Microsoft®

# tech·days | 15 From the Desktop to the Cloud Years of Turning Vision into Value

Hong Kong | 2012

Microsoft® SQL Server® 2012

Microsoft® System Center 2012

Microsoft® Exchange · Microsoft® Lync · Microsoft® Office · Microsoft® Visual Studio · Windows® 7 · Windows® Internet Explorer · Windows® Phone Put people first. · Windows Azure · Microsoft® SQL Azure

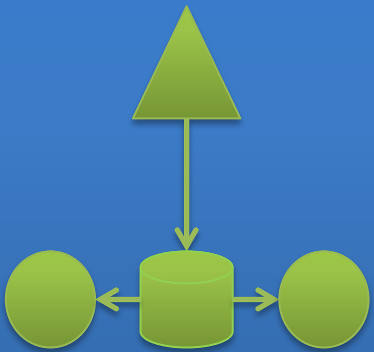
# Windows Azure Service Bus: Advanced Messaging Features

Abhishek Lal
Senior Program Manager,
Windows Azure Application Platform

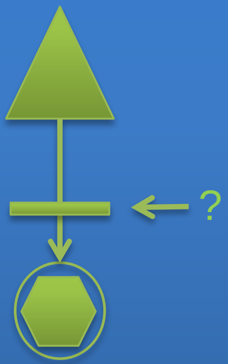
Microsoft® tech·days Hong Kong | 2012

# What is Service Bus?

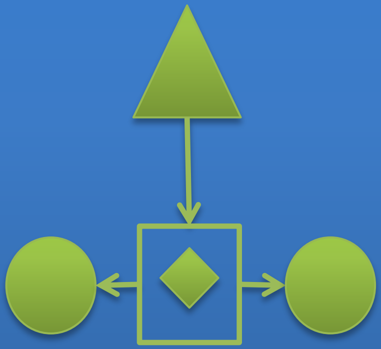
**Connectivity**
Service Relay
Protocol Tunnel
Eventing



**Messaging**
Queuing
Pub/Sub
Reliable Transfer



**Svc Management**
Naming, Discovery
Monitoring



**Integration**
Routing
Coordination
Transformation

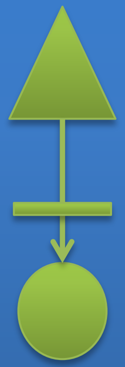
Rich options for interconnecting apps across network boundaries

Reliable, transaction-aware cloud messaging infrastructure for business apps.

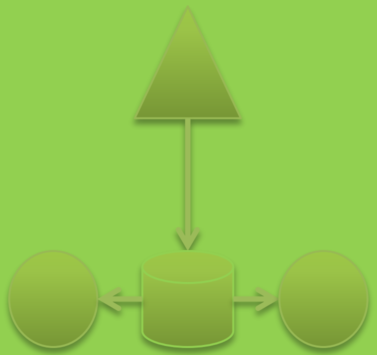
Consistent management surface and service observation capabilities

Content-based routing, document transformation, and process coordination.

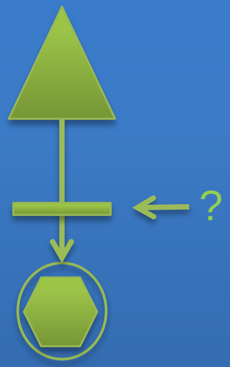
# Service Bus Messaging



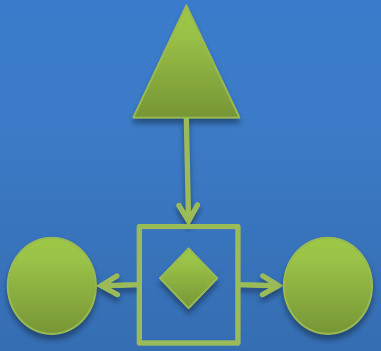
**Connectivity**
Service Relay
Protocol Tunnel
Eventing



**Messaging**
Queuing
Pub/Sub
Reliable Transfer



**Svc Management**
Naming, Discovery
Monitoring



**Integration**
Routing
Coordination
Transformation

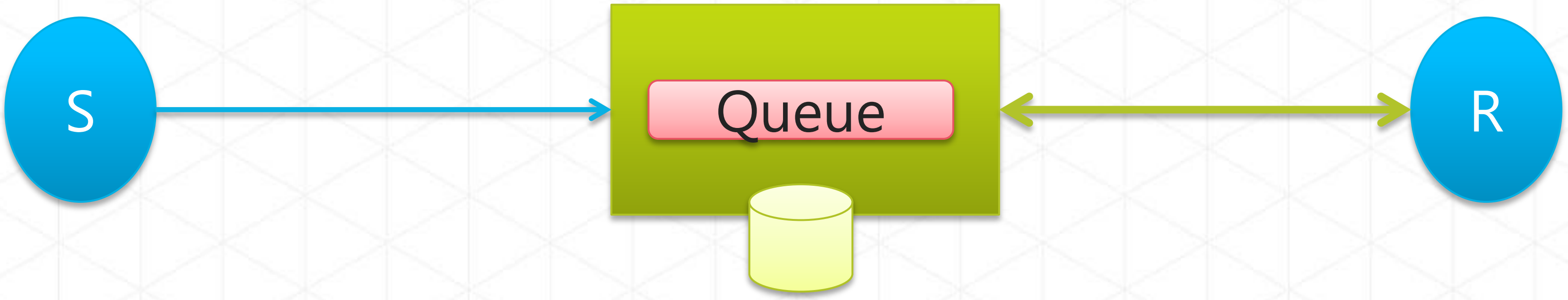
Rich options for interconnecting apps across network boundaries

Reliable, transaction-aware cloud messaging infrastructure for business apps.

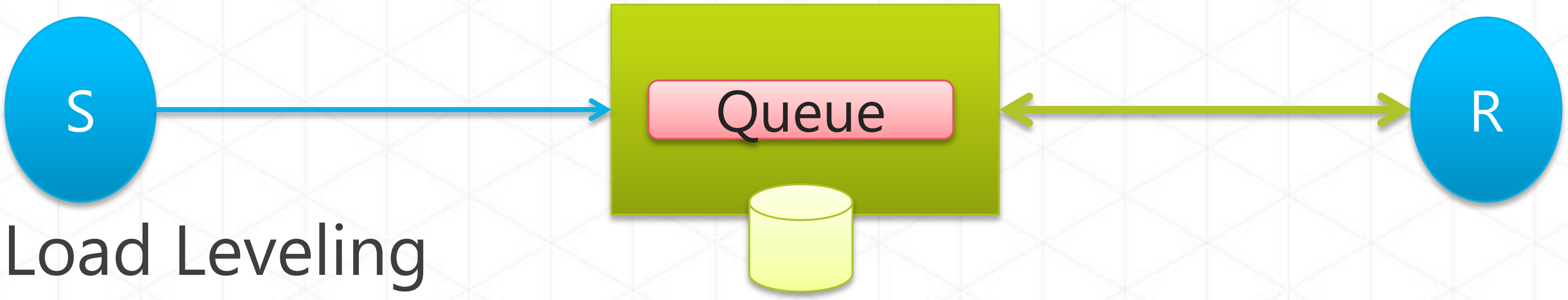
Consistent management surface and service observation capabilities

Content-based routing, document transformation, and process coordination.

# Brokered Transfer

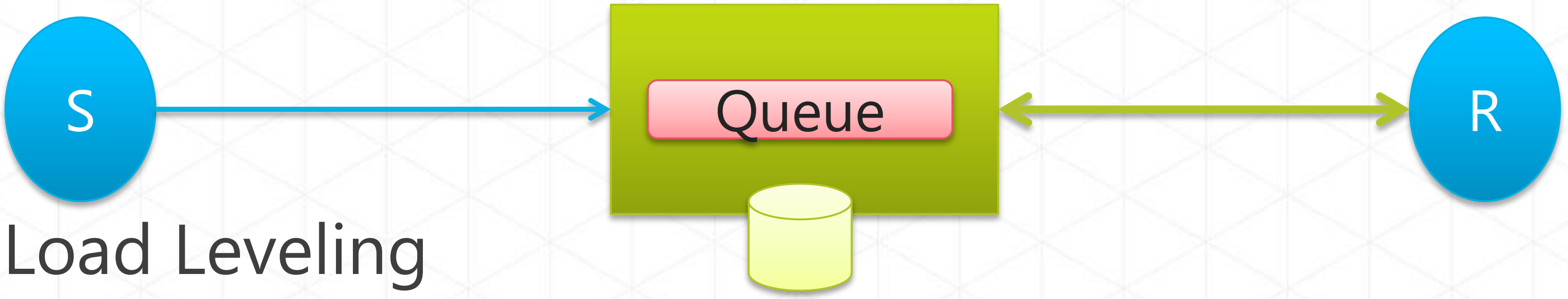


# Brokered Transfer



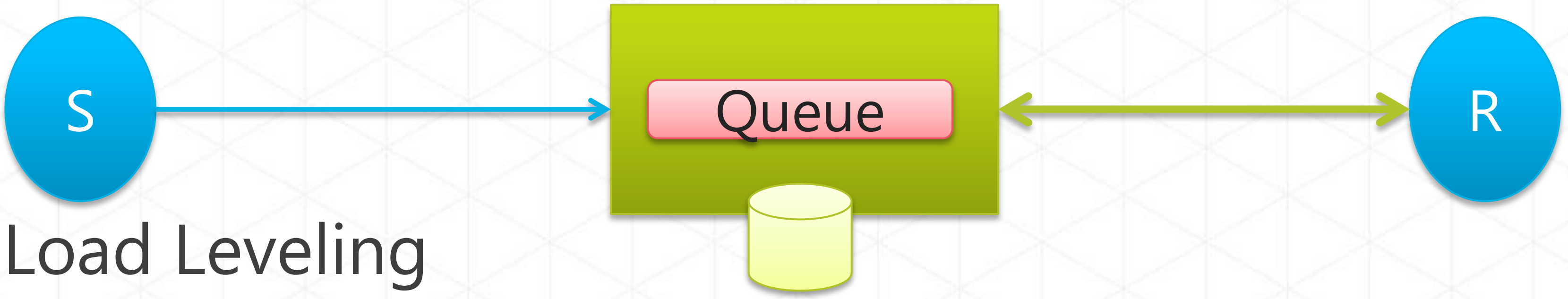
- Load Leveling

# Brokered Transfer



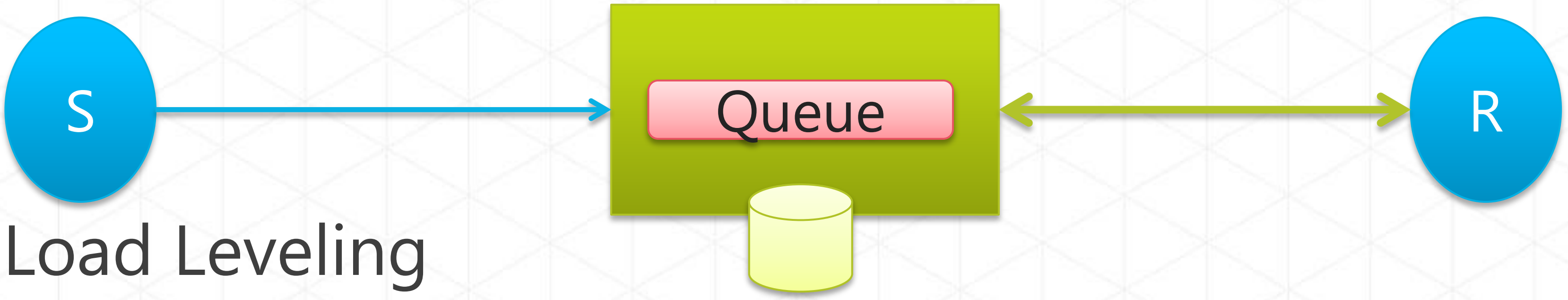
- Load Leveling
  - Receiver receives and processes at its own pace and can never be overloaded

# Brokered Transfer



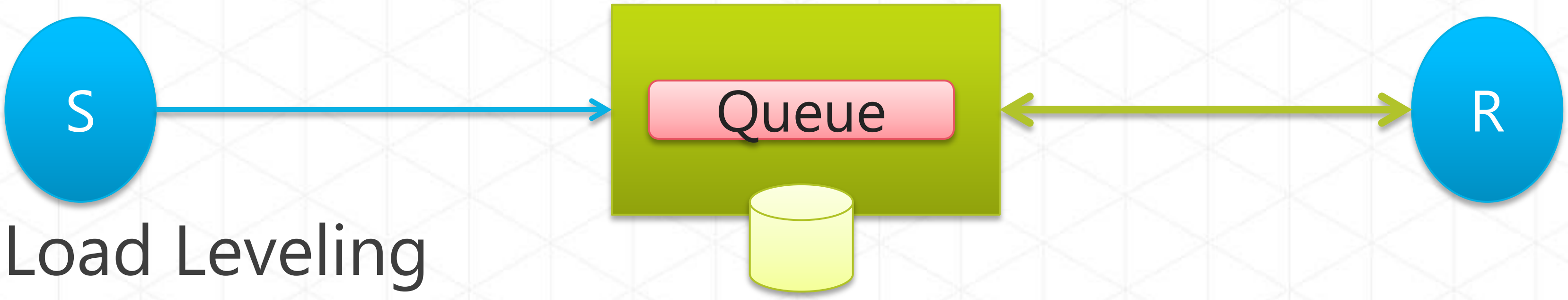
- Load Leveling
  - Receiver receives and processes at its own pace and can never be overloaded
  - Can add receivers as queue length grows, reduce receiver if queue length is low or zero

## Brokered Transfer



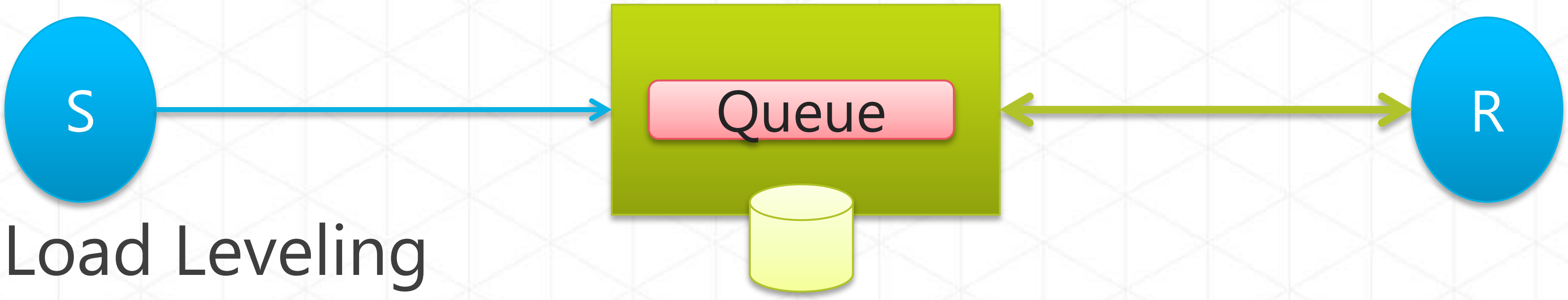
- Load Leveling
  - Receiver receives and processes at its own pace and can never be overloaded
  - Can add receivers as queue length grows, reduce receiver if queue length is low or zero
  - Gracefully handles traffic spikes by never stressing out the backend.

