

Enterprise Development with NServiceBus

Authored by: Udi Dahan

Topics Covered

- ◆ NServiceBus Basics
- ◆ Architectural Implications
- ◆ Business Processes
- ◆ Scalability & Monitoring
- ◆ Designing distributed systems

Course Programme

◆ Day 1

- ◆ One-way messaging
- ◆ Queues
- ◆ Request / Response
- ◆ Exceptions & faults
- ◆ Web Apps
- ◆ WS Integration

◆ Day 2

- ◆ SOA Intro
- ◆ SOA & Events
- ◆ Publish / Subscribe
- ◆ CQRS

Course Programme

- ◆ Day 3

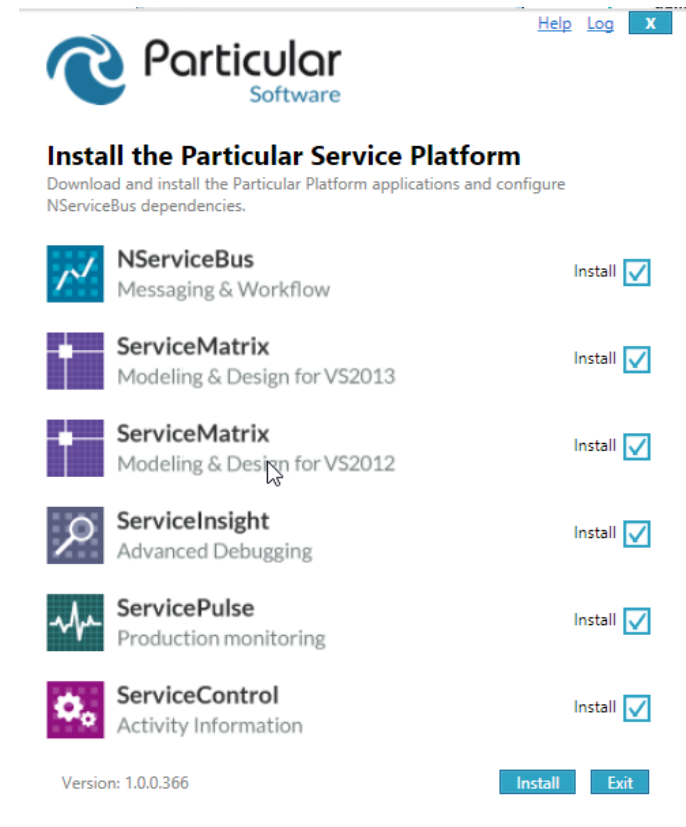
- ◆ Sagas
- ◆ Timeouts
- ◆ Unit testing

- ◆ Day 4

- ◆ Scaling & Monitoring
- ◆ Multi Site operations
- ◆ Group exercise

Hello Distributed World

- ◆ Downloading NServiceBus
- ◆ Using NServiceBus with Visual Studio
- ◆ NuGet
- ◆ Logging

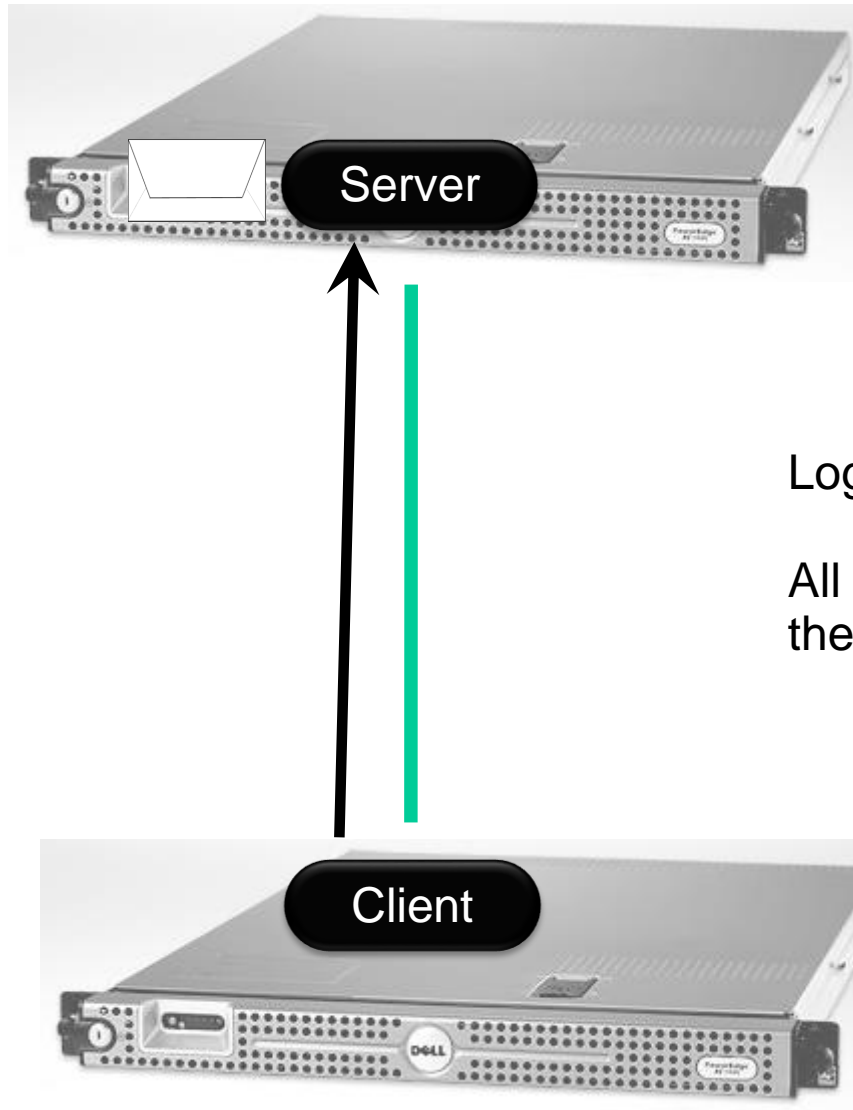


Exercises 1, 2

- ◆ Exercise 1:
 - ◆ Hello Distributed World
 - ◆ Getting ready
 - ◆ Hosting NServiceBus
- ◆ Exercise 2:
 - ◆ Logging

One-Way Messaging

Messaging & Queues



Logically one-way:

All information transferred in the message

How to: Define a message

```
public class MyMessage : IMessage { }
```

Or:

```
public interface IMyMessage : IMessage { }
```

Or:

```
public interface IMyMessage { }
```

```
configuration.Conventions()  
    .DefiningMessagesAs(t=>MyOwnConvention(t))
```

- ◆ Add properties like a regular class/interface
- ◆ **Keep messages in their own assembly/project**
- ◆ **Use conventions whenever possible**

How to: Instantiate a message

```
var m = new MyMessage();
```

How to: Send a message

```
Bus.Send(messageObject);
```

- ◆ Can instantiate and send together
 - ◆ Useful for interfaces:

```
Bus.Send<IMyMessage>(m =>  
{  
    m.Prop1 = v1; m.Prop2 = v2;  
});
```

How to: specify destination

`Bus.Send(string destination, object msg);`

- ◆ Requires that application manages routing
- ◆ Configure destination for message type:
 - ◆ In `<UnicastBusConfig>`, under `<MessageEndpointMappings>`

```
<add Assembly="assembly" endpoint="destination">
```

Or:

```
<add Assembly="asm" Namespace="ns" endpoint="dest">
```

```
<add Assembly="asm" Type="fqn" endpoint="dest">
```

How to: specify destination

- ◆ QueueName@ServerName
- ◆ Or
- ◆ Just QueueName for the same machine

How to: handle a message

- Write a class that implements `IHandleMessages<T>` where T is message type

```
public class H1 : IHandleMessages<MyMessage>
{
    public void Handle(MyMessage message)
    {
    }
}
```

Exercises 3 - 6

- ◆ Exercise 3:
 - ◆ Unobtrusive mode, message conventions
- ◆ Exercise 4:
 - ◆ Customization of serializer
- ◆ Exercise 5:
 - ◆ Routing
- ◆ Exercise 6:
 - ◆ Message processing

Dangers of Store & Forward

- ◆ If target is offline for an extended period of time messages can fill up the disk
 - ◆ Can cause a server to crash
- ◆ Especially problematic in B2B integration
 - ◆ Example:
 - ◆ 1 MB / message, 100 messages/sec = 6GB / minute
- ◆ Solution – discard messages after a while

How to: Specify time to discard

- ◆ Use the `TimeToBeReceivedAttribute`:

```
[TimeToBeReceived("00:01:00")] // one minute  
public class MyMessage : IMessage { }
```

Unobtrusive:

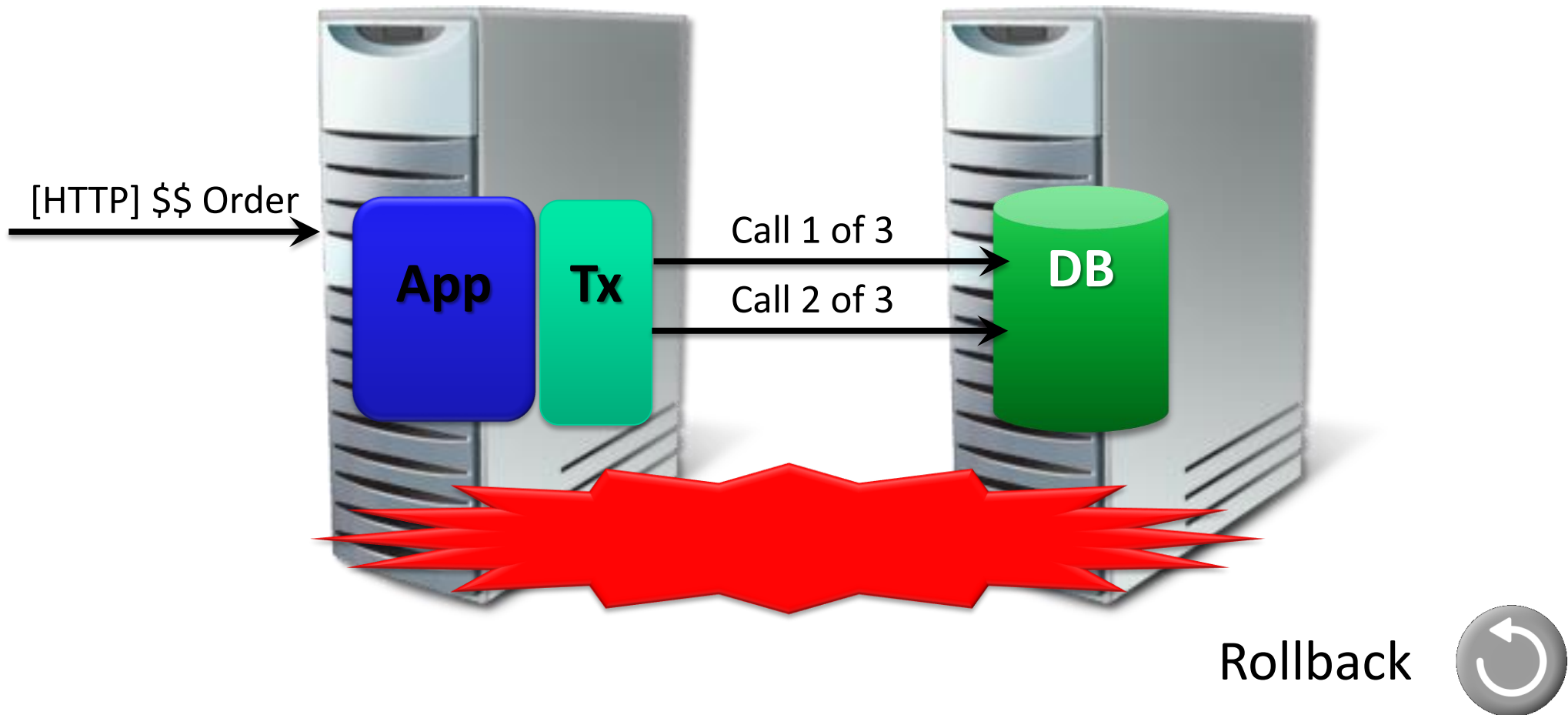
```
configuration.Conventions()  
    .DefiningTimeToBeReceivedAs(...)
```

Exercise 7

- ◆ Conventions configuration for expiry
- ◆ Centralized conventions

Fault Tolerance

Fault Tolerance

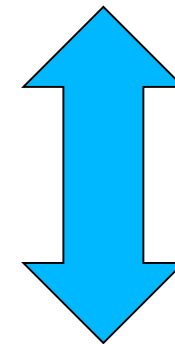


Where's the order!?

