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

Presenter: Shegufta Bakht Ahsan

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

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

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

ILLINOIS

Microsoft Azure

Microsoft Service Fabric

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

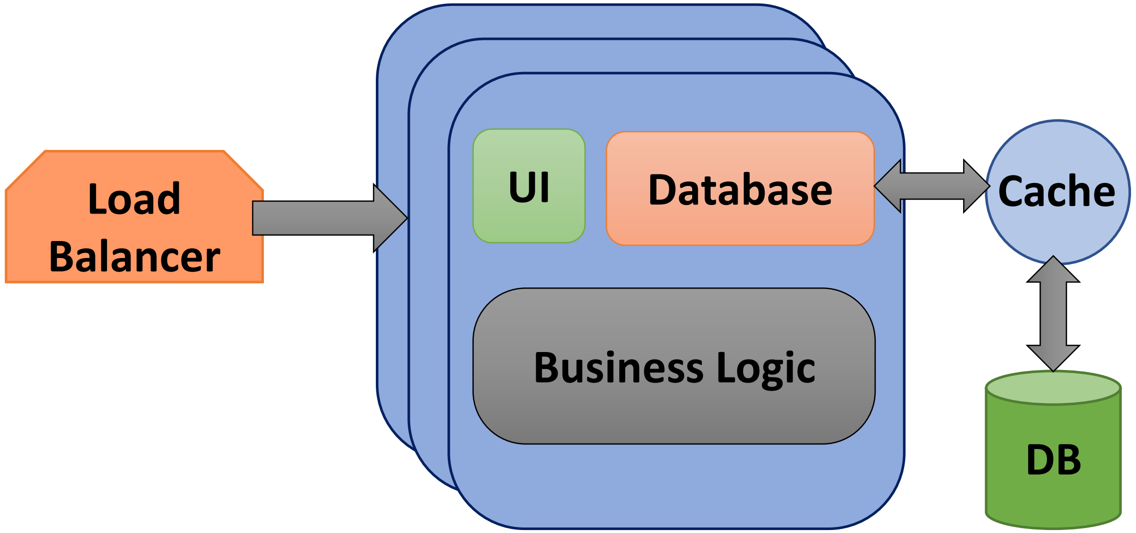
Culmination of **over 15 years** of design and development

Skype	Microsoft Intune	Azure Cosmos DB	TalkTalk TV
Cortana	Microsoft IoT Suite	BMW	And More…

- **Microsoft Azure SQL DB:**
  - Hosts ~2 Million DBs | Containing 3.5 PB of data | Spans over 100K machines
- **Azure Cosmos DB:**
  - Utilizes 2 million cores | Spans over 100K machines
- **Cloud Telemetry Engine:**
  - Processes 3 Trillion events/week

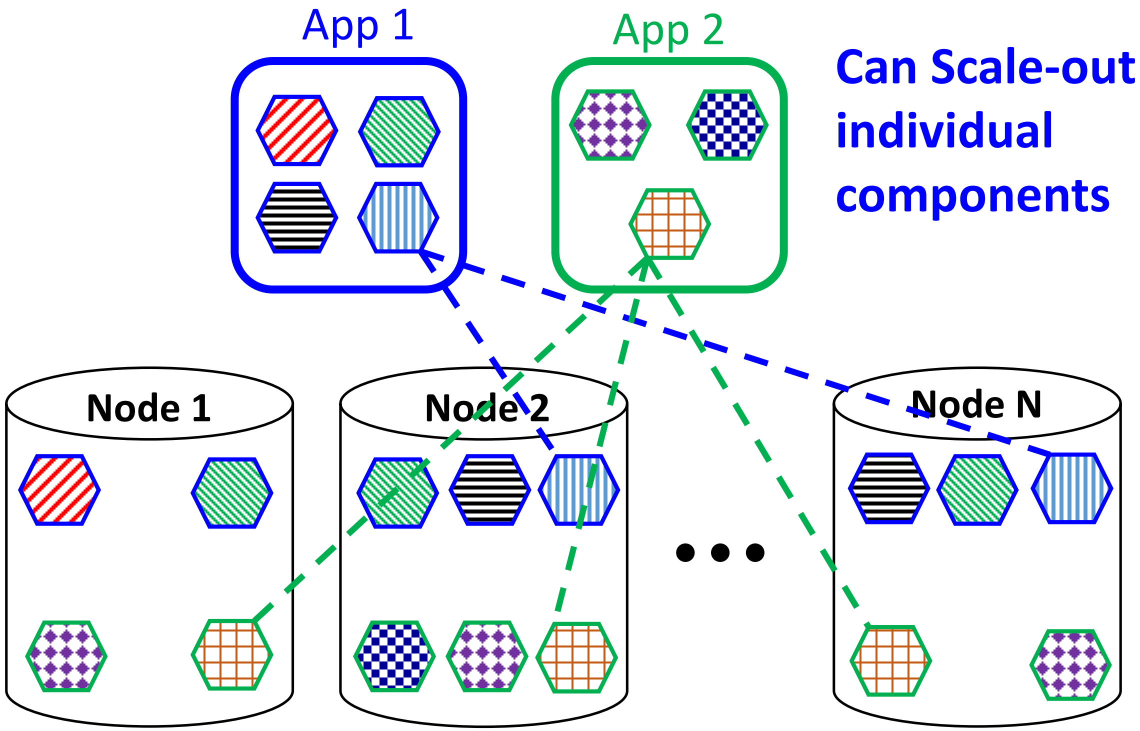
Monolithic Vs. Microservice Based Approach

