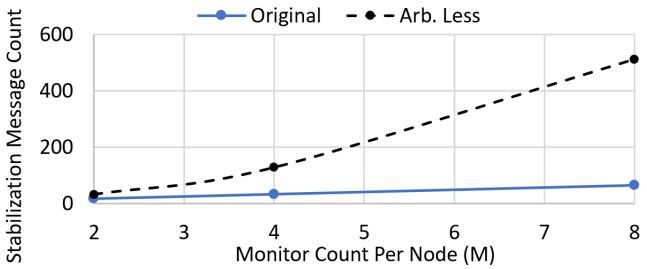

Figure 7: Stabilization Message Count: Arbitrator Vs. Arbitrator less scheme. *Crashed Node set {1, 10, 20, 30}. M = number of monitor per node.*

Fig. 7 shows how many messages are needed to stabilize the ring, after a failure. As we increase the number of monitors per node (M), SF's arbitrator's overhead grows slower than the arbitrator-less scheme. In fact, analytically, these overheads are linear and quadratic respectively. When using arbitrators, a failure causes the

M monitors of a failed node to perform a request-reply to the arbitrator ($2M$ messages). In arbitrator-less approaches, a failure causes all M monitors to communicate with each other ($2M^2$ messages).

8.2 Failure Detector Overhead

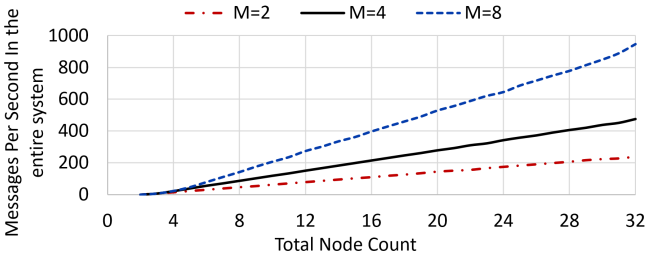

Figure 8: Failure Detector (FD) message overhead. *Cluster messages increase linearly with cluster size. M = number of monitors per node.*

Fig. 8 shows the *total cluster* overhead of the leasing mechanism (Sec. 4.2.1). For each M, cluster load scales linearly with the number of nodes (production SF uses $M = 4$ monitors per node). Hence SF's leasing mechanism incurs per-node overhead that is constant and scalable, independent of cluster size.

8.3 In Production: Arbitrator Behavior

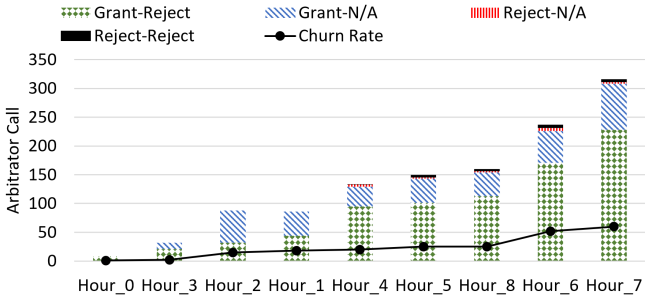

Figure 9: Arbitrator Call count per hour (total 9 hours of traces). *Hours are rearranged based on the churn rate.*

Fig. 9 evaluates the load on the arbitrator group. The data is from 9 hours of a 225+ machine production cluster. Each machine hosts an expected 4 SF instances. Below, we call each of these instances a "node". We sort trace hours in increasing order of churn, and the plot shows both event counts and hourly churn rate.

We explain the 4 event types. When a node A detects failure of node B and contacts the arbitrator, there are four possible outcomes: *i) Grant-Reject:* both nodes A and B send arbitration requests and only one is granted; *ii) Reject-Reject:* both nodes A and B send arbitration requests, and both of them are rejected; *iii) Grant-N/A:* only one node in a pair sends request and succeeds; *iv) Reject-N/A:* only one node in a pair sends request and is rejected.

First we observe that a majority of hours have medium churn with between 10-15 nodes churned per hour (Hours_2, 1, 4, 5, 8). Only 22% of hours (Hours_6, 7) have very high churn, and another 22% have low churn (Hours_0, 3). Second, the number of duplicate messages received at the arbitrator, and the wholesale rejections at arbitrator startup (first T_m time units: see Sec. 4.2.2), are together captured by the sum of Reject-Reject and Reject-N/A events (two topmost bar slivers). This is small at medium churn (Hours_4, 5, 8), and does not increase much at high churn (Hours_6, 7). Hence we conclude that: *i)* the arbitrator's effort is largely focused on resolving new detection conflicts rather than re-affirming past decisions to errant nodes; and *ii)* false positives due to arbitrator startup are small in number. Our data also indicates SF nodes leave quickly after they are asked to.