Exceptions

- ◆ Reasons exceptions happen:

- ◆ Deadlock in the database
- ◆ Database is down
- ◆ Message deserialization fails



Transient

Permanent

- ◆ Retrying can resolve transient exceptions

```
<TransportConfig MaxRetries="5" />
```

- ◆ Second Level Retries (SLR)

- ◆ Messages that always fail are moved to an error queue

```
<MessageForwardingInCaseOfFaultConfig ErrorQueue="error"/>
```

Deserialization exception: message moved to error queue right away

Consistency

```
configuration.Transactions()  
    Enable();
```

OR

```
IConfigureThisEndpoint, AsA_Server
```

Exercise 8

- ◆ Exceptions
- ◆ First-Level and Second-Level Retries

Authorization, Impersonation, Auditing

Headers

```
Bus.OutgoingHeaders["user"] = "udi";
```

- ◆ Affects all messages sent

```
Bus.SetHeader(msg, "key", "value");
```

- ◆ Headers attached only to specific message
- ◆ For server processing, headers cleared when a new message arrives

Message Processing Pipeline

- ◆ Multiple handlers can process the same message
 - ◆ Useful for having validation handlers separate from handlers that perform business logic
- ◆ Implement `IHandleMessages<IMessage>` or `IHandleMessages<object>` to handle all types of messages
 - ◆ Useful for things like authentication and authorization

Set Pipeline Order

configuration

```
.LoadMessageHandlers(  
    First<A>.Then<B>().AndThen<C>() );
```

OR

Inherit from `ISpecifyMessageHandlerOrdering`

```
order.SpecifyFirst<A>();
```

OR

```
order.Specify(First<A>.Then<B>().AndThen<C>());
```

How to: reference IBus in handler

```
public class H1 : IHandleMessages<MyMessage>
{
    public IBus Bus { get; set; }

    public void Handle(MyMessage message)
    {
        // use Bus to do something
    }
}
```

How to: stop the pipeline

```
public class Auth : IHandleMessages<IMessage>
{
    public IBus Bus { get; set; }
    public void Handle(IMessage message)
    {
        if ( /* some logic */ )
            Bus.DoNotContinueDispatchingCurrentMessageToHandlers();
    }
}
```

How to: get additional meta data

```
public class H1 : IHandleMessages<MyMessage>
{
    public IBus Bus { get; set; }
    public void Handle(MyMessage message)
    {
        Console.WriteLine("Received " +
            Bus.CurrentMessageContext.Id + " from " +
            Bus.CurrentMessageContext.ReplyToAddress);
        // Bus.CurrentMessageContext.Headers (dictionary of strings)
    }
}
```

Message Auditing

- Can forward messages received to another queue
- In config:

```
<AuditConfig
```

```
    QueueName="AuditQueue@AuditServer">
```

Exercise 9

- ◆ Authorization
- ◆ Message Headers

Infrastructure Extension

Dependency Injection

- ◆ NServiceBus makes heavy use of IoC
- ◆ Uses a container, and exposes its own registration API:

```
configuration.RegisterComponents(c =>  
    c.ConfigureComponent<T>(DependencyLifecycle lifecycle)  
        .ConfigureProperty(o =>  
            o.Property, someValue));
```

Dependency Injection

- ◆ Locator pattern is no longer supported

~~Configure.Instance.Builder.Build<T>();~~

- ◆ Supports constructor or property setter injection
- ◆ Constructor or setter injection is usually better

Unit of work management

- ◆ Called once per message
- ◆ Called before and after all handlers
 - ◆ A way to perform things like session management
- ◆ Implement `IManageUnitsOfWork`
- ◆ Needs to be registered in the container explicitly
 - ◆ Should be registered as instance per call

Message Mutators

- ◆ Implement `IMessageMutator` to mutate individual messages
 - ◆ Useful for things like validation, encryption
- ◆ Implement `IMutateTransportMessages` to mutate transport messages
 - ◆ Useful for things like compression
- ◆ Both on incoming and outgoing messages
- ◆ Needs to be registered in the container explicitly

Exercise 10

- ◆ Dependency Injection

Encryption

Encryption

- ◆ Use the `WireEncryptedString` type for the message properties you want encrypted

```
public class MyMessage
{
    public string UnencryptedProperty { get; set; }
    public WireEncryptedString Encrypted { get; set; }
}
```

- ◆ Get the value using the `.Value` property
- ◆ Unobtrusive:
 - ◆ `configuration.Conventions().DefiningEncryptedPropertiesAs(...)`

Encryption Service

- ◆ Pluggable IEncryptionService
- ◆ Default implementation is the RijndaelEncryptionService
 - ◆ Based on a shared key
- ◆ Enabled by calling
`configuration.RijndaelEncryptionService();`

Encryption and Topology

- ◆ Same-site / LAN
 - ◆ Full message encryption unnecessary
- ◆ Cross-site / WAN
 - ◆ Full message encryption may be required
 - ◆ Configure the Gateway to communicate using HTTPS

Exercise 11

- ◆ Encryption
- ◆ Custom convention for encrypted properties

Adjusting runtime behaviour

Profiles

- ◆ Change endpoint behavior without recompiling
- ◆ Built-in profiles
 - ◆ Environment profiles:
 - ◆ Lite | Integration | Production
 - ◆ Feature profiles (moved out of the core):
 - ◆ MultiSite | Distributor | Master | etc
- ◆ Activated by the command line:

`.\NServiceBus.Host.exe nservicebus.production nservicebus.multisite`

Defining your own profiles

- ◆ Defined by implementing `IProfile`
- ◆ Add behavior by implementing:
 - ◆ `IHandleProfile<T>`
- ◆ Can also add behavior to existing profiles
 - ◆ Implement `IHandleProfile<ExistingProfile>`

Convention over Configuration

Endpoint name

- ◆ Is the base for all the conventions
- ◆ Must be logically unique
- ◆ Conventions
 - ◆ Endpoint input queue
 - ◆ TimeoutManager input queue
 - ◆ Name of Raven database
 - ◆ Distributor input, control and storage queues
 - ◆ Subscription storage
 - ◆ Gateway input queue

Defining the endpoint name

- ◆ Ways to define the name
 - ◆ Namespace of the endpoint configuration class
 - ◆ `[EndpointName("X")]` attribute
 - ◆ `/serviceName:"X" | /endpointName:"X"`
- ◆ Or define your own convention
 - ◆ `configuration.EndpointName("X");`

Customization

Extending the Fluent DSL

- Write an extension method for the NServiceBus.Configure class:

```
public static class MyExtensions
{
    public static BusConfiguration
        Something(this BusConfiguration c)
    {
        // register something in the container
        c.Configurer.RegisterComponents(
            r => r.ConfigureComponent<SaySomething>(...);
        }
    }
}
```

Accessing Configuration Sections

```
public static class MyExtensions
{
    public static BusConfiguration
        Something(this BusConfiguration c)
    {
        var section = c.GetSettings()
            .GetConfigSection<MySection>();
        configuration.RegisterComponents(r => r
            .ConfigureComponent<SaySomething>(...)
            .ConfigureProperty(p => p.Prop, section.Value));
    }
}
```

Diverting App.config

- ◆ Only holds for config sections accessed via `configuration.GetSettings().GetConfigSection<T>configuration`

```
.CustomConfigurationSource(new MySource());
```

```
public class MySource : IConfigurationSource
{
    public T GetConfiguration<T>() where T : class, new()
    {
        // override everything
    }
}
```

Diverting individual sections

- ◆ Use `IProvideConfiguration` to control individual sections

```
class MySource : IProvideConfiguration<SomeSection>
{
    public SomeSection GetConfiguration()
    {
        return new SomeSection
        {
            Setting = ""
        };
    }
}
```

Russian dolls

- ◆ Pipeline of NServiceBus can be bent to your will, implement `IBehavior<IncomingContext|OutgoingContext>`

```
public class IncomingBehavior : IBehavior<IncomingContext>
{
    public void
        Invoke(IncomingContext context, Action next)
    {
        next();
    }
}
```

Russian dolls

- ◆ Register (before or after) the behavior in the pipeline with a RegisterStep

```
public class IncomingBehaviorStep : RegisterStep
{
    public IncomingBehaviorStep()
        : base("Id", typeof (IncomingBehavior), "Description")
    {
        InsertBefore(WellKnownStep.ExecuteUnitOfWork);
    }
}
```


Russian dolls

- ◆ Register the step in the container

```
public class IncomingBehaviorStepRegistration
    : INeedInitialization
{
    public void Customize(BusConfiguration configuration)
    {
        configuration.Pipeline
            .Register<IncomingBehaviorStep>();
    }
}
```

Exercise 12

- ◆ Override Configuration

Custom Hosting

Custom Hosting

- ◆ Can host NServiceBus in your own process
 - ◆ References required:
 - ◆ NServiceBus.dll
 - ◆ NServiceBus.Core.dll
 - ◆ NuGet: `install-package NServiceBus`
 - ◆ Don't need to reference NServiceBus.Host.exe

Hosting in a Web Application

- ◆ Set up the bus in the Global.asax:
 - ◆ Using the “Application_Start” built-in method;
- ◆ Set up the bus in a separate assembly:
 - ◆ Defining a public static class decorated with the “PreApplicationStartMethod”:
 - ◆ The attribute is defined in “System.Web” assembly;
 - ◆ Your class will be called at application startup time by the asp.net infrastructure;

Fluent Initialization DSL

```
var configuration = new BusConfiguration();  
configuration.UsePersistence<RavenDBPersistence>();  
configuration.UseTransport<MsmqTransport>();  
Bus.Create(configuration).Start();
```

Send only endpoints

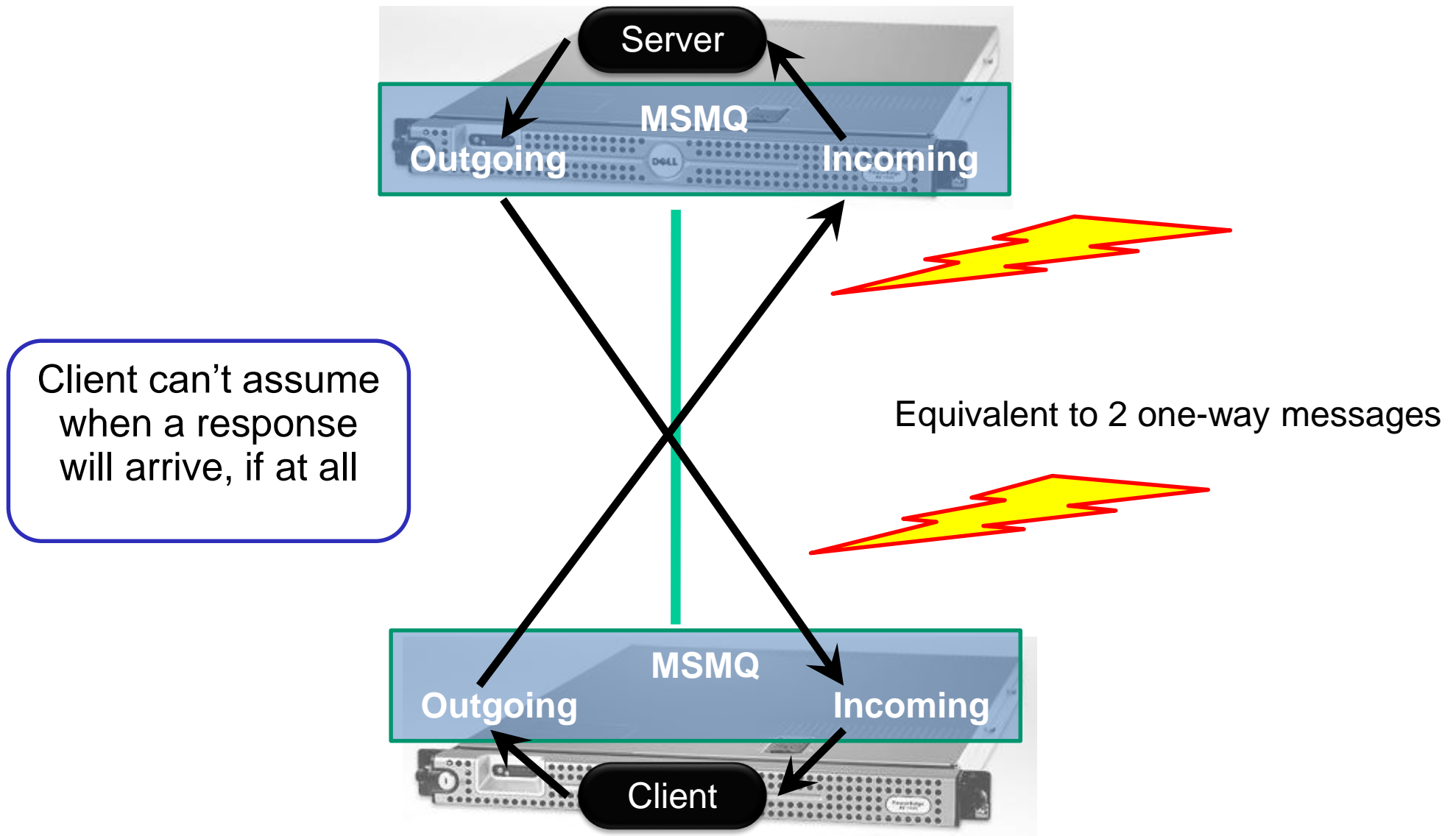
- ◆ Useful for endpoints that only sends messages
- ◆ Configured by calling
 `Bus.CreateSendOnly(configuration)`
- ◆ Doesn't need settings for
 - ◆ Input queue
 - ◆ Error queue
 - ◆ Worker threads, Retries

