.NET Enterprise Services

Cleveland .NET SIG, January 2004

Juval Löwy
www.idesign.net

About Juval Löwy

- Software architect
  - Consults and trains on .NET migration and design
- Microsoft's Regional Director for the Silicon Valley
- Authored
  - Programming .NET Components (2003, O'Reilly)
  - COM and .NET Component Services (2001, O'Reilly)
- Participates in the .NET design reviews
- Contributing editor and columnist to the Visual Studio Magazine
  - Publishes at MSDN and other magazines
- Speaker at the major international software development conferences
- Recognized Software Legend by Microsoft
- Contact at www.idesign.net

What are Enterprise Applications?

- The term Enterprise means different things for different people
  - Large number of users, scalability and throughput a must
  - Fewer users, with drastic spikes in load
  - Few users using many expensive resources
  - Mission critical, 24x7, zero down time
  - Sensitive information
  - Interoperate with a wide range of platforms
- Where quality and productivity are top priority
- Any application that is not a toy program

What are .NET Enterprise Services?

- Set of component services designed to ease considerably developing Enterprise applications
- The result of integrating COM+ into .NET
- Components using these services called Serviced Components
- .NET assemblies mapped to COM+ applications
  - A COM+ application can contain components from multiple assemblies

.NET Enterprise Services

- Instance management
  - Pooling
  - JITA
- Transactions
- Concurrency management
- Loosely coupled events
- Queued components
- Security
  - Authorization
  - Authentication
  - Identity

.NET Enterprise Services

- Remote calls and Web Services
- Combination of the services
- Services are configured:
  - Administratively
  - Declaratively (attributes)
  - Programmatically

Evolution of Enterprise Services

- MTS 1.0 1996
- MTS 2.0 1998
- COM+ 1.0 2000
- COM+ 1.5 2001
- .NET Enterprise Services 2002
- Indigo 2006 (?)
- MTS and COM+ were bad technology monikers

Serviced Components

- Serviced component must derive from **`ServicedComponent`**
  - **`System.EnterpriseServices`**
  - Cannot use static members/methods
  - Cannot have parameterized constructors
  - Should use interfaces (not have to)
  - Should use class libraries (not have to)
- Use Component Services Explorer or special attributes
  - Some with Explorer only (deployment specific)
  - Some programmatic only
  - Everything else use attributes
- .NET integration with COM+ is better than COM/VS6.0

Serviced Components


```
using System;
using System.EnterpriseServices;

public interface IMessage
{
   void ShowMessage();
}
public class MyComponent:ServicedComponent,IMessage
{
   public MyComponent() {}//constructor
   public void ShowMessage()
   {
      MessageBox.Show("Hello!","MyComponent");
   }
}
```

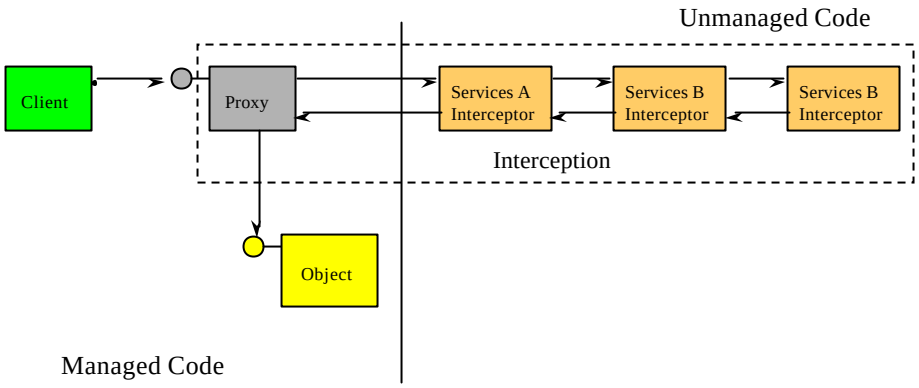
- Use the Component Services Explorer for this component

Interception

- .NET does not use COM interop for serviced components
- Special context interceptors configure services
- Call itself doesn't leave managed code
- Remote calls can use
  - DCOM
  - Remoting
  - Web services

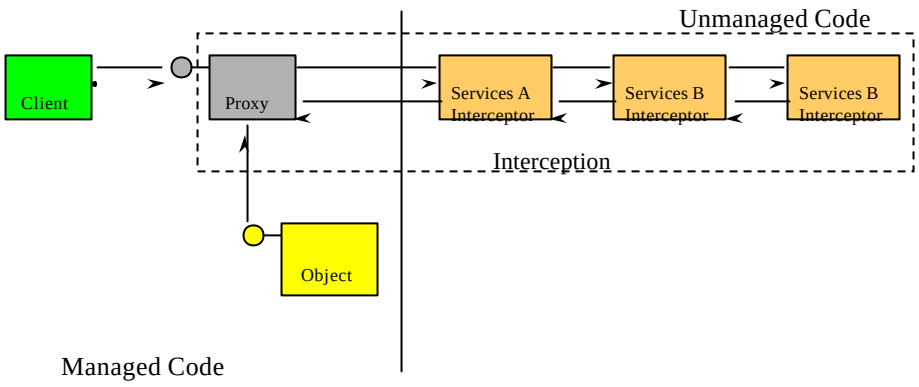
Interception (today)

- Call itself doesn't leave managed code
  - No parameters marshaling conversion penalty