8.4 In Production: Message Delay Under Churn

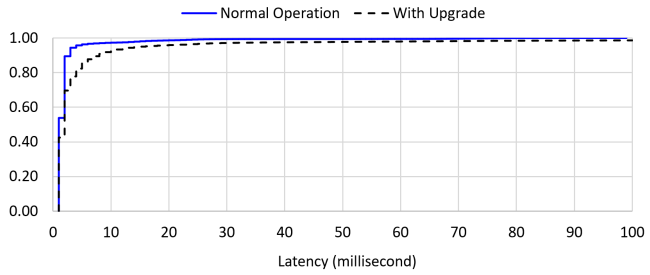

Figure 10: CDF of Message Delay under Churn. *Normal Operation has lower churn than with Upgrade.*

	1^{st} Perc.	5^{th} Perc.	50^{th} Perc.	95^{th} Perc.	99^{th} Perc.
No Churn	1	1	1	4	24
Churn	1	1	2	14	175

Table 2: Tail latency: *Message Delay (millisecond).*

Fig. 10 measures the total message delay (including routing latency) in a 24 hour trace of 205 VMs across 3 data-centers. Each VM is equipped with 24 cores, 168 GB RAM, 3×1.81 TB HDD and 4×445 GB SSD.

The plot shows the latency CDF for two scenarios: i) *Normal Operation*, prone to natural churn, e.g., due to failures; and ii) *With Upgrade*, when there is higher churn due to node upgrades, system upgrades, service upgrades, etc. Going from normal operation to upgrades, the 80^{th} percentile latency remains largely unaffected. Table 2 shows the median latency rises only two-fold, from 1 ms to 2 ms. $95th$ percentile latency rises a modest 3.5×. We conclude that SF deals with churn and upgrades in a low-overhead way.

8.5 In Production: Reconfiguration Time

Reconfigurations triggered by service failure, overloaded nodes, service upgrade, machine failure, system upgrades, etc., are handled by the FM and PLB (Sections 5.1, 5.3).

Failover	SwapPrimary	Other
1%	20%	79%

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

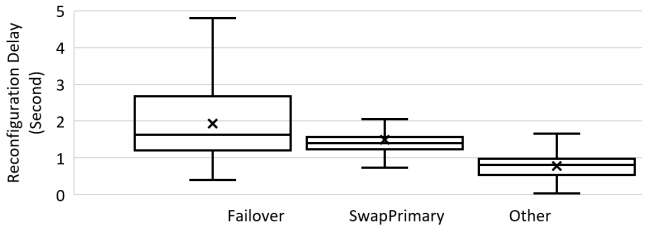

Figure 11: Statistics of different Reconfiguration Delays in the 20 day trace. *Candlestick plots show the 1^{st} and 99^{th} percentiles, 1^{st}, 2^{nd} and 3^{rd} quartiles and the average (X's).*

We collected a 20 day trace with 3 Million events from the same production cluster as Sec. 8.4. Table 3 shows a breakdown by reconfiguration type. Only Failover and SwapPrimary events affect availability (total 21%). Fig. 11 shows that SF makes control decisions about these two types of events quickly. The average time to perform failover is 1.9 s, and 99^{th} percentile is 4.8 s. While "Other" events constitute 79%, they do not affect data availability as they deal with per-replica reconfiguration, and are quite fast.

Fig. 12 depicts a timeline over 6 days of these 3 event types. Large spikes are due to pre-planned upgrades of infrastructure, application, and SF. Otherwise, we observed no fixed or predictable patterns (e.g., periodic, diurnal). This indicates that modeling+prediction approaches would be excessive, and instead SF's reactive approach is preferable.