# Brokered Transfer



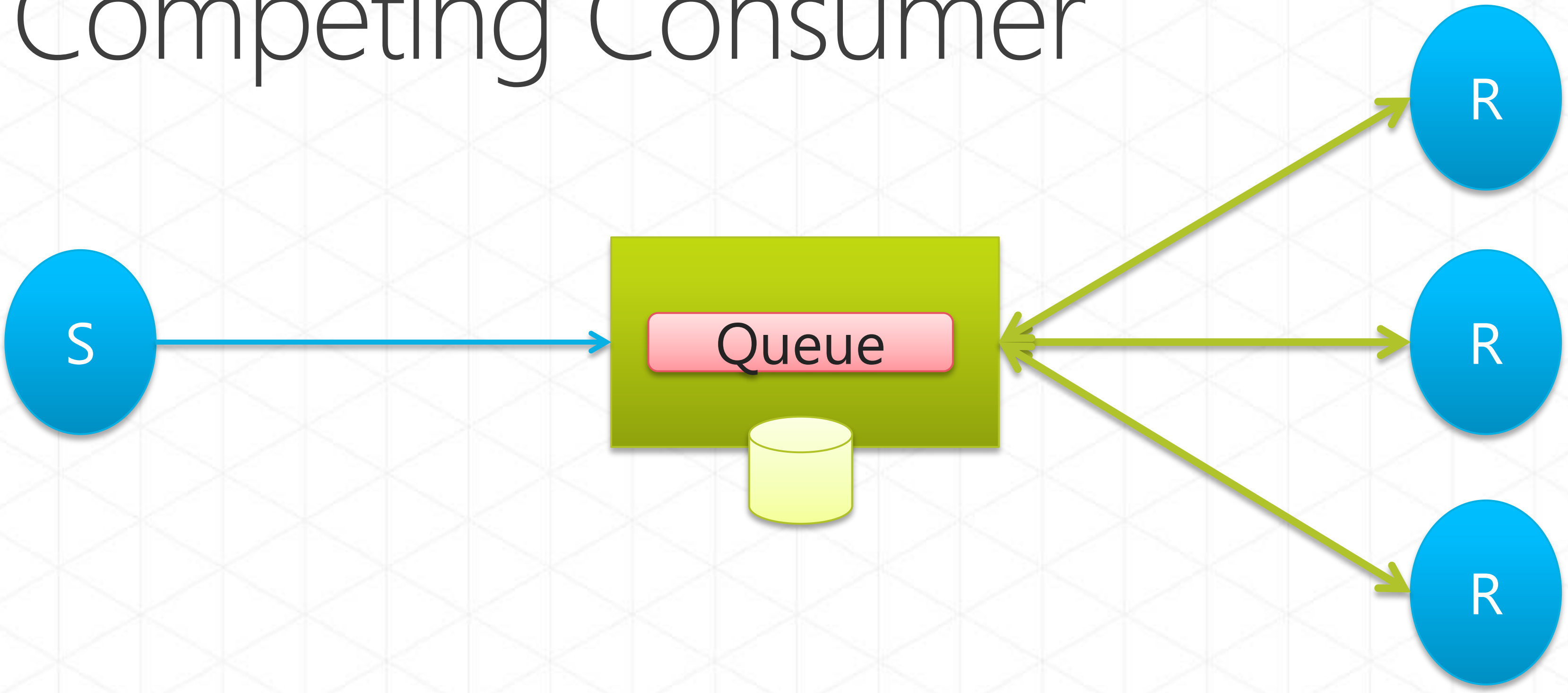
- Load Leveling
  - Receiver receives and processes at its own pace and can never be overloaded
  - Can add receivers as queue length grows, reduce receiver if queue length is low or zero
  - Gracefully handles traffic spikes by never stressing out the backend.
- Offline/Batch

# Brokered Transfer

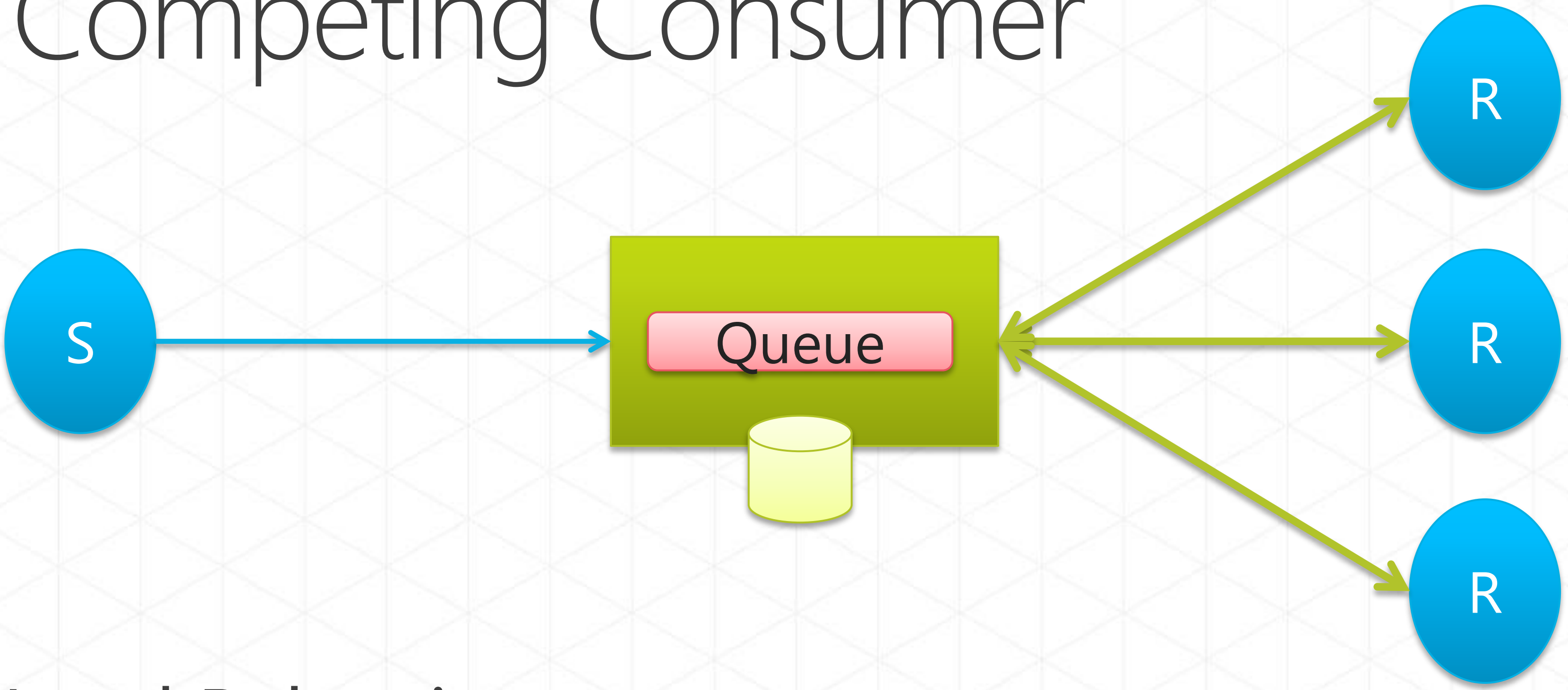


- Load Leveling
  - Receiver receives and processes at its own pace and can never be overloaded
  - Can add receivers as queue length grows, reduce receiver if queue length is low or zero
  - Gracefully handles traffic spikes by never stressing out the backend.
- Offline/Batch
  - Allows taking the receiver offline for servicing or other reasons. Requests are buffered up until the receiver is available again.

