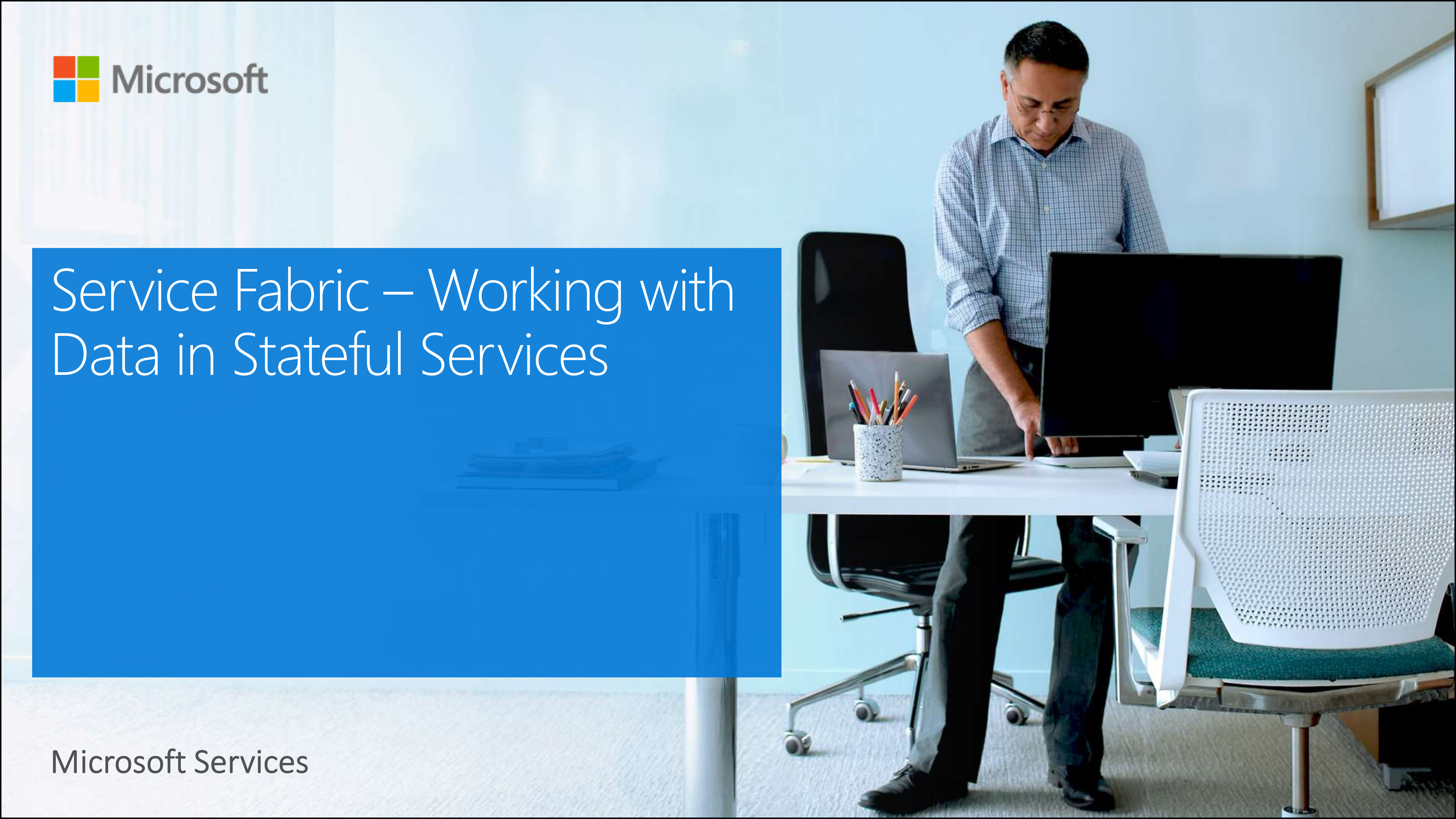


Service Fabric – Working with Data in Stateful Services

Microsoft Services

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

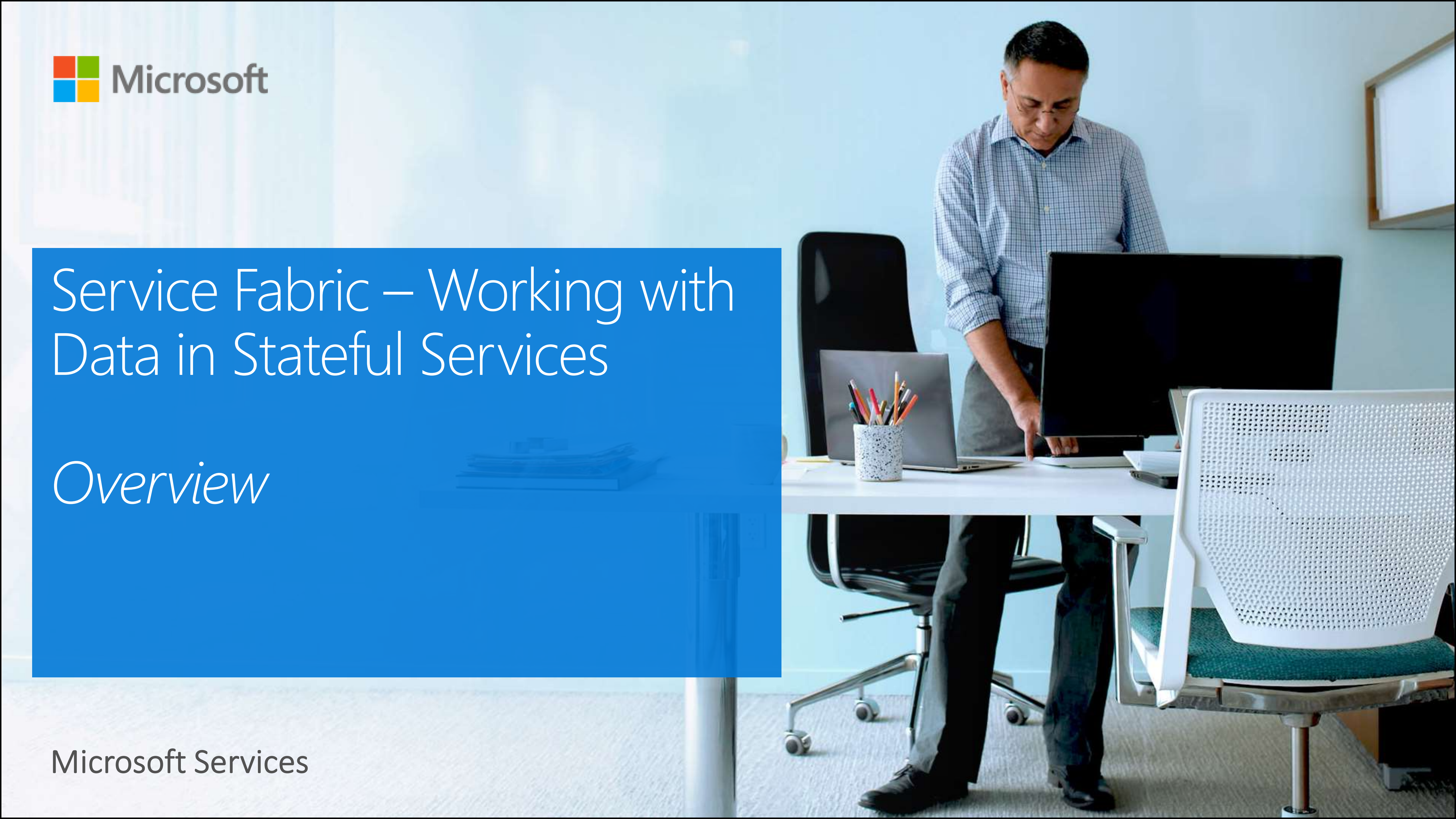
Training package content, including URLs and other Internet website references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

Agenda

- Stateful Services details
- Partitioning and Scale
- Data and State
- Backup Reliable Services and Reliable Actors