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



Classic Monolithic Approach

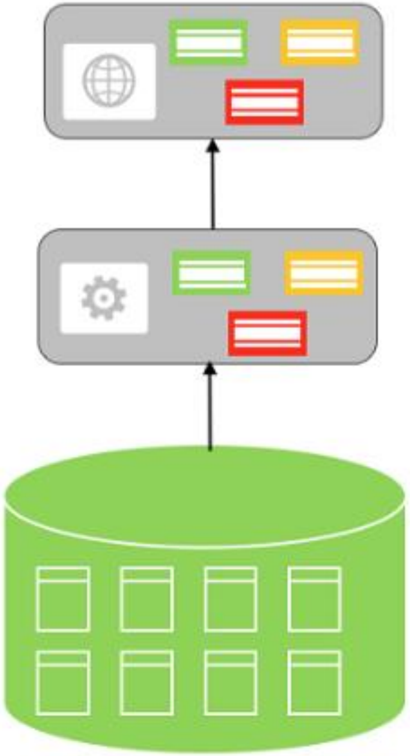
Not Cloud Friendly



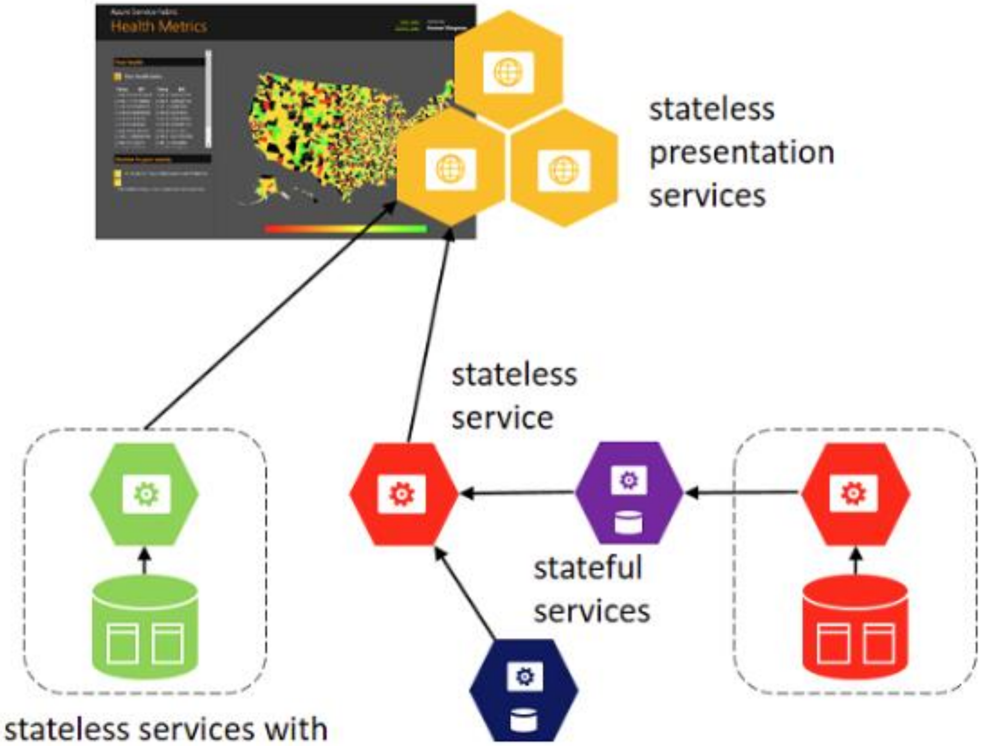
Microservice Based Approach

Cloud Friendly

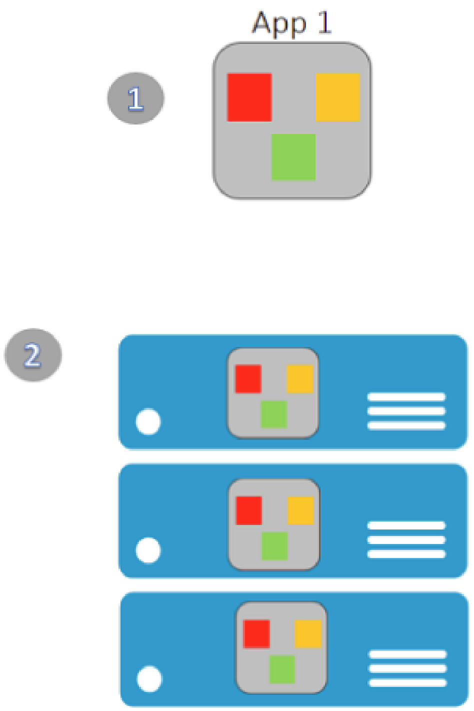
State in Monolithic approach



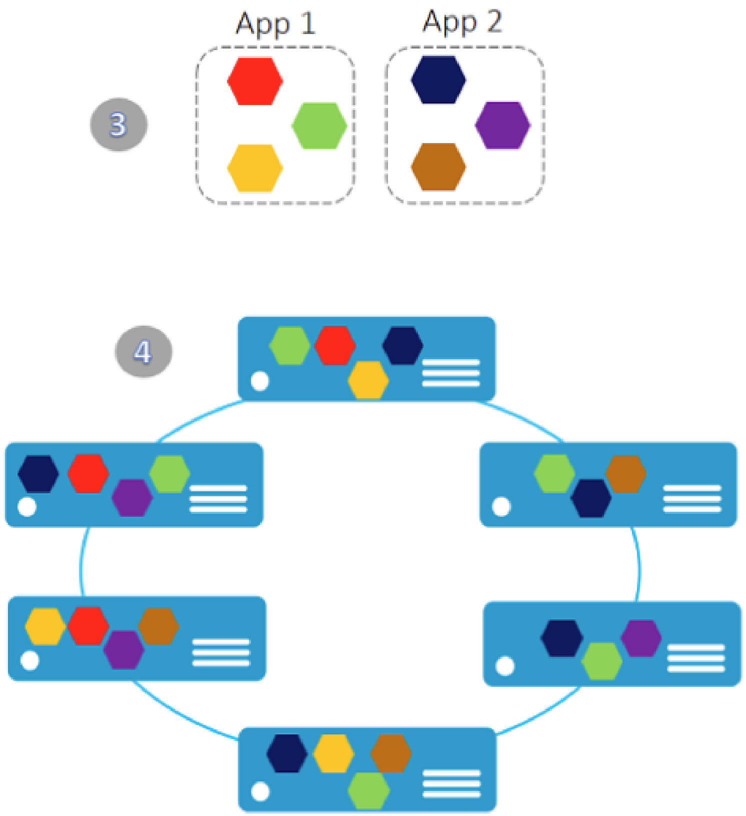
State in Microservices approach



Monolithic application approach