# Competing Consumer

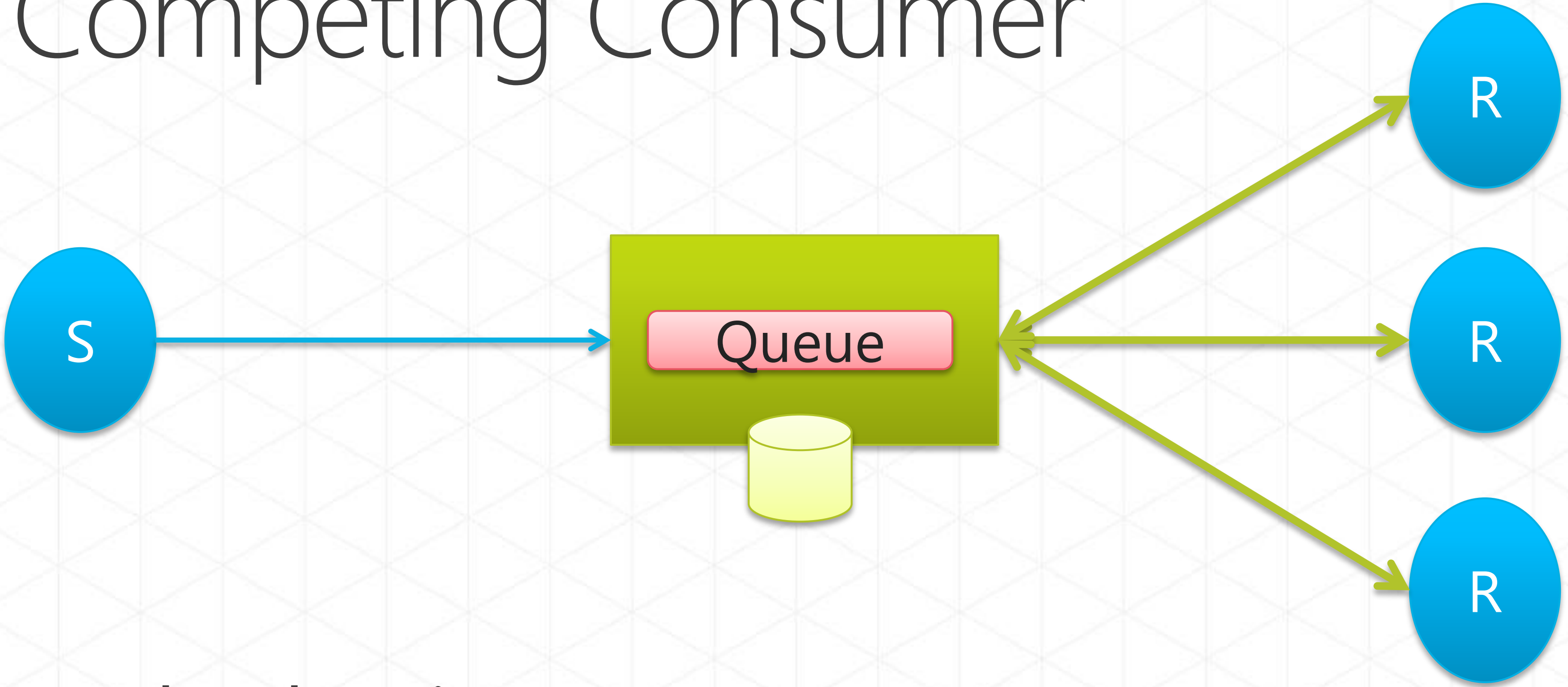


# Competing Consumer



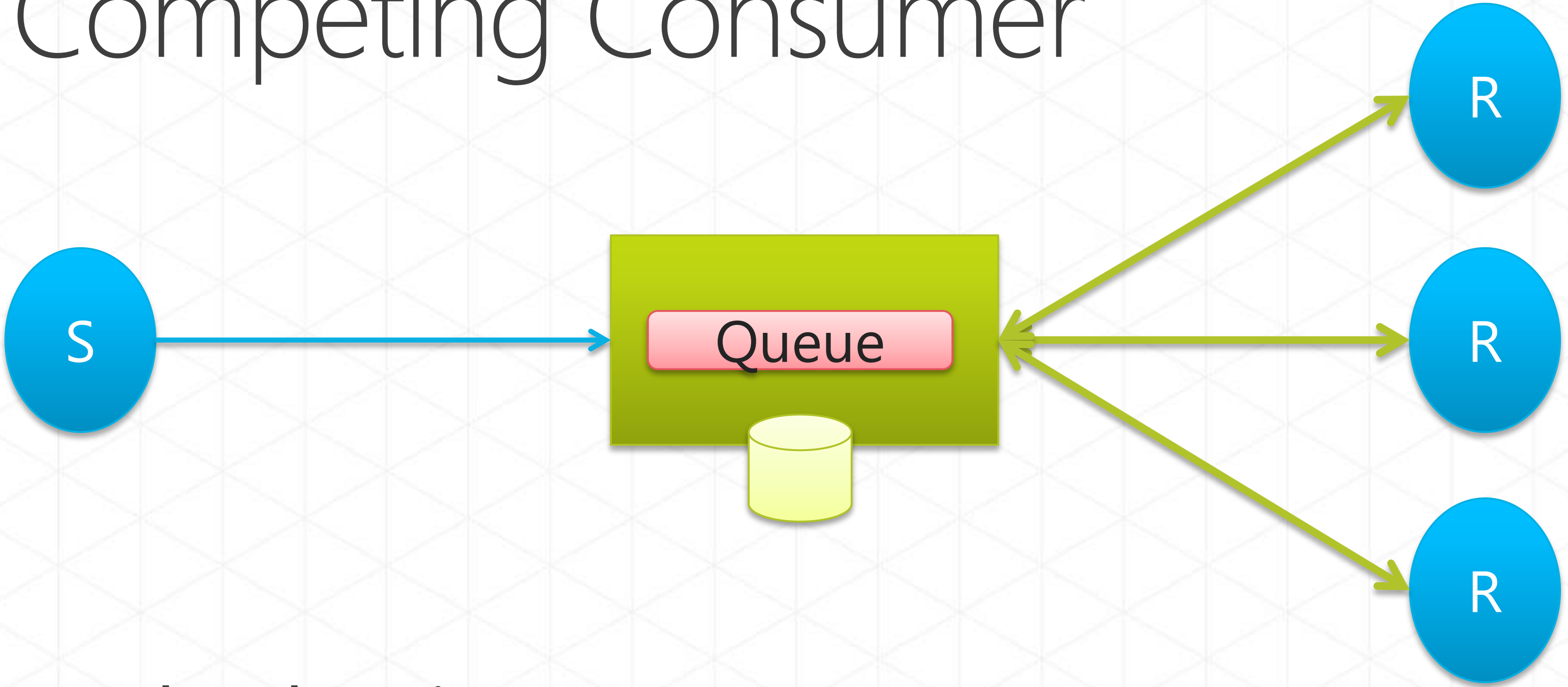
- Load Balancing

# Competing Consumer



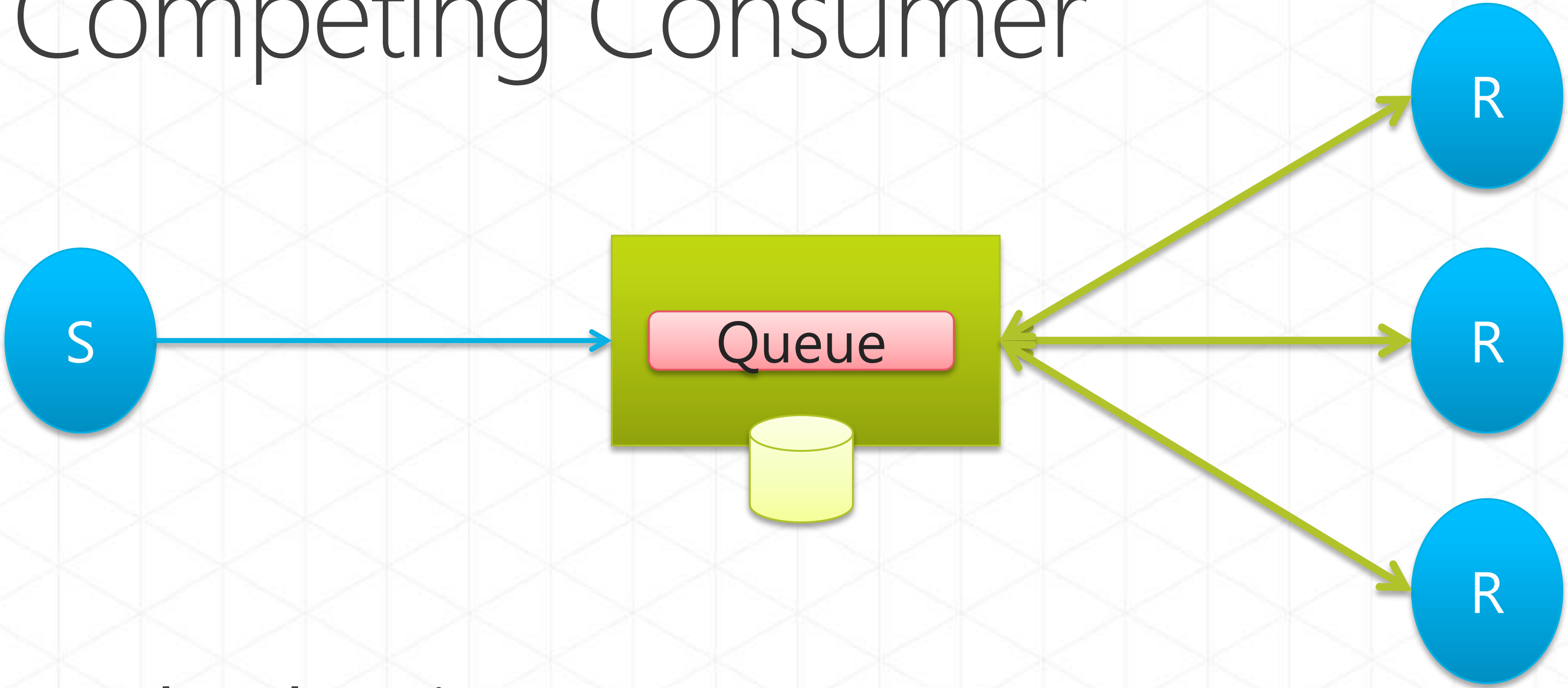
- Load Balancing
  - Multiple receivers compete for messages on the same queue (or subscription).

# Competing Consumer



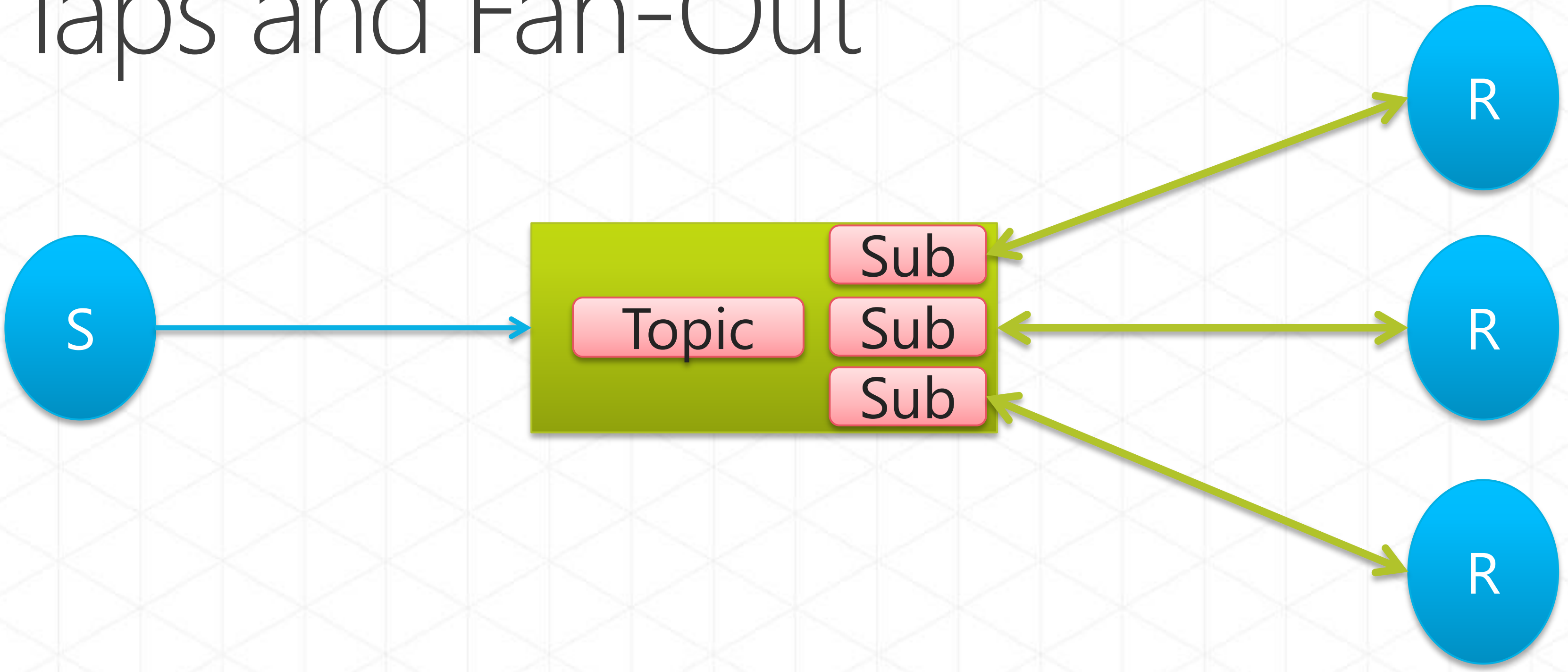
- Load Balancing
  - Multiple receivers compete for messages on the same queue (or subscription).
  - Provides automatic load balancing of work to receivers volunteering for jobs.

# Competing Consumer

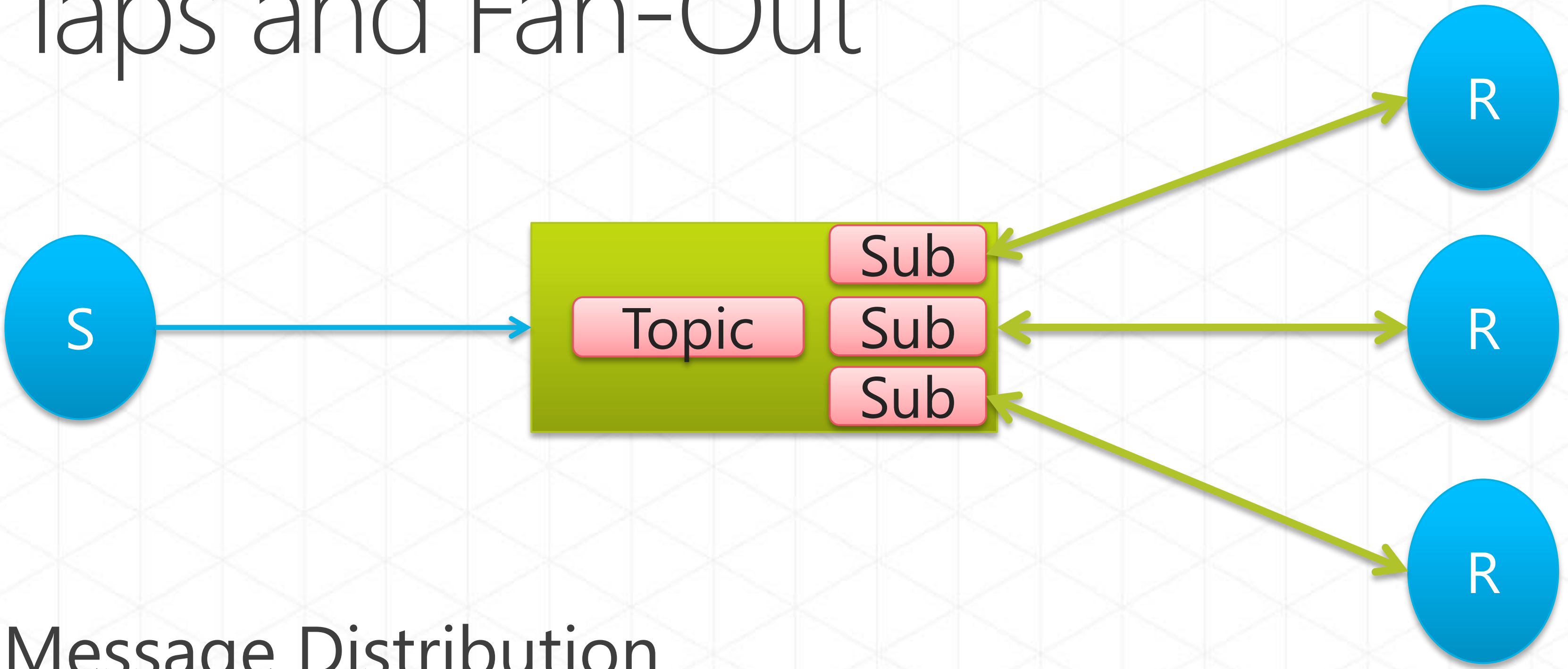


- Load Balancing
  - Multiple receivers compete for messages on the same queue (or subscription).
  - Provides automatic load balancing of work to receivers volunteering for jobs.
  - Observing the queue length allows to determine whether more receivers are required.

# Taps and Fan-Out

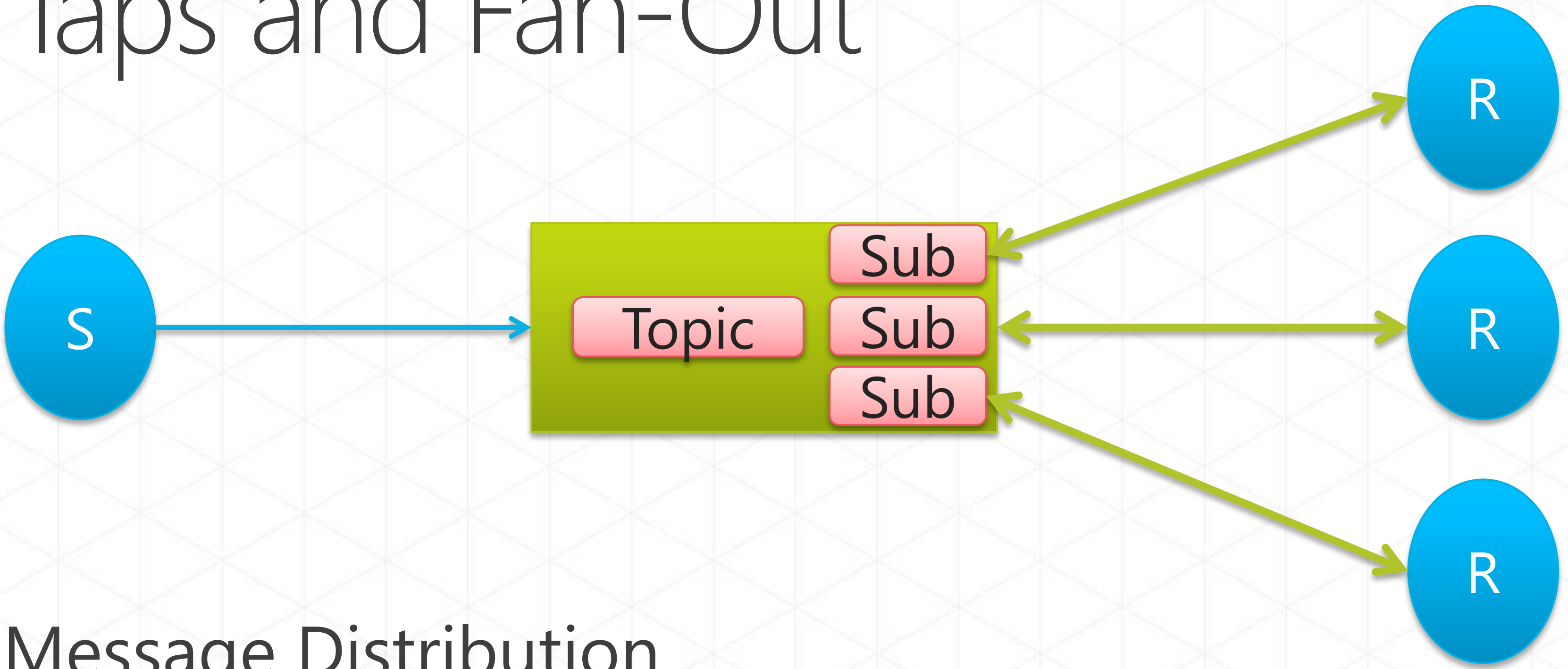


# Taps and Fan-Out



- Message Distribution
  - Each receiver gets its own copy of each message. Subscriptions are independent.
  - Allows for many independent 'taps' into a message stream. Subscriber can filter down by interest.

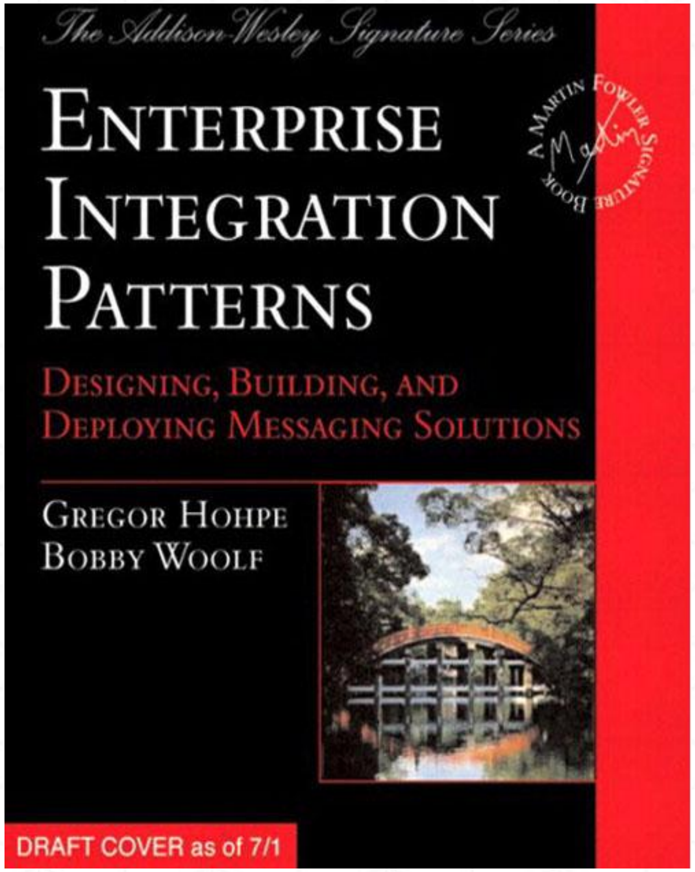
# Taps and Fan-Out



- Message Distribution
  - Each receiver gets its own copy of each message. Subscriptions are independent.
  - Allows for many independent 'taps' into a message stream. Subscriber can filter down by interest.
- Constrained Message Distribution (Partitioning)
  - Receiver get mutually exclusive slices of the message stream by creating appropriate filter expressions.