Exercise 13

- Web Application Hosting

Request / Response

Request / Response



Request / Response

- ◆ Message is sent from the server to the client's queue
 - ◆ If the client is offline, message sits in the server machine's outgoing queue
- ◆ Client is not blocked until response arrives
- ◆ If two requests were sent, responses may arrive out of order

Warning! This is NOT RPC

- ◆ Do NOT try to implement regular request/response patterns on top of messaging
- ◆ The client should be designed so that it can continue operating if a response never comes

How to return data

- ◆ `Bus.Reply(messages);`
 - ◆ Remember: You don't know when this will arrive
- ◆ `Bus.Return(int errorCode);`
 - ◆ Embeds the `errorCode` in a control message
 - ◆ Prefer to define an enumeration:
 - ◆ `Bus.Return(ErrorCodes.NoSuchUser);`

Handling responses client-side

- ◆ Can handle like any regular message:
 - ◆ Have a class implement `IHandleMessages<Response>`
- ◆ Can register a callback when sending request:
 - ◆ `Bus.Send(request).Register(asyncCallback, state)`
 - ◆ Callback only fires on first response
 - ◆ Then is cleaned up to prevent memory leaks
 - ◆ Doesn't survive restarts – not suitable for server-side
 - ◆ Useful for feedback on a command (success/failure)

Client Startup

- ♦ Servers shouldn't throw away messages - ever
- ♦ Clients may not care what's in their queue at startup – preferring not to waste resources

```
configuration.PurgeOnStartup(true);
```

Exercise 14

- ◆ Full Duplex Request / Response

Unit Testing

Unit Testing Message Handlers

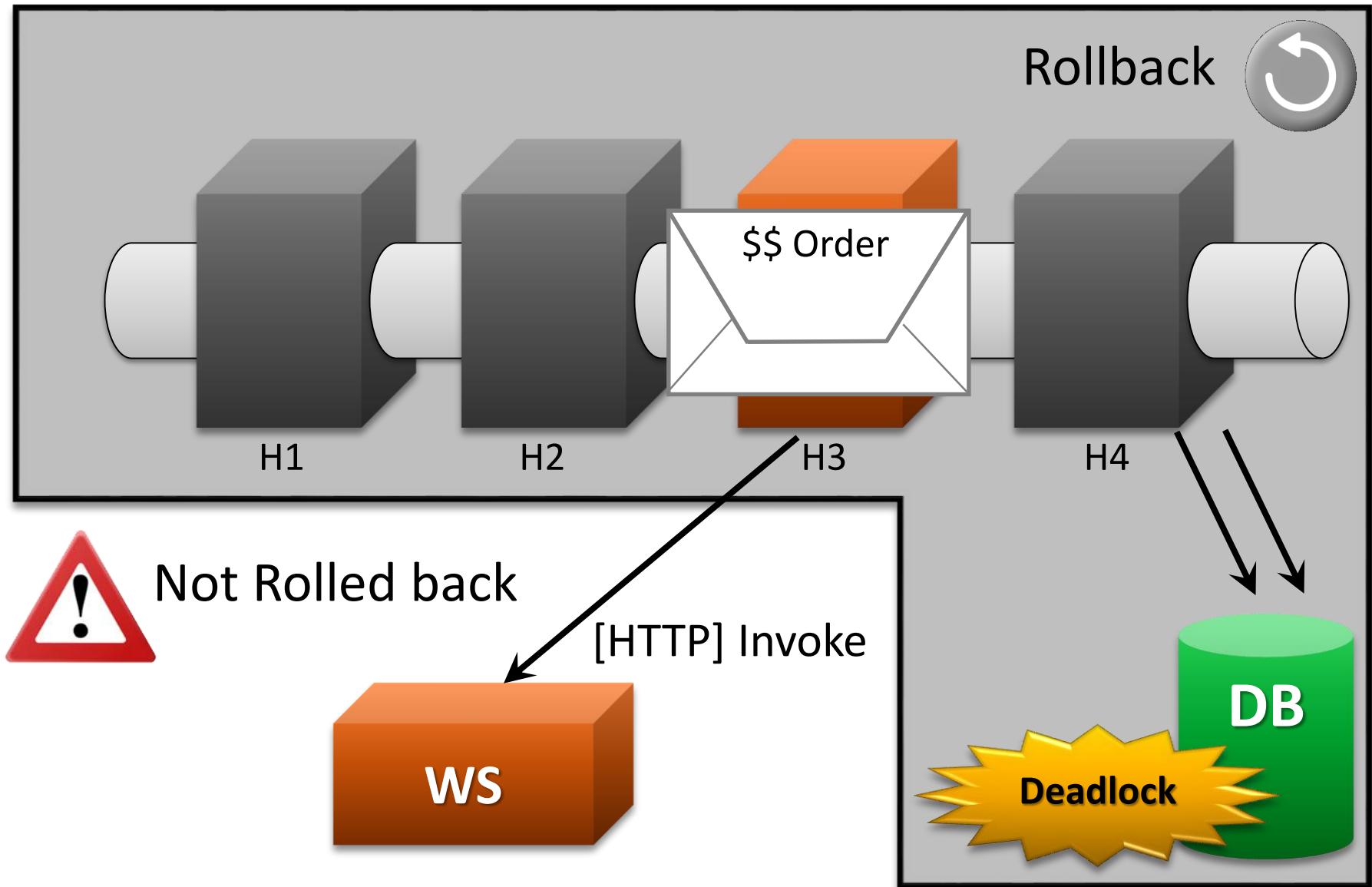
- ◆ NServiceBus.Testing.dll
- ◆ Provides the ability to set expectations around how message handlers handle messages
 - ◆ Expect: Send, Reply, Publish, etc...

Exercise 15

- ◆ Unit Test

Exceptions, Consistency, Integration

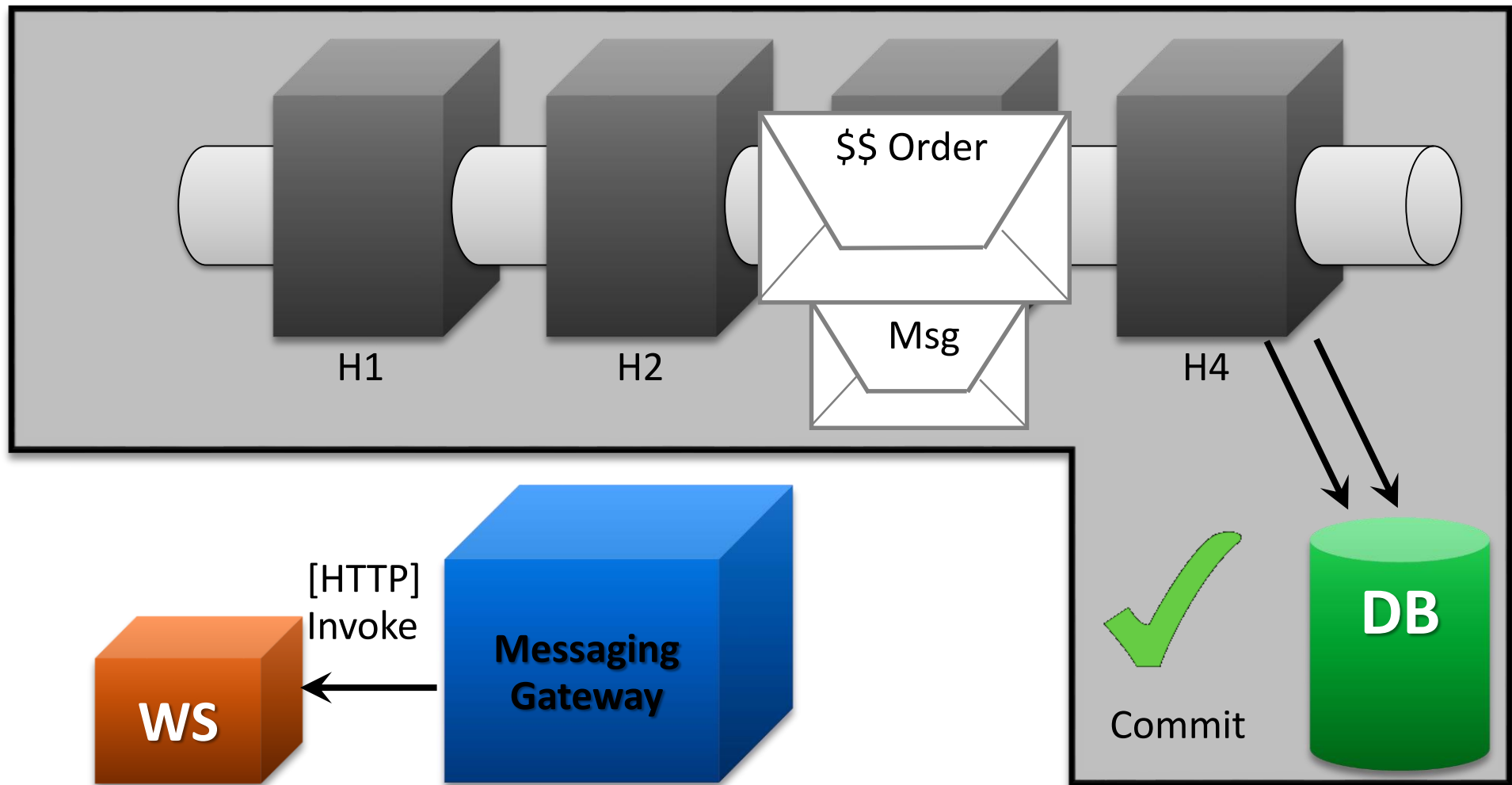
Invoking web services from handlers



Integrating messaging & WS

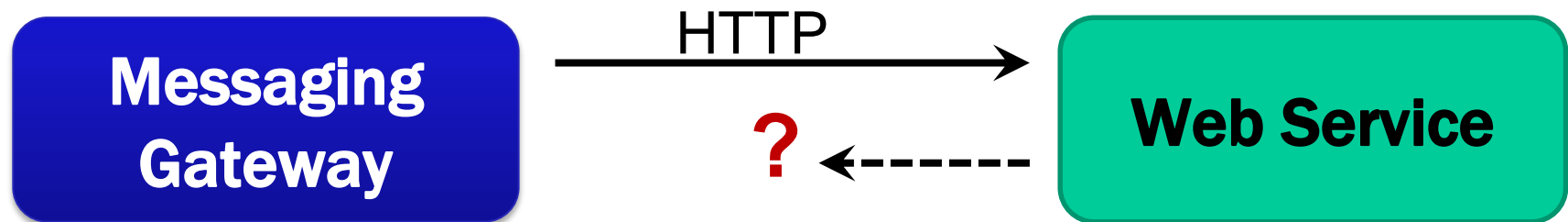
The right way

The message won't be sent if there's a failure



Messaging to WS Integration

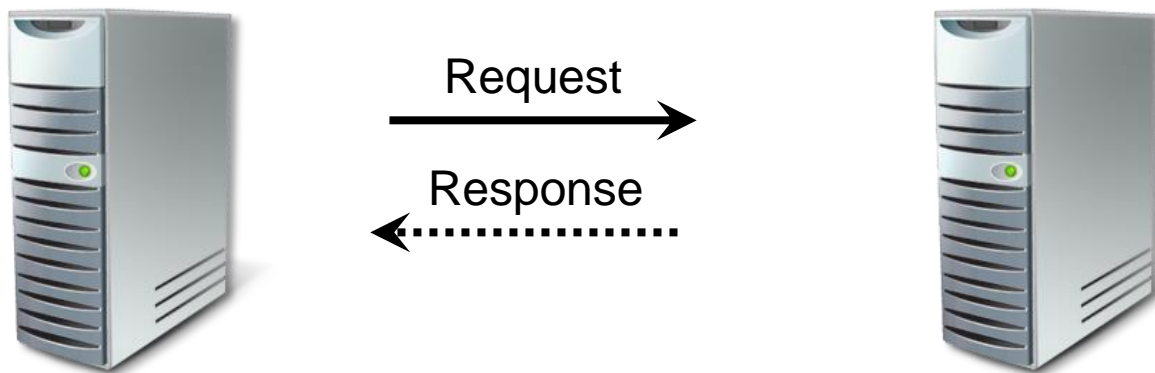
- When calling external web services from a messaging endpoint, if they're down, regular retry logic kicks in.