Microsoft

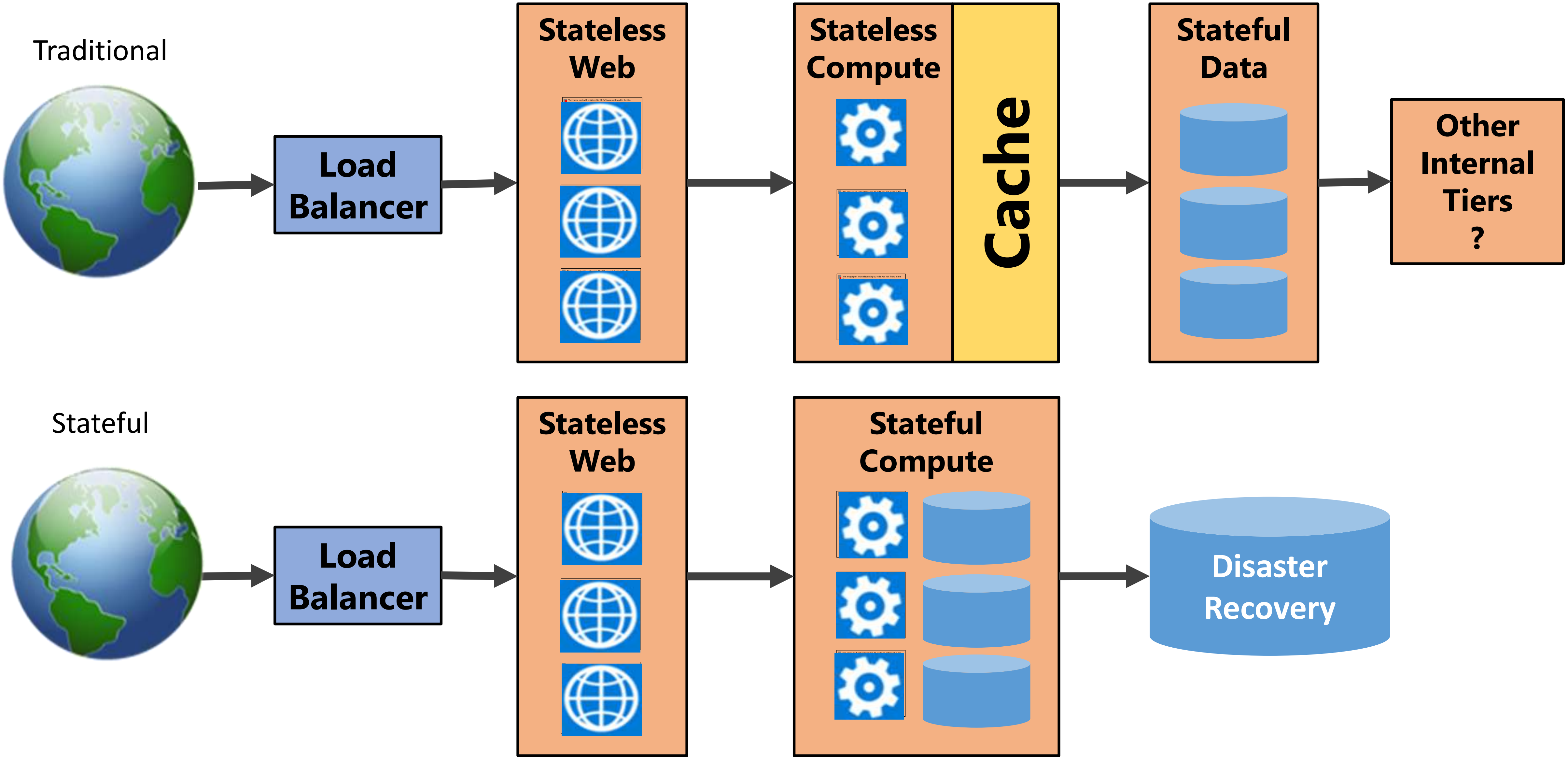


Service Fabric – Working with Data in Stateful Services

Overview

Microsoft Services

State Architectures: Traditional vs Service Fabric



Traditional

Load Balancer

Stateless Web

Stateless Compute

Cache

Stateful Data

Other Internal Tiers ?

Stateful

Load Balancer

Stateless Web

Stateful Compute

Disaster Recovery

Stateful Reliable Services Review

- Pros
  - Compute code has low-latency, strongly-consistent read access to hot data
  - Reduces dependency on external storage services
- Cons
  - Keeping (replicated) data on compute nodes can be costly
- Named services’ Reliable Dictionary/Queue collections
  - Partitioned for data scalability
  - Replicated for availability
  - Transacted (within a partition) for ACID semantics
  - Asynchronous for efficiency
  - Persisted for quick node failure recovery
  - Provides automatic locking for multi-threaded access