# Enterprise Integration Patterns

http://www.eaipatterns.com/
by Gregor Hohpe


The Addison-Wesley Signature Series
ENTERPRISE INTEGRATION PATTERNS
DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS
GREGOR HOHPE
BOBBY WOOLF
A MARTIN FOWLER SIGNATURE BOOK
DRAFT COVER as of 7/1

## Need for integration

# Need for integration

- Enterprises typically comprised of hundreds of applications
  - Custom built
  - Acquired from third parties
  - Part of legacy systems

# Need for integration

- Enterprises typically comprised of hundreds of applications
  - Custom built
  - Acquired from third parties
  - Part of legacy systems
- Customers do not think about these system boundaries
  - They interact with the business
  - Common processes and data sharing needs to be supported = Integration

# Need for integration

- Enterprises typically comprised of hundreds of applications
  - Custom built
  - Acquired from third parties
  - Part of legacy systems
- Customers do not think about these system boundaries
  - They interact with the business
  - Common processes and data sharing needs to be supported = Integration
- This is not an easy task
  - Different data types and formats
  - Different types of extensibility / states of modifications possible
  - Different application platforms and systems

## Common types of Integration

## Common types of Integration

- Information portals

# Common types of Integration

- Information portals
- Data replication

## Common types of Integration

- Information portals
- Data replication
- Shared business function

## Common types of Integration

- Information portals
- Data replication
- Shared business function
- Service oriented architectures

# Common types of Integration

- Information portals
- Data replication
- Shared business function
- Service oriented architectures
- Distributed business processes

## Common types of Integration

- Information portals
- Data replication
- Shared business function
- Service oriented architectures
- Distributed business processes
- Business-to-business integration

## Options for integration

## Options for integration

- File transfer
  - allow systems to share data/state but not functionality

## Options for integration

- File transfer
  - allow systems to share data/state but not functionality
- Shared Database
  - allow systems to share data/state but not functionality

# Options for integration

- File transfer
  - allow systems to share data/state but not functionality
- Shared Database
  - allow systems to share data/state but not functionality
- Remote procedure invocation
  - enables shared functionality but tightly couples applications

# Options for integration

- File transfer
  - allow systems to share data/state but not functionality
- Shared Database
  - allow systems to share data/state but not functionality
- Remote procedure invocation
  - enables shared functionality but tightly couples applications
- Messaging
  - Use messaging when you need to transfer packets of data
    - Frequently
    - Immediately
    - Reliably
    - Asynchronously
    - In customizable formats

## Messaging Concepts

## Messaging Concepts

- Channels – a virtual pipe that connects a sender to a receiver

# Messaging Concepts

- Channels – a virtual pipe that connects a sender to a receiver
- Messages – an atomic packet for data that can be transmitted on a channel

## Messaging Concepts

- Channels – a virtual pipe that connects a sender to a receiver
- Messages – an atomic packet for data that can be transmitted on a channel
- Pipes & Filters – that perform certain actions on messages

## Messaging Concepts

- **Channels** – a virtual pipe that connects a sender to a receiver
- **Messages** – an atomic packet for data that can be transmitted on a channel
- **Pipes & Filters** – that perform certain actions on messages
- **Routing** – allows the message to navigate the channel topology

# Messaging Concepts

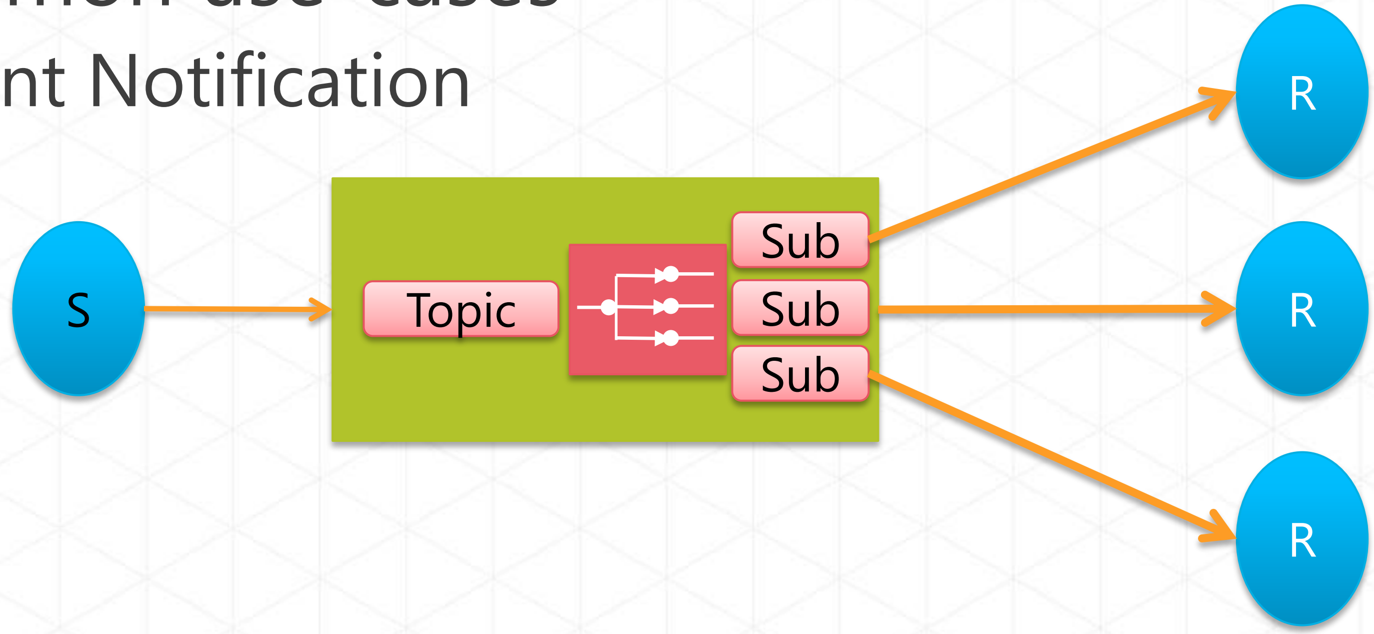
- Channels – a virtual pipe that connects a sender to a receiver
- Messages – an atomic packet for data that can be transmitted on a channel
- Pipes & Filters – that perform certain actions on messages
- Routing – allows the message to navigate the channel topology
- Transformation – converts messages from one format to another

## Messaging Patterns

- Publish Subscribe
- Content Based Router
- Recipient List
- Fan-In
- Update/Read Separation
- Update Notifications
- Diagnostics / Statistics
- Correlation

# Publish-Subscribe

- Scenario
  - Sender broadcasts event to all interested receivers
- Common use-cases
  - Event Notification



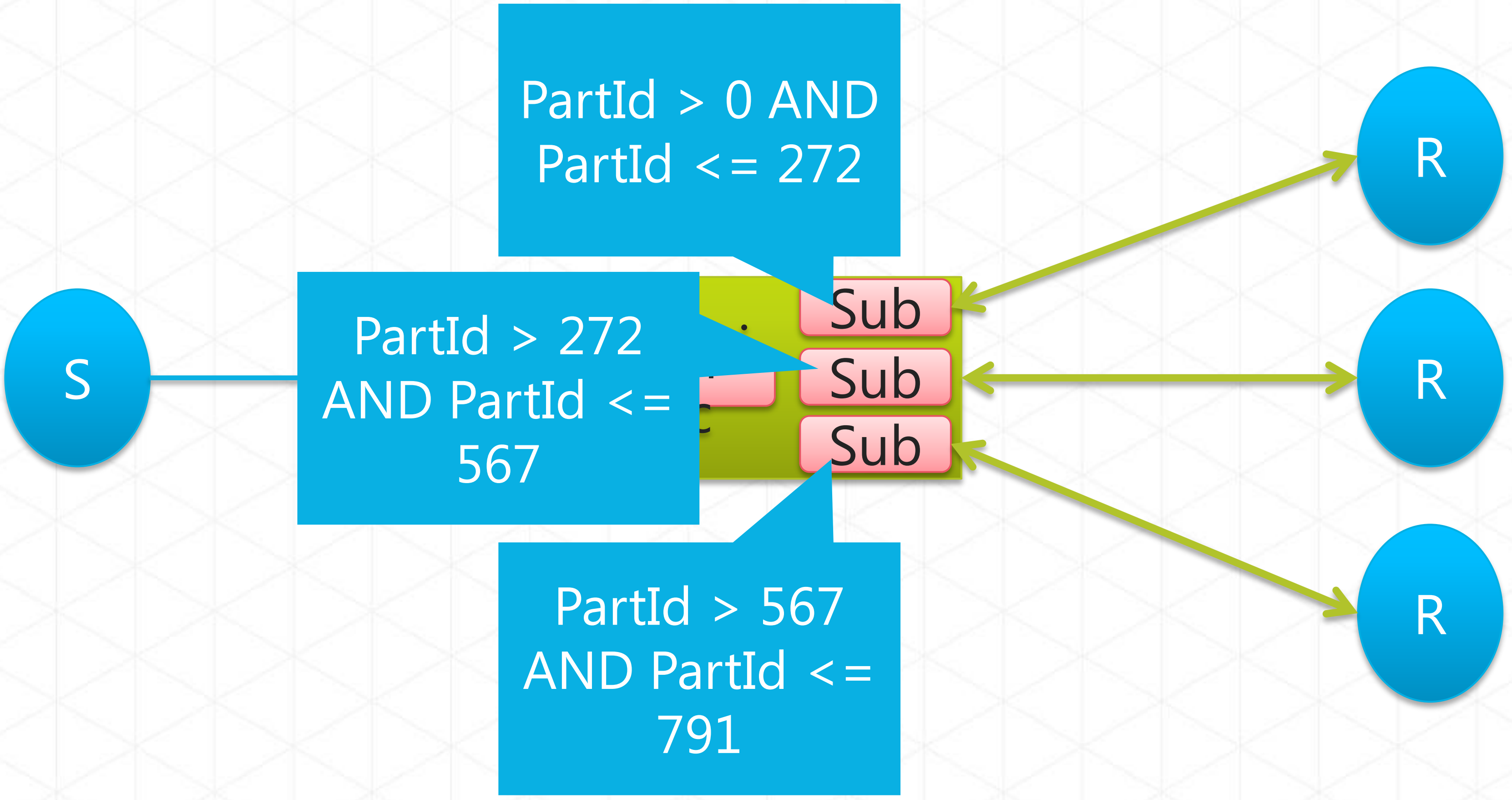
# Create Topics and Subscriptions

```
 Uri managementUri =
ServiceBusEnvironment.CreateServiceUri("sb",
                                       ServiceBusNamespace,
                                       string.Empty);

NamespaceManager namespaceManager =
new NamespaceManager(managementUri,
TokenProvider.CreateSharedSecretTokenProvider
(ServiceBusIssuerName,
ServiceBusIssuerKey));

 TopicDescription mainTopic =
namespaceManager.CreateTopic(“topicName");

 namespaceManager.CreateSubscription(“topicName",
“FirstSubscription");
```

demo

## Publish Subscribe

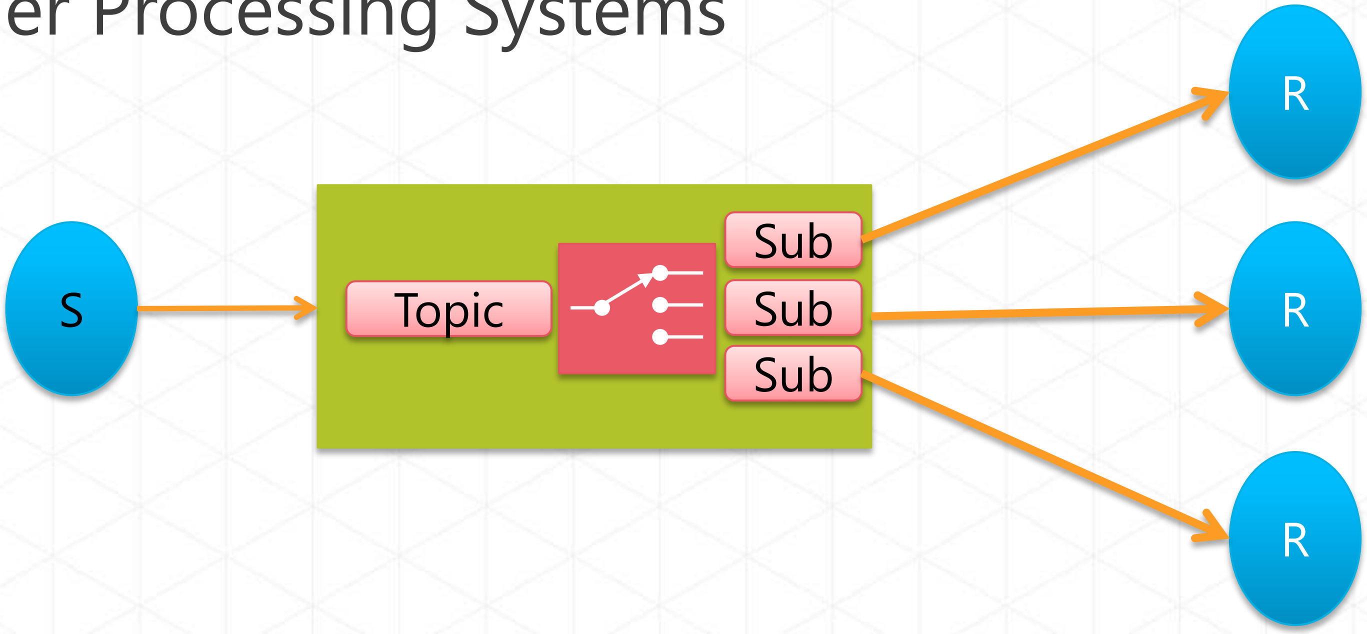
# Partitioning


PartId > 0 AND PartId <= 272
PartId > 272 AND PartId <= 567
PartId > 567 AND PartId <= 791
S
Sub
Sub
Sub
R
R
R

- Rule conditions form mutually exclusive ranges
- Allows partitioning-aware message distribution
- No need for sender to be aware of partitioning

# Content-based router

- Scenario
  - Route a message to different recipients based on data contained in the message
- Common use-cases
  - Order Processing Systems



