Group Technology and Support Services

Technical End to End Design Specification

CCPS (Convergence) - TED - Infrastructure

Document Version: 0.9
Document Version Date: 9 Feb 2005

TABLE OF CONTENTS

1 DOCUMENT CONTROL

1.1 Sign-Off

	OWNER / AUTHOR	REVIEWER	AUTHORIZER
Name Designation Area Date	Marius Snel Technology Architecture	Kim Muller Technology Architecture	
Name Designation Area Date	Louis Steyn Technology Architecture	Philip Putter Technology Architecture	

1.2 Version Control

VERSION	DATE	CQ #	AUTHOR	SECTIONS	CHANGE
0.2	2005-01-07		Marius Snel		Created doc.
0.9	2005-02-04		Louis Steyn		Internal version for review by architecture and operational stakeholders. Feedback will be considered and the document will be updated accordingly before any further releases.

2 INTRODUCTION

2.1 Document Background

Architecture has been working on application architecture for service-orientation for some time now. The channel convergence project has been selected as a significant candidate to start implementing this architecture. Having a separate TED just for application infrastructure shows the importance of this decision. The designs in this document fundamentally change the way applications will be designed going forward. It departs from the conventional Nedcor application architecture and specifies a new way to design applications that will eventually take the company's systems towards a service-oriented architecture.

2.2 Document Purpose

This document details common and re-usable application infrastructure portions of the solution for the self-service channel convergence project. The different TED documents that will be developed for each of the channel convergence DRS documents will reference this TED, as it will contain application infrastructure design that should be common to all the TED documents. The intention is to document the application infrastructure part of the solution to a sufficient level for detail design to commence. During detail design this document will be re-factored if and when significant architectural problems arise that cannot be resolved.

The key purpose of this document is to:

- Elaborate on the key architectural principles that governed the decisions in this document.
- Explain and elaborate on Service-Orientation and Services and the rules that govern it.
- Elaborate on the key architectural patterns to be applied by both the infrastructure and the components that makes use of it.
- Set the boundaries for detail design of the individual architectural building blocks.
- Describe the interfaces between the architectural building blocks comprising the solution in terms of flow and data.

2.3 Document Scope

This document is concerned with the high-level end-to-end solution design for the common application infrastructure aspects within the context of the self-service channel convergence project.

Any non-infrastructure application components will be addressed as part of other TED documents and have been explicitly left out of this document.

Being a high-level design, only architecturally significant packages (package as defined in UML) are included, with special attention given to the relationships between these packages. Where components are defined within a package, their interfaces and interactions will also be defined at a high level. Where applicable the document will also provide guidance to authors of detail designs.

NOTE: Although this document is concerned with the self-service channel convergence project, the designs it contain are re-usable and should be used in TED and detail design documents for all projects going forward.

2.4 Definitions, Acronyms and Abbreviations

BRS	Business Requirements Specification
DRS	Detailed Requirements Specification
DSF	Distribution Services Framework
UML	Unified Modelling Language
TED	Technical End-to-End Design
EAI	Book -> Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions
GoF (Gang of Four)	Book -> Design Patterns, Elements of Reusable Object-Oriented Software

2.5 References

Document	Version	Date	Author	Organisation
Design Patterns, Elements of Reusable Object-Oriented Software			GoF	
Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions			Gregor Hohpe Bobby Woolf	
Distribution Services Strategy Future View	6.0	September 5, 2002	Martin Krsek	Nedcor
Architectural Principles	6.0	June 2, 2004	Les Williams	Nedcor
Application Instrumentation Architecture.doc	1.0	June 4, 2004	Louis Steyn	Nedcor
TED – Convergence – Security And Profile v0-1c.doc	0.2	February 8, 2004	Beyers Coetzee	Nedcor
A lightweight conversation controller	1.0	April 4, 2001	HP Labs	HP
WSCL description	1.0	March 14, 2002	HP Labs	W3C, HP

2.6 Overview of the Document Structure

This document is ordered and structured as follows:

- An introduction -> this section.
- A project overview is provided including the high-level project scope and various stakeholder objectives.
- A description of the architectural requirements for this document.
- A description of the design requirements for this document.
- An overview of key concepts that are used throughout this document.
- A description of the End-to-End Technical Infrastructure Context.
- The logical view of the infrastructure solution. This section describes the key components of the infrastructure and the relationships and interactions between them.
- The physical view of the infrastructure solution. This section is concerned with the physical infrastructure required to support solution functionality.