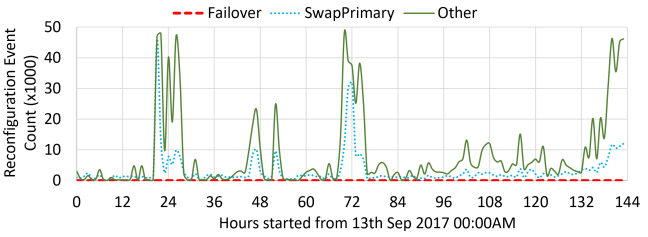

Figure 12: Reconfiguration Event count per hour. *Started from 13^{th} September 2017 00:00AM.*

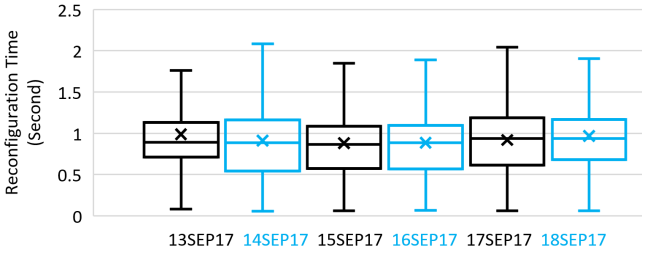

Figure 13: Statistics of Reconfiguration Delay (at placement) across six days. *Candlestick plots show the 1^{st} and 99^{th} percentiles, 1^{st}, 2^{nd} and 3^{rd} quartiles and the average (X's).*

Fig. 13 shows the time to execute a reconfiguration. Across the week, we observe very stable reconfiguration times. Tail latency is within 2.2 s and the average latency hovers at around 1.0 s. This is the time to execute control actions for the reconfiguration, after simulated annealing and in parallel with data transfer (which itself is dependent on the size of the object). Overall, we conclude that SF reconfigures replicas very quickly and predictably.

8.6 SF-Ring vs. Chord

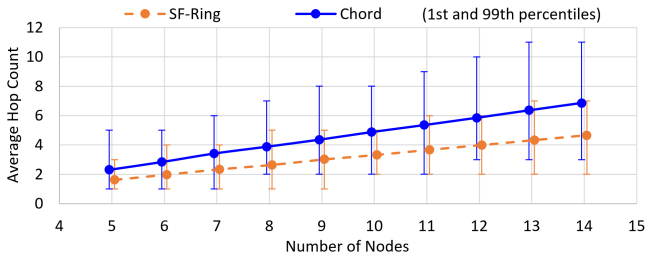

Figure 14: SF-Ring vs. Chord: Hop Count as function of system size (log scale). *Points perturbed slightly (±0.05 on X-axis) for clarity.*

We faithfully implemented a simulation of both SF-Ring and Chord [79] routing. Fig. 14 shows that SF-Ring messages transit 31% fewer hops in the ring than Chord. At the 1^{st} percentile the savings is 20% and at the 99^{th} percentile it is 34%. The slope of the SF-Ring and Chord lines are respectively 0.34 and 0.5. Therefore when the number of nodes doubles Chord requires 49.27% more hops than SF-Ring.

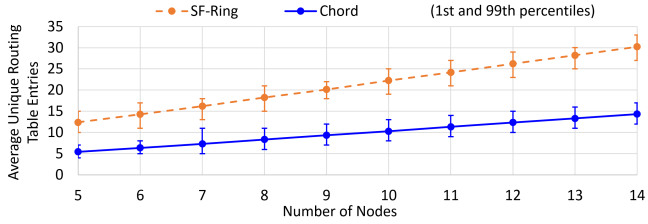

Figure 15: SF-Ring vs. Chord: Memory. *Unique Routing Table Entry Count as a function of system size (log scale).*

Fig. 15 compares the memory cost, calculated as the number of unique routing table entries, vs. cluster size (log scale). SF-Ring utilizes 117% higher memory than Chord. This is the cost to achieve faster routing latency. Yet, practically speaking SF-Ring's memory overhead is quite small–most containers/VMs today have many GBs of memory. In comparison, in an SF-Ring with 16K nodes, 99% of nodes store on average 33 routing table entries. Conceptually, with 100 B per entry, this comes out to only 3.3 KB of total memory (SF memory is higher in practice, but still small).

9 RELATED WORK

Microservice-like Frameworks: Nirmata [64] is a microservice platform built atop Netflix open-source components [65]: gateway service Zuul [90], registry service Eureka [37], management service Archaius [6], and REST client Ribbon [74]. Unlike Service Fabric, Nirmata does not have consistency and state built into the system.

It also has external dependencies. Other microservices platforms include Pivotal Application Service [67] and Spring Cloud [78]. However, none of these support stateful services out of the box. Akka [3] is a platform that embraces actor-based programming to build microservices. These systems do not solve the hard problems related to state or consistency, and do not take as principled an approach to design as SF. AWS Lambda [7] and Azure functions [13] both provide event-driven, serverless computing platforms for running small pieces of short lived code. SF is differentiated because it is the only data-aware orchestration system today for stateful microservices.