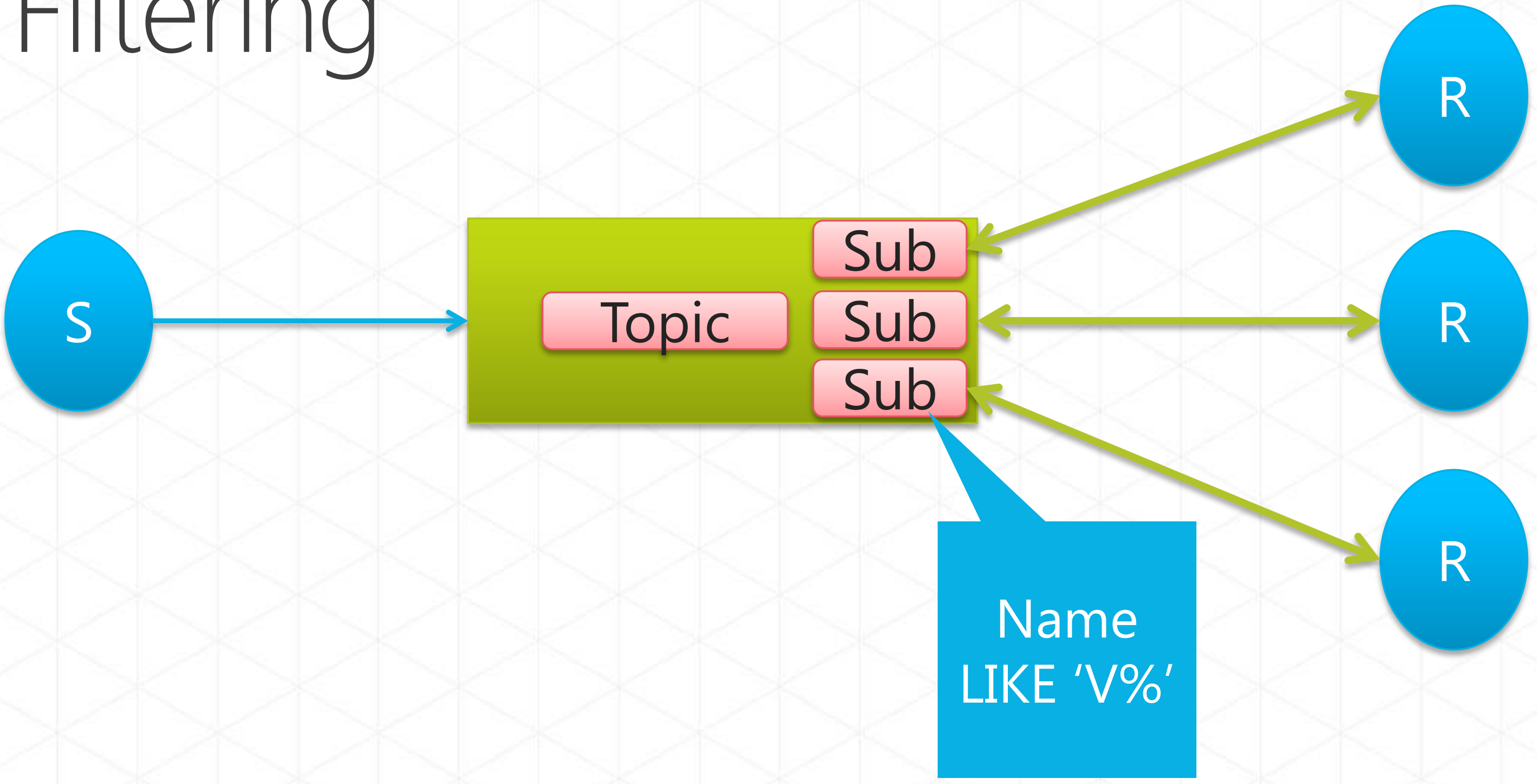
## Create Subscriptions with Rules (Filters)

```
 TopicDescription mainTopic =
namespaceManager.CreateTopic(“topicName");

 namespaceManager.CreateSubscription(“topicName",
“AuditSubscription");

 namespaceManager.CreateSubscription(“topicName",
“Category1Subscription",
      new SqlFilter(“Category = 1"));

 namespaceManager.CreateSubscription(“topicName",
“CategoryNot1Subscription",
      new SqlFilter(“Category <> 1"));

 BrokeredMessage  myMessage = new BrokeredMessage();
        myMessage.Properties.Add(“Category”, 1);
                     or
        myMessage.Properties.Add(“Category”, 2);
                     or
        myMessage.Properties.Add(“Category”, 3);
```

demo

# Content Based Router

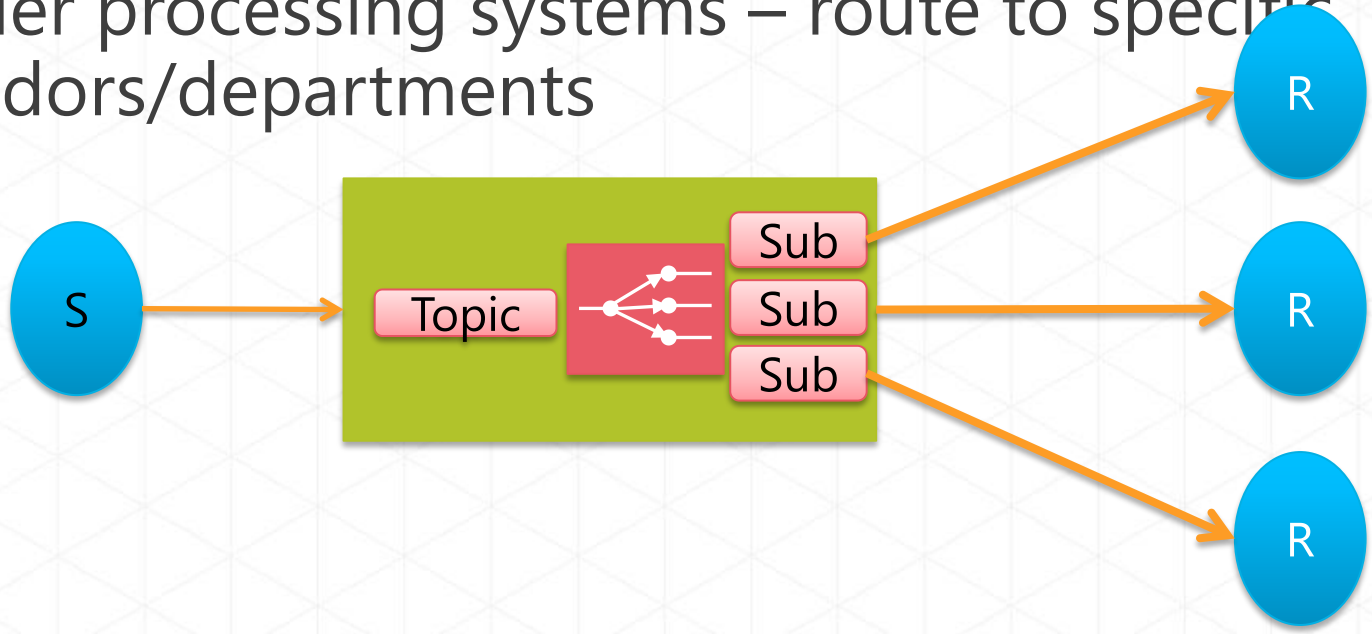
# Filtering



- Up to 2000 rules per topic
- Each matched rule yields a message copy
- SQL'92 expressions over message properties

# Recipient List

- Scenario
  - The sender wants to send the message to a list of recipients
- Common use-cases
  - Order processing systems – route to specific vendors/departments



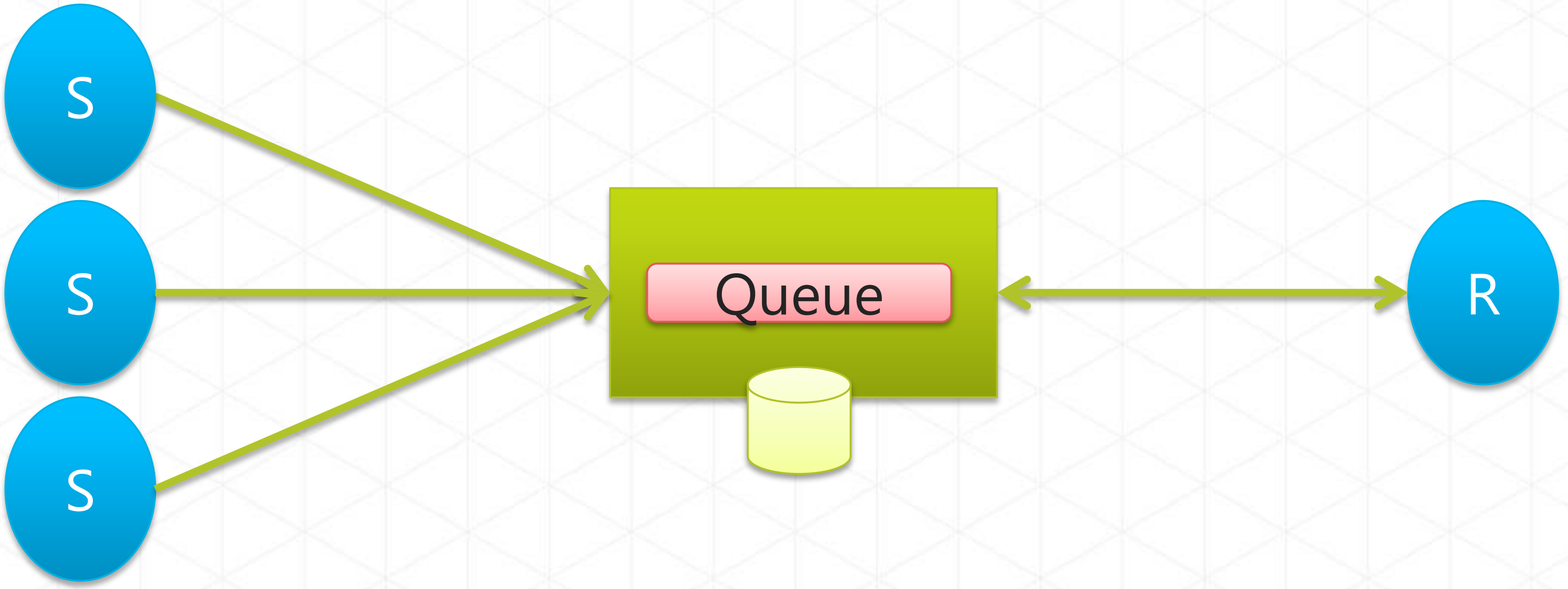
# Create Rules (with SQL Filters)

```
 TopicDescription mainTopic =
namespaceManager.CreateTopic(“topicName");

 namespaceManager.CreateSubscription(“topicName",
“AuditSubscription");

 namespaceManager.CreateSubscription(T“topicName",
"FirstSubscription",
      new SqlFilter("Address LIKE '%First%'"));

 namespaceManager.CreateSubscription(T“topicName",
“SecondSubscription",
      new SqlFilter("Address LIKE '%Second%'"));

 BrokeredMessage  myMessage = new BrokeredMessage();
       myMessage.Properties.Add(“Address”, “First”);
                    or
       myMessage.Properties.Add(“Address”, “Second”);
                    or
       myMessage.Properties.Add(“Address”, “First,Second”);
```