Microsoft

Service Fabric – Working with Data in Stateful Services

Partitioning and Scale

Microsoft Services

A Named Service's Partitions

- You can scale a named services data by splitting it across partitions
  - Allows data/computation to be spread across nodes for appropriate storage/speed
  - A partition must fit in 1 node; but, 1 node can hold multiple partitions
- Each partition has 0+ Reliable Collection objects.
- Architecting a service's partitions is often very hard
  - Cross-partition operations require network hops and different transactions
  - How many partitions depends on how much data you'll have ***in the future***
  - By default, Service Fabric balances partitions across nodes so try to keep the partitions data size as even as possible
    - Report capacity load metrics to better control balancing
- Service Fabric identifies each partition with a static globally unique identifier (GUID)

Service Fabric Offers 3 Partition Schemes

- Uniform Int64 Range (key range and ***n*** partitions) ~ Ranged partitioning

Distribution:
Data → Algorithm → Int64 key → Partition #

Key range=0-99; Partitions=5				
0-19	20-39	40-59	60-79	80-99
0	1	2	3	4

- Singleton partitioning (1 partition)

Distribution:
Data → Algorithm → Partition #

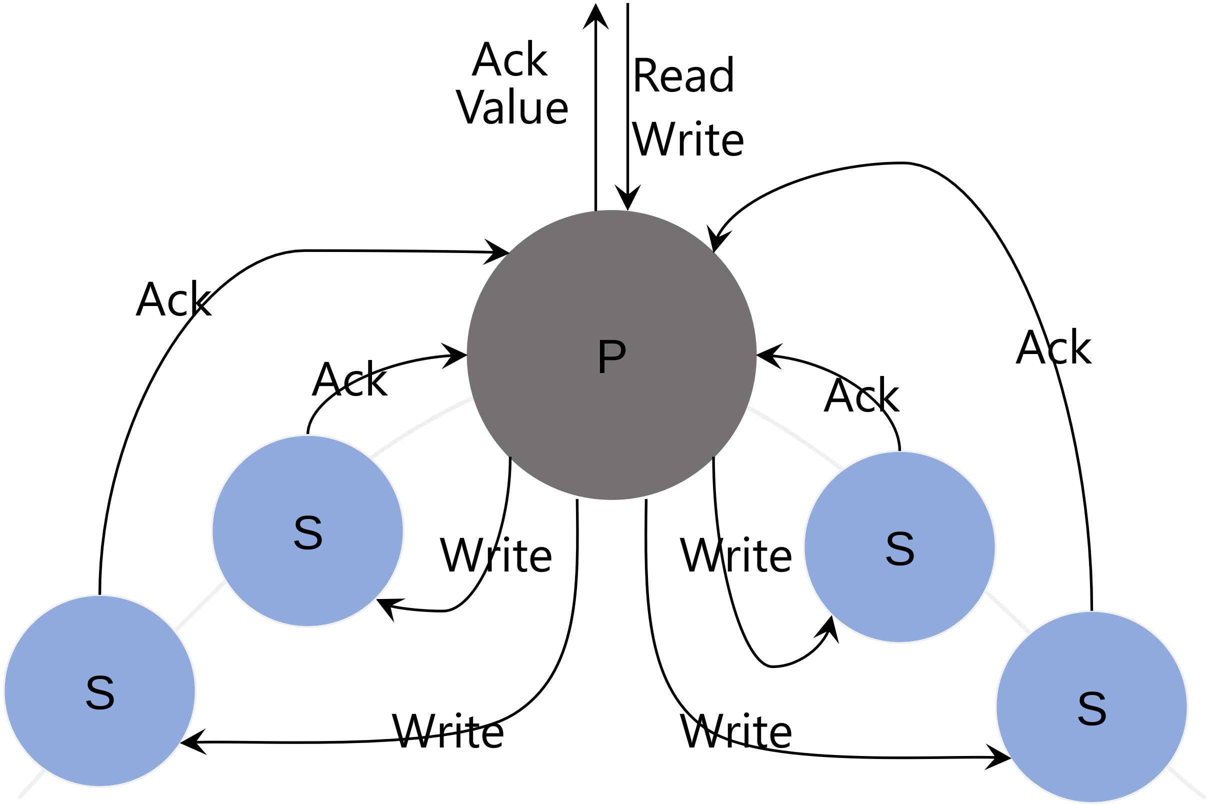
- String (1 string per partition) ~ Named partitioning