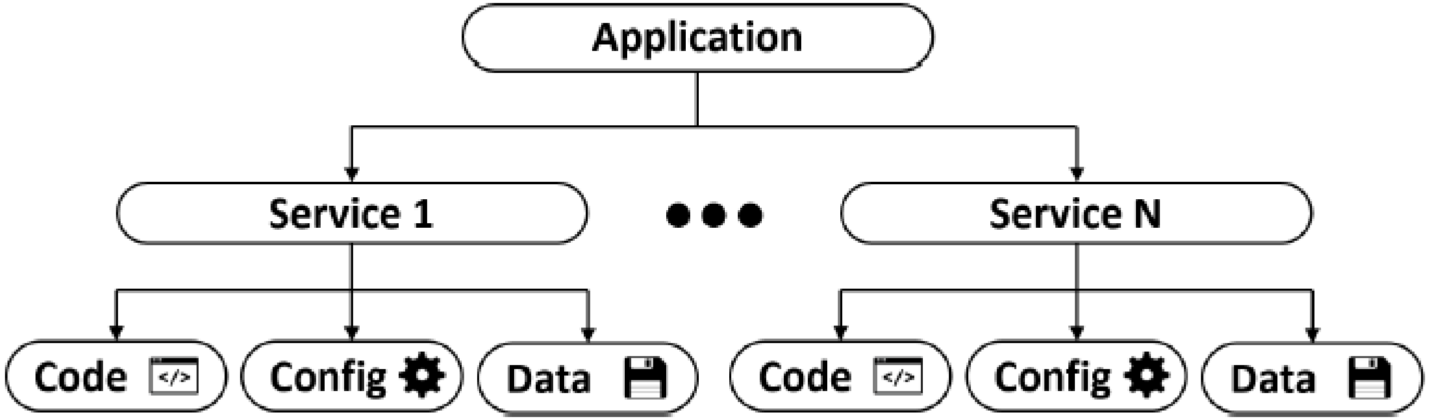
Microservices application approach



Monolithic vs. Microservice Applications

	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

Application Model



Service Fabric and Its Goals

➢ **Support for Strong Consistency:**

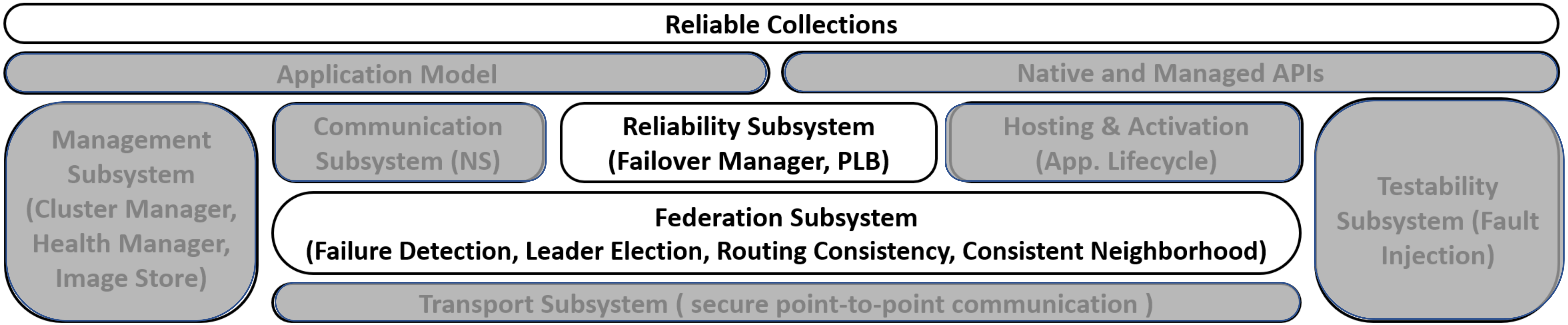
- Ground Up
- Higher layer focuses on "their" relevant notion of consistency (ACID at Reliable Collections)