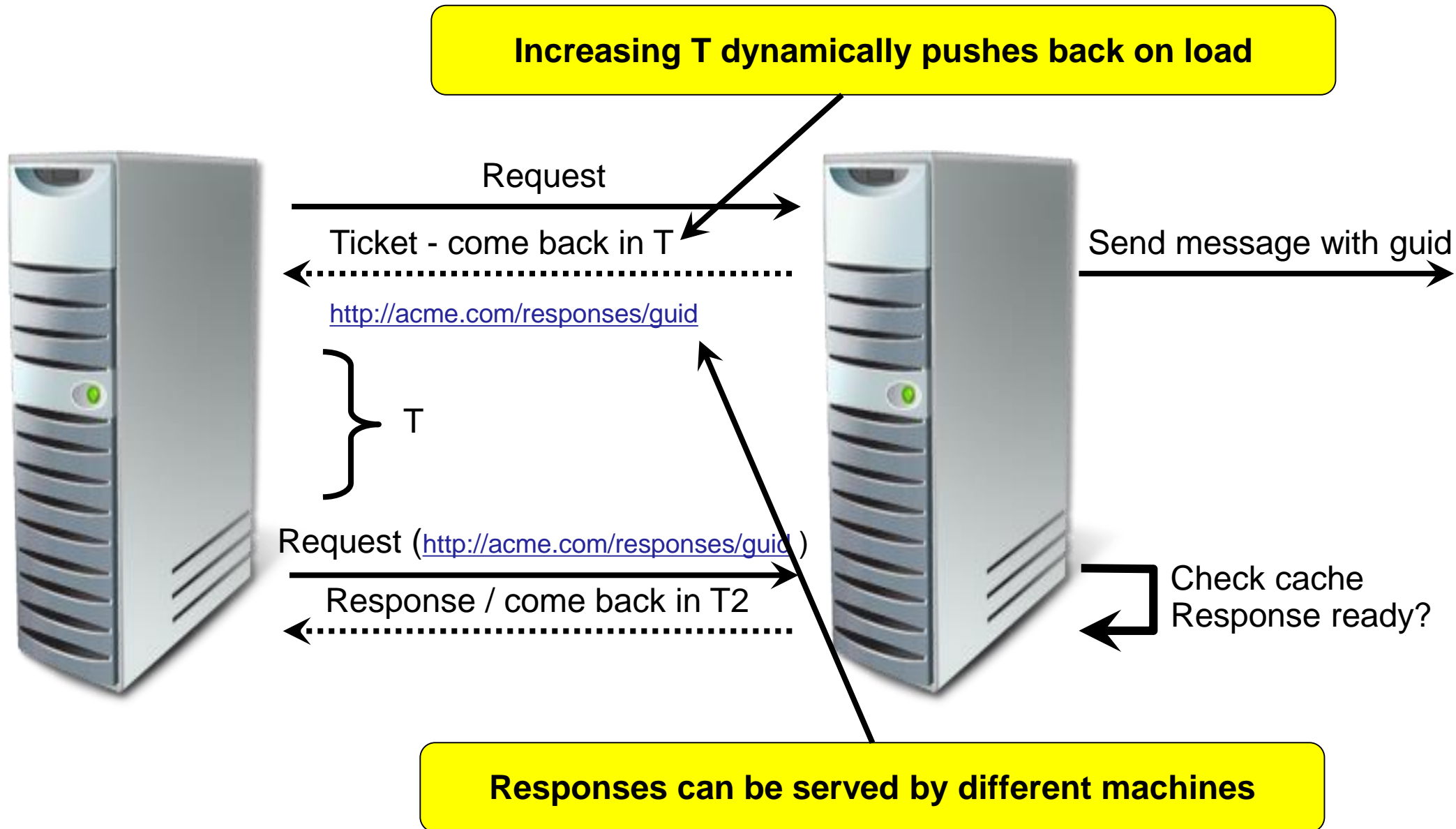
HTTP Timeout Exception & Retry

Web Services & Integration

- ◆ Request/response synchronous web service most broadly known
 - ◆ Makes sense to expose this kind of endpoint
 - ◆ Throttle it – not a scalable solution
 - ◆ Direct users to a different integration *protocol*



Web Integration Protocols

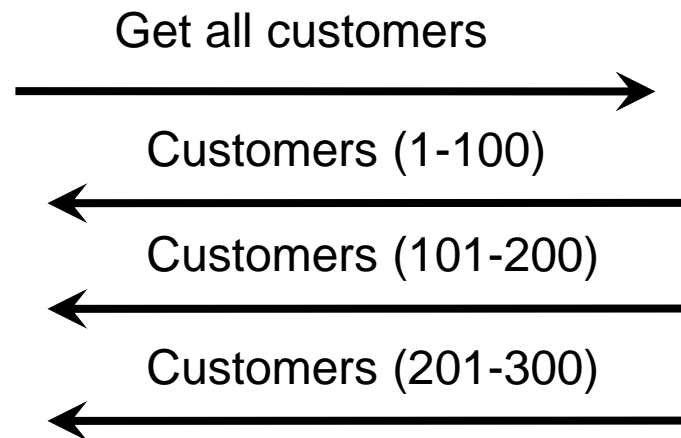


Exceptions & Request / Response

- ◆ As opposed to RPC, an exception doesn't return to the endpoint which sent the request
- ◆ If there's a problem with the system, users don't need to call the helpdesk – administrators can see that there's a problem from the error queue.
- ◆ Consider responding to the user via email

Multiple Responses

- ◆ Can call `Bus.Reply` multiple times
 - ◆ Useful for “streaming” back a lot of data



Better user experience

Caps server-side
memory use

- ◆ Can respond with multiple message types too
 - ◆ Each handler in the pipeline can reply differently

Streaming Responses and CQRS

- ◆ When the endpoint is transactional, messages are not sent until processing is complete
- ◆ Query endpoints do not require transactional fault-tolerance, and **MUST** not be transactional in order to stream back responses

Exercise 16, 17

- ◆ Exercise 16:
 - ◆ Multiple Responses
- ◆ Exercise 17:
 - ◆ Streaming

Administration & Monitoring

Installing your endpoint

- ◆ NServiceBus.Host.exe -install
 - ◆ Installs the host as a windows service
 - ◆ Add -sideBySide if you want to run in parallel while upgrading
 - ◆ Executes the installers
- ◆ NServiceBus.Host.exe -uninstall
 - ◆ Uninstalls the service
 - ◆ It recommended to install/uninstall on each deploy

Installers

- ◆ Used by NServiceBus to install runtime dependencies like queues, databases etc
- ◆ Implement `INeedToInstallSomething` to create your own installer
 - ◆ Used for endpoint related dependencies

Error Queues

- ◆ 1 error queue usually enough for the whole system
- ◆ Administrators should monitor the error queue
 - ◆ First indication that something is wrong
 - ◆ Like database going down
- ◆ Especially after installing a new version of the system side by side with the old version

Error Queues & Administration

- ◆ Deserialization exceptions can be caused by incompatible versions running side-by-side
- ◆ After fixing and redeploying, messages from error queue can be retried:
 - ◆ ServicePulse or ServiceInsight
 - ◆ Can return single or all messages

Monitoring

- ◆ Use performance counters exposed by the queuing system to monitor number of messages waiting in queue
- ◆ But various parts of the system process different loads, and have varying computing power

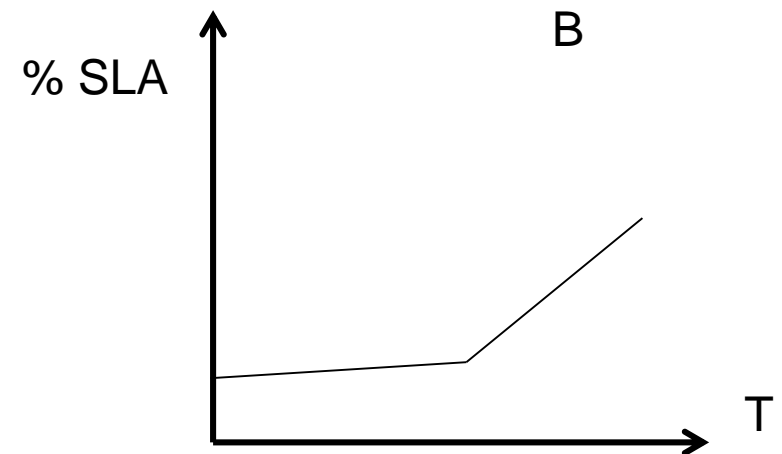
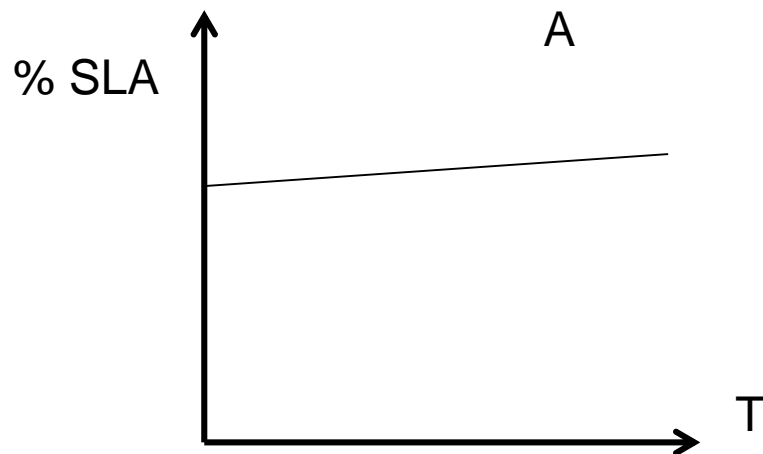


Time is the key

- ◆ Age of oldest message equalizes everything

Critical Time Performance Counter

- Can use WMI to pull into existing monitoring apps
- Measure against business-defined SLA