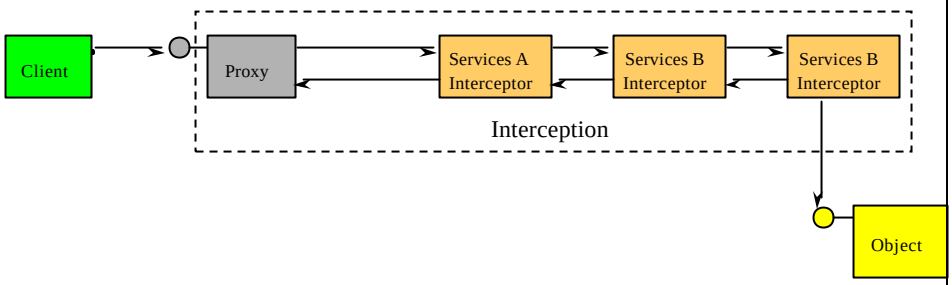


Interception (today)

- Call itself doesn't leave managed code
  - No parameters marshaling conversion penalty



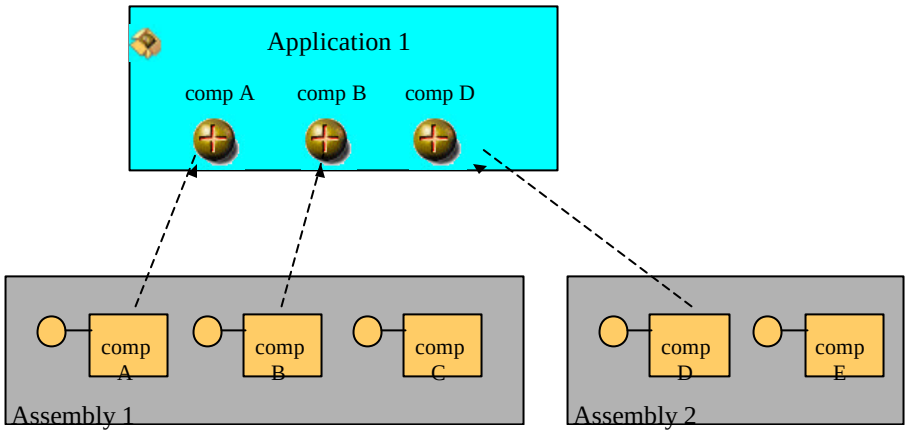
Interception - Indigo



Managed Code

Assemblies and Applications

- COM+ application can contain components from multiple assemblies



Dynamic Registration

- When no need for deployment specific configuration
- At runtime, .NET verifies current assembly version is mapped to COM+ application
  - New version/signatures triggers dynamic registration of all components in the assembly
    - RegSvcs.exe /reconfig /fc MyAssembly.dll
- If app exists but components are not in, .NET adds them to app
- Only available for managed clients
  - Administrator rights
- COM+Application stays in the Component Services Explorer

Application Activation Type

- Can specify application activation type
  - Server or a library application
  - Default is library

```
[assembly: ApplicationActivation(ActivationOption.Server)]
//or
[assembly: ApplicationActivation(ActivationOption.Library)]
```

  - **`ActivationOption.Service`** is not available

Versioning

- RegSvcs creates CLSID for components
  - Any change registers a new CLSID, to avoid versioning conflicts with existing clients
  - Clients automatically use highest compatible version
- Can specify that CLSID as a class attribute

```
using System.Runtime.InteropServices;

[Guid("260C9CC7-3B15-4155-BF9A-12CB4174A36E")]
public class MyComponent :ServicedComponent,IMyInterface
{…}
```

Versioning

- Specifying CLSID forces RegSvcs to use it in spite of changes
  - Very handy for development
- Can also specify the component name (prog-ID)
  - **`ProgidAttribute`**
  - **`System.Runtime.InteropServices`**

```
using System.Runtime.InteropServices;

[ProgId("My Serviced Component")]
public class MyComponent :ServicedComponent,IMyInterface
{…}
```

Assembly Identity

- Assembly must have a strong name
  - To ensure versioning
- Assembly should be in the GAC
  - Must be in a known location otherwise
  - Server applications/remote calls

Remote Calls

- Can use .NET remoting
- Can generate ES proxy application
- Can use web services
  - Windows 2003 Server only

Accessing ES Context

- Not the same as .NET context
- **`ContextUtil`** access context object and its interfaces
  - Static methods and properties
- Example: tracing context ID

```
public interface IMyInterface
{
   void MyMethod();
}
public class MyComponent :ServicedComponent,IMyInterface
{
  public void MyMethod()
  {
    Guid contextID = ContextUtil.ContextId;
    String traceMessage = "Context ID is " + contextID.ToString();
    Trace.WriteLine(traceMessage);
  }
}
```

Activation Context Attribute

- **`MustRunInClientContext`** attribute controls whether object must be activated in creator's context

```
[MustRunInClientContext(true)]
public class MyComponent :ServicedComponent,IMyInterface
{}
```

  - Default is true

```
[MustRunInClientContext]
//same as
[MustRunInClientContext(true)]
```