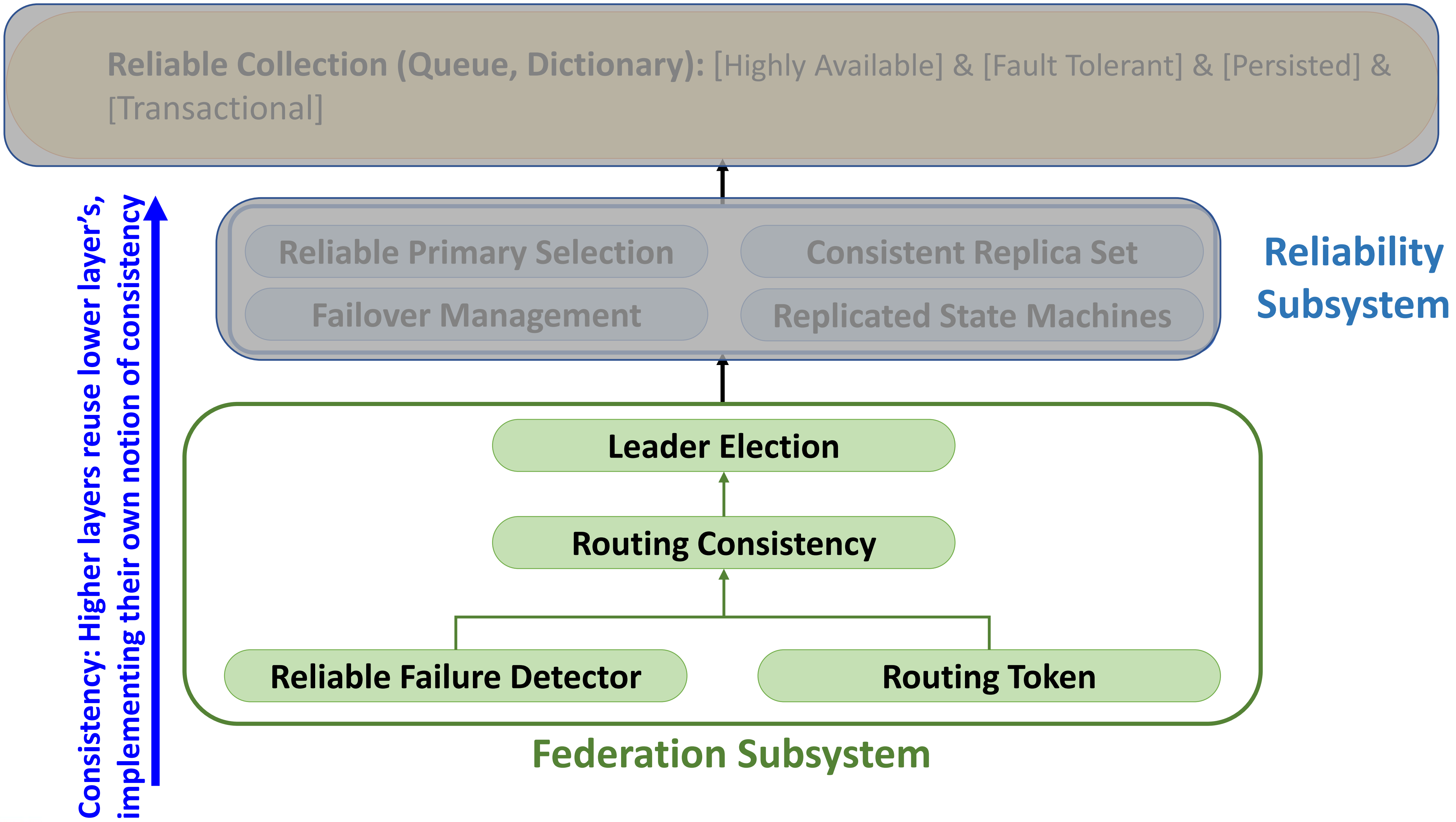
➢ **Fault Tolerance**

➢ **Support for Stateful Microservices:**

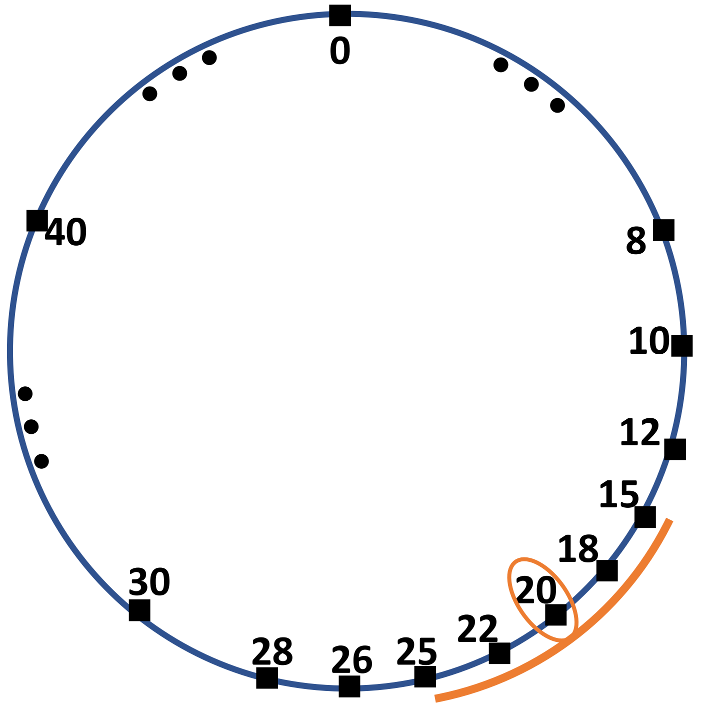
- Microservices can have their **own state**

Service Fabric Major Subsystems




Reliable Collection (Queue, Dictionary): [Highly Available] & [Fault Tolerant] & [Persisted] & [Transactional]
Reliable Primary Selection
Consistent Replica Set
Failover Management
Replicated State Machines
Reliability Subsystem
Leader Election
Routing Consistency
Reliable Failure Detector
Routing Token
Federation Subsystem
Consistency: Higher layers reuse lower layer's, implementing their own notion of consistency

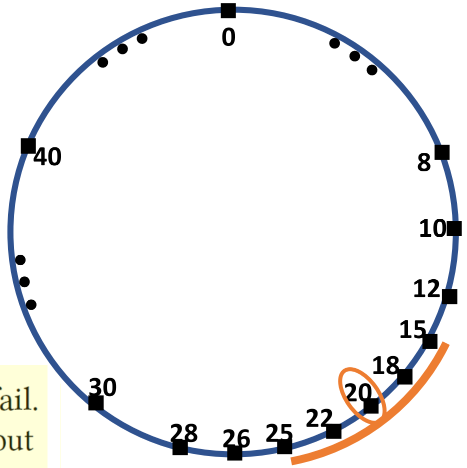
Federation Subsystem



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

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

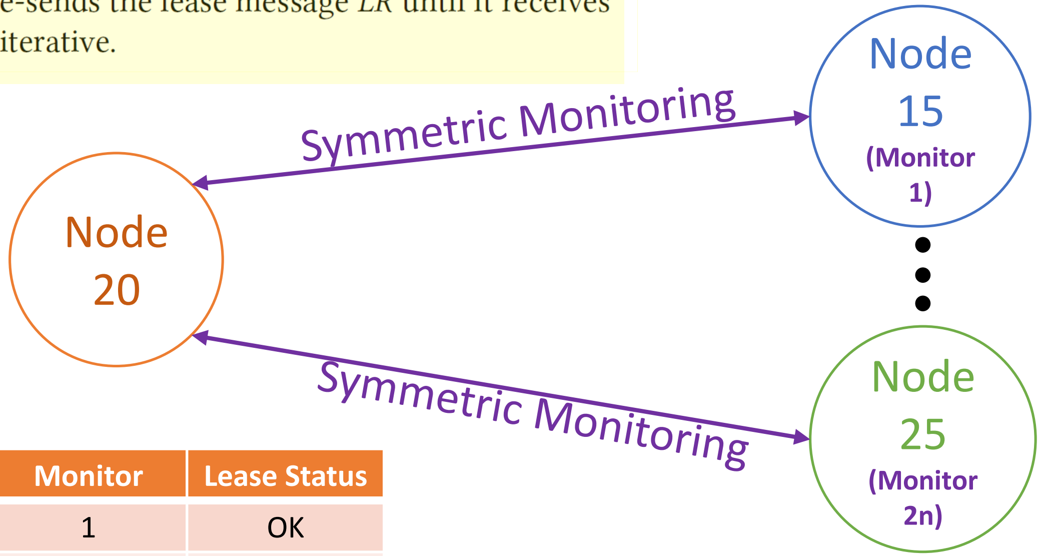
Consistent Membership and Failure Detection



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

Lease 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.

- **Lease Based Monitoring:**
  - Node A sends Lease Request to Node B
  - If Node A receives ACK, lease stablishes