Distribution:
Data → Algorithm → String-key → Partition
#

Strings=5; Partitions=5				
Arctic	Atlantic	Indian	Pacific	Southern
0	1	2	3	4

Replication – Stateful Services

- Reads are completed at the primary
- Writes are replicated to the write quorum of secondary's



A Service's Replicas

- Replicating state increases chance of data surviving 1+ ***simultaneous*** node failures
  - But, more replicas increase cost & network latency to sync replicas
  - The less replicas that exist, the more risk for data loss
  - Consider writing to external state (reducing replica costs & failure recovery) and reading from a replica (for speed)
- Replicas go across FDs/UDs; avoids single point of failure
- Service Fabric identifies each replica with a dynamic 64-bit integer (changes on create/move)

Configuring Partitions & Replicas

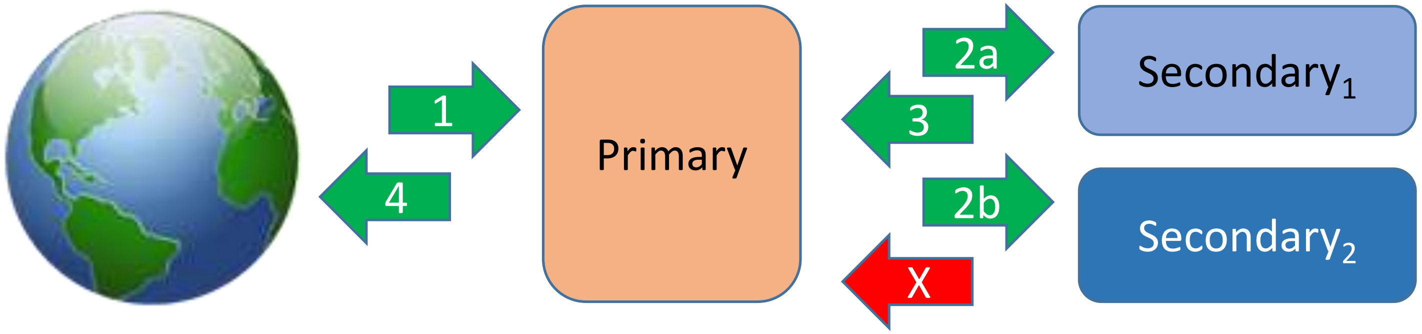
- New-ServiceFabricService cmdlet requires
  - Partition scheme (low key, high key, & partition count)
  - Replica counts (minimum & target)
- Update-ServiceFabricService cmdlet lets you change
  - Replica counts (minimum & target)
- Partition settings are in the ApplicationManifest.xml

NOTE: You can’t update/change partition scheme

- It’s OK to have many partitions since smaller partitions are fairly cheap
  - But, if a node fails, even empty partitions have to be re-built (some performance hit)

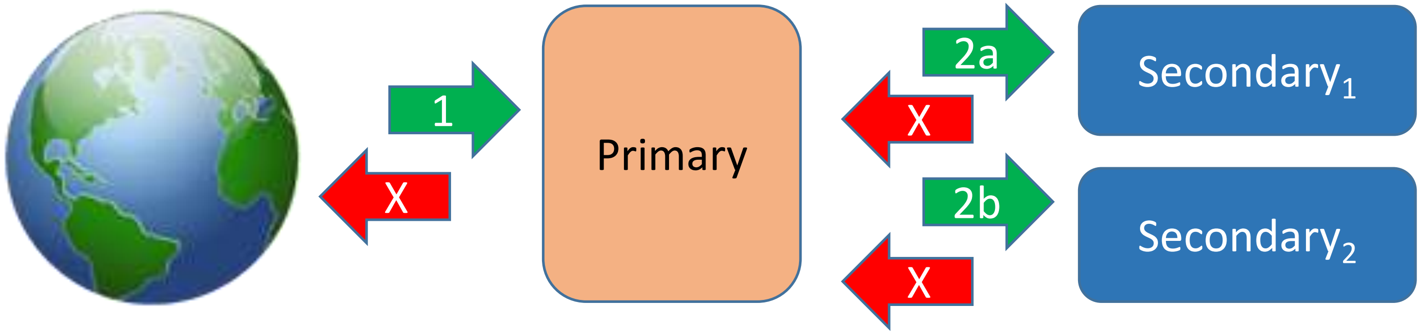
Partition Write Scenarios (Replicas=3, Quorum=2)

Write with P and S_1 up, S_2 is down



Either S_2 comes back up or Service Fabric will create another Secondary

Write with P up, S_1 and S_2 down (quorum loss)



Client won't get a response if Service Fabric can't create new Secondary. NOTE: Clients can still read from Primary.