Instance Management

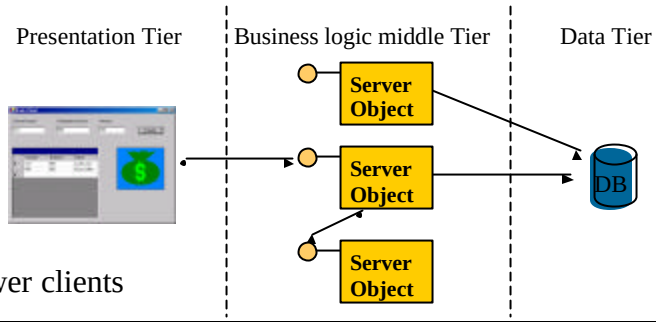
- The single most important driving force behind the move to three tiers is the need for scalability
- You no longer can expect to dedicate a server object per client

Client Types

- Needs to handle two kinds of clients
  - Intranet clients
  - Internet clients
- Differ not only in the way they connect with application, but also in the interaction pattern
- Must scale up to both kinds, and compensate for the differences

Intranet client

- Like classic client/server model
- Usually a rich UI (Window Forms or browser with ActiveX controls)
  - Richer user experience, more privileges and features
- Connects directly to server objects in middle tier



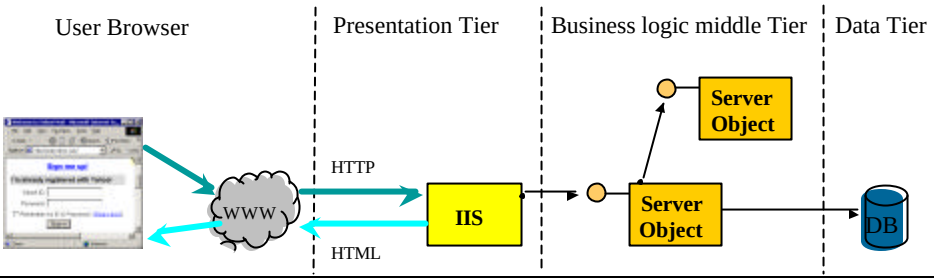
- Relatively fewer clients

Intranet client

- Calling pattern
  - Create object
  - Use it
  - Dispose
- Relatively fewer Intranet clients
  - Overhead of creating objects and clean up per client is not a scalability limitation
- What impends scalability
  - Holding objects for long time, while using objects only in a fraction of that time
  - When Intranet application starts, it gets all objects it needs (perforce,responsiveness) and dispose at shutdown
  - If object per client, you tie in crucial resources for long time,
  - and will eventually run out of resources

Internet client

- A browser or client connect to Internet server (IIS)
- Page or web service creates objects and when the request is done, disposed resources
- The client connection is stateless – no object references are maintain outside the scope of a request



Internet client

- The duration page uses the objects is usually not a scalability limitation
- Relatively many Intranet clients
  - Overhead of creating objects and clean up per client is a scalability limitation
  - Service will appear to be unavailable or slow response time
  - At peak demand periods, may kill your ASP.NET app

Object Pooling

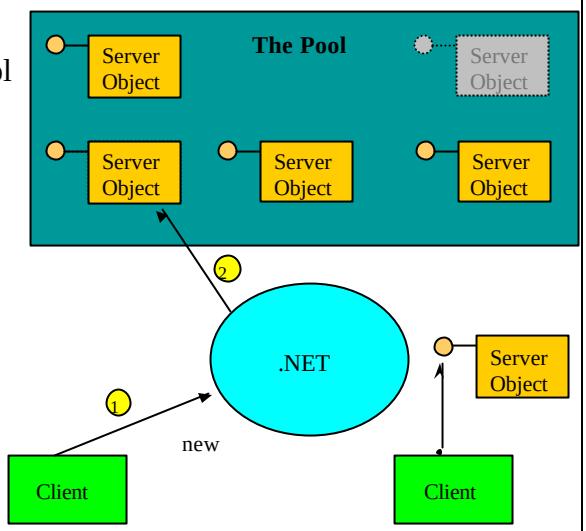
- ES maintain a pool of objects, ready to serve clients
- When asked to create new object, .NET checks if there is an object in the pool, and will return it to client
  - Else: create a new object, up to the pool limit
- Pool is per object type
  - In one application can have components with no pool, or as many pools as services components types

Object Pooling

- Internet clients
- When instantiating object is a costly generic operation
  - Or when need to pool resources
- Constructor does as much of the time-consuming work uniform for all clients
  - Acquiring connections
  - Running scripts
    Fetching initialization data from files or across network

