Dura Trans Project

Architecture Report

Draft 1.0

By Stuart Snaddon

Outreal

Contents

[Project Overview 3](#_Toc221897554)

[Proposed Architecture 6](#_Toc221897555)

[Call Chains 11](#_Toc221897556)

[Publish Call Chain 12](#_Toc221897557)

[Distribute Call Chain 14](#_Toc221897558)

[Stream File Call Chain 16](#_Toc221897559)

[Packaging and Deployment 18](#_Toc221897560)

[Assembly Allocation 18](#_Toc221897561)

[WCF Services 20](#_Toc221897562)

[Run-Time Process Allocation 21](#_Toc221897563)

[Client Application Processes 21](#_Toc221897564)

[Service Allocation 21](#_Toc221897565)

[Security 22](#_Toc221897566)

[Identity Boundaries 22](#_Toc221897567)

[Identity Boundaries 23](#_Toc221897568)

[Authentication and Authorization 26](#_Toc221897569)

[Authentication Boundaries 27](#_Toc221897570)

[Authorization Boundaries 30](#_Toc221897571)

[Logical Thread of Execution 33](#_Toc221897572)

[Transaction Management 37](#_Toc221897573)

[Use-Cases 40](#_Toc221897574)

[Publication 41](#_Toc221897575)

[Distribution 42](#_Toc221897576)

[Summary 43](#_Toc221897577)

[Appendix A: Compensating Behavior 44](#_Toc221897578)

# Project Overview

This is a project that demonstrates how to build a service-oriented application that is a fully transactional, secure, scalable, durable (in the event of catastrophic failure) application architecture that leverages the power of WCF and MSMQ.

The durable aspect to this project is where it really begins to shine. Applications are usually built with the assumption that they are going to work. Unit tests mean the code does what it’s expected to do, and end-to-end testing proves the application works at run-time. But what happens if the server your application is running on crashes when it’s trying to process 50 concurrent transactions? Is the data lost? Is the system in an inconsistent state? Is there a contingency plan to automatically recover?

With a durable architecture transactions are fundamental to answering all of those questions satisfactorily. Transactions ensure the system remains in a consistent state. Transactions also work symbiotically with the durable MSMQ to guarantee that messages are not lost during transmission or incomplete processing. The MSMQ can be configured to allow a pre-determined number of attempts to deliver the message. Upon failure, a contingency plan takes over and the message is sent to a DLQ (or poison queue). The DLQ is simply a service that logs the failed message so it can be handled appropriately. All this takes place within transactions, and all this takes place automatically, with no complex plumbing in your code.

The resilience of the system is in its architecture, and the boundaries in which the services are contained. If the services are down (off-line for maintenance, re-starting, power failure, etc), this doesn't mean the system no longer works, it merely means that those particular aspects of it are unavailable for now. It's a completely distributed system where each service/client application can run independently of everything else. The client applications can still submit message for processing, and when the services come back on-line, processing continues as normal.

The project follows a closed architecture pattern which helps to reduce internal coupling and is achieved by making each internal component (Manager, BLL, DAL) a WCF service. The benefits of WCF can now be pushed down throughout the tiers with in-proc hosting.

For example:

* system.diagnostics can now be used to trace calls throughout an application and log errors, with no change to your code.
* WCF automatically propagates the client security context across service boundaries, with no change to your code.
* Services can participate in ambient transactions (where desired), with no change to your code.
* Services can be extended by adding attributes to the method/interface, with no change to your code.

What this now means is that developers can spend more time implementing business logic rather than plumbing. This where real productivity gains can be made, as developers are usually domain experts and not infrastructure (transactions, security, threading and synchronization) experts. What's more, is that the developers don't need to learn a myriad of technologies to do this, because if they know .Net then they already have the lion’s share of the skills required.

There’s no steep learning curve for implementing infrastructure anymore, as these aspects are now out-of-the-box features that the developer gets for free. Well almost for free, as it’s now just a case of the developer changing their mindset about how they build their application. Once you understand that these infrastructure aspects are now taken care of for you, there’s no need to re-invent the wheel by trying to program them yourself anymore. And when it comes to advanced and complex things to implement like distributed transactions and synchronization, you really don’t want to be doing it yourself.