SF is the only standalone microservice platform today. The systems just listed usually require an external/remote drive for state. Akka sits atop a JVM. Spanner [27] relies on Colossus, Paxos, and naming. In SF, beyond the OS/machine, there are no external dependencies at the distributed systems layer.

Container Orchestrators: Container services like Kubernetes [51], Azure Container Service [8], etc., allow code to run and be managed easily, but they are typically stateless. SF supports state, which entails further challenges related to failover, consistency, and manageability (our paper addressed these goals). Further, container systems do not provide prescriptive guidance on writing applications; SF provides full lifecycle management.

Strong Consistency in Storage Systems: It is clear that many users and applications prefer strong notions of consistency alongside high performance. Distributed storage systems have come full circle from relational databases to eventually consistent databases [4, 5, 16, 26, 28, 33, 56, 61, 73] to recently, high throughput transactional databases. After eventually consistent databases, stronger models of consistency emerged (e.g., red-blue [54], causal+ [55], etc.). Many recent systems provide strong consistency and transaction support at high throughput: 1) systems layered atop unreliable replication, e.g., Yesquel [2], Callas [88], Tapir [89]; and 2) systems layered atop strong hardware abstractions, e.g., FaRM [34], RIFL [53], DrTM [86].

Cluster OSes: Prominent among cluster OSes that manage multitenancy via containers are: Apache YARN [84] which is used underneath Hadoop [41], Mesos [44] that provides dominant resource fairness, and Kubernetes [51].

Distributed Hash Tables and NoSQL: In the heyday of the P2P systems era, many DHTs were invented including: i) those that used routing tables (Chord [79], Pastry [75], Kademlia [57], Bamboo [72], etc.), and ii) those that used more memory for faster routing [39, 40]. P2P DHTs influenced the design of eventually consistent NoSQL storage systems including Dynamo [33], Riak [73], Cassandra [52], Voldemort [80], MongoDB [61], and many others.

10 SUMMARY

This paper presented Service Fabric (SF), a distributed platform at Microsoft running on the Microsoft Azure public cloud. SF enables design and lifecycle management of microservices in the cloud. We have described several key components of SF showing their modular design, self-* properties, decentralization, scalability, and especially the unique properties of strong consistency, and stateful support from the ground up. Experimental results from real production traces reveal that Service Fabric: *i)* reconfigures quickly (within seconds); *ii)* efficiently uses an arbitrator to resolve failure detection conflicts, in spite of high churn; and *iii)* routes messages efficiently, quickly, and using small amounts of memory.

Future Directions: 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. They deploy their applications and containers against a single consolidated system. Other interesting and longer-term models revolve around having customers owning servers, but also being able to run microservice management as a service where these 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 Service Fabric's Reliable Collections. This requires tackling many interesting problems ranging from logging, byte vs. block oriented storage, efficient encryption, and transaction-aware memory allocations.

ACKNOWLEDGMENTS

Work by Shegufta Bakht Ahsan and Indranil Gupta was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, and a generous gift from Microsoft. We are grateful to the following for help with experiments and traces: Tanvir Tanviruzzaman, Ketaki Joshi, Leon Mai, Gayathri Sundararaman, Hend Kamal Eldin, Hareesh Nagaraj. We thank the reviewers and our shepherd Romain Rouvoy for their insightful comments.

REFERENCES

[1] Adding nodes to an existing cluster. https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_add_node_to_cluster_t.html. Last accessed February 2018.
[2] Aguilera, M. K., Leners, J. B., and Walfish, M. Yesquel: Scalable SQL storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 245–262.
[3] Akka. http://akka.io/. Last accessed February 2018.
[4] Amazon SimpleDB. https://aws.amazon.com/simpledb/. Last accessed February 2018.
[5] Andler, S. F., Hansson, J., Eriksson, J., Mellin, J., Berndtsson, M., and Eftring, B. DeeDS : Towards a distributed and active real-time database system. *ACM SIGMOD Record 25*, 1 (Mar. 1996), 38–51.
[6] Archaius. https://github.com/Netflix/archaius. Last accessed February 2018.
[7] AWS Lambda. https://aws.amazon.com/lambda/. Last accessed February 2018.
[8] Azure Container Service. https://azure.microsoft.com/en-us/services/container-service/. Last accessed February 2018.
[9] Azure Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/. Last accessed February 2018.
[10] Azure Table Storage. https://azure.microsoft.com/en-us/services/storage/tables/. Last accessed February 2018.
[11] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/. Last accessed February 2018.
[12] Azure Event Hubs. https://azure.microsoft.com/en-us/services/event-hubs/. Last accessed February 2018.
[13] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Last accessed February 2018.
[14] Azure IoT. https://azure.microsoft.com/en-us/suites/iot-suite/. Last accessed February 2018.
[15] Azure SQL DB. https://azure.microsoft.com/en-us/services/sql-database/. Last accessed February 2018.
[16] Bailis, P., and Ghodsi, A. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM 56*, 5 (May 2013), 55–63.