3 PROJECT OVERVIEW

3.1 Project Overview and Background

Currently, Nedbank offers a number of different electronic banking solutions to its clients. In the Business and Corporate markets these solutions include NedInform, NedExec, Netbank, NedTreasury, CPS, EPS, NedAcad, Probanker and Corporate Saver.

Having as many different solutions in the market presents problems to both Nedbank and its clients. For the bank it is both difficult and expensive to maintain as many systems and infrastructure components. Staying compliant to industry rules requires development on multiple systems and therefore time to market is slow. Creating new products and feature in this environment presents the same problems. For the bank's clients it is inconvenient to run multiple systems and interface with multiple support areas to address their electronic banking. Service levels are consistently below optimal due to the multiple systems being supported causing client dissatisfaction.

A proposal has been put forward to, over time, converge all self-service channels onto a single front end and to rationalise the supporting back-end profile and payment systems onto a single supporting application infrastructure. Another initiative is also currently underway to develop a similar strategy for staff-assisted channels. It has been proposed that the Netbank system is evolved to cater for all market segments.

3.2 Project Scope

The scope of this phase is to detail the application infrastructure for all the components that is needed to complete the first business release of the self-service channel convergence project.

3.2.1 In Scope

- Develop an enterprise framework to enable service-oriented integration.
- Develop a touch-point layer as described by the DSF.
- Re-factor the current Netbank application server software to flow the enterprise context.

3.2.2 Out of Scope

The following is not within the scope of the Release 1 project:

- Develop interaction layer / tier.

3.3 Project Stakeholder Objectives

The various key stakeholder objectives of this project is still under discussion and this document will be updated as soon as it is finalised.

4 ARCHITECTURAL REQUIREMENTS

4.1 Adherence to Architectural Principles

All architecture principles are well documented and published. This section will not elaborate in the same detail, but just refer to the principle and how they drove decisions in this document.

Principle	Reference	Relation to this TED
The primacy of architecture principles.	5.1	• All decisions in this document are as a direct result of the architecture principles. • These principles are accepted at management level and are not negotiable. • All principles must be adhered to without exception.
The architecture must have a planned evolution that is governed across the enterprise.	5.2	• This document detail the full architecture needed to complete the self-service channel convergence project. • It allows for a staged implementation and does not require all parts of it to be implemented simultaneously.
Accountability of assets.	5.3	• This document explicitly draws the boundary between application infrastructure and the application itself. • It also details application infrastructure and design that span systems and projects. • The accountability for application infrastructure design should be with architecture.
Standards based.	5.4	• The design patterns in this document are based on industry accepted open design patterns. • Suggested mechanisms and standards are also based on current industry standards. • Detail designs based on this document must ensure that compliance with industry standards is an ultimate goal.
Interoperability.	5.5	• A large portion of this document is dedicated to integration based on industry accepted design patterns. • Once these design patterns are implemented across the enterprise, interoperability will be much easier to achieve, both internally and externally to our organisation.
Operational effectiveness.	5.6	• The design patterns used in the document aims to make the application infrastructure and the applications that run on top of it, more maintainable and easier to extend. • Although the extra features will use resources, it will be limited and refined to be as effective as possible. To understand the effectiveness of the application infrastructure part, one need to measure its resource usage against a currently accepted alternative.

Principle	Reference	Relation to this TED
Adding value.	5.7	• This principle clearly states that any application that is developed must add explicit value to the stated business requirements and objectives. • This document and the designs in it do not add direct business value, because the components in these designs do not operate on business entities and processes. They rather direct the design patterns of other application, which do add direct business. • The indirect value that this infrastructure design will deliver, when it is fully implemented and utilised by business applications, is: o Less complex systems architecture. o Applications will be easier to maintain. o Broader re-use of business applications. o Changes to business applications and processes will become easier to introduce.
Systemic view of systems.	5.8	• By drawing explicit boundaries around systems and specifying the integration patterns and infrastructure between them, this document will enable a planned evolution towards a stable core for both infrastructure and the application services that utilise it. • End goal 1: Explicitly defines system boundaries. • End goal 2: Establish a stable core infrastructure for service oriented design patterns to be implemented. • End goal 3: Establish stable core sets of services on top of witch other services and process can be built. • These three goals reaffirm the fact that this document takes an enterprise view, rather than a project specific one.