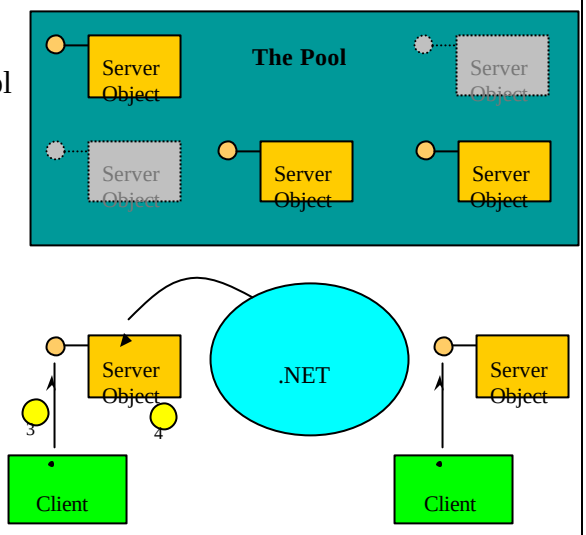
Object Pooling

1. Client calls new
2. .NET retrieves from pool
3. Client use the object
4. GC
5. .NET returns object to pool



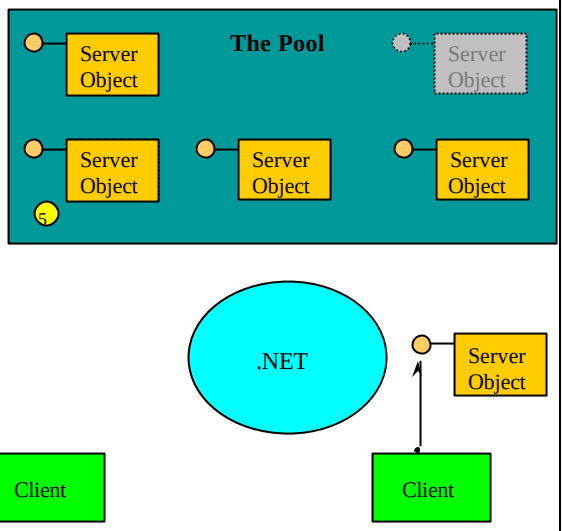
Object Pooling

1. Client calls new
2. .NET retrieves from pool
3. Client use the object
4. GC
5. .NET returns object to pool

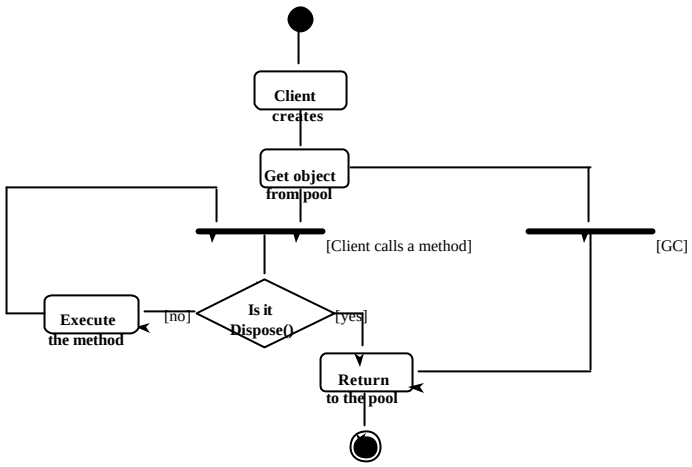


Object Pooling

1. Client calls new
2. .NET retrieves from pool
3. Client use the object
4. GC
5. .NET returns object to pool



Object Pooling



Object Pooling

- Min pool size
  - Start pool with that number
  - Mitigate sudden spikes in demand
- Max pool size
  - Total number of objects created
  - Once reached, further requests for objects blocked for 'Creation timeout'
  - If in timeout object returned to pool, the client gets it
  - Clients servers FIFO

Object Pooling

- If the object is in server application, pool is per machine
  - Potentially per LAN
- If the object is in library application, pool per app domain
  - Two clients in different app domain will use two distinct pools

Object Pooling

- **`ObjectPooling`** attribute controls pooling aspects

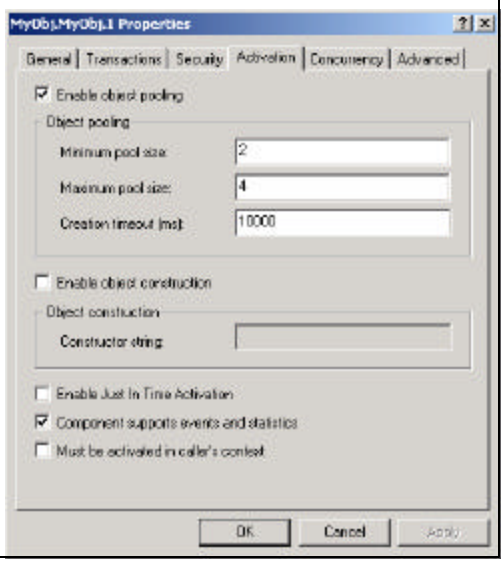
```
[ObjectPooling(MinPoolSize = 3,MaxPoolSize = 10,CreationTimeout = 20)]
public class MyComponent :ServicedComponent,IMyInterface
{}
```

- A few overloaded constructors:

```
//first three are equivalent
[ObjectPooling]
[ObjectPooling(true)]
[ObjectPooling(Enabled = true)]
[ObjectPooling(MinPoolSize = 2,MaxPoolSize = 10,CreationTimeout = 700)]
[ObjectPooling(MinPoolSize = 2)]
[ObjectPooling(true,MinPoolSize = 0,MaxPoolSize = 10)]
[ObjectPooling(Enabled = true,MinPoolSize = 0,MaxPoolSize = 10)]
```

Object Pooling

- Set the min pool size (low water mark) or the max pool size (high water mark)
- Set the creation timeout
  - Not the same as the time it takes to create an object



Disposing of a Pooled Object

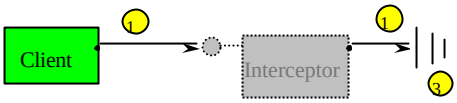
- Object returns to pool when garbage collected
- Client can call **`Dispose()`** in **`ServicedComponent`** to expedite

Just In Time Activation (JITA)

- .NET can dedicate object per client only while a call in progress
- When instantiating object is not costly, but object uses expensive/scarce resources and when clients can hold the object reference
  - Intranet clients
- Client would not know the difference
  - References a proxy

Just In Time Activation (JITA)