- **Symmetric Monitoring (SM)**
  - Node A and Node B monitor each other

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

Monitor	Lease Status
1	OK
…	…
2n	OK

Arbitrator – Decouple Detection From Decision

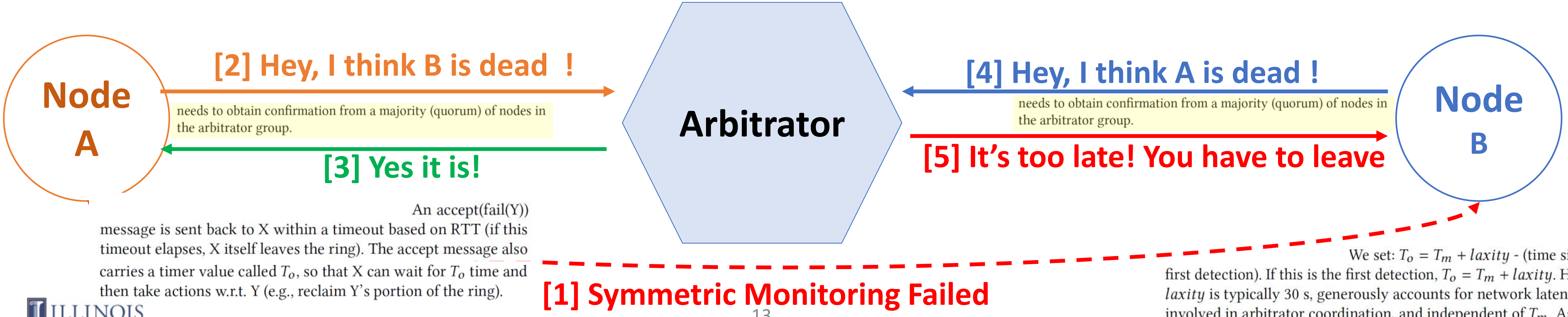
Recently-failed list

<u>Arbitration Log</u>

Log 1: Time T : Node B declared dead

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

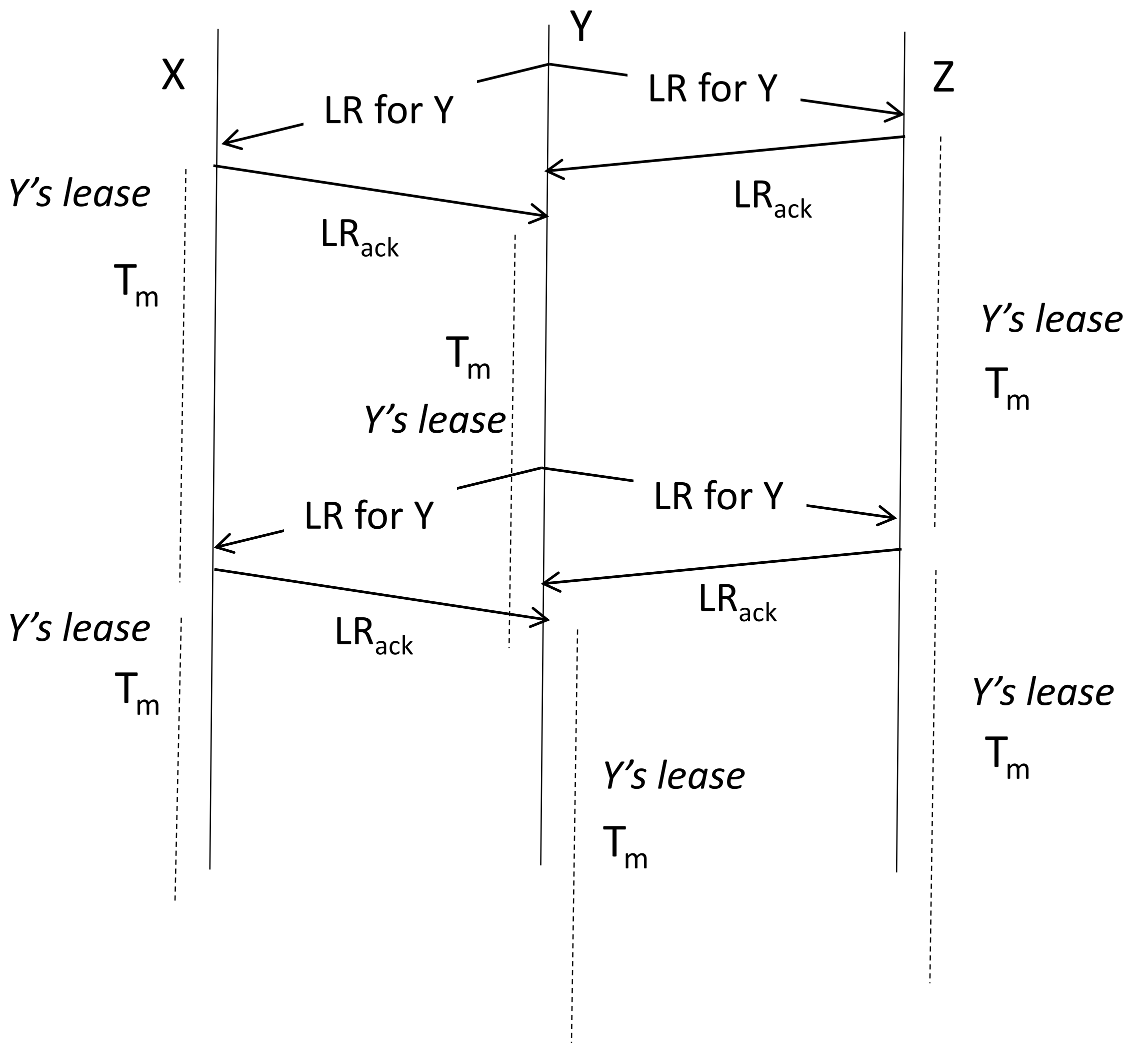
In Production: Multiple Arbitrators, Quorum Based approach



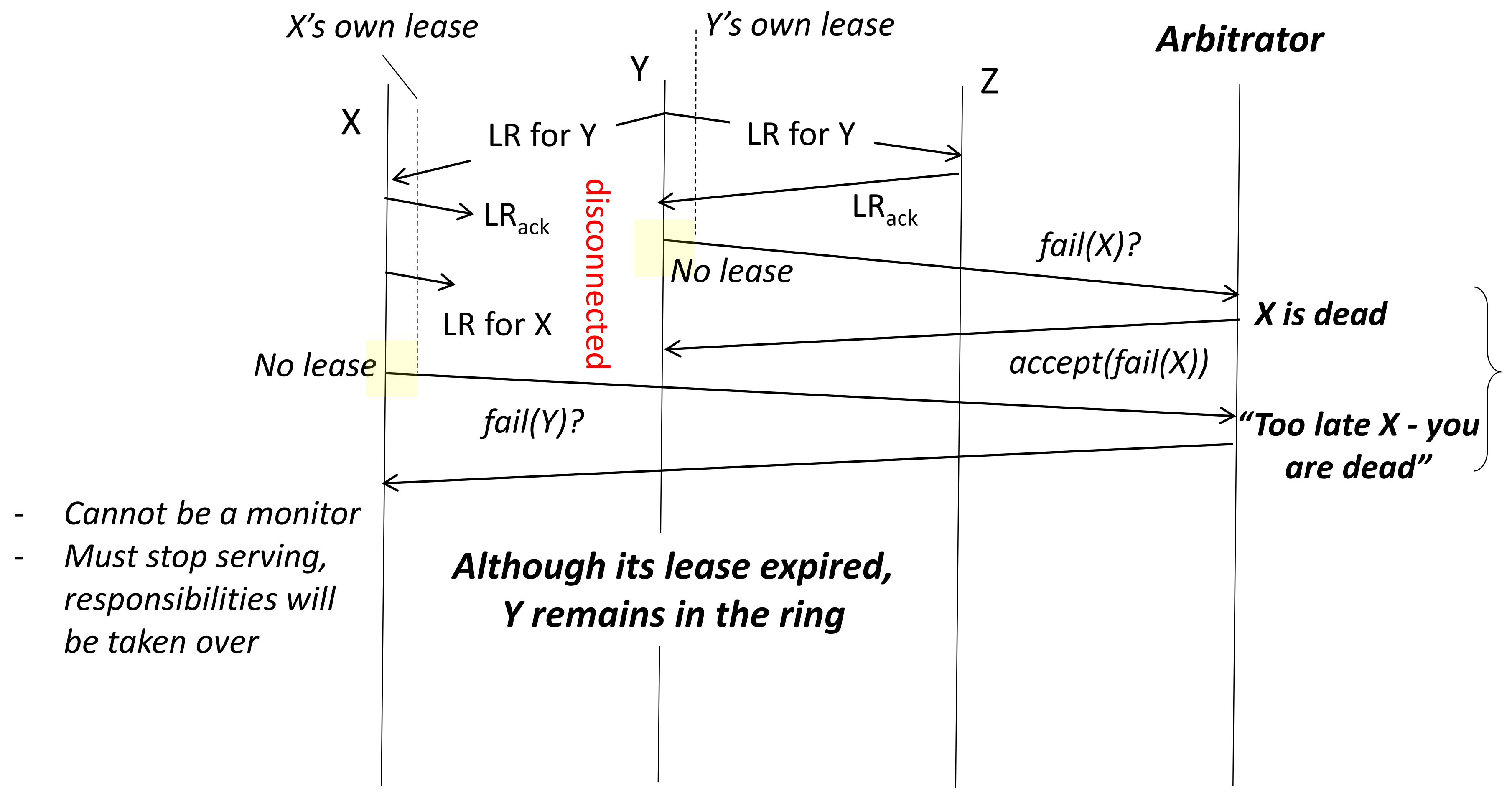
An accept(fail(Y)) message is sent back to X within a timeout based on RTT (if this timeout elapses, X itself leaves the ring). The accept message also carries a timer value called T_o, so that X can wait for T_o time and then take actions w.r.t. Y (e.g., reclaim Y's portion of the ring).