DuraTrans will use this new application architecture from the ground up using .NET 3.0 and 2.0 technologies, MSMQ and Microsoft SQL Server 2005.

Motivations for the new architecture include making maintenance and enhancement of the application capabilities less costly and more agile, and to allow the application to scale to much larger number of users, and process higher numbers of concurrent transactions during the publication process.

DuraTrans will be designed as a Service Oriented Architecture, taking advantage of the capabilities of Windows Communication Foundation (WCF) for developing, deploying, and hosting services, to achieve better decoupling of different parts of the application infrastructure for maintainability reasons. Most of the services will just be consumed internally by the client applications or by other services in the architecture. As a result, the tenets of SOA will be followed as much as possible, but for internal services explicit compromises will be made for the purposes of efficiency in terms of the autonomy of the back-end data stores and the coupling of types in the clients and the services.

# Proposed Architecture

The static view of the DuraTrans architecture is shown in Figure 1. The major components of the system were identified through use case analysis of the top level use cases specified by the business analysts. These use cases (Figure 23 : Publication Use-Case and Figure 24 : Distribution Use-Case) support the overall workflow of publication processing. The static views identify the major system components needed to satisfy the use cases with an emphasis on separation of concerns, high cohesion, and low coupling to enable better maintainability and agility in development. The static views are technology independent and represent the logical structuring of the application into components and layers. A number of different development platforms could be used to implement the architecture without modification of the architecture diagrams.



Figure : DuraTrans Architecture Static View

The architecture follows the principals of layered application architecture. Different application concerns are separated into distinct layers of component to ensure high coherency with low coupling. The layers in the DuraTrans system include the following:

* Presentation Layer

Only UI focused code should be in the applications this layer. The client depicted in this layer, simply calls the publication service to trigger the publication process.

* Service Agent

This is the proxy to the appropriate service. Where multiple assemblies need to use the same service it can be a good idea to place the proxy code in a DLL which can be referenced where required. Any changes to the proxy can then be made in one place and the DLL distributed appropriately.

* Service Layer

In order to maintain a separation of concerns, the managers should be separate from the code that provides the service. This provides a very thin service layer that is completely independent from any workflow. This allows changes to the workflow to be made without affecting the service (interface, versioning, etc).

* Business Logic

Components containing application logic including business rules, and workflow go in this layer. Sub-tiers can include managers (for workflow), controlling access to the business logic components themselves.

* Data Access

Components that use data access providers in .NET to talk directly to a database go in this layer.

* Data

Databases and other persistent storage such as files go in this layer.

* Utilities

The components shown on the side of the diagram are not really a separate layer, but represent components that are used throughout multiple layers of the architecture.

In addition to being separated into logical layers, the application layers can also be treated as tier boundaries if it makes sense to separate them out to separate physical machines for scalability or security reasons. In the case of the DuraTrans architecture, the client will run in process on the client side, and the business, and data access layers will run on a middle tier application server. The servers hosting these services may be run as a farm to scale to large throughput demands.

There is a single client for the purposes of the DuraTrans application as identified by the use-cases.

The service layer components should not contain any business logic. They should just delegate down to the business layer components to get the real work of the use cases done. The service layer exists to isolate the business logic from needing to know about the service boundary specifics and they should be designed so they could be run in-process in the client applications if it makes sense to do so.

The business layer components are broken into two main types: manager and engine components. Manager components are focused on implementing the workflow or business process for the use case to which it is aligned and determine the API exposed to the services. The engine components encapsulate the business rules that support the business process. The separation is intentional for maintenance reasons. Business rules are more likely to change frequently than the business process. Keeping them isolated in their own component allows the developer to more easily locate and update the business rule code without needing to check in many locations throughout the architecture. The engine components can potentially be deployed and updated separately from the rest of the architecture to minimize the potential breakage of deploying updates. The implementation of manager components can be done with workflows, and the implementation of business rules can be done with rules within the workflow environment if the use case is complex enough to warrant implementation through a workflow engine technology such as Windows Workflow Foundation or K-2.

The data access components isolate the business layer from the details of data retrieval in the same way that the service layer isolates the business layer from the details of exposing remotely accessible services. No data access code should reside outside the data access layer. Specifically, in a .NET application, no ADO.NET types other than DataSets should be used outside the data access layer.