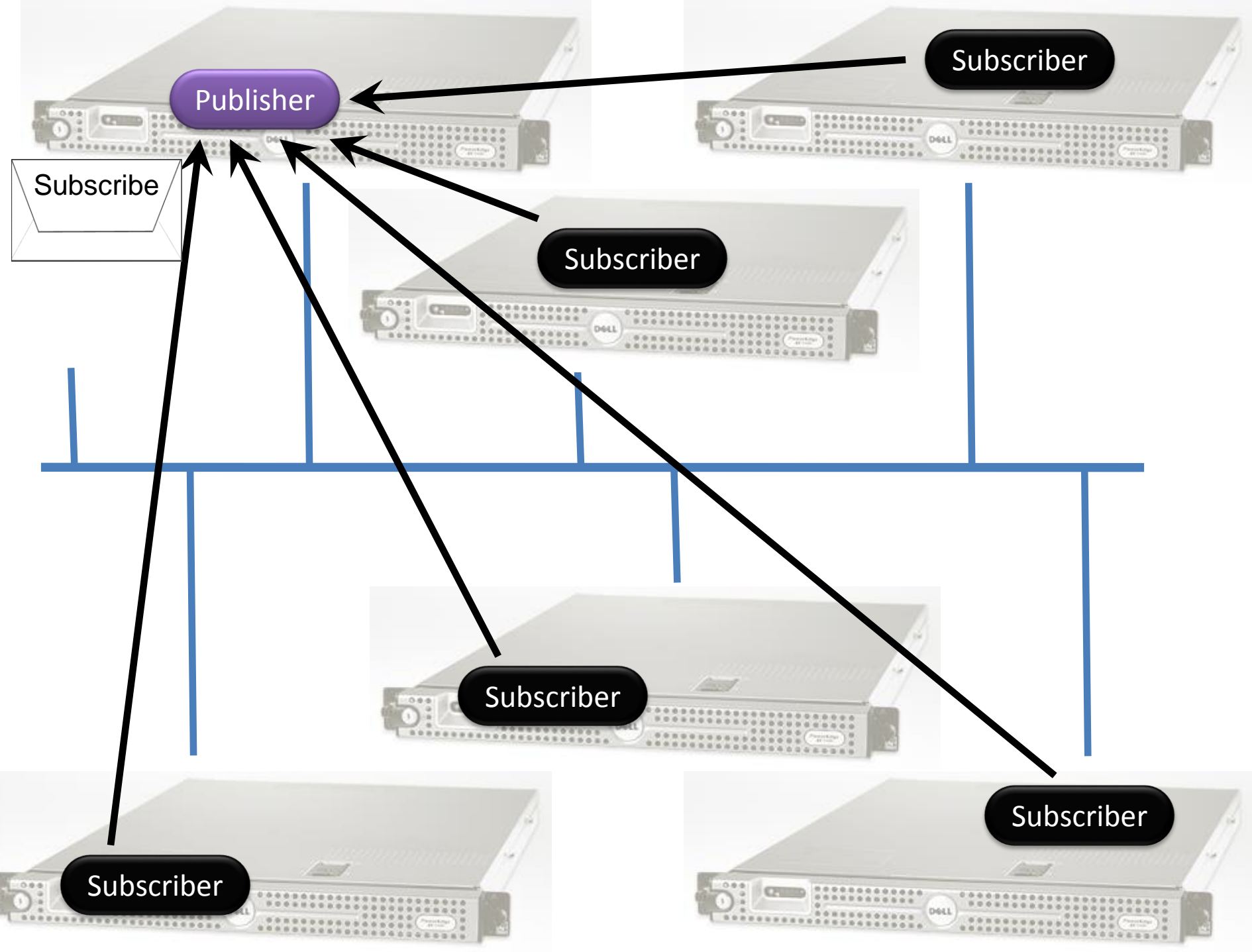


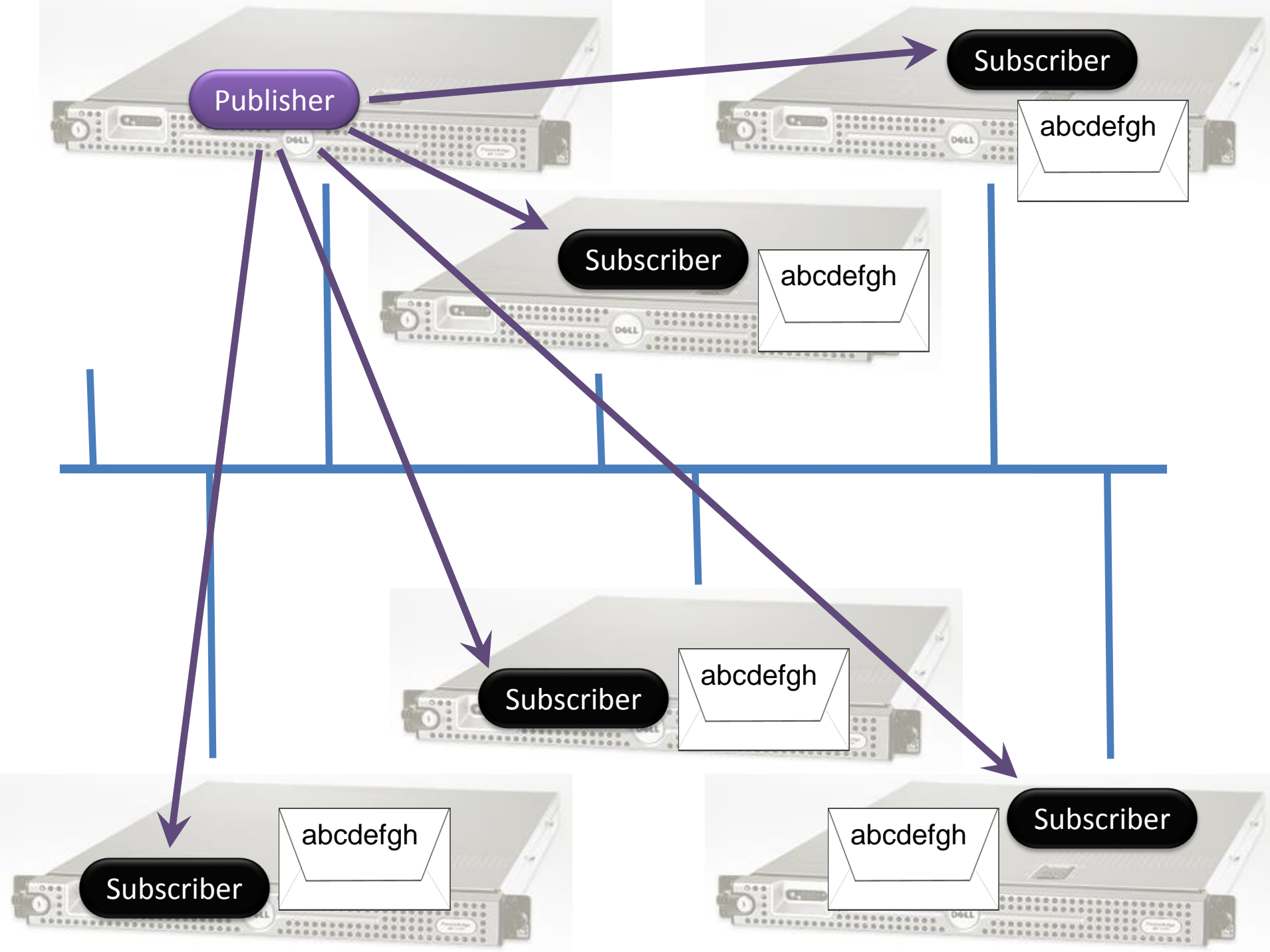
- Calculate “Time to exceed SLA”

Publish/Subscribe

Advertise / Subscribe / Publish

- ◆ Messages represent logical events
 - ◆ Technically, there's no difference.
- ◆ Publisher advertises these events
- ◆ Subscribers express interest in an event
- ◆ When a publisher publishes an event, a message arrives in the queues of subscribed parties





Advantages

- ◆ Extensible
 - ◆ Adding subscribers without bringing anything down
- ◆ Robust
 - ◆ publishers & subscribers can operate independently of each other
- ◆ Easy to patch, upgrade, and administer

Advertise

- ◆ NServiceBus is not involved in this step
- ◆ May be as simple as a wiki
 - ◆ Hosts the schema as well as where to go to subscribe
- ◆ Configure like regular messaging:
 - ◆ In <UnicastBusConfig>, under <MessageEndpointMappings>
 <add Assembly="schema assembly"
 endpoint="subscribe-here@Publisher">

Automatic Subscriptions

- ◆ Write a handler for the event
 - ◆ NServiceBus will contact the configured publisher at startup to subscribe to that message
 - ◆ Unless:
`configuration.DisableFeature<AutoSubscribe>();`

Manual Subscription Management

- ◆ `Bus.Subscribe<T>();`
- ◆ `Bus.Unsubscribe<T>();`
- ◆ Still requires a handler for the given message
- ◆ Consider unsubscribing clients when they shut down

Subscription Storage

- ◆ Publisher-side storage of who is interested in what
 - ◆ Can be stored in MSMQ (not scale-out friendly)
`configuration.UsePersistence<MsmqPersistence>();`
 - ◆ Configured:
`<MsmqSubscriptionStorageConfig Queue="" />`
 - ◆ Can be stored in a database
 - ◆ RavenDB is the default
`configuration.UsePersistence<RavenDBPersistence>();`
 - ◆ NHibernate is still supported but in a separate assembly
`NServicebus.Configure.With().DbSubscriptionStorage();`

Publishing Events

- ◆ Similar to sending a message:

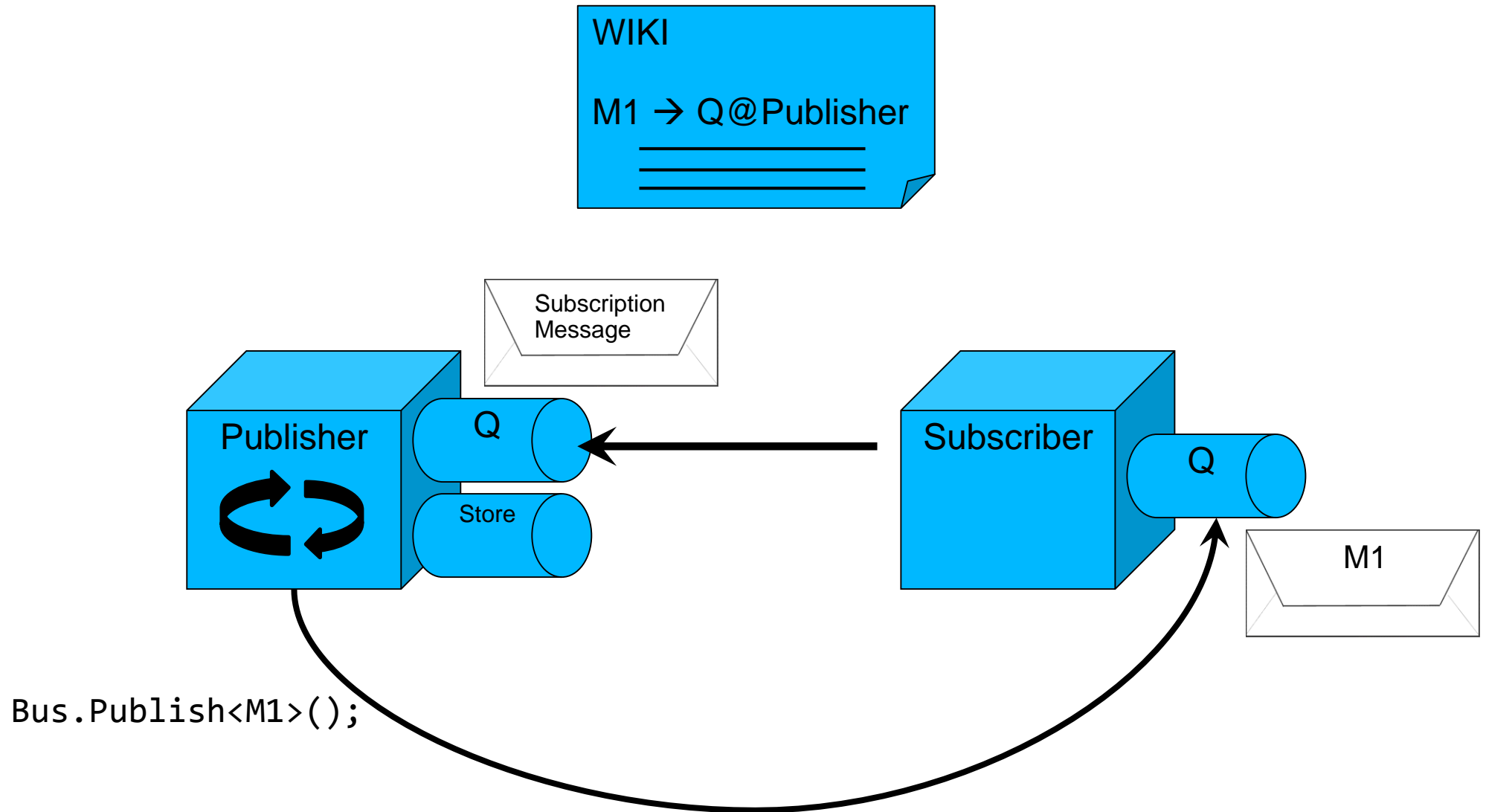
```
Bus.Publish(msg);
```

Or:

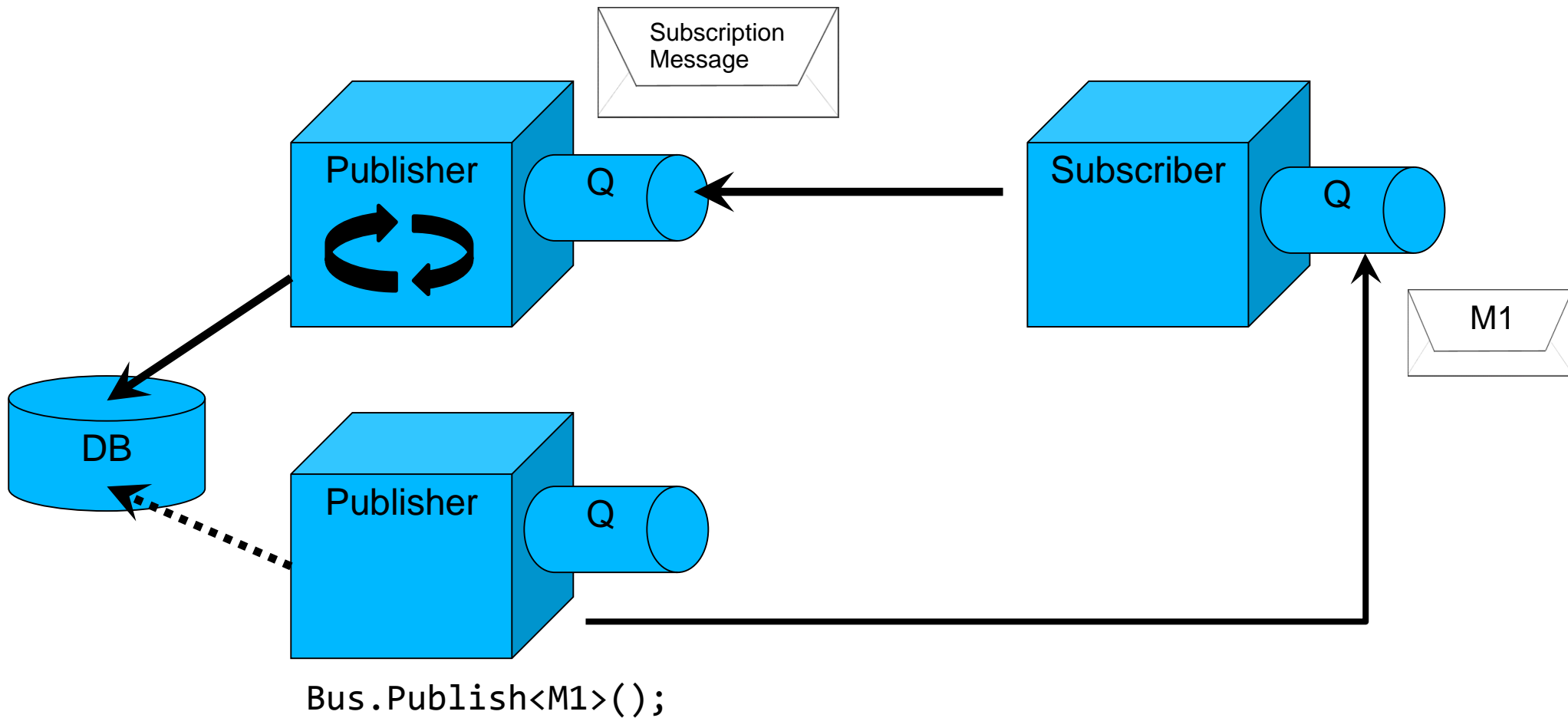
```
Bus.Publish<IMyEvent>( (m) =>  
    { m.Prop1 = val1; m.Prop2 = val2; } );
```

- Consider having your events inherit from `IEvent` or define convention `configuration.Conventions().DefiningEventsAs(t =>...)`

Advertise / Subscribe / Publish



Publisher Scale-Out



Exercise 18

- Publish / Subscribe

Web, Pub/Sub, Caching

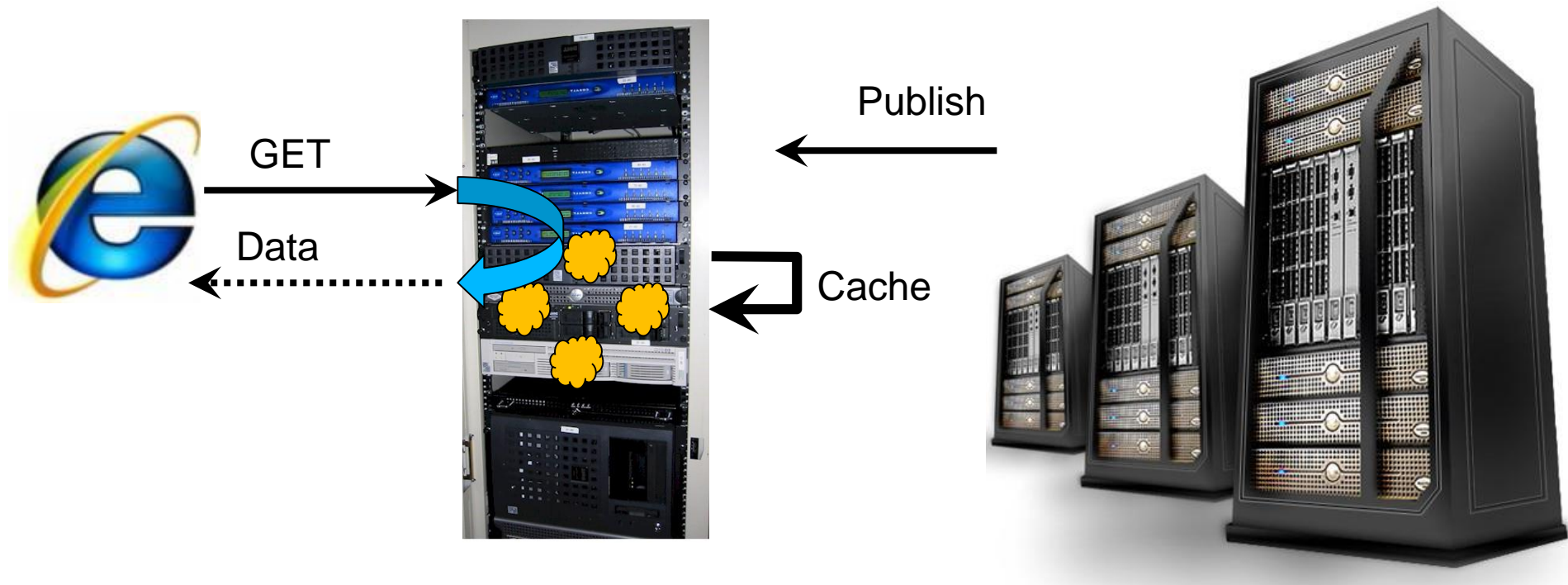
Web Apps

- ◆ HTTP request/response with browsers
- ◆ Should avoid request/response with backend
 - ◆ Easier said than done



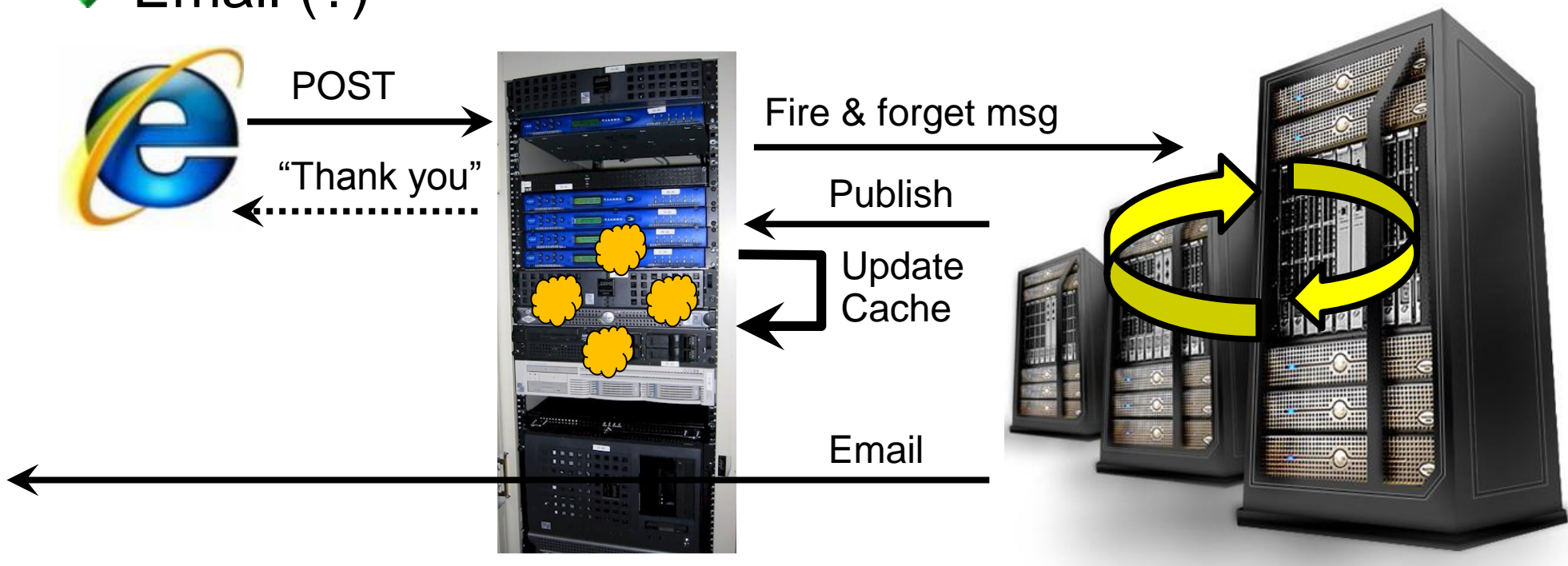
Web Apps, Pub/Sub, and Caching

- ◆ Web app front end subscribes to events
 - ◆ Caches data from events in the web tier
 - ◆ Serves data from the cache to browsers



Changing data

- ◆ User submits changes to data
 - ◆ Send message and show user “thank you” page
 - ◆ Successful processing causes publish/cache update
 - ◆ Email (?)

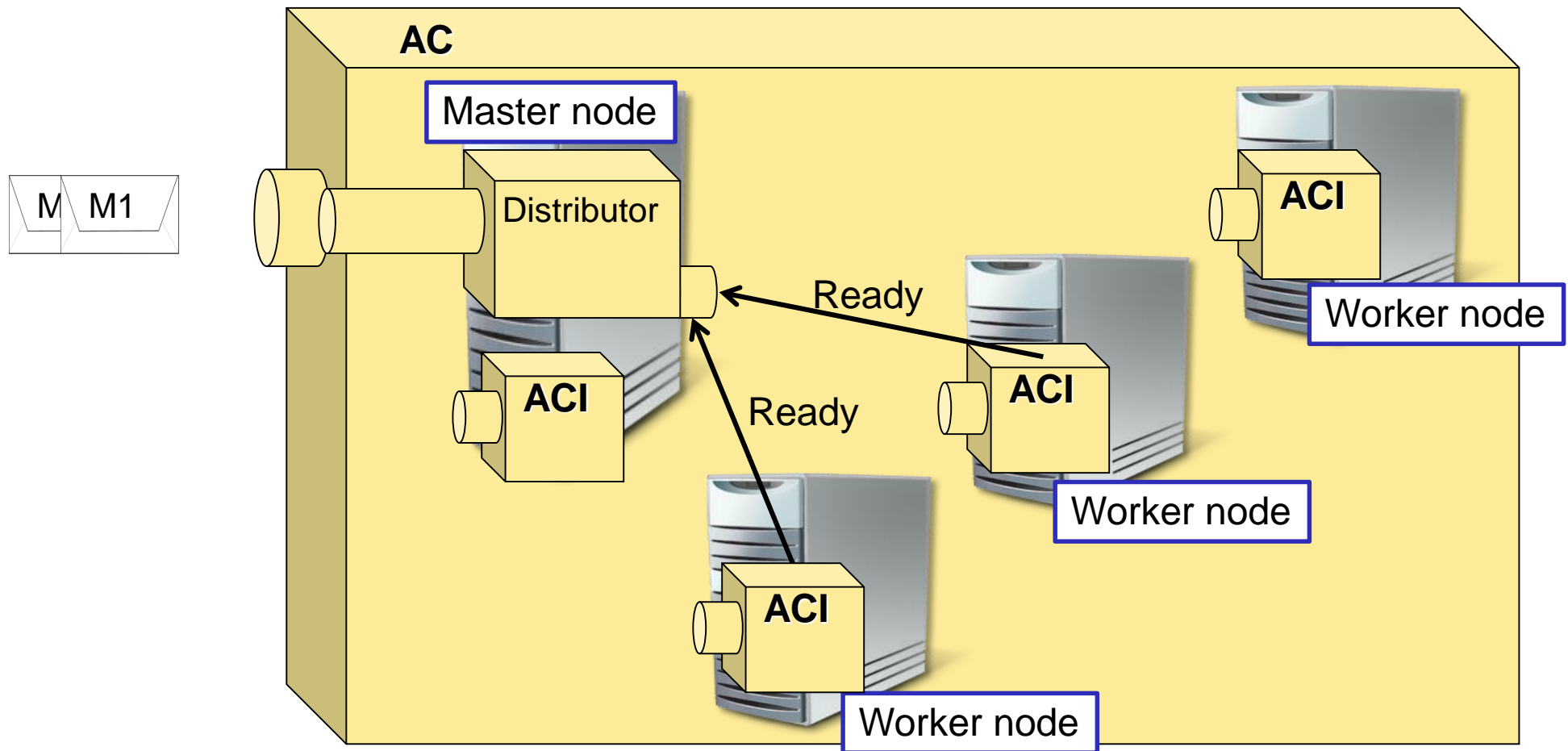


Scaling-out and Multi-Site Messaging

Master nodes and worker nodes

- ◆ Master node provides infrastructure capabilities
 - ◆ Distributor, RavenDB, TimeoutManager, etc
 - ◆ Needs to be highly available
- ◆ Worker nodes – process messages delivered from the master node
- ◆ Enables scale out with minimal configuration