We set: $T_o = T_m + laxity$ - (time since first detection). If this is the first detection, $T_o = T_m + laxity$. Here, $laxity$ is typically 30 s, generously accounts for network latencies involved in arbitrator coordination, and independent of T_m. As all timeouts are large (tens of seconds), loose time synchronization suffices.

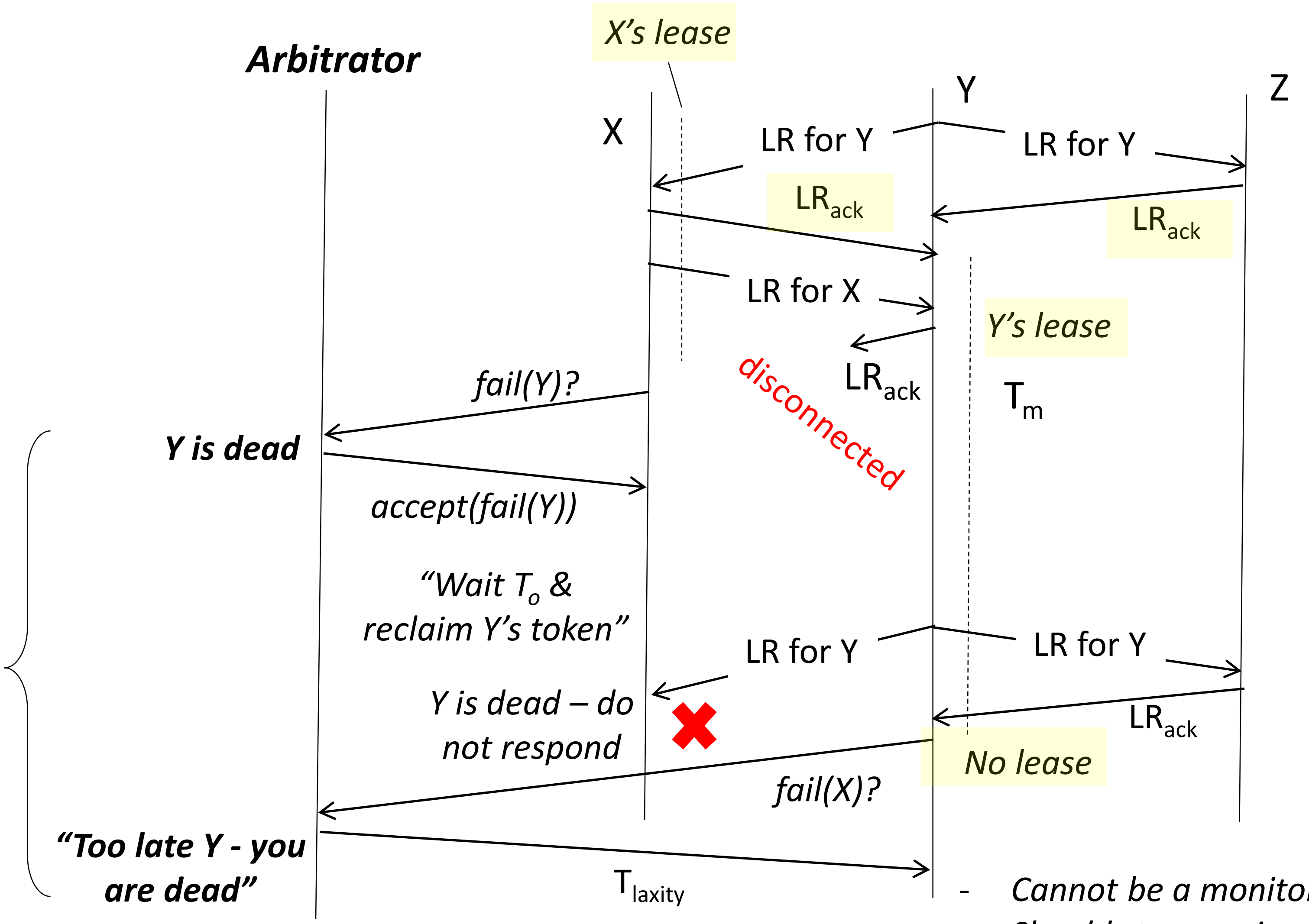
Each node Y and its monitors (X, Z) maintain Y's lease



X and Y's LR's are almost simultaneous and both fail: <u>only one of them is kicked out</u>, <u>situation is resolved fast</u>



Y's lease is renewed, then Y suffers temporary disconnection: Y can be kicked out, <u>may only find out in (up to) T_m time</u>



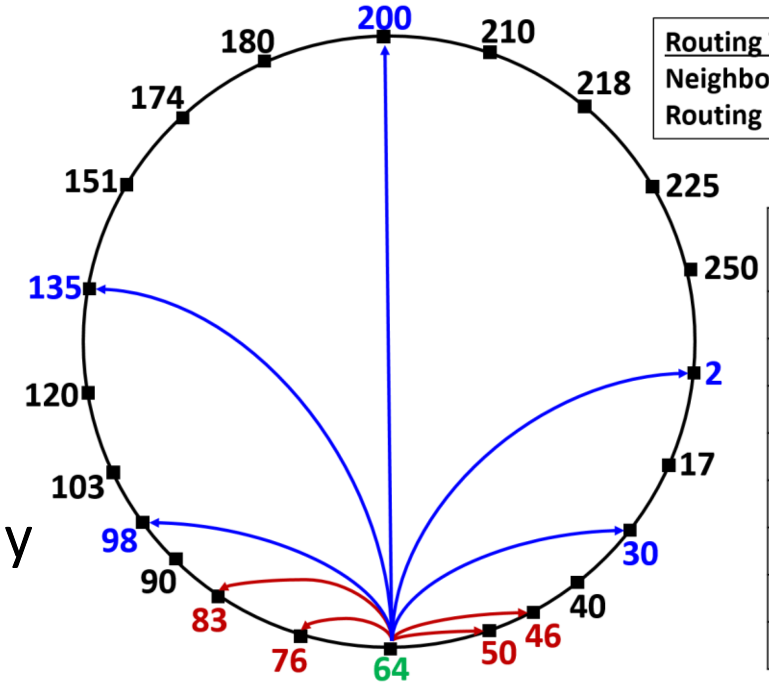
- *Cannot be a monitor*
- *Should stop serving, responsibilities will be taken over*