Principle	Reference	Relation to this TED
Application architecture.	5.9	• This document is based on current theory around service-oriented architecture and it details a new design approach and patterns for implementing services. In a sense it paves the way for service-oriented architecture to be adopted into the organisation. • The design for infrastructure and the definition of the boundary between application infrastructure and application helps with proper layering of both applications and the infrastructure that they run on in each tier of the enterprise architecture. This architectural layering model must be kept constant across all tiers to ensure proper consolidation of the design patterns that they comprise of. • Most of the designs in this document is based on industry accepted design patterns, that have been proven to a certain extent by multiple vendors. • The proper application of these patterns should enhance re-use and the development of re-usable components as long as all projects adhere to these principles and the design patterns used in this document. • Developing applications as services implicitly make them event driven as prescribed by this principle. Operations on services can only be invoked through events (more specifically messages). • Service characteristics also dictates the way parameterisation works. Parameters for service operations can have only two sources: o In the message event that caused the invocation of the service operation. o Internal to the encapsulation of the service interfaces (call to other service, file, database or other internal application state).
Data is an asset.	5.10	• A large part of the data design in this document is centred on an integration design pattern called the “Canonical Data Model”. This pattern governs the way integration data are defined and enable the re-use of these definitions with an enterprise centric view in mind. Therefore, once application data has been defined using this pattern, it becomes a re-usable asset for the whole enterprise to consume and use. • In accordance with the guidance in this principle, this document states that the architecture department governs the enterprise “Canonical Data Model”. • Applying this pattern must implicitly create common data vocabulary, rules and definitions.
Architecture driven organisation.	5.11	• In this document architecture details the new way to design applications that will enable us to migrate towards a service-oriented architecture.

4.2 Architecture Objectives

1. Adhere to architecture principles. This must drive all decisions taken throughout the project.
2. Consolidate application design patterns. Having fewer types of application designs make the system landscape less complex. This allows for easier introduction of changes and enhances maintainability.

3. Consolidate application integration patterns and mechanisms. Fewer integration patterns and mechanisms will allow for quicker introduction of changes and will enhance maintainability.
4. Establish looser coupling at architectural level of decomposition as a minimum. Loose coupling reduces the amount of dependency and level of assumption between system components at the cost of overhead. It helps when parts of the architecture need to be evolved or replaced. This will also make changes easier to introduce and contain the effect of change. Further design should not introduce additional loose coupling without architecture's consent.
5. Establish enterprise re-usable application infrastructure. In this context re-use is defined as the exact opposite of duplication. When there is a large amount of duplication in systems, changes typically require huge effort. Generally changes can be introduced easier into an environment with a higher level of re-use. The same argument holds true for maintainability.
6. Ensure that the application infrastructure is easily replaceable with the minimum effect on the applications themselves.

4.3 Lessons learnt from past experiences

The following lessons are pertinent:

1. Do not re-invent the wheel. Where designs, design patterns and implemented components can fit the architecture, make use of them. In principle it still stays an architecture decision.
2. Exposed application functionality must not be developed with only the scope of the current project in mind, but with enterprise re-use in mind.
3. Loose coupling should exist between layers, tiers and systems. Loose coupling reduces the amount of assumption between these architectural parts and as a result enable the IT organisation to evolve them separately while limiting the impact on other parts.
4. Usage patterns that force multiple calls across platform boundaries to resolve a single request have severe and adverse consequences for performance. Patterns of usage should be designed to optimise this.
5. Over parameterisation makes testing very complex and expensive. It also makes the system very difficult to set up and in the end limited functionality may be used due to complex set up requirements.