1. Client calls a method on proxy, that delegates the call to the object
2. When method returns, object indicates it can be deactivated
3. Interceptor release object, and nulls its pointer to it
4. Client makes another call, the interceptor creates a new object and delegates the call



Just In Time Activation (JITA)

1. Client calls a method on proxy, that delegates the call to the object
2. When method returns, object indicates it can be deactivated
3. Interceptor release object, and nulls its pointer to it
4. Client makes another call, the interceptor creates a new object and delegates the call

Just In Time Activation (JITA)

- .NET can not arbitrarily kill objects, that may not be ready to be shut down
  - Object has to tell ES it is willing to be destroyed
- Requires object to be state aware
  - At beginning of every method should initialize its state from a durable storage, and at the end should save its state

Just In Time Activation (JITA)

- Letting .NET object is ready to be destroyed by setting the 'Done' bit on the object context
- By default, done bit is set to false
  - JITA object created in its own context
- Can set done bit programmatically or configure a method to automatically deactivate on return

Just In Time Activation (JITA)

- **`JustInTimeActivation`** attribute turns JITA on or off
  - Default constructor turns on
- **`ContextUtil`** property **`DeactivateOnReturn`** sets context object done bit

```
[JustInTimeActivation]
public class MyComponent :ServicedComponent,IMyInterface
{
   public void MyMethod(long objectIdentifier)
   {
     GetState(objectIdentifier);
     DoWork();
     SaveState(objectIdentifier);
     //inform .NET to deactivate the object upon method return
     ContextUtil.DeactivateOnReturn = true;
   }
}
```

Just In Time Activation (JITA)

- Auto-deactivation:

```
[JustInTimeActivation]
public class MyComponent :ServicedComponent,IMyInterface
{
   [AutoComplete]
   public void MyMethod(long objectIdentifier)
   {
     GetState(objectIdentifier);
     DoWork();
     SaveState(objectIdentifier);
   }
}
```

Just In Time Activation (JITA)

- JITA object get disposed on deactivation
- Put cleanup in destructor
  - Deterministic destructor!

```
[JustInTimeActivation]
public class MyComponent :ServicedComponent,IMyInterface
{
   ~MyComponent()
   {
      //put cleanup code here
   }
}
```

JITA with Object Pooling

- Useful when the initialization is generic and expensive (just JITA would not make sense)
- Instead of creating and release object on each method, .NET grabs it from the pool and return the object to the pool

Transactions

- Execution requires intermediate inconsistent system state
  - Money deducted but no bills yet
  - Must roll back changes in case of error
- Multiple users may access system at the same time
  - Their access and changes should be isolated from each other
  - The resource must synchronize access to information

Transactions

- It's impractical to try and write error handling code
  - Very complex scenarios – bound to miss some
  - Tons of extra code
  - Productivity hit
  - Performance hit
  - Fragile solution
  - Impossible to test or debug
- .NET simplifies using transactions
  - Administrative configuration
  - Auto-enlistment
  - Distributed transactions

What is a Transaction?

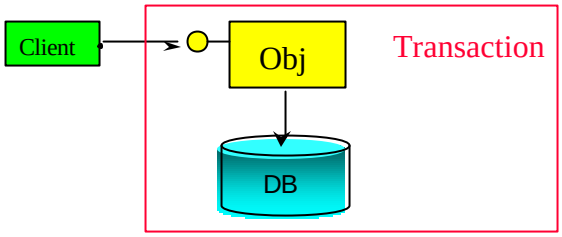
- A set of potentiality complex operations, that will all succeed or fail, as one atomic operation
- Transactions have been around since the early 60's
  - Introduced by databases, but other resources such as messaging systems support them as well
  - Complex Transaction Processing Monitors (TPM) coordinate transactions across databases and resources
  - A transaction is executed on behalf of one client only
- Transaction is usually of short duration

What is a Transaction?

- Transactions can spread across multiple machines and resources
  - Any resource can fail
  - All resources determine overall success or failure (vote on the transaction outcome)
  - Coordination challenge
- While transaction in progress, system can be in an inconsistent state
  - But the transaction must leave the system in a consistent state
- Faster transaction == scalability and throughput
- In general, whenever you update persistent storage, do it under the protection of a transaction

Single Component/Single Resource Transaction

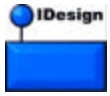
- One component, accessing just one resource (such as a database) that should take part in a transaction
- Component informs resource a transaction has started (enlist the resource)
- Resource starts recording operations you ask it to do, but does not actually do it yet



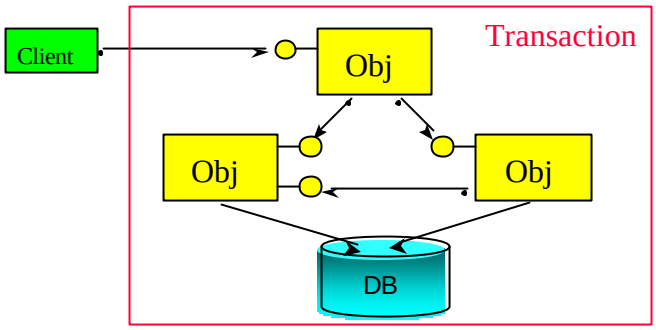
Single Component/Single Resource Transaction