Routing is Bidirectional and Symmetric (SF-Routing)

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

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

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



Routing Table at node 64:
Neighborhood = {83,76,50,46}
Routing Partner = {200,2,30,98,135,200}

Routing Partner Calculation

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

Consistent Routing

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

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

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

- Efficiently Handle: Node Join, Leave and Fail

Consistent Routing

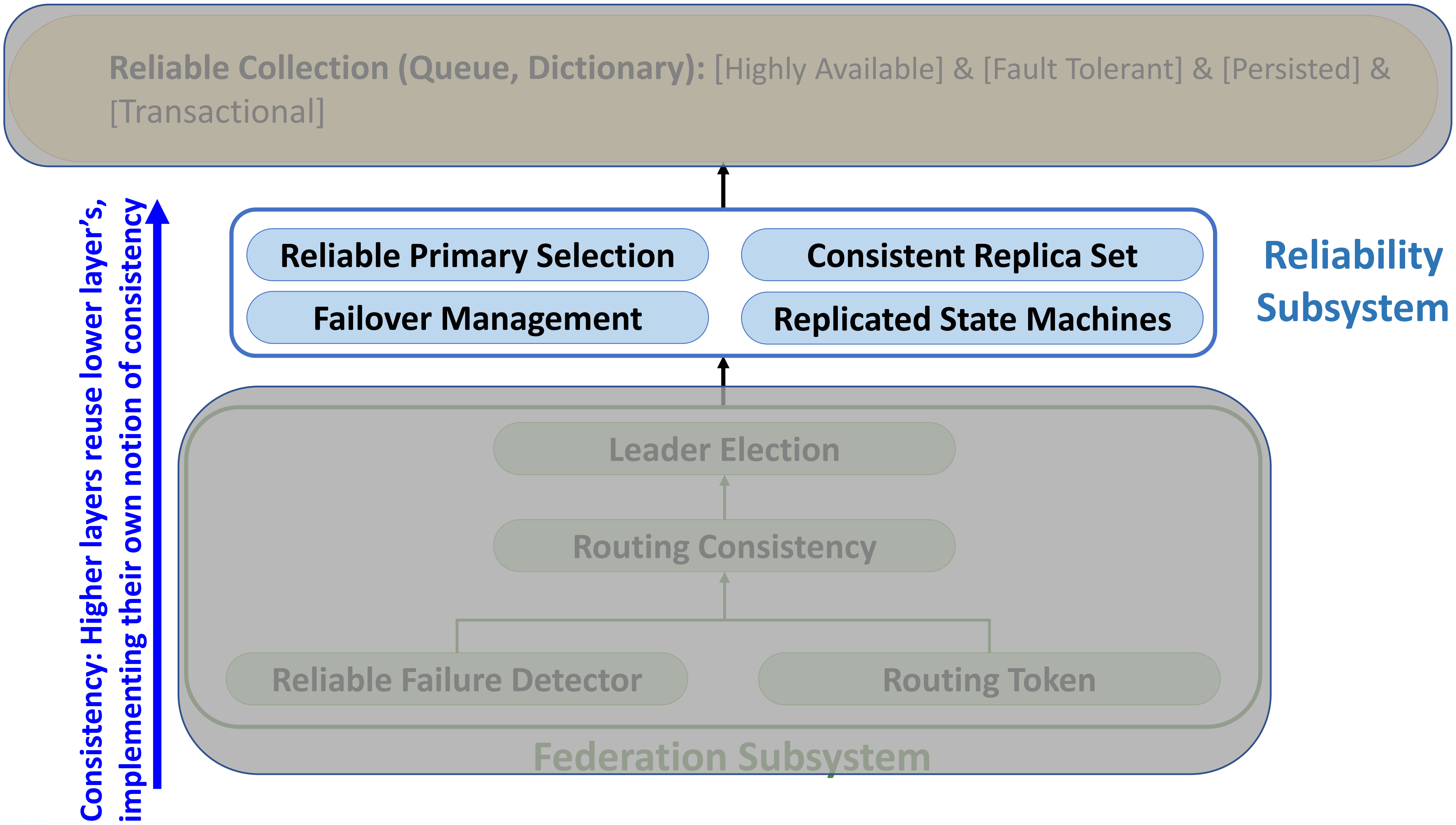
- At any given time all messages sent to key 'K' will be received by **a unique** Node. If that node crashes, a new node will take the responsibility

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

- **Invented concurrent with Chord and Pastry**

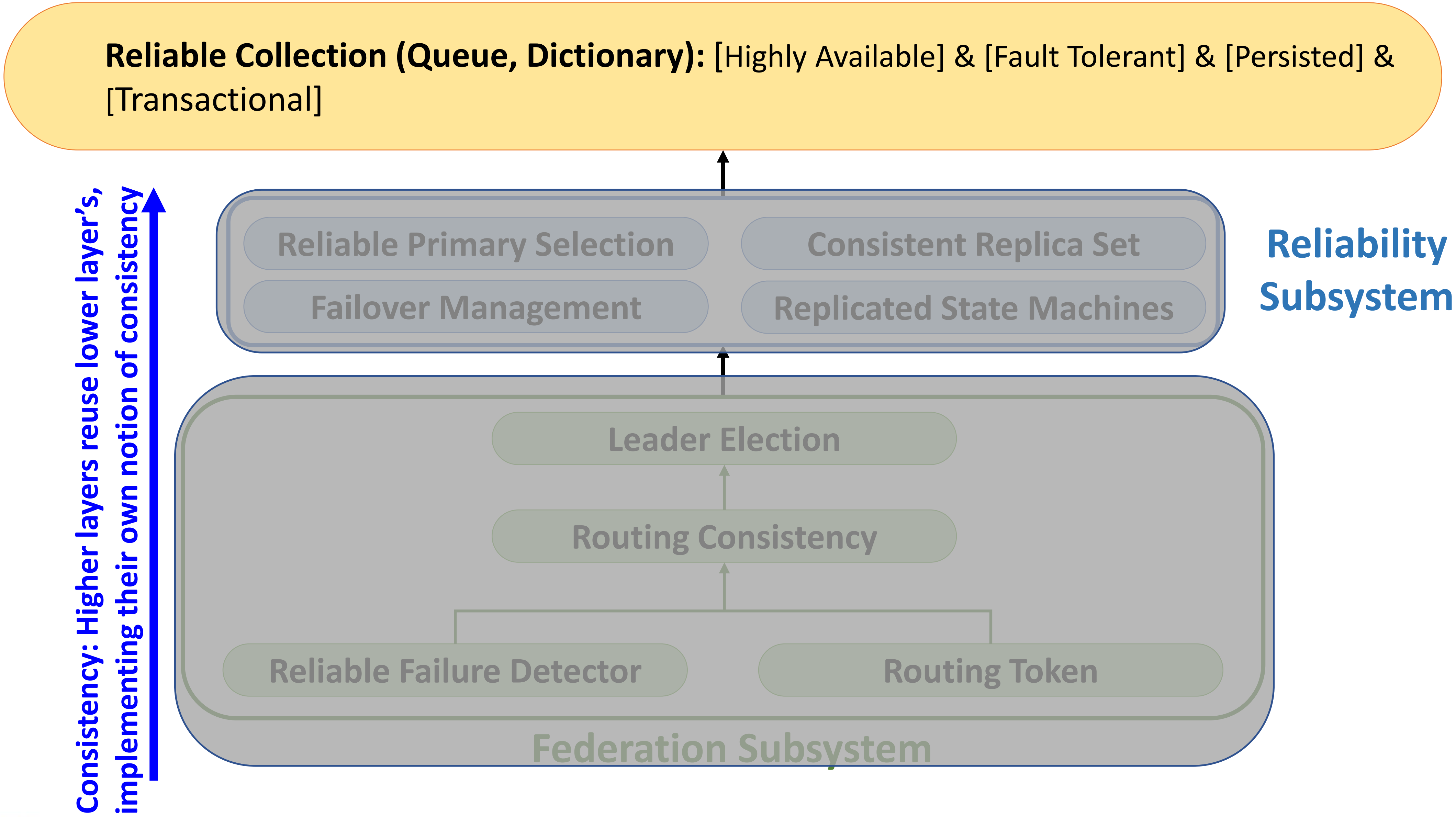
- **Chord/Pastry do not support Strong Consistency**