Calculating Partitions / Capacity Planning

- Partitioning your service does NOT scale out the service itself
- Each partition must fit within a single VM, but multiple (small) partitions can be placed on a single VM
- Having a larger number of small partitions gives you greater flexibility than having a small number of larger partitions.
- Trade-offs
  - Increases Service Fabric overhead
  - You cannot perform transacted operations across partitions.
  - More potential network traffic if your service code frequently needs to access pieces of data that is located in different partitions

Scaling at the service name level

- Another option for scaling
- Data partitions must be decided at build time, but more service instances can be added dynamically
- Services can be added via PowerShell or by API
- Example – embed dates in service name
  - Customers who joined in Year 2015 get Service1, customers who joined in 2016 get Service2
  - Both have underlying partition schemes but partition size may be different

Demonstration

Partitioning

Service Fabric – Working with Data in Stateful Services

Data and State

Microsoft Services

Types of Data

Data usage categories

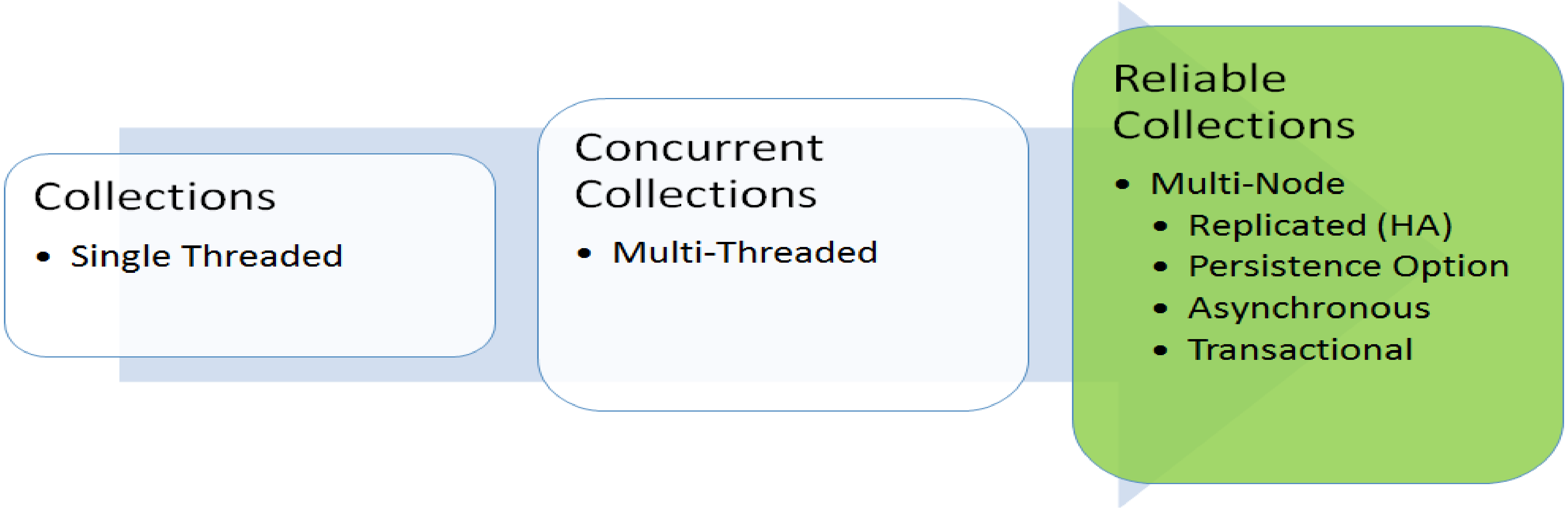
- Hot path data
- Warm path data
- Cold path data

Data stores

- Reliable Collections – Service State
- Traditional Stores
- Static Data – Configuration and Settings

Reliable Collections

Enables you to write highly available, scalable, and low-latency cloud applications as though you were writing single computer applications



Collections

- Single Threaded

Concurrent Collections

- Multi-Threaded

Reliable Collections

- Multi-Node
  - Replicated (HA)
  - Persistence Option
  - Asynchronous
  - Transactional