demo

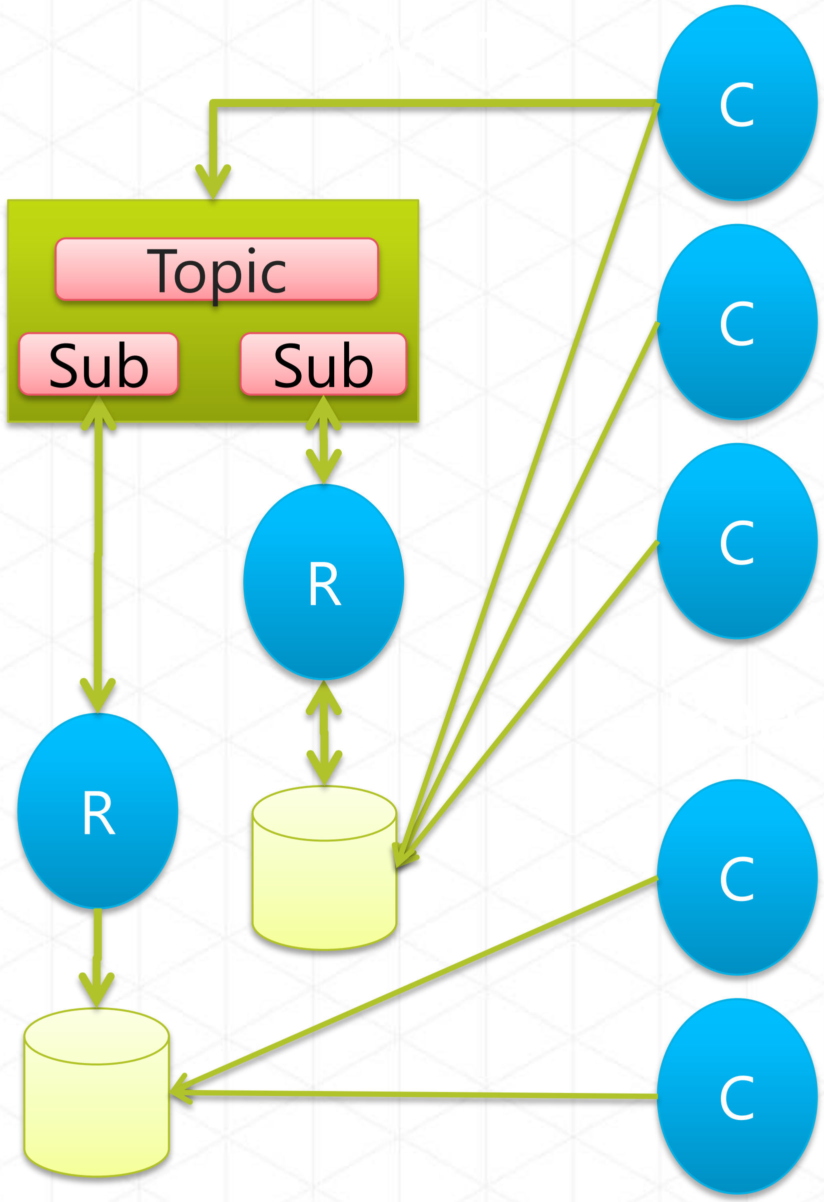
## Recipient List

## Fan-In



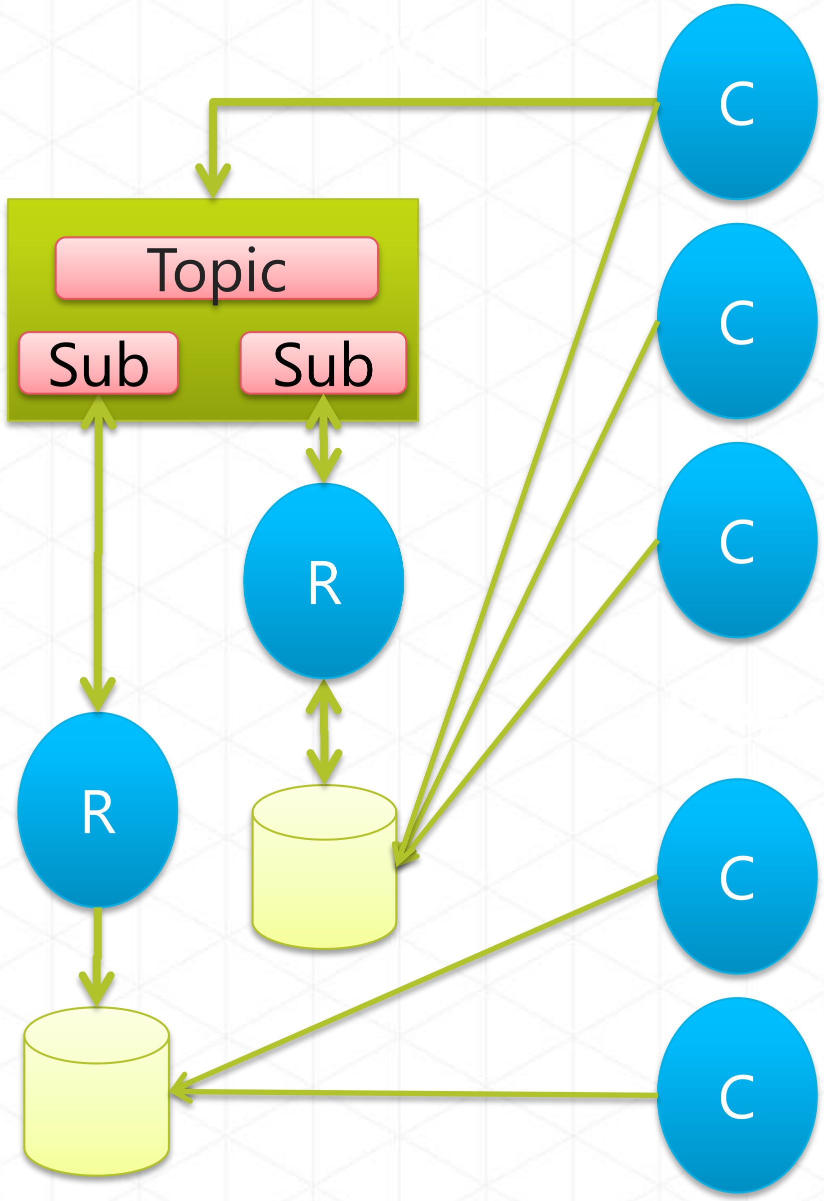
- Concentrator
  - Fan information into a single queue from a range of data sources
- Multi-Stage Aggregration / Rollup
  - Fan into a set of queues, perform aggregation/roll-up/reduction and fan further.

# Update/Read Separation

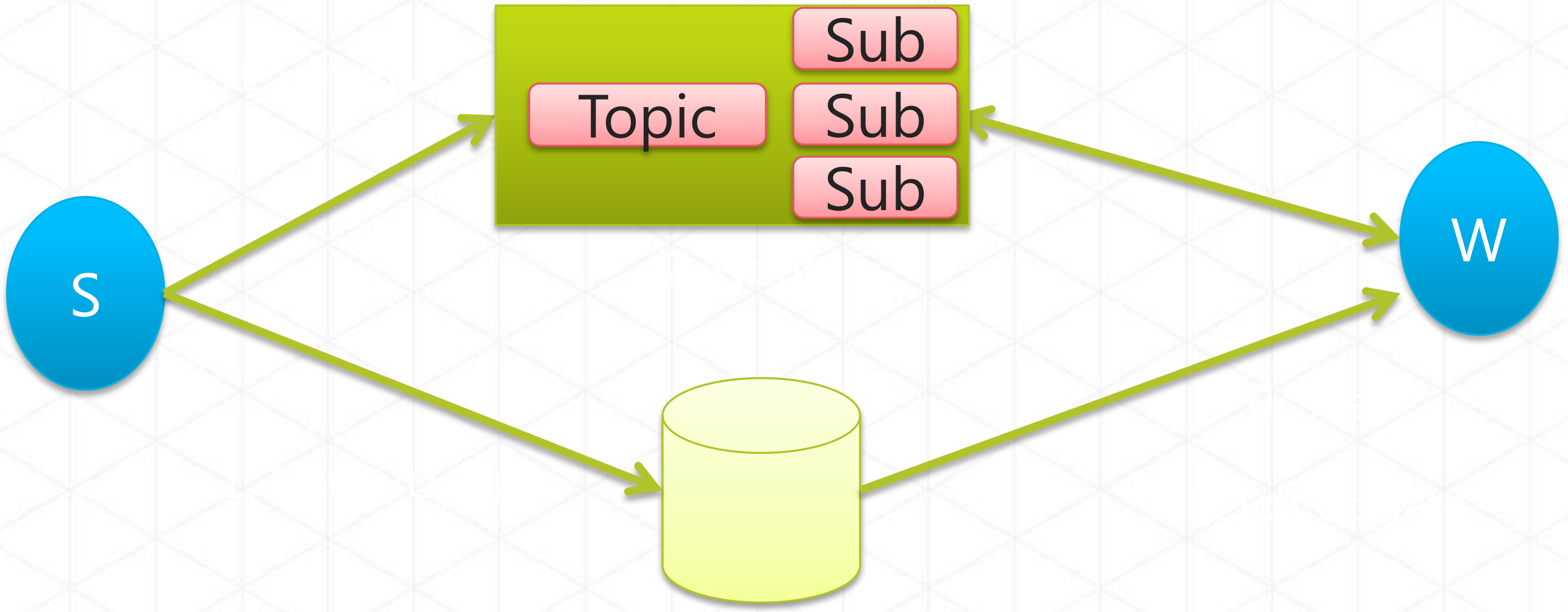


# Update/Read Separation

- Reads on partitioned stores
- All writes through messages
- Distribution via fan-out
- Trades timeliness and instant feedback for robustness and scale

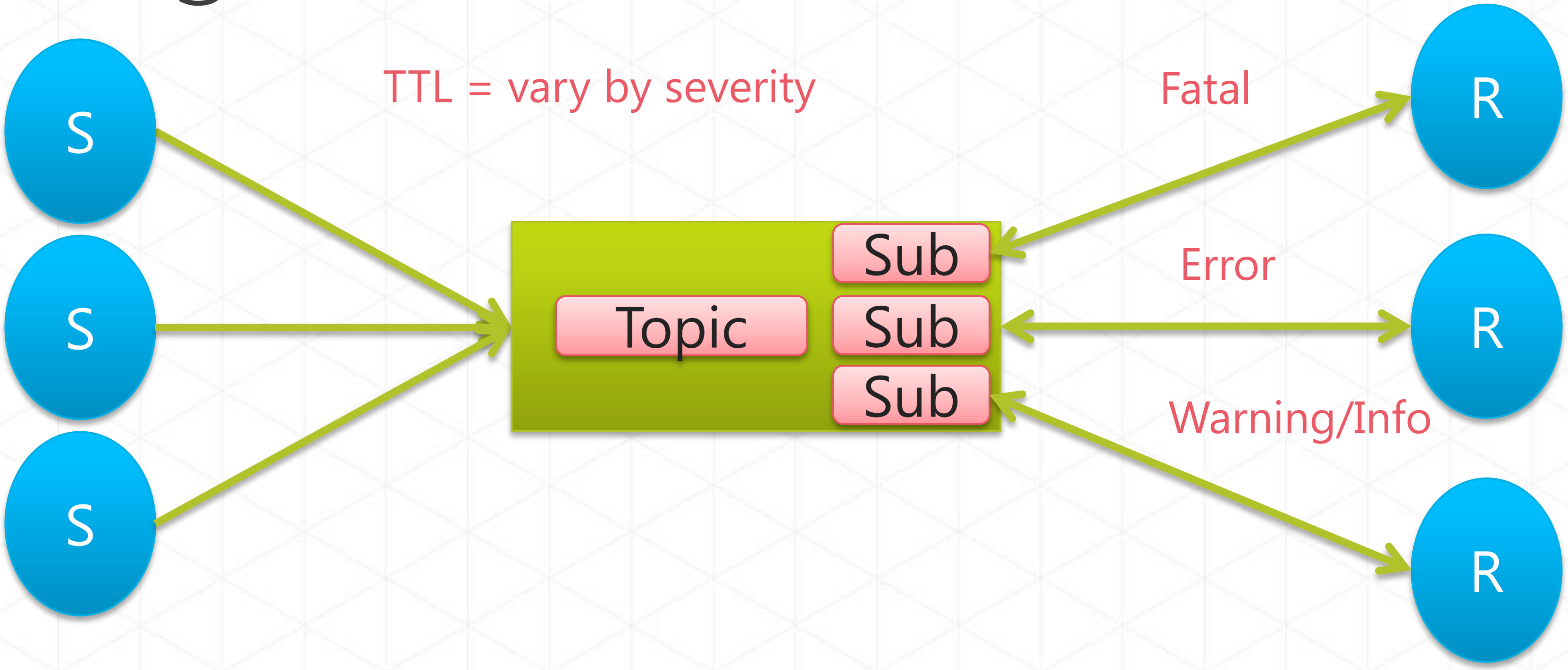


# Update Notification



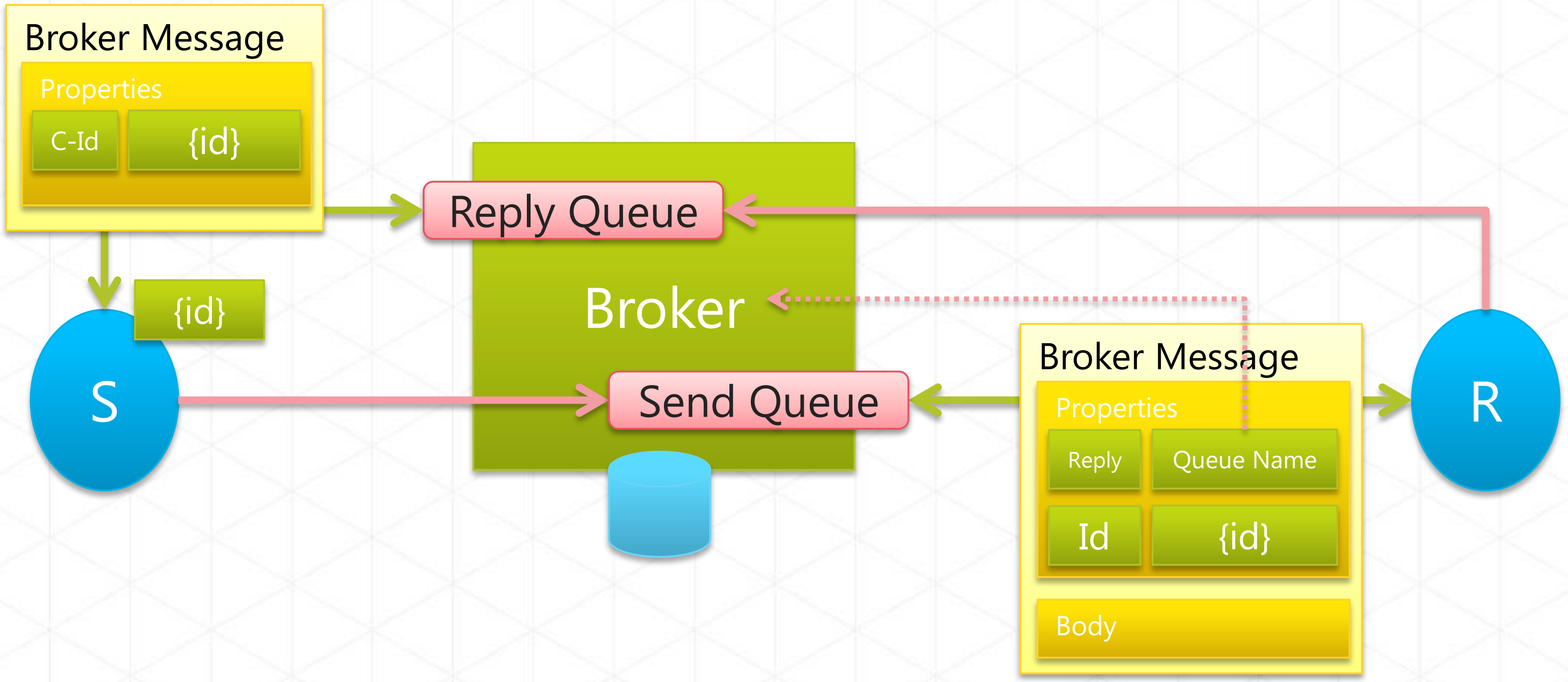
- Long-running workers seeded with state from a database
- Other part of the system inserts new jobs into the database
- Notification routed via message (uni/multi) to trigger fetch

# Diagnostics / Statistics



- Flow diagnostics events from backend services
- Vary TTL by severity. Verbose logs very short lived, fatal error reports long-lived.
- Filter by severity or needs of different audiences

# Correlation



- Correlation is required to set up reply paths between sender and receiver.
- Three correlation models in Service Bus: Message-correlation, subscription-correlation, session-correlation

## Correlation in Service Bus

## Correlation in Service Bus

- Message Correlation (Queues)
  - Originator sets Message- or CorrelationId, Receiver copies it to reply
  - Reply sent to Originator-owned Queue indicated by ReplyTo
  - Originator receives and dispatches on CorrelationId

## Correlation in Service Bus

- Message Correlation (Queues)
  - Originator sets Message- or CorrelationId, Receiver copies it to reply
  - Reply sent to Originator-owned Queue indicated by ReplyTo
  - Originator receives and dispatches on CorrelationId
- Subscription Correlation (Topics)
  - Originator sets Message- or CorrelationId, Receiver copies it to reply
  - Originator has Subscription on shared reply Topic w/ rule covering Id
  - Originator receives and dispatches on CorrelationId

## Correlation in Service Bus

- Message Correlation (Queues)
  - Originator sets Message- or CorrelationId, Receiver copies it to reply
  - Reply sent to Originator-owned Queue indicated by ReplyTo
  - Originator receives and dispatches on CorrelationId
- Subscription Correlation (Topics)
  - Originator sets Message- or CorrelationId, Receiver copies it to reply
  - Originator has Subscription on shared reply Topic w/ rule covering Id
  - Originator receives and dispatches on CorrelationId
- Session Correlation
  - Originator sets some SessionId on outbound session
  - Receiver reuses SessionId for reply session
  - Originator filters on known SessionId using session receiver

## When use Which?

## When use Which?

- Message Correlation (Queues)
  - High throughput needs; work usually completes in minimal time
  - It’s ok for the replying party to directly know of the reply destination