The data tier is depicted in Figure 1 as a single database and a file system. This too may be factored out as needed. To the extent possible without creating data duplication, the database schema should be designed so that each service has its own independent schema of information saved in the database to support the service. If the underlying tables are decoupled from other tables used by other services, then those tables and their supporting database objects could be factored into a separate autonomous database for the service. In the interest of efficiency, there are going to be a set of tables that represent shared schema across all the services (containing information about orders, clients, and other shared entity types. For these, a single common schema can be maintained that is shared across all the services. Although this does reduce the autonomy of the services, it will greatly reduce the data maintenance effort required compared to having each service contain its own copy of the shared data or the performance impact of always having to call a service to obtain the data through the service.

The utilities involved include class libraries for shared data entity / data contract definitions (one per service), a support library for providing diagnostics and end user support for deployed applications, and a utilities library for shared types that don’t belong in other libraries. The configuration service depicted in Utilities is really another service that gets called from all services and from the client machines to keep their configuration files up to date based on a centralized management system used by the configuration management team.

The factoring of components into multiple logical tiers will enable the following benefits in the context of the DuraTrans project:

Consistency

The middle tier will be able to consolidate multiple transactions initiated by the various business logic objects into a single, logical, distributed transaction, which spans multiple resources and machines.

Scalability

The middle tier will allow avoiding allocating a business object per client request. This will conserve system resources, and improve throughput.

Security

The middle tier will perform authentication and authorization of incoming callers, to verify the callers are allowed to perform the requested services. The middle tier components will run under designated identities, distinct from those of the incoming callers. This in turn will cater for resource pooling and scalability.

Separation of presentation from logic

This will allow decoupling of the user interface used to manage and invoke the services from the service management components. This in turn will enable using multiple presentation frameworks such as Windows Forms for rich clients, ASP.NET, web services or even mobile devices. Another benefit is that separating the user interface from the business logic will allow both to evolve and be maintained and extended separately.

Availability

The middle tier can use load-balancing servers to channel the load to available machines in a cluster. SO autonomous tenet.

Throughput

The ability to instantiate or use pooled objects for service execution will greatly improve the system throughput. Doesn’t get clogged up with IO from all layers.

Responsiveness

The combination of both just-in-time activation and object pooling will result with minimizing the response time per service invocation.

Synchronization

The middle tier will synchronize access to the underlying resources used for both DuraTrans management and for the services themselves. The synchronization used will be automatic via the transaction management and coordination. This will reduce the likelihood of deadlock due to multiple threads of execution accessing resources or objects, significantly simplify the application and reduce its time to market.

# Call Chains

The top level business process for processing publication tasks was decomposed into a sequence of individual business processes that implement the steps of stages of publication processing. These processes were used to identify the factoring into services represented by the vertical slices of the static view. Each of these was analysed to identify the basic call chains that would support those business processes to identify the relationships and interaction of components. The following sections describe the call chains that were elaborated on during the architecture engagement.

There are still any number of additional call chains that will be needed to support the full functionality of the application. Many of those call chains will be simple data retrieval or data persistence call chains that look the same at the architecture level for each call chain, even though a different set of components will be involved depending on the specific part of the business process and the entities that are being manipulated.

## Publish Call Chain

Figure 2 shows the call chain for the publish task use case. This is the use case that kicks off the overall workflow for handling the publication process. The publication request comes from the client application. The entry of a publication request starts the workflow that manages the rest of the business process.

The PublicationManager validates the publication task is in a publishable state (Unpublished or In queue). The complete list of possible states is:

* Cancelled
* Unpublished
* In queue
* Published
* Error

If the PublicationManager declares the publication task valid and in a publishable state, the PublicationManager sets the state of the task to In queue by calling the TaskDAL component. Status and state in the database are updated so that subsequent steps in the process, which are driven by lists of orders in a given status, can be performed. The status and state of the order can be maintained by using workflow tracking mechanisms if implemented using workflow technology such as K-2 or Windows Workflow Foundation.