Reliable Collections

- Strong consistency is achieved by ensuring transaction commits finish only after the entire transaction has been applied on a quorum of replicas, including the Primary
    - To achieve weaker consistency, applications can acknowledge back to the client/requester before the asynchronous commit returns
- Two supported isolation levels
    - Repeatable read:
        - Outside transaction cannot read anything modified and not yet committed by transaction
        - No other transaction can modify anything until read by current transaction is finished
    - Snapshot
        - Data transaction sees will be the data that existed at transaction start (like it gets a snapshot)

Isolation Levels

- Repeatable Read – can’t read from incomplete transaction
- Snapshot – data at the beginning of the transaction is the same as at the end

Reliable Dictionary

Operation/Role	Primary	Secondary
Single Entity Read	Repeatable Read	Snapshot
Enumeration	Snapshot	Snapshot

Reliable Queue

Operation/Role	Primary	Secondary
Single Entity Read	Snapshot	Snapshot
Enumeration	Snapshot	Snapshot

NOT Your Typical .NET Collections

- .NET collections hold **references**
- Reliable Collections hold **objects** (think database hand-offs)
    - Misusing a reliable collection **will corrupt your data!**

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   await m_dic.AddAsync(tx, name, user1);
   user1.LastLogin = DateTime.UtcNow;  // Corruption!

   ConditionalResult<User> user2 = await m_dic.TryGetValueAsync(tx, name);
   if (user2.HasValue) user2.Value.LastLogin = DateTime.UtcNow; // Corruption!

   await tx.CommitAsync();
   // Of course, if you modify an object after CommitAsync, corruption!
}
```

- Correct: Get reference, copy/change object, write new object

Adding a Key/Value to a Dictionary

```
retry:
try {
   // Create a new Transaction object for this partition
   using (ITransaction tx = StateManager.CreateTransaction()) {
      // AddAsync takes key's write lock; if >4 secs, TimeoutException
      // key & value put in temp dictionary (read your own writes),
      // serialized, redo/undo record is logged & sent to secondary replicas
      await m_dic.AddAsync(tx, key, value);

      // CommitAsync sends Commit record to log & secondary replicas
      // After quorum responds, all locks released
      await tx.CommitAsync();
   }
   // If CommitAsync not called, Dispose sends Abort record
   // to log & secondary replicas, all locks released
}
catch (TimeoutException) { await Task.Delay(ms, cancellationToken); goto retry; }
```

Define Immutable Types to Force Correct Behavior

```
// If you don’t seal, derived classes must also be immutable
[DataContract] public sealed class UserInfo
 {

   public UserInfo(Email email, IEnumerable<ItemId> itemsBidding = null)
   {
      // We can assign to the read-only properties only in the ctor
      Email = email;
      ItemsBidding = itemsBidding ?? new ItemId[0];
   }

   // Read-only properties (you can set default values):
   [DataMember] public readonly Email Email; // Value type
   [DataMember] public readonly IEnumerable<ItemId> ItemsBidding = null;