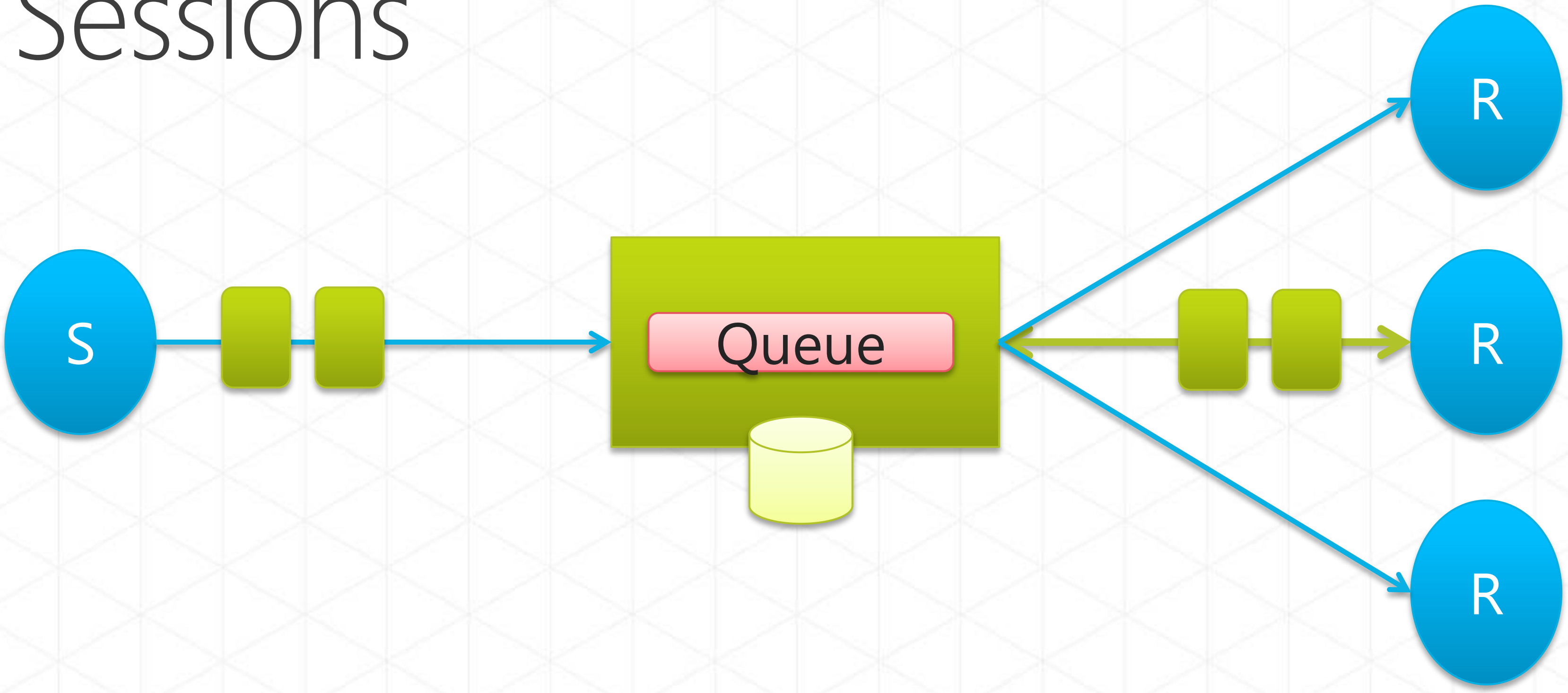
## When use Which?

- Message Correlation (Queues)
  - High throughput needs; work usually completes in minimal time
  - It's ok for the replying party to directly know of the reply destination
- Subscription Correlation (Topics)
  - Decoupling of replying party and destination
  - Longer lived jobs that may require moving handling between subscriptions by ways of moving rules

# When use Which?

- Message Correlation (Queues)
  - High throughput needs; work usually completes in minimal time
  - It’s ok for the replying party to directly know of the reply destination
- Subscription Correlation (Topics)
  - Decoupling of replying party and destination
  - Longer lived jobs that may require moving handling between subscriptions by ways of moving rules
- Session Correlation
  - Reliable multiplexed duplex communication

## Sessions



- Work-Set Pinning
  - Sessions allow pinning sets of related sets of related messages to a particular receiver even when using competing consumers.

## Sessions – Creating Session-Aware Entities

```
 namespaceManager.CreateQueue(
         new QueueDescription(queueName)
         { RequiresSession = true });


     namespaceManager.CreateSubscription(
         new SubscriptionDescription(topicName,
subName)
         { RequiresSession = true });
```

## Sessions – Sending Messages

```
var msg = new BrokeredMessage
          {
            SessionId = sessionId,
            Properties = {
                           { "JobId", jobId },
                           { "Result", result }
                         }
          };
```

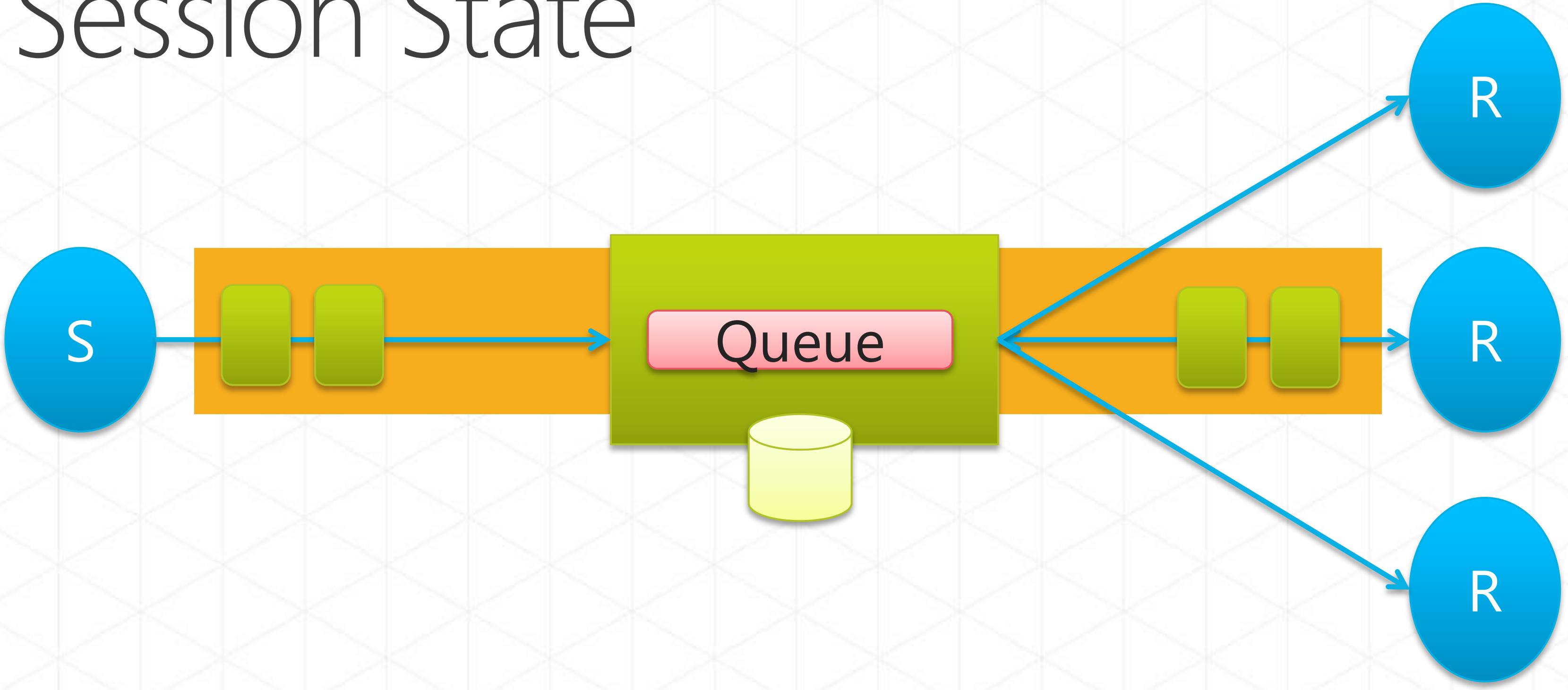
## Sessions – Receiving Messages

```
var qc =
messagingFactory.CreateQueueClient(queueName);

var session =
replyQueueClient.AcceptMessageSession(sessionId);

var msg = session.Receive();
```

demo

## Service Bus Sessions

## Session State



- Allows storing session state in Service Bus
  - Size limit equivalent to one message (256KB)
  - Enables Work Set pinning with safe failover to secondary

## Sessions – Storing Processing State

```
var qc =
messagingFactory.CreateQueueClient(queueName);
var session =
replyQueueClient.AcceptMessageSession(sessionId);
var msg = session.Receive();

session.SetState(serializedProcessingState);
```

demo

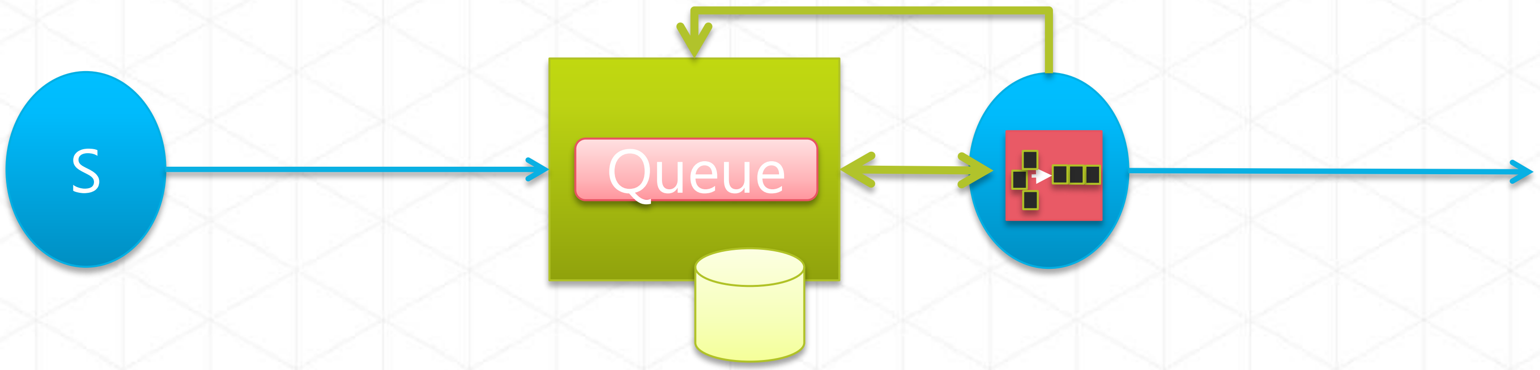
## Service Bus Session State: Re-sequencer

# Re-sequencer

- Scenario
  - A statefull filter which collects and reorders messages
- Common use-cases
  - Get a stream of related but out-of-order messages back into the correct order

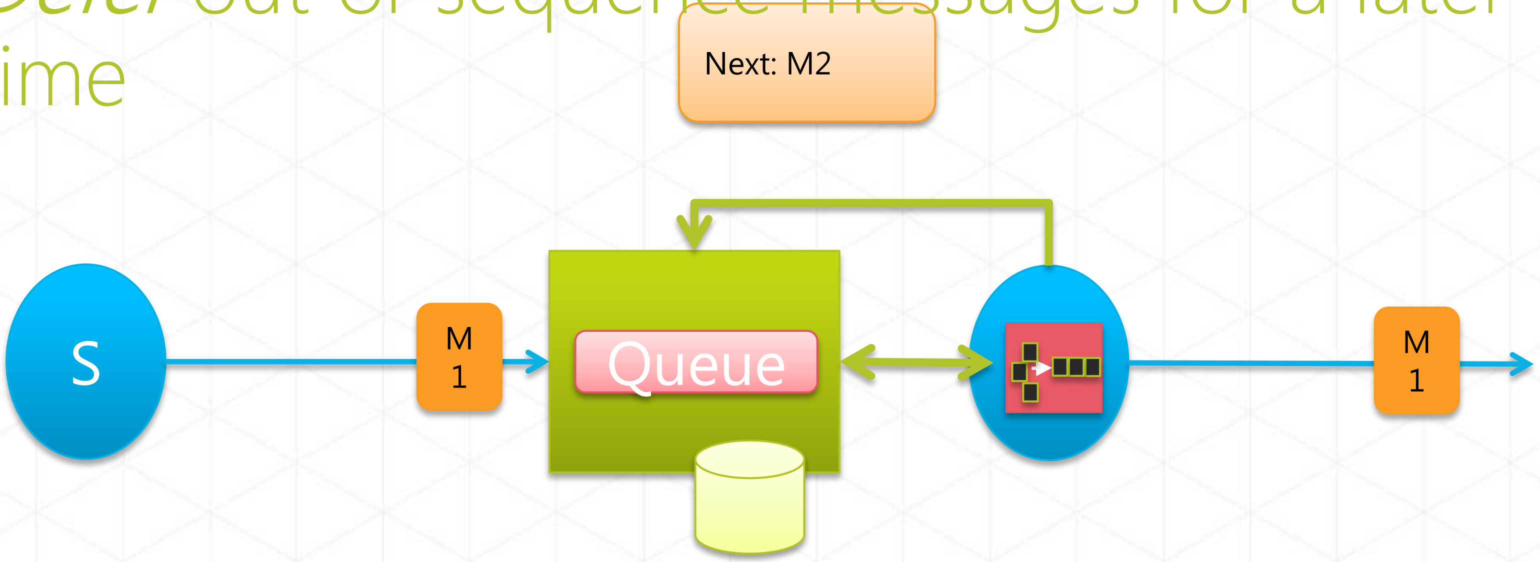
## Implementation

- Correlate messages with *sessions ID.* Identify sequence with a sequence ID property
- Use *session state* to store out-of-sequence messages
- *Defer* out-of-sequence messages for a later time



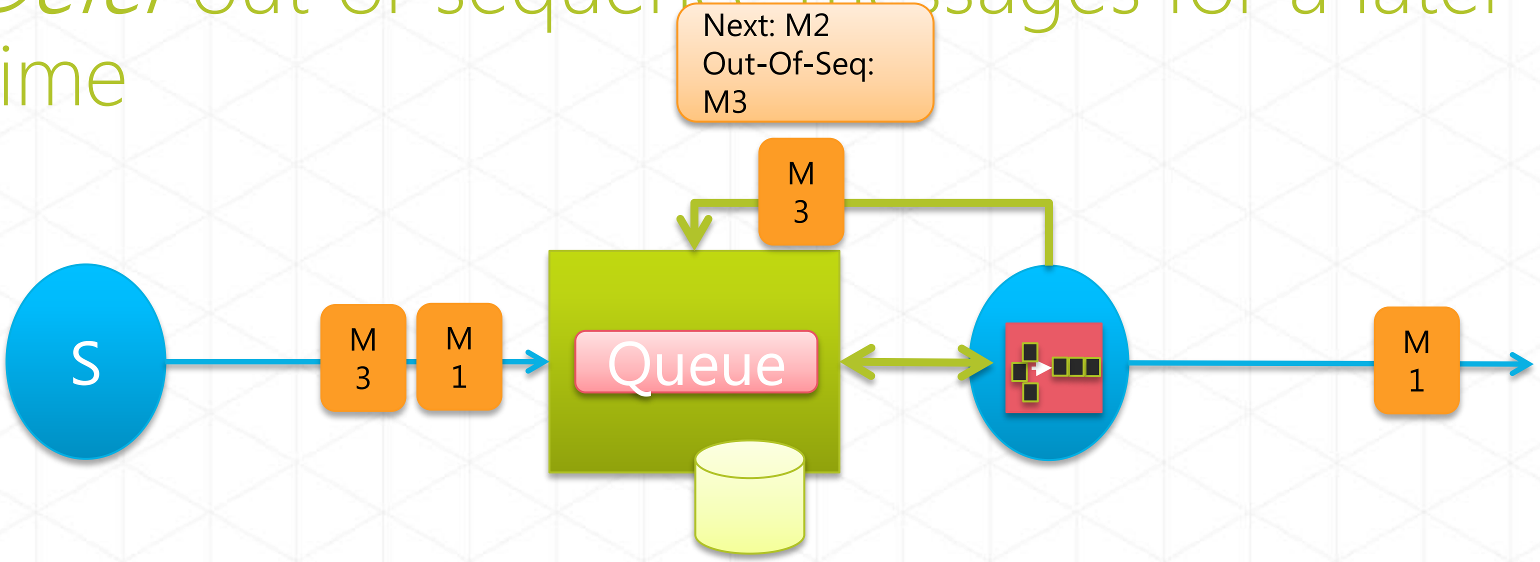
# Implementation

- Correlate messages with *sessions ID.* Identify sequence with a sequence ID property
- Use *session state* to store out-of-sequence messages
- *Defer* out-of-sequence messages for a later time