The collection of publishable items related to the task is retrieved from the database. The PublicationManager will then call the DistribtionHubManager service for each item to move the workflow on to the next stage using a queued call. The use of queued call facilitates the on-line/off-line nature and guaranteed delivery of messages. Once the publication item message has been placed, the PublicationManager proceeds to queue the next publication item until all items are on the queue.

The workflow then validates the publication status of the current task. If there are still items related to the task that do not have a published status, then the transaction is blown leaving the message back on the message queue for a further processing attempt. This processing message loop continues until the workflow decides the task is published, or the current message is deemed to be a poison message or moved to the DLQ. Poison messages or dead-letters are handled by the appropriate message handlers as part of a compensation plan for failed publication processing attempts. Compensating behavior is described in more detail in Appendix A.

If the task has no outstanding items to be published, then the task status is set to published and the current transaction is allowed to commit successfully. This results in the publication task message being removed from the message queue. At this point the task will have been successfully published.



Figure : Publish Call Chain

## Distribute Call Chain

The next stage of the workflow is triggered by a message being taken off the Distribution message queue, Figure 3. The message simply contains the ID for the publication item to be distributed from its source to its destination. This process allows the workflow to transition to the next milestone in the publication process.

Initial validation checks are performed to make sure the item to be published is in a publishable state (Unpublished or In-Queue). If the item is valid, its status is set to “**In-Queue**” to indicate that the item is to be published when resources are available. It is questionable that another status (In-Progress) could be used to indicate that the item is actively being distributed at this point in time. However, if the distribution process fails the status needs to be switched back to In-Queue. To avoid this switching, the In-Queue status will suffice. This part of the workflow needs further analysis to determine if the status switch is required to help with workflow processing.

Due to the unknown size of the physical asset to be published, streaming is used to keep processing overhead to a minimum. The stream can be configured to make best use of available bandwidth from the source server, without consuming too much memory. This can be coupled with load-balancing to reduce bottlenecks and create a free-flowing system.

Transactions are used once again to guarantee message delivery, and allow for retry attempts should system resources not be available. Should the attempt to distribute item fail, the transaction is blown up and the message is moved back to the message queue.

If the item is successfully distributed to its destination the DistributionHubManager will move the workflow on to the next step in the processing and will persist the change of status to the database by calling the ItemDAL component. Successful publication will set the status to Published and the transaction is allowed to commit, thus removing the message from the message queue.

Compensating behavior is facilitated by the transactional nature of the MSMQ and can be configured to allow different behavior depending on system requirements. Compensating behavior is described in more detail in Appendix A.



Figure : Distribute Call Chain

# Stream File Call Chain

Figure 4



Figure : Stream File

Notes

Describe the following:

How the service is accessed

* Clients
* Workflow
* Notifications
* Call types
* Calls to other services, components

Process flow

* Steps
* Workflow transitions

Other

* Data access

# Packaging and Deployment

An important aspect of the architecture is the physical packaging of the logical components and how they will be deployed to different services or processes. The following sections describe those aspects.

## Assembly Allocation

The logical components identified through the use case analysis need to be developed and compiled into a deployable assembly. Each logical component in the architecture diagrams will likely be factored into multiple real classes as the development of the detailed use cases progresses. The assembly boundary in .NET maps one-to-one with a Visual Studio project, and is the boundary at which security and versioning can be applied.

The assembly allocation of these components is almost a one-to-one location where each component becomes an assembly container for those classes. There are a few deviations from that however, described below. Figure 5 shows the assembly allocation of the vertical slices of the system.



Figure : Services Assembly Allocation

The assemblies are still broken into their respective layers vertically, and separated across vertical slices horizontally so that different services can be deployed to different servers easily. A WCF service is composed of a service contract, a service implementation class, and data contracts used by the service contract. Because most services will be consumed internally by Customer, the service contract and data contract types are shared between the client and the service. As a result each vertical slice has its own entities DLL (depicted as a single DLL for all at the bottom of Figure 5) which contains the data contracts and a separate service interfaces assembly.

## WCF Services

Each of the back-end services will need its own host process, although multiple services could be hosted in the same process if desired. The service boundary contains all the components needed to support that service. The services can be hosted in IIS6 for the best manageability, but will only be able to use HTTP bindings if so. To use other bindings, such as the queued MSMQ binding chosen for the publication and distribution services when developing the vertical slices, the service will have to be self-hosted in a custom .NET process such as an NT service. Figure 6 shows the service boundaries and the assemblies that will run within them to support the use cases diagrammed during the architecture engagement.