[17] Balalaie, A., Heydarnoori, A., and Jamshidi, P. Migrating to cloud-native architectures using microservices: An experience report. *Computing Research Repository abs/1507.08217* (2015).

[18] Balalaie, A., Heydarnoori, A., and Jamshidi, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software 33*, 3 (2016), 42–52.

[19] Birman, K., and Joseph, T. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1987), SOSP '87, ACM, pp. 123–138.

[20] Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS) 17*, 2 (May 1999), 41–88.

[21] Bluemix. https://www.ibm.com/cloud-computing/bluemix. Last accessed February 2018.

[22] BMW Connected App. http://www.bmwblog.com/2016/10/06/new-bmw-connected-app-now-available-ios-android/. Last accessed February 2018.

[23] BMW Open Mobility Cloud. http://www.bmwblog.com/tag/open-mobility-cloud/. Last accessed February 2018.

[24] Burrows, M. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[25] Carretero, J., and Xhafa, F. Genetic algorithm based schedulers for Grid computing systems. In *International Journal of Innovative Computing, Information, and Control ICIC 3* (01 2007), vol. 5, pp. 1053–1071.

[26] Carstoiu, B., and Carstoiu, D. High performance eventually consistent distributed database Zatara. In *Proceedings of the 6th International Conference on Networked Computing* (May 2010), pp. 1–6.

[27] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[28] CouchDB. http://couchdb.apache.org/. Last accessed February 2018.

[29] Service Fabric Customer Profile: BMW Technology Corporation. https://blogs.msdn.microsoft.com/azureservicefabric/2016/08/24/service-fabric-customer-profile-bmw-technology-corporation/. Last accessed February 2018.

[30] Service Fabric Customer Profile: Mesh Systems. https://blogs.msdn.microsoft.com/azureservicefabric/2016/06/20/service-fabric-customer-profile-mesh-systems/. Last accessed February 2018.

[31] Service Fabric Customer Profile: TalkTalk TV. https://blogs.msdn.microsoft.com/azureservicefabric/2016/03/15/service-fabric-customer-profile-talktalk-tv/. Last accessed February 2018.

[32] Das, A., Gupta, I., and Motivala, A. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks* (2002), DSN '02, pp. 303–312.

[33] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[34] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 54–70.

[35] Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow. *Computing Research Repository abs/1606.04036* (2016).

[36] Esposito, C., Castiglione, A., and Choo, K. K. R. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing 3*, 5 (Sept 2016), 10–14.

[37] Eureka. https://github.com/Netflix/eureka. Last accessed February 2018.

[38] Ge, Y., and Wei, G. GA-Based Task Scheduler for the Cloud Computing Systems. In *Proceedings of International Conference on Web Information Systems and Mining* (Oct 2010), vol. 2, pp. 181–186.

[39] Gupta, A., Liskov, B., and Rodrigues, R. One hop lookups for peer-to-peer overlays. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 2–2.

[40] Gupta, I., Birman, K., Linga, P., Demers, A., and van Renesse, R. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems* (2003).

[41] Hadoop. http://hadoop.apache.org/. Last accessed February 2018.

[42] Hasha, R., Xun, L., Kakivaya, G., and Malkhi, D. Allocating and reclaiming resources within a rendezvous federation. https://patents.google.com/patent/US20080031246 A1, 2008. US Patent 11,752,198.

[43] Hasha, R. L., Xun, L., Kakivaya, G. K. R., and Malkhi, D. Maintaining consistency within a federation infrastructure. https://patents.google.com/patent/US20080288659 A1, 2008. US Patent 11,936,589.

[44] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI '11, USENIX Association, pp. 295–308.

[45] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association, pp. 11–11.

[46] Johnson, D. B., and Maltz, D. A. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing* (1996), Kluwer Academic Publishers, pp. 153–181.

[47] Kakivaya, G., Hasha, R., Xun, L., and Malkhi, D. Maintaining routing consistency within a rendezvous federation. https://patents.google.com/patent/US20080005624 A1, 2008. US Patent 11,549,332.