- Efficiently Handle: Node Join, Leave and Fail


Reliable Collection (Queue, Dictionary): [Highly Available] & [Fault Tolerant] & [Persisted] & [Transactional]
Reliable Primary Selection
Consistent Replica Set
Failover Management
Replicated State Machines
Reliability Subsystem
Leader Election
Routing Consistency
Reliable Failure Detector
Routing Token
Federation Subsystem
Consistency: Higher layers reuse lower layer's, implementing their own notion of consistency

Reliability Subsystem

- **Provides:**
  - Replication
  - High Availability
  - Load Balancing



Reliable Collection (Queue, Dictionary): [Highly Available] & [Fault Tolerant] & [Persisted] & [Transactional]
Consistency: Higher layers reuse lower layer's, implementing their own notion of consistency
Reliable Primary Selection
Consistent Replica Set
Failover Management
Replicated State Machines
Reliability Subsystem
Leader Election
Routing Consistency
Reliable Failure Detector
Routing Token
Federation Subsystem

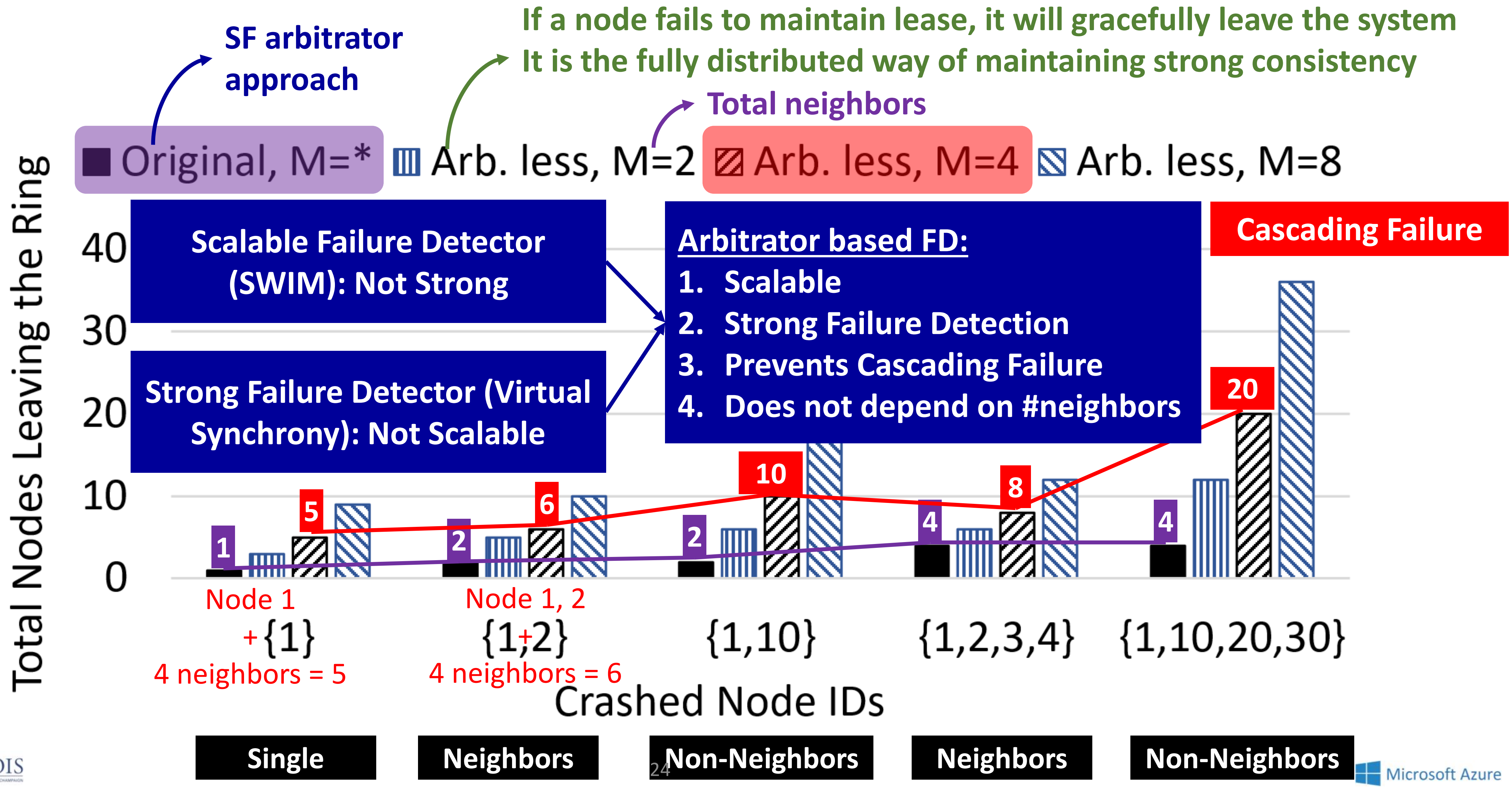
Reliable Collection (Queue, Dictionary)

- **Reliable Collections:**
  - Fault Tolerant
  - Highly Available
  - Persisted, Replicated
  - Transactional

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

- **Used in Stateful Microservices**

Evaluation – SF Arbitrator vs. Fully Distributed Scheme



ILLINOIS

Microsoft Azure

Summary

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

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

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

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