Figure : Service Boundaries

# Run-Time Process Allocation

## Client Application Processes

The client application needs to run a number of components in process to support the UI functionality and to make remote calls to the services. Figure 7 shows the process boundaries of the client application and the assemblies that are loaded into those processes.



Figure : Client Application Process Boundaries

## Service Allocation



Figure : Service Process Boundaries

# Security

Important architecture aspects regarding security include where the identity boundaries exist, where authentication and authorization are performed. These aspects are described in the following sections.

## Identity Boundaries

Each process on a Windows system has an identity under which it runs. Additionally, each thread in .NET can take on a distinct identity. Depending on the impersonation settings, those identities can flow out of process when remote calls are made and can be used for authentication and authorization on the receiving side. They can also be used to restrict access to resources that the program calls out to (such as the file system or registry) from within the process.

Figure 9, Figure 10 & Figure 11 shows the identity boundaries for all of the call chains. Each of the client processes run with the logged in identity of the user. The windows client application will run under the logged in Windows identity of the user. In the case of the Customer Web Application, the process identity will be the identity of the ASP.NET worker process configured in IIS when IIS hosted, or the configured identity of the NT Service when self-hosted. However, the logged in identity of the user can also be captured by the Web application through username/password authentication or through Windows integrated security if they are calling from within the domain tree of the network. This identity is used to service the request within the web application and can be used to restrict resource access from within the web application.

Each back-end service will be configured to run under a distinct middle tier identity so that identity can be used for authentication against the database and other internal services. The database runs in its own separate identity, as do the other services that other call chains use as part of their processing.

## Identity Boundaries



Figure : Publish Call Chain Identity Boundaries



Figure : Distribution Call Chain Identity Boundaries



Figure : Stream File Call Chain Identity

## Authentication and Authorization

Authentication is the process for identifying the caller and verifying that they are who they say they are. Authorization is the process of granting access to resources or operations based on that known identity. Users will have to authenticate with the client or web applications to use them. In the case of the internal client application, this is done automatically by Windows – the process starts under the logged in identity of the user. For the web application, a number of options exist including Windows integrated security, username/password authentication against a custom store, or using CardSpace and federated identities.

Authentication should be performed at each process boundary crossing. This includes coming into the client applications, as well as entering each service and entering the database. Authentication at the service and client boundaries will likely be done with federated identities based on discussions with the architecture team. Authentication at the database boundary will be done using Window integrated security based on the configured middle tier identities of the services.

Authorization can be done separately at different layers in the application where appropriate based on the identity of a caller, even across different layers within a process. However, in the client application it should be sufficient to apply authorization access controls primarily at the entry to each process, at the same location as authentication. As a result, separate graphics are not used to depict the authorization boundaries.

Figure 12, Figure 13 & Figure 14 show the authentication boundaries, Figure 15, Figure 16 & Figure 17 the authorization boundaries for the DuraTrans call chains.

### Authentication Boundaries



Figure : Publish Authentication Boundaries



Figure : Distribution Authentication Boundaries



Figure : Stream File Authentication Boundaries

### Authorization Boundaries



Figure : Publish Authorization Call Boundaries



Figure : Distribute Authorization Call Boundaries



Figure : Stream File Authorization Call Boundaries

# Logical Thread of Execution

In all workflow there is cause and effect, and every action has a traceable root cause. This root identifies the logical thread of execution through the process.

Any re-entrant cyclic path implies:

* Deadlock
* Poor design and re-entrancy
* Need for queuing
* Need for asynchronous event publishing

The causality for each use-case can be seen in Figure 18, Figure 19 & Figure 20.