- When component is done, it informs resource to commit changes
  - If errors, it should instruct to abort, or rollback
- Even if component wants to commit, the resource could have errors, and the transaction aborts
- Observation: only the application can request to commit but both application and resource can abort the transaction
- Requires explicit programming, to enlist a resource in a transaction and inform it to commit or rollback
  - **`BeginTransaction()`** and **`EndTransaction(commit/abort)`** calls
  - Most resources support this sort of interaction

Multiple Components/Single Resource Transaction



- Multiple components accessing just one resource that should take part in a transaction
- Things get a lot more complicated



Multiple Components/Single Resource Transaction

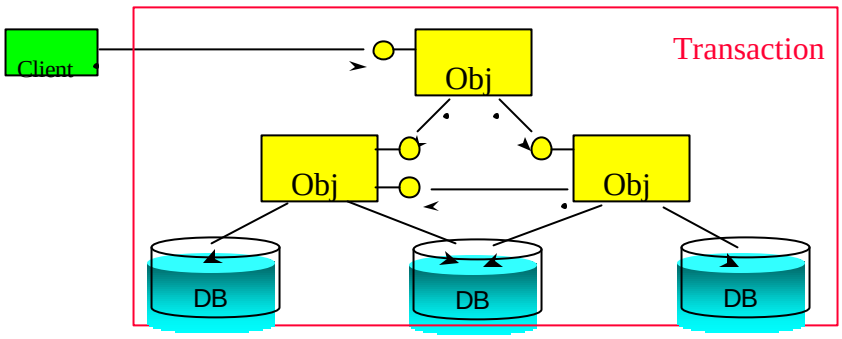
- Resource should be enlisted just once
  - By whom? first to access it? first created? How would components know all that?
- Components can be on different machines
  - Transaction has to flow across machine boundary
  - One machine can crush, while the other continue processing
- Each component can encounter error and abort
  - Only if they all succeed ask the resource to commit
  - Somebody has to collect the votes

Multiple Components/Single Resource Transaction

- The resource should be notified about the vote just once
  - But who knows what is the right thing to do?
- Resource can still refuse to commit changes
- .NET make this situation as easy as the previous one

Multiple Components/Multiple Resources Transaction

- Multiple components accessing multiple resources, all taking part in same transaction



Multiple Components/Multiple Resources Transaction

- Multiple points of failure
- Resources must be enlisted, and just once
  - Who keeps track of what resources are used?
  - Put that knowledge in your code ?
- Components and resources can be on different machines
  - Transaction has to flow across machine boundary
- Each resource can encounter error with the requested changes (wrong account number) and abort the transaction
- .NET make this as easy as the first one

Transactions

- **`TransactionOption`** enums declares COM+ transaction support

```
public enum TransactionOption
{
   Disabled,
   NotSupported,
   Supported,
   Required,
   RequiresNew
}
[Transaction(TransactionOption.Required)]
public class MyComponent : ServicedComponent
{…}
```

  - Default constructor is “Required”

```
[Transaction]
[Transaction(TransactionOption.Required)]
```

Voting On Transaction

- Set the static property **`MyTransactionVote`** of **`ContextUtil`**

```
ContextUtil.MyTransactionVote = TransactionVote.Commit;
```

- Can use **`ContextUtil.SetComplete()`** or **`ContextUtil.SetAbort()`**
- Be mindful of exceptions

```
[Transaction]
public class MyComponent : ServicedComponent
{
   public void MyMethod(long objectIdentifier)
   {
      try
      {
         GetState(objectIdentifier);
         DoWork();
         SaveState(objectIdentifier);
         ContextUtil.MyTransactionVote = TransactionVote.Commit;
      }
      catch
      {
         ContextUtil.MyTransactionVote = TransactionVote.Abort;
      }
      //Let COM+ deactivate the object once the method returns
      ContextUtil.DeactivateOnReturn = true;
   }
   //other methods
   protected void GetState(long objectIdentifier){…}
   protected void DoWork(){…}
   protected void SaveState(long objectIdentifier){…}
}
```

Voting On Transaction

- Voting without exception-handling

```
[Transaction]
public class MyComponent :ServicedComponent
{
   public void MyMethod(long objectIdentifier)
   {
      //Let COM+ deactivate the object once the method returns
      //and abort. Can use ContextUtil.SetAbort() as well
      ContextUtil.DeactivateOnReturn = true;
      ContextUtil.MyTransactionVote = TransactionVote.Abort;

      GetState(objectIdentifier);
      DoWork();
      SaveState(objectIdentifier);

      ContextUtil.MyTransactionVote = TransactionVote.Commit;
    }
}
```

- Offers exception propagation (good for performance)

Auto Complete Attribute

- **`AutoComplete`** method-attribute sets done and consistency bits to true if the method did not throw an exception, and the consistency bit to false if it did
  - Similar the COM+ Method auto-deactivation
  - Deactivates JITA objects

```
[Transaction]
public class MyComponent : ServicedComponent
{
   [AutoComplete]
   public void MyMethod(long objectIdentifier)
   {
      GetState(objectIdentifier);
      DoWork();
      SaveState(objectIdentifier);
   }
}
```

Auto Complete Attribute

- Overloaded constructor:

```
[AutoComplete]
[AutoComplete(true)]
```

- Avoid using at interface definition:
  - Works, but confused design and contract with implementation

```
public interface IMyInterface
{
   //Avoid this:
   [AutoComplete]
   void MyMethod(long objectIdentifier);
}
```