Scaling a Autonomous Component



Configuring an ACI as the master

- ◆ Use the Master profile
 - ◆ Must be running on the master node
 - ◆ Runs the distributor in-process
 - ◆ Enlists it self as a worker
-
- ◆ If you want to run the distributor by itself, use the Distributor profile

Configuring an ACI as a worker

- ◆ Use the Worker profile
- ◆ Must know the master node

```
<MasterNodeConfig Node="MyCluster"/>
```

- ◆ Sends timeout/gateway requests to the master
- ◆ Connects to RavenDB on the master for shared subscriptions and sagas

Distributor – Group Exercise

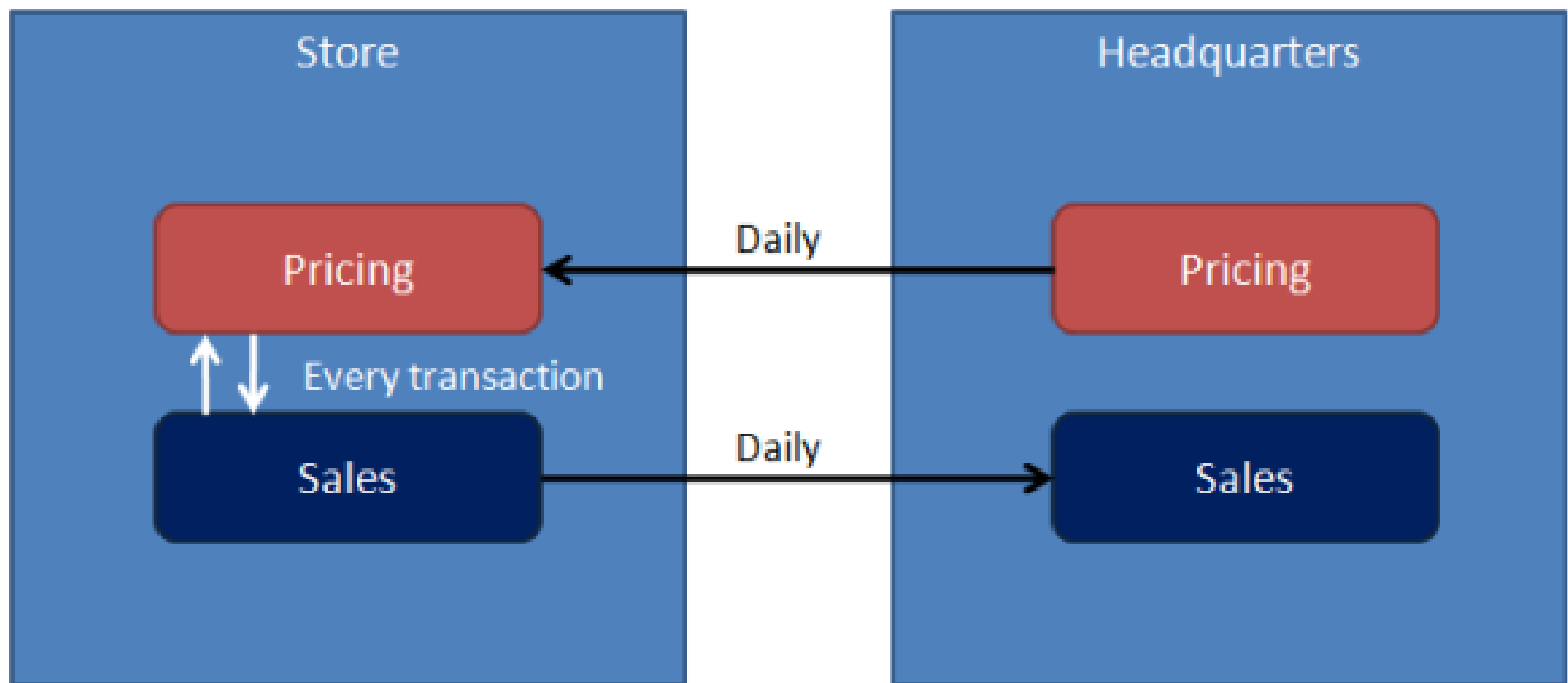
- ◆ Use the scale out sample from the download
- ◆ Run only the Orders.Handler project
- ◆ Have one machine be a master node
 - ◆ *NServiceBus.Production NServiceBus.Master*
- ◆ Configure the other machines to be workers
 - ◆ *NServiceBus.Production NServiceBus.Worker*
 - ◆ *<MasterNodeConfig Node="smt68"/>*

Cross-site communication

- ◆ When your sites are logically different
 - ◆ Headquarter vs Stores
- ◆ For High Availability (HA) scenarios infrastructure solutions should be used
- ◆ Prefer explicit message types for your cross-site communication

```
public class EndOfDaySalesReport : IMessage
{
    public List<Purchase> Purchases{ get;set; }
}
```

Logically significant sites



The Gateway

- ◆ Runs in-process with your endpoint
 - ◆ Enabled by the MultiSite profile
- ◆ Use the gateway by calling

```
Bus.SendToSites(endOfDayReport,  
                new[] { "Headquarter" })
```
- ◆ Builtin channels are Http/Https
 - ◆ Create you own by implementing IChannel1
- ◆ Handles de-duplication for you

Sagas

Saga: Definition

- ◆ A saga is pattern for implementing long-lived transaction by using a series of shorter transactions
- ◆ Sagas hold relevant state needed to process multiple messages in a “saga entity”
- ◆ Sagas are initiated by a message

Saga: Declaration

```
public class MySaga : Saga<MySagaData>,
    IAmStartedByMessages<M1>, IHandleMessages<M2>
{
    public void Handle(M1 message) {}
    public void Handle(M2 message) {}
}
```

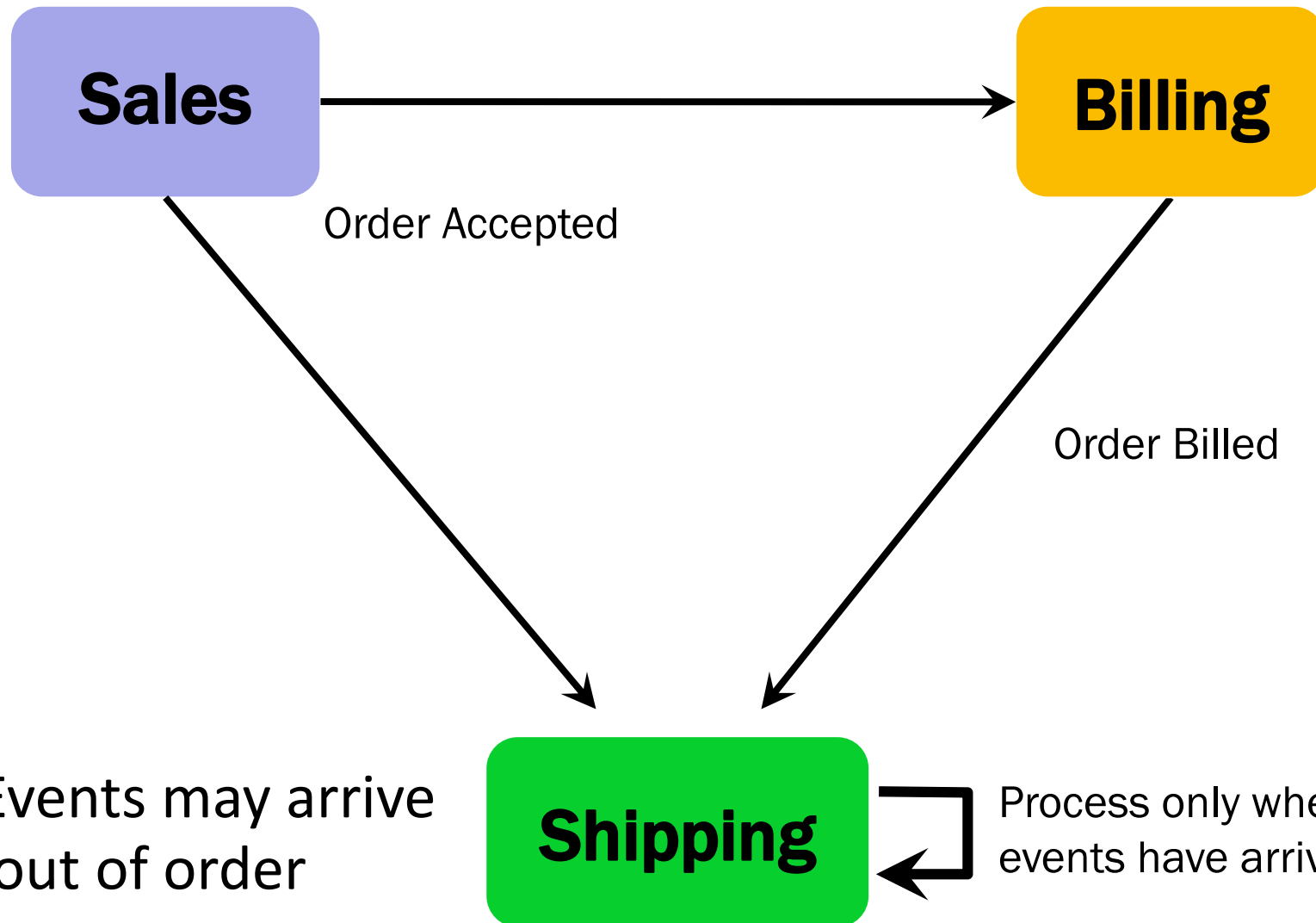
- Methods are like regular message handling logic

SagaEntity: Declaration

```
public class MySagaData : IContainSagaData
{
    public Guid Id { get; set; }
    public string Originator { get; set; }
    public string OriginalMessageId { get; set; }
}
```

- ◆ Must have these properties
- ◆ Easier to inherit ContainSagaData
 - ◆ This was a bad idea prior to V4 due to NHibernate saga storage implementation

Sagas and Services



Starting Sagas

- ◆ Sagas can be started by multiple messages
 - ◆ Implement `IAmStartedByMessages<>` for each
- ◆ First messages should start saga, following messages should be processed by the same one

Finding Sagas

- ◆ Saga can declare how to find it using messages:

```
public class MySaga : Saga<MySagaData>,
                    IAmStartedByMessages<M1>,
                    IAmStartedByMessages<M2>
{
    public override void ConfigureHowToFindSaga()
    {
        ConfigureMapping<M1>(m => m.SomeId)
            .ToSaga(s => s.MyId);
        ConfigureMapping<M2>(m => m.SomeId)
            .ToSaga(s => s.MyId);
    }
}
```

Finding Sagas

For greater flexibility:

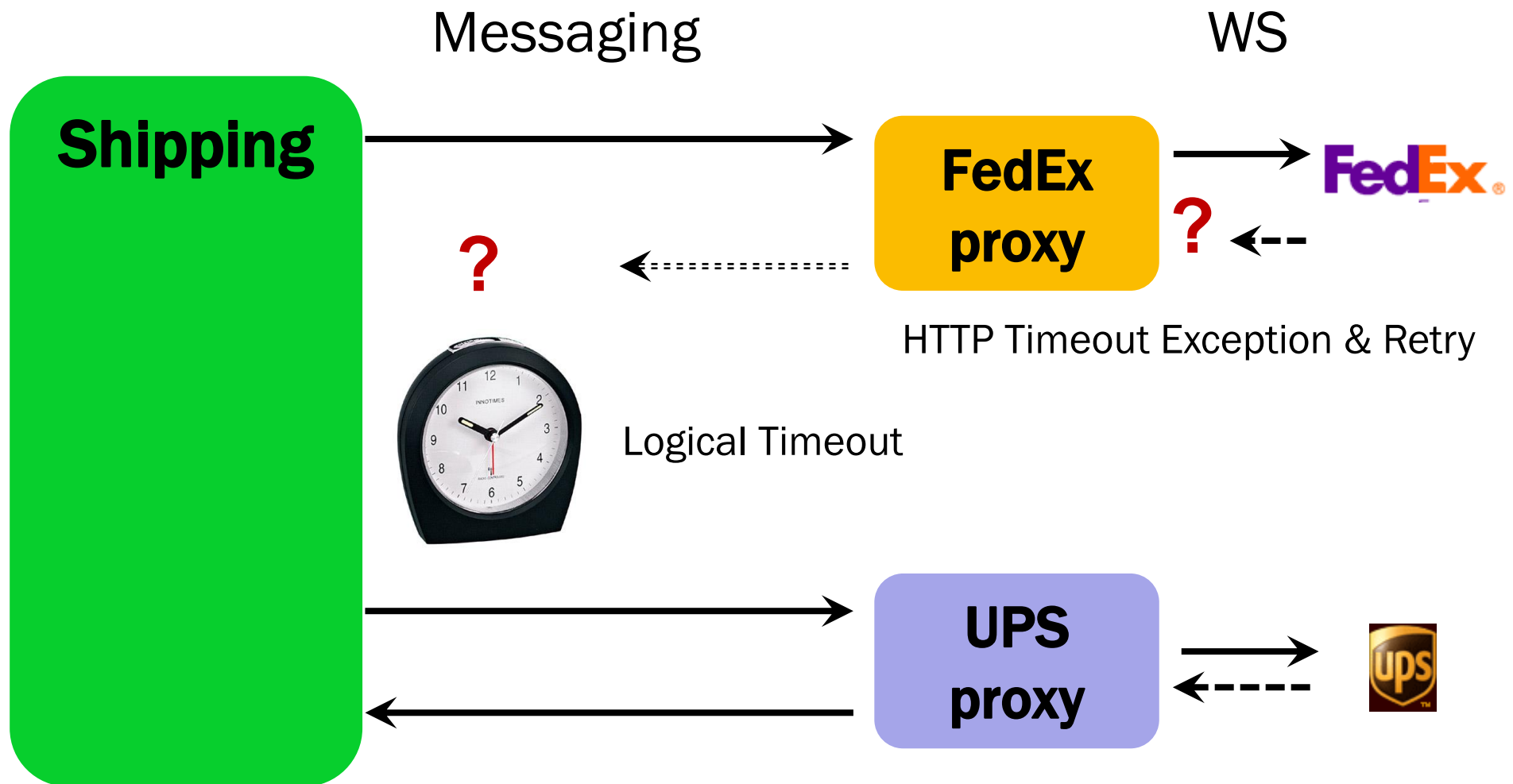
```
class MySagaFinder : IFindSagas<MySagaData>.Using<M1>
{
    public MySagaData FindBy(M1 message)
    {
        //Go to DB and find it using any/all data from message
    }
}
```

Ending a saga

- ◆ `MarkAsComplete();`
- ◆ Can call this from any method
- ◆ Causes the saga to be deleted
- ◆ Any data that you want retained should be sent on (or published) via a message

Timeouts

Sagas and Integration



Remember: when handling a response from a partner, if you want to reply to your client, use `Saga.ReplyToOriginator(msg);`

Sagas and Integration

```
public class ShippingSaga : Saga<ShippingSagaData>,
    IAmStartedByMessages<M1>, IHandleMessages<FedExResponse>,
    IHandleTimeouts<FedexTimedOut>
{
    public void Handle(M1 msg)
    {
        Data.Order = msg.Order;
        Bus.Send<ShipToFedEx>(m => m.Order = msg.Order);
        RequestTimeout<FedexTimedOut>(
            TimeSpan.FromMinutes(5));
    }
    public void Handle(FedExResponse msg)
    {
        this.MarkAsComplete();
    }

    public void Timeout(FedexTimedOut state)
    {
        Bus.Send<ShipToUps>(m => m.Data = Data.Order);
    }
}
```

Custom timeout state

```
public class FedexTimedOut
{
    public string SomeState {get;set;}
}
```

```
public void Timeout(FedexTimedOut state)
{
    if(state.SomeState)
        ...
}
```

Timeout Management

- ◆ Runs in process with your endpoint by default
 - ◆ Enabled by default
- ◆ Easily replaced with your own implementation
- ◆ Maintains timeout data in RavenDB

Timeout Configuration

- ◆ Convention based configuration
 - ◆ Input queue:
`{endpointname}.timeouts@{masternode}`
- ◆ Override using:

```
<UnicastBusConfig
```

```
TimeoutManagerAddress="timeouts@myserver"/>
```

Unit testing

Unit Testing Sagas

```
Test.Saga<MySaga>(sagaId)
    .WhenReceivesMessageFrom(clientEndpoint)
        .ExpectReplyToOriginator<M1>( /* check data */ )
        .ExpectPublish<M2>( /* check data */ )
        .ExpectSend<M3>( /* check data */ )
        .When( saga => saga.Handle(firstMessage) );
```

Check Data:

```
m => return (m.Data == Something)
```

Unit Testing Saga Timeouts

```
MyTimeoutState state;
```

```
Saga.WhenReceivesMessageFrom(clientEndpoint)  
    .ExpectTimeoutToBeSetAt<MyTimeoutState>(...)  
    .When( saga => saga.Handle(firstMessage) );
```

```
Saga.Expect...  
    .When( saga => saga.Timeout(state) );
```


Exercise: Sagas

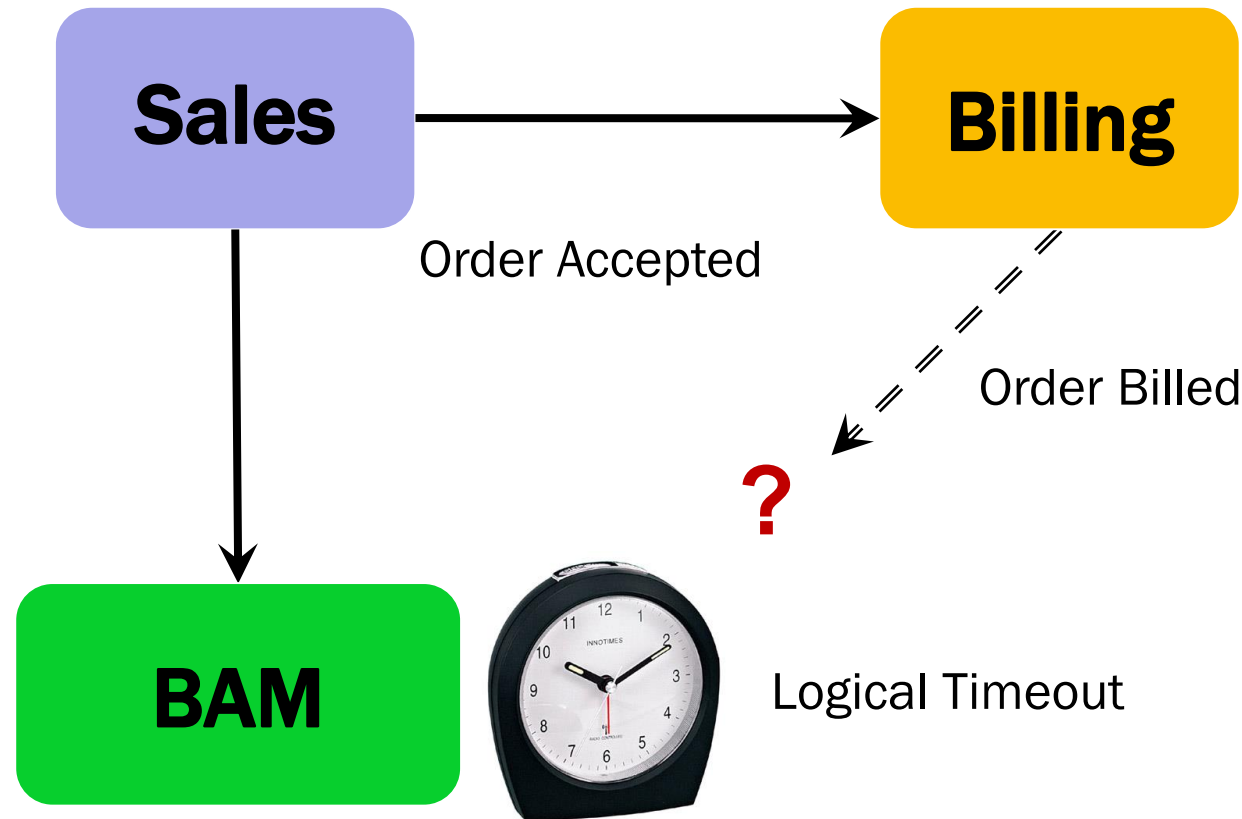
- ◆ Implement and unit test the shipping saga described

Business Activity Monitoring

BAM & Monitoring

- ◆ Business Activity Monitoring
 - ◆ High-level “what’s going on” from a business view
- ◆ Monitoring
 - ◆ Lower-level “what’s going on” from a technical view
- ◆ Important to be able to tie one to the other

When Events Don't Happen In Time



Often makes use of sagas

How big of a timeout?

- ◆ The service's SLA specifies time
- ◆ BAM bases timeout choices on that SLA

Dashboards

- ◆ BAM can show:
 - ◆ How many cross-service processes are in progress
 - ◆ How many did we do yesterday, last week, etc
 - ◆ Statistics around completion time
 - ◆ Trends
- ◆ In real time
- ◆ Useful for administrators to see problematic trends
 - ◆ Zoom in on an AC of problem service

SOA, BAM, and Centralization

- ◆ There is no central authority which orchestrates autonomous services
- ◆ Business Activity Monitoring provides a central point of visibility on what's going on

SOA, CQRS, and Sagas

All together now

Race Conditions – the secret sauce

- ♦ Race conditions may indicate a collaborative domain – fertile ground for CQRS
- ♦ May even make you think your service boundaries are wrong

Real World Requirements

- ◆ Do not allow users to cancel shipped orders
- ◆ Don't ship cancelled orders
- ◆ As we shrink the time between actions, a race condition presents itself

Service Boundary Issues

- ◆ Cancelling an order is in the Sales service
- ◆ Shipping an order is in the Shipping service
- ◆ Requirements seem to imply need for consistency/transactions between services

Implementation is simple with 3-Tier

```
public class Order
{
    public void Cancel()
    {
        if (status != OrderStatusEnum.Shipped)
            //cancel
    }

    public void Ship()
    {
        if (status != OrderStatusEnum.Cancelled)
            //ship
    }
}
```

Remember

- ◆ In CQRS, commands don't fail
- ◆ Race conditions don't exist in business
- ◆ A microsecond either way shouldn't change business objectives

Find underlying business objectives

Rules:

1. Cannot cancel shipped orders

Why?

Because shipping costs money

So?

That money would be lost
if the customer cancelled

Why?

Because we refund the customers money

2. Don't ship cancelled orders

Refund Policies

Analyze

- ◆ When an order is cancelled, does the refund need to be given immediately?

No

- ◆ Can we give a partial refund?

Yes

Dig Deeper

- What does a customer have to do in order to get a refund?

Return the products

Most orders cancelled soon after they were made – buyer's remorse

Implement a saga for buyer's remorse in the Sales service

Consider Service Boundaries

Order Accepted
Order Cancelled

Sales

Products Returned

Shipping

Customer Charged
Refund Policy

Billing

Implement a saga for the refund policy in Billing

Summary and Q&A

- ◆ NServiceBus is *one piece* of the puzzle
- ◆ Points you in the right direction
 - ◆ Creating friction if you're going the wrong direction
- ◆ Keep in mind the importance of the UI
- ◆ Q&A ...

So long, and thanks for all the fish



`www.particular.net`

`/groups.google.com/forum/#!forum/particularso`
`stackoverflow.com/questions/tagged/nservicebu`