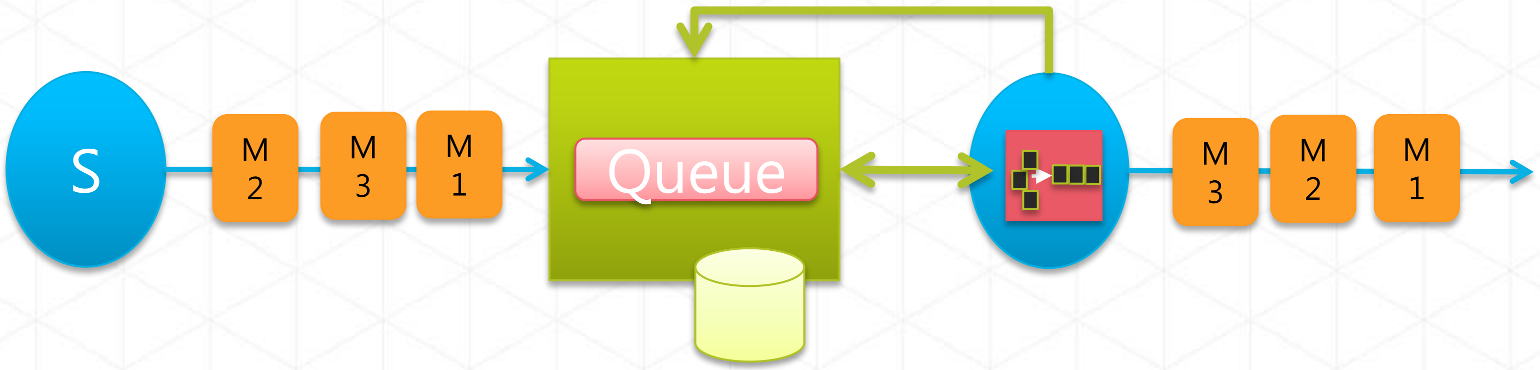
# Implementation

- Correlate messages with *sessions ID.* Identify sequence with a sequence ID property
- Use *session state* to store out-of-sequence messages
- *Defer* out-of-sequence messages for a later time



# Implementation

- Correlate messages with *sessions ID.* Identify sequence with a sequence ID property
- Use *session state* to store out-of-sequence messages
- *Defer* out-of-sequence messages for a later time

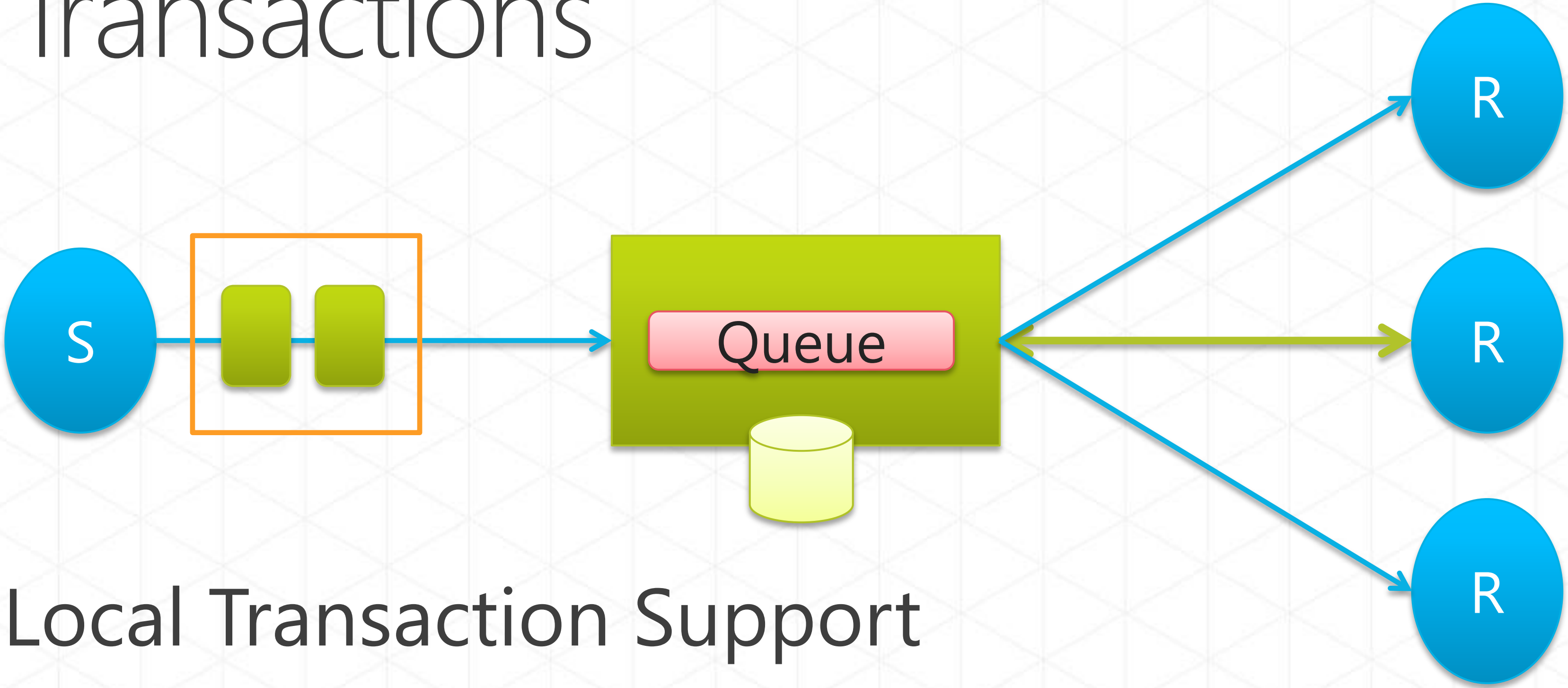


# Deferring messages

```
 if (sessionState.GetNextSequenceId() != messageId)
                {
                    Console.WriteLine("Defering message: Category {0}, Message sequence
{1}",                                session.SessionId, messageId);
                    // Deferring the message, and setting sessions state.
                    // Note: Use transaction scope to ensure consistency
                    message.Defer();
                    sessionState.AddOutOfSequenceMessage(messageId,
message.SequenceNumber);
                    SetState(session, sessionState);
                }
…

 while (sessionState.GetNextOutOfSequenceMessage() != -1)
                {
                    //Call back defered messages
                    Console.WriteLine("Calling back for deferred message: sequence {0}",
                                          sessionState.GetNextSequenceId());
                    receivedMessage =
receiver.Receive(sessionState.GetNextOutOfSequenceMessage());
                    ProcessMessage(receivedMessage, ref sessionState, receiver);
                }
```

## Advanced Features

# Transactions

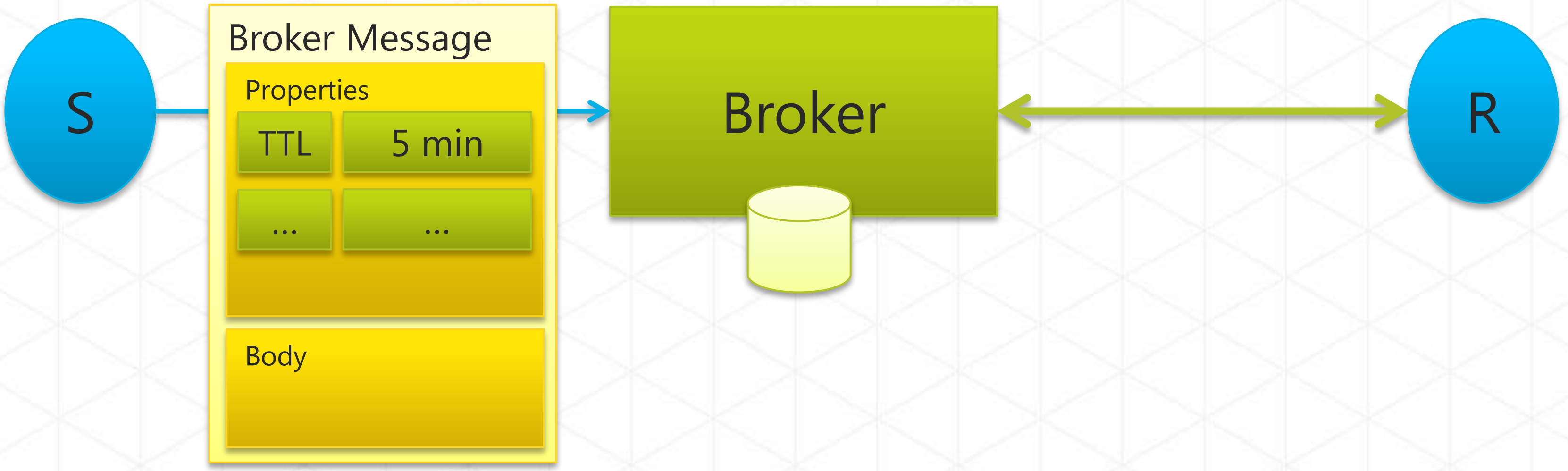


- Local Transaction Support
  - Create message batches that must only be sent together and are not sent in case of a transfer failure
  - Enable transactional operations on a single entity, e.g. receiving a message and deleting a rule from a subscription or store session state
- No distributed Tx support

# Transactions

```
using (TransactionScope scope = new
TransactionScope())
{
    sender.Send(msg1);
    sender.Send(msg2);

    scope.Complete();
}
```

# Time To Live

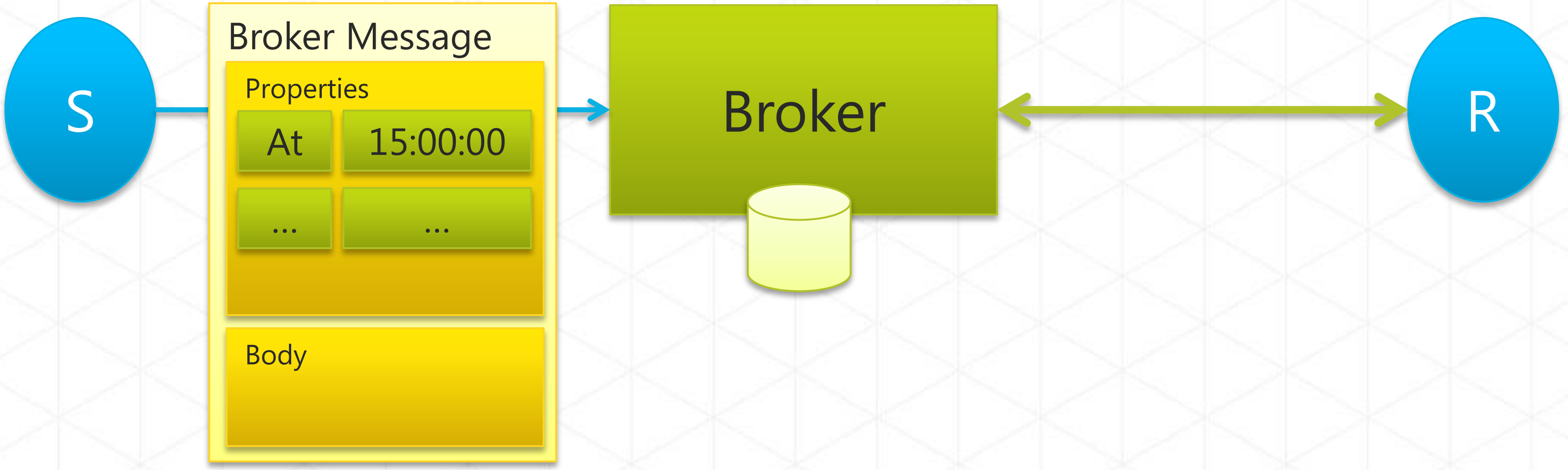


- Messages disappear once TTL expires
- TTL protects against stale information, especially against clogging auxiliary queues/topic with outdated info
  - Diagnostics, audit, errors, update notifications, statistics

## Sending Messages (TTL)

```
var message = new BrokeredMessage(msgBody)
              {
                MessageId = msgId,
                TimeToLive =
TimeSpan.FromMinutes(1)
              };
```

# Scheduling



- Scheduled messages appear at a certain point in time
- Very nice way to implement a simple distributed timer/scheduler.

## Sending Messages (Scheduled)

```
 var message = new BrokeredMessage(msgBody)
       {
            MessageId = msgId,
            ScheduledEnqueueTimeUtc =
DateTime.UtcNow.AddHours(2)
       };
```

## Dead-Lettering

- Allows safely discarding messages that cannot be processed for any reason and require some form of manual intervention.
- Discarded messages are available in the 'dead-letter queue'

```
receivedMessage.DeadLetter(
   "UnableToProcess",
   "Unrecoverable exception while processing");

QueueClient deadLetterClient =
    messagingFactory.CreateQueueClient(
         QueueClient.FormatDeadLetterPath(queueClient.Path),
         ReceiveMode.ReceiveAndDelete);
```

# Duplicate Detection

- Automatically detects duplicates (using the message-id) on the Service Bus server side.
- Eliminates doubt on whether a message has already been sent in case of disconnects/retries

```
namespaceManager.CreateQueue(
    new QueueDescription(queueName)
    {
        RequiresDuplicateDetection = true,
        DuplicateDetectionHistoryTimeWindow =
                           TimeSpan.FromHours(1)
    });
```

## Prefetch

- Optimized network behavior for high-throughput scenarios
- Fetches 'n' messages into the client even if the client hasn't yet explicitly called 'Receive'
- May cause load imbalance and, potentially, message loss or hoarding messages with expired locks

```
QueueClient queueClient =

messagingFactory.CreateQueueClient(Program.QueueName,

ReceiveMode.PeekLock);
queueClient.PrefetchCount = 50;
```

## Resources

- MSDN Docs: http://msdn.microsoft.com/sb
- SDK: http://windowsazure.com
- Samples – http://servicebus.codeplex.com
- Blog: http://blogs.msdn.com/windowsazure
  - Abhishek Lal: http://abhishekrlal.wordpress.com/
  - Clemens Vasters: http://blogs.msdn.com/clemensv/
  - Will Perry: http://blogs.msdn.com/willpe/

Microsoft®