   // "Modify" the object by creating a new one with the desired new state
   public UserInfo AddItemBidding(ItemId itemId) =>
      new UserInfo(Email, ItemsBidding.Concat(new[] { itemId }));
}
```

Querying Reliable Collections

- Both IReliableDictionary and IReliableQueue implement IAsyncEnumerable
- Microsoft is working on setting up async LINQ, in the meantime there are workarounds
  - Wrap the async calls with synchronous methods
  - Use library on GitHub Gist which supports Select, SelectMany, and Where
- ReliableDictionary supports enumeration through CreateEnumerableAsync
- Note that IEnumerables returned by CreateEnumerableAsync can only be enumerated within a transaction scope, so if you intend to use them elsewhere, you will need move the results into a temporary collection, such as a List.
- Snapshot isolation – lock free
  - Structure does not reflect changes that happen after the start of enumeration

Persistence Model Details

- State Provider stores data in the service
- Can be in-memory only or in-memory + local disk
- Default Actor state provider = in-memory + local disk but keeps hot data in memory so your storage requirements are not memory bound.
- Reliable Collections state provider stores all data both in-memory and on local disk
    - May be configurable in future release

Serialization

- Objects are serialized for persistence and wire transfer
- Persistent serializers must maintain infinite backward compatibility and +1 forward compatibility
    - Because data gets stored at lots of places (log, checkpoints, backups, etc.) and is retained for a very long time
- Wire transfer serializers must maintain +1/-1 compatibility
    - Because upgrading clusters have old and new code running simultaneously

Reliable Collections

- Recommendations
  - Do not modify an object of a custom type returned by read operations (e.g., TryPeekAsync or TryGetAsync)
  - Do a deep copy of the returned object of a custom type before modifying it. Since structs and built-in types are pass-by-value, you do not need to do a deep copy on them
  - Do not use TimeSpan.MaxValue for time-outs. Time-outs should be used to detect deadlocks
  - Do not create a transaction within another transaction’s using statement because it can cause deadlocks

Reliable Collections

- Considerations
    - The default time-out is 4 seconds for all the Reliable Collection API operations. Most users should not override this
    - The default cancellation token is CancellationToken.None in all Reliable Collections APIs
    - Enumerations are snapshot consistent within a collection. However, enumerations of multiple collections are not consistent across collections
    - To achieve high availability for the Reliable Collections, each service should have at least a target and minimum replica set size of 3

Configuration and static data

- Service Fabric xcopys all data in the package directory including the config, code, and data directories
- Any static data placed here will be available to all instances of the service
- Settings files and dependencies use this mechanism

Demonstration

Query Reliable Collections

Microsoft

Service Fabric – Working with Data in Stateful Services

Backup And Restore Reliable Services and Actors

Microsoft Services

Why backup reliable service or reliable actor data?

- In the event of the permanent loss of an entire Service Fabric cluster or nodes in a given partition
- Administrative errors whereby the state accidentally gets delete or corrupted
- Bugs in the service that causes data corruption
- Offline data processing, ie, for nightly batch runs or BI
- When moving to a new cluster environment

Backup Types

- Full – contains all data required to restore the state of the replica
  - Challenge – If the checkpoint for the data is large, a short Recovery Point Objective can cause excessive data collection
- Incremental – Only the log records since the last backup are backed up
  - Can not be restored on its own, ie, the entire backup chain is required

How does Backup and Restore work?

- Currently, you can only perform backup and restores by using API calls, no ARM templates or PowerShell
- Backup
  - To start a backup, the service needs to invoke the inherited member function BackupAsync
  - When BackupAsync is called, the Reliable State Manager instructs all Reliable objects to copy their latest checkpoint files to a local backup folder. The reliable services knows what is in the reliable objects already
  - The Reliable State Manager copies all log records, starting from the "start pointer" to the latest log record into the backup folder
- Restore
  - The service author needs to override the base class method OnDataLossAsync
  - A RestoreContext is provided by the OnDataLossAsync method
  - Call the RestoreAsync API on the RestoreContext to restore data

Demonstration

Backup and Restore Reliable Service Data

Recovery scenarios

- Partial data loss in Reliable Services
  - Service Fabric runtime automatically detects the data loss and calls OnDataLossAsync that the service author has provided
- Deleted or lost service
  - This typically happens when a service is removed
  - Recreate the service first then invoke OnDataLossAsync on each partition
- Replication of corrupt application data
  - Could be caused by a bug in an updated service
  - You need to first freeze the service at the application level
  - Per partition, start restoring the most recent data down to the least
  - Find the most recent backup that does not have corruption

Testing Backup and Restore

- Data loss in a particular partition can be invoked by calling the Invoke-ServiceFabricPartitionDataLoss cmdlet in PowerShell
- Programmatic API access can also be used to invoke data loss

```
$s = "fabric:/WebReferenceApplication/InventoryService"

$p = Get-ServiceFabricApplication | Get-ServiceFabricService -ServiceName $s | Get-ServiceFabricPartition | Select -First 1

$p | Invoke-ServiceFabricPartitionDataLoss -DataLossMode FullDataLoss -ServiceName $s
```

How often should I do a backup?

- Determining a backup frequency is very workload specific and therefore is hard to determine
- Generally, the frequency of your backups (whether full or incremental) depends on your Recovery Point Objective (RPO)
- Also how many incremental backups to take before a full backup depends on the following factors:
  - Your Recovery Time objective (RTO) – It takes more time to recover using incremental backups
  - The storage requirement – Full backups require more storage than incremental ones
  - Configured value for **MaxAccumulatedBackupLogSizeInMB**. If this setting is exceeded by the incremental backup(s) a full backup must be taken. This setting can be configured.

Microsoft