[48] Kakivaya, G. K. R., and Xun, L. Neighborhood maintenance in the federation. https://patents.google.com/patent/US20090213757 A1, 2009. US Patent 12,038,363.

[49] Kerberos. https://web.mit.edu/kerberos/. Last accessed February 2018.

[50] Khachaturyan, A., Semenovsovskaya, S., and Vainshtein, B. The thermodynamic approach to the structure analysis of crystals. *Acta Crystallographica Section A 37*, 5 (Sep 1981), 742–754.

[51] Kubernetes. https://kubernetes.io/. Last accessed February 2018.

[52] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *Operating Systems Review 44*, 2 (2010), 35–40.

[53] Lee, C., Park, S. J., Kejriwal, A., Matsushita, S., and Ousterhout, J. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 71–86.

[54] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI '12, USENIX Association, pp. 265–278.

[55] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[56] MariaDB. https://mariadb.org/. Last accessed February 2018.

[57] Maymounkov, P., and Mazières, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, UK, 2002), IPTPS '01, Springer-Verlag, pp. 53–65.

[58] Mesh Systems. http://www.mesh-systems.com/. Last accessed February 2018.

[59] Microsoft cortana. https://www.microsoft.com/en-us/mobile/experiences/cortana/. Last accessed February 2018.

[60] Microsoft Intune. https://www.microsoft.com/en-us/cloud-platform/microsoft-intune. Last accessed February 2018.

[61] MongoDB. https://www.mongodb.org/. Last accessed February 2018.

[62] Microsoft Service Fabric. https://azure.microsoft.com/en-us/services/service-fabric/. Last accessed February 2018.

[63] ning Gan, G., lei Huang, T., and Gao, S. Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In *2010 International Conference on Intelligent Computing and Integrated Systems* (Oct 2010), pp. 60–63.

[64] Nirmata. http://www.nirmata.com/. Last accessed February 2018.

[65] Netflix Open Source Software Center. https://netflix.github.io/. Last accessed February 2018.

[66] Perkins, C. E., and Royer, E. M. Ad-hoc on-demand distance vector (AODV) routing. In *In Proceedings of the 2nd IEEE Workshop On Mobile Computing Systems and Applications* (1997), pp. 90–100.

[67] Pivotal Application. https://pivotal.io/platform/pivotal-application-service. Last accessed February 2018.

[68] Quorum Business Solutions. https://www.qbsol.com/. Last accessed February 2018.

[69] Service Fabric Customer Profile: Quorum Business Solutions. https://blogs.msdn.microsoft.com/azureservicefabric/2016/11/15/service-fabric-customer-profile-quorum-business-solutions/. Last accessed February 2018.

[70] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), NSDI '04, USENIX Association, pp. 8–8.

[71] Redis. https://redis.io/. Last accessed February 2018.
[72] Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. Handling churn in a DHT. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), ATEC '04, USENIX Association, pp. 10–10.
[73] Riak. http://basho.com/products/. Last accessed February 2018.
[74] Ribbon. https://github.com/Netflix/ribbon. Last accessed February 2018.
[75] Rowstron, A. I. T., and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg* (London, UK, UK, 2001), Middleware '01, Springer-Verlag, pp. 329–350.
[76] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984), 277–288.
[77] Skype for Business. https://www.skype.com/en/business/skype-for-business/. Last accessed February 2018.
[78] Spring Cloud. http://projects.spring.io/spring-cloud/. Last accessed February 2018.
[79] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), SIGCOMM '01, ACM, pp. 149–160.
[80] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 18–18.
[81] Talk Talk TV. http://www.talktalk.co.uk/. Last accessed February 2018.
[82] Tonse, S. Scalable microservices at Netflix. challenges and tools of the trade. https://www.infoq.com/presentations/netflix-ipc. Last accessed February 2018.
[83] van Renesse, R., Minsky, Y., and Hayden, M. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (London, UK, UK, 1998), Middleware '98, Springer-Verlag, pp. 55–70.
[84] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.
[85] Wang, A., and Tonse, S. Announcing Ribbon: Tying the Netflix mid-tier services together. http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html. Last accessed February 2018.
[86] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.
[87] Wheeler, B. Should your apps be cloud-native? https://devops.com/apps-cloud-native/. Last accessed February 2018.
[88] Xie, C., Su, C., Littley, C., Alvisi, L., Kapritsos, M., and Wang, Y. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 279–294.
[89] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A., and Ports, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 263–278.
[90] Zuul. https://github.com/Netflix/zuul. Last accessed February 2018.