Figure : Publish Causality



Figure : Distribute Causality



Figure : Stream File Causality

# Transaction Management

In a complex system, transactions are essential for ensuring the integrity and consistency of the data modified by the call chains. Directly managing the transactions is error prone and extremely labor intensive, if not impossible. The transaction support in the .NET Framework and WCF make this considerably easier. Decisions still need to be made on where to apply transactional boundaries and when to span those transactions to other services. The decision includes not only what call chains to include in a transactional scope, but also what level of isolation to use for that transaction. The default in .NET is a Serializable transaction, which provides the maximum protection for consistent data resulting from a transaction. However, Serializable transactions also use the most locking at the database level to ensure that integrity so will have throughput impacts. Those impacts are generally best addressed by scaling out the application as needed for throughput rather than relaxing the isolation unless detailed analysis of each call chain is done by data access experts. Certain calling patterns, such as a read-only SELECT style call chain do not need to participate in a transaction. But if there is any potential for database modifications to be made as part of the overall business process of a call chain, then starting with a serializable transaction encapsulating all the activity of the call chain is the safest starting point. Through analysis, the decision to back off on the span of the transaction scope or the isolation level may be made if performance challenges dictate and the query pattern is permissive. These have to be treated on a case by case basis.

Figure 21 & Figure 22 show the transaction boundaries for each of the DuraTrans use cases. Where transactions were deemed appropriate, they are started at the entry to the service. Some call chains include flowing the transaction to other service calls. This is done when the call to the other service will include persistence of information passed through the call chain to ensure that the persistence is rolled back if something goes wrong with the completion of the call chain on the originating service. Additionally, anywhere that calls are made as queued calls to the Publication or Distribution services, the delivery of the message into the outgoing queue is performed as part of the transaction by the calling proxy, but the delivery of the message will not be part of the originating transaction.



Figure : Tranaction Boundaries For Publish Call Chain



Figure : Transaction Boundaries For Distribute Call Chain

# Use-Cases

The following use-cases as shown in Figure 23 & Figure 24 were identified after analysis of the clients requirements.

## Publication



Figure : Publication Use-Case

## Distribution



Figure : Distribution Use-Case

# Summary

This architecture report provides a summary of the architecture decisions made, diagrammed and discussed during the architecture engagement with the customer in August 2008. Each architectural aspect was discussed and the resulting diagrams for the call chains presented. These aspects are important for allocating the functionality of the system to the appropriate set of components and assemblies to maintain loose coupling of the business functionality of the system. Adhering to this approach will help ensure that future changes to the business logic and functionality of the services can be evolved more rapidly with minimal impact on other parts of the system.

This system demonstrates how to build a fully transactional, durable, partially connected (on-line/off-line) and multi-threaded system (using no explicit threads).

This architecture report is uses the iDesign Method to capture critical design decisions as taught on the Architects Master Class. For more information about this methodology visit the <http://www.idesign.net> web-site or download an overview of from:

<http://www.idesign.net/idesign/download/IDesignMethod.zip>

The complete solution source code and documentation is available for download. Send an email to [stuart@snaddon.com](mailto:stuart@snaddon.com) for download details.

# Appendix A: Compensating Behavior

Compensating behavior is an important aspect for any system that is often overlooked and neglected. The importance of compensating behavior becomes more evident when building on-line/off-line systems. A system that is on-line and/or connected can receive feedback from components as to whether or not the call was successful. In a partially connected system, the opportunity to send feedback or handle a failed message isn’t always available.

What you need in this scenario is a guaranteed technique for handling failed messages. MSMQ 4.0 attempts to deliver messages in a queue for the specified number of times (receiveRetryCount). If the message cannot be delivered it is moved to the retry subqueue. After a configurable delay, the message is then moved back to the application queue when another attempt to deliver it can be made. This later time should be configured to allow the system to recover before further delivery attempts for the failed message are made. Failed message delivery doesn’t necessarily mean the service is down, just that the message could not be delivered. There are a number of reasons for this, including specific resources required by the message not be available at the point in time.

If the message still fails to be delivered after the configured maxRetryCycles value, it is removed from the retry queue and sent to a DLQ. Messages that end up in the DLQ are effectively never going to be processed as part of a successful use-case. This is not a catastrophic failure, as the DLQ is still a durable and transactional MSMQ. This means the dead message is never lost and can be retrieved by some compemsating behavior logic. This may in turn make available information about the status of the system for the client to consume. This means the system, even in failure, is still partially connected and access to resources can resume when on-line again.

More details for configuring MSMQ compensating behavior can be found at <http://www.outreal.com/CompensatingBehavior.html>.