Enterprise Services Security

- Only way in .NET 1.1 for authentication of remote calls out of the box
  - Granular control
  - Encryption
- Rich role-based security
  - Independent of Windows groups
  - Full security call context propagation

Application Security

- **`ApplicationAccessControl`** attribute
  - Turning authorization on/off
    - Constructor takes a Boolean flag
  - Security level **`AccessChecksLevel`** property set to **`AccessChecksLevelOption.ApplicationComponent`** or **`AccessChecksLevelOption.Application`**
  - Authentication level **`Authentication`** property accepts enum values of **`AuthenticationOption`**
  - Impersonation level **`ImpersonationLevel`** property accepts the enum values of **`ImpersonationLevelOption`**

Application Security

- Server app

```
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(
           true,//Authorization
           AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
           Authentication=AuthenticationOption.Packet,
           ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

Application Security

- Library app:

```
[assembly: ApplicationActivation(ActivationOption.Library)]
[assembly: ApplicationAccessControl(
           true,// Authorization
           AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
          //use AuthenticationOption.None to turn off authentication,
           //and any other value to turn it on
           Authentication=AuthenticationOption.None)]
```

Component Access Checks

- Turn component level access checks on or off using **`ComponentAccessControl`** attribute

```
[ComponentAccessControl(true)]
public class MyComponent :ServicedComponent,IMyInterface
{}
```

  - Default constructor turns security on **`[ComponentAccessControl]`**

Adding Roles to Application

- Use **`SecurityRole`** attribute

```
[assembly: SecurityRole("Manager",
                        Description = "Can access all components")]
[assembly: SecurityRole("Teller",
                        Description = "Can access IAccountsManager only")]
```

  - Description property is optional

Adding Roles to Application

- Overloaded constructors

```
[assembly: SecurityRole("Manager")]
[assembly: SecurityRole("Manager", false)]
[assembly: SecurityRole("Manager",SetEveryoneAccess = false)]
```

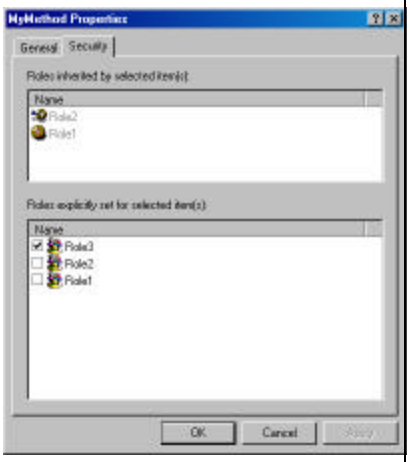
- Use the Marshaler role to create components

```
[assembly: SecurityRole("Marshaler",SetEveryoneAccess = true)]
```

- **`SecureMethod`** attribute to prevent marshaler from abusing reflection
  - At class or method level

Assigning Roles

- Use **`SecurityRole`** attribute
  - No use for users/description properties at that level




```
[assembly: SecurityRole("Role1")]
[assembly: SecurityRole("Role2")]
[assembly: SecurityRole("Role3")]

[SecurityRole("Role2")]
interface IMyInterface
{
  [SecurityRole("Role3")]
  void MyMethod();
}

[SecurityRole("Role1")]
public class MyComponent :ServicedComponent,IMyInterface
{}
```

Verifying Caller's Role Membership

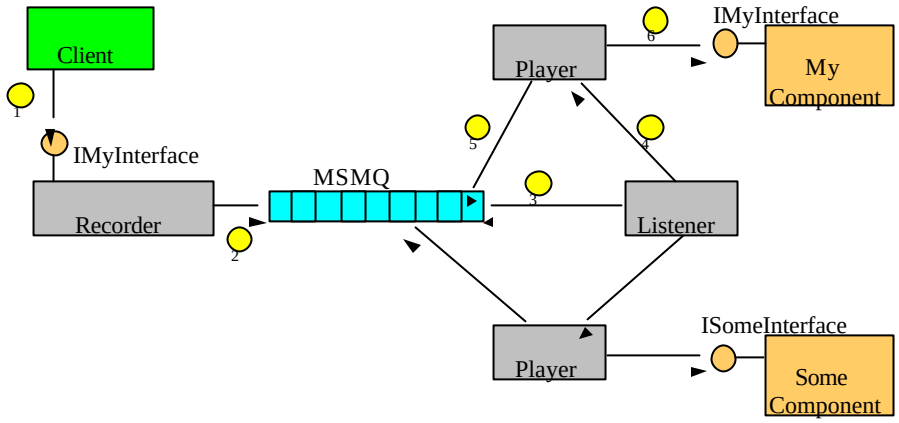
- Done via **`SecurityCallContext`** object
  - Current call is a static property of same type

```
[SecurityRole("Manager")]
public class Bank : ServicedComponent
{
   [SecurityRole("Customer")]
   void TransferMoney(int sum,long accountSrc,long accountDest)
   {
      bool callerInRole = false;
      callerInRole = SecurityCallContext.CurrentCall.IsCallerInRole("Customer");
      if(callerInRole)//The caller is a customer
      {
         if(sum > 5000)
           throw(new UnauthorizedAccessException(@"Caller does not have sufficient
                                                 credentials to transfer this sum"));
      }
      DoTransfer(Sum,accountSrc,accountDest);//Helper method
    }
    //Other methods
}
```

Queued Components

- Disconnected work
  - Orders submitted without access to remote resources
  - Most business communications are async by nature (email, vmail…)
  - ~40% of new computers sold are mobile
- Workload buffering
  - When workload is uneven during the day
- Transactional work

QC Architecture



Queued Components

- **`ApplicationQueuingAttribute`** configures application level QC support
  - Accept queued calls and enable a listener:

```
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationQueuing(Enabled = true,QueueListenerEnabled = true)]
```

  - Default only accepts calls, no listener:

```
[assembly: ApplicationQueuing]
//same as:
[assembly: ApplicationQueuing(Enabled = true,QueueListenerEnabled = false)]
```

Queued Components

- **`InterfaceQueuingAttribute`** configures interface to support queued calls

```
[InterfaceQueuing]
public interface IMyInterface
{
   void MyMethod();
}
```

  - Overloaded constructors:

```
//all are equivalent
[InterfaceQueuing]
[InterfaceQueuing(true)]
[InterfaceQueuing(Enabled = true)]
```

QC-Client Side

- Client uses **`Marshal.BindToMoniker()`** to record calls

```
IMyInterface obj;
obj=(IMyInterface)Marshal.BindToMoniker("queue:/new:MyAssembly.MyComponent
obj.MyMethod();//call is recorded
```

- Recorder adds message to queue when garbage collected
- Client can expedite by forcing a release:

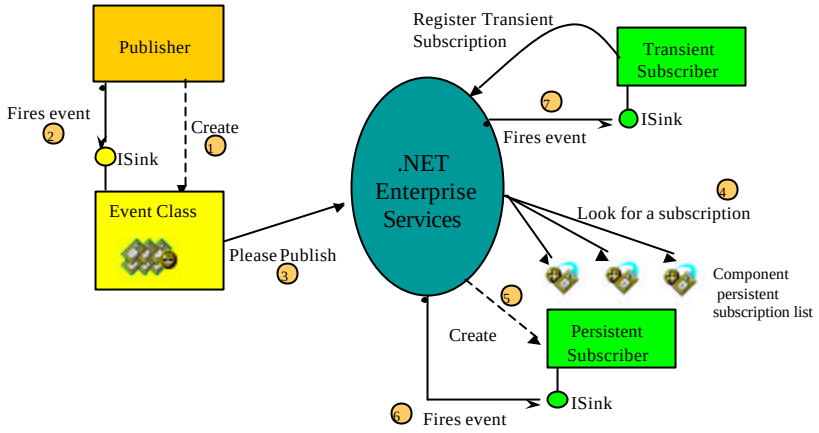
```
IMyInterface obj;
obj=(IMyInterface)Marshal.BindToMoniker("queue:/new:MyAssembly.MyComponent
obj.MyMethod();//call is recorded

//Expedite dispatching the recorded calls by releasing the recorder
Marshal.ReleaseComObject(obj);
```

Loosely Coupled Events

- Effective way of de-coupling components
- Other capabilities such as security, queuing, transactions
- With delegate-based events:
  - Client has to subscribe per event per publisher
  - Coupled life line
  - Cannot subscribe to type
  - No administrative setting of connection
- LCE lets you configure subscriptions
- LCE has separate life line
- Can subscribe existing objects as well

Loosely Coupled Events

- AKA COM+ Events



Loosely Coupled Events

- **`EventClass`** attribute denotes a managed class as an event class

```
public interface IMySink
{
   void OnEvent1();
   void OnEvent2();
}
[EventClass]
public class MyEventClass : ServicedComponent,IMySink
{
   public void OnEvent1()
   {
      throw new NotImplementedException(exception);
   }
   public void OnEvent2()
   {
      throw new NotImplementedException(exception);
   }
   const string exception = @"You should not call an event class
               directly. Register this assembly using RegSvcs /reconfig";
}
```

Loosely Coupled Events

- **`EventClass`** attribute constructor is overloaded

```
//These are equivalent:
[EventClass]
[EventClass(AllowInprocSubscribers = true,FireInParallel=false)]
```

- Publishing event

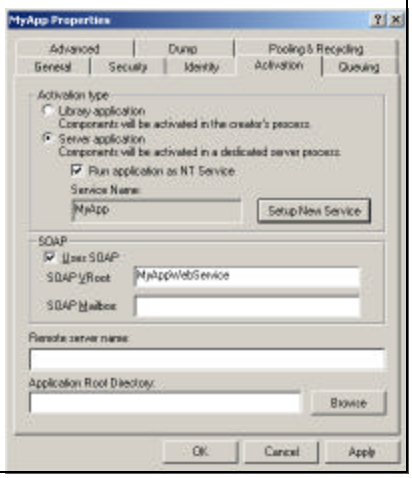
```
IMySink sink;
sink = new MyEventClass();
sink.OnEvent1();
```

Loosely Coupled Events

- Persistent subscribers
  - Get created to handle event
  - Admin support
  - Persist machine reboot
- Transient subscribers
  - Notifying existing subscribers
  - No out-of-the-box or admin support
    - Use my helper class
  - Gone after machine reboot

Web Services Support

- When set:
  - Use WS for all invocation
  - Depending on remoting call, can maintain state
- Great for migration
- Windows Server 2003 only



Resources

- Programming .NET components
  - By Juval Lowy, O'Reilly 2003
- www.idesign.net
  - Code library
  - Coding standard
- .NET Master Class
  - Regular and advanced
  - 3-4 annually
  - Upcoming events on www.idesign.net