4.4 Architecture Requirements

4.4.1 Supplementary Requirements for the Application Infrastructure

From an architectural perspective the supplementary requirements can be divided into a number of categories. The following categories and corresponding requirements must be addressed by the solution:

1. **Availability:** The new converged self-service channel is a critical component of the bank's future architecture. The solution implemented must not compromise the availability of the banks systems i.e. the solution should not be so rigid that a failure in one of its components (single point of failure) could prevent the IT infrastructure from supporting the Banks core business. This channel and the infrastructure it is built on must be available 24/7. It is appreciated that current back-end functionality might not have the same level of availability.
2. **Scalability:** The new converged self-service channel should support the seamless scaling without the need for costly development, testing, and business re-engineering. It must support separate scaling for different architectural tiers and should lean towards a "scale out" rather than a "scale up" model, since scaling out will enhance availability as well.
3. **Flexibility:** The design and implementation of the application infrastructure should cater for sufficient extensibility points. In addition, parameterisation should be used to enable adjustment of important aspects of the application infrastructure. One caveat: With increased flexibility comes increase complexity and cost. This must be considered when designing and implementing the solution i.e. tradeoffs will be required.
4. **Maintainability:** The application infrastructure and the applications that run on top of it must be easy to understand and maintain.

5. **Performance:** The design and implementation of the solution must ensure that the current performance criteria are not significantly adversely affected.
6. **Security:** see the Security TED for full details.

4.4.2 Architectural non-Negotiable aspects for the ultimate infrastructure

The following are architectural non-negotiable aspects:

1. All architecture principles.
2. Application infrastructure may not be part of the application that runs on top of it and must be seen as a separate set of applications altogether. Preferably its delivery must be run in a separate project.
3. All definitions of key concepts.

5 DESIGN REQUIREMENTS

5.1 Design Patterns

Two types of trace-ability attributes are listed for each pattern. One points to other patterns discussed in this document and the other points to Architecture Objectives under the Architectural Requirements section.

Pattern	Reference	Rationale
Messaging	EAI	Answers the following question: • How can I integrate multiple applications so that they work together and can exchange information? This is not actually a pattern, but more an integration style. This whole document and application infrastructure design is based on this style of integration to enable proper service oriented integration. Only evaluate the other patterns and therefore the design in this context. Other integration styles noted in EAI are File Transfer, Shared Database and Remote Procedure Call. The main reasons the Messaging style was chosen for this design are: • It can be platform independent and therefore be more flexible. • It supports both synchronous and asynchronous semantics. • It can support multiple dialog patterns i.e. request-reply or one-way. • Most major vendors support this style of integration. Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Canonical Data Model	EAI	Answers the following question: • When designing several applications (consumers and services) to work together trough Messaging where each application has its own internal data format, how can you minimise dependencies when integrating these applications? This is the main information integration pattern utilised by this design. Defining information in a Canonical Data Model makes it more re-usable since messages are built up using a set of common definitions and data types. It also enhances platform independence, because the definition of the information is not based on any specific platform, but rather on the needs of the enterprise as a whole. It enhances loose coupling, since applications only need to support translation between its own internal representation and that of the Canonical Data Model, without ever requiring any knowledge of the internal representation of any other application. Together with the Message Bus pattern, this makes for a very scalable integration infrastructure. This pattern must apply to all interfaces / messages between systems. Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Command Message	EAI	Answers the following question: • How can messaging be used to invoke a procedure in another application? This pattern is introduced to assist with implementing Services. Services expose contracts. Part of the contract is a lists of the operations it support with the [in], [out] and [resulting] parameters that these operations require and provide. When a message is dispatched on the Message Bus, its destination is typically a Service or a set of Services. When interpreting the message, a Service needs to know which operation it is destined for. This pattern solves this problem by embedding the logical name of the operation into the actual message. There need not exist a direct mapping between the logical operation name and the physical operation name. This is an implementation detail and the Service Activator and Message Translator patterns can be used to resolve differences if they exist. This pattern is used to support the following patterns: • Canonical Data Model • Messaging Gateway • Service Activator • Message Translator • Envelope Wrapper Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling.

Pattern	Reference	Rationale
Messaging Mapper	EAI	Answers the following question: • How do you move data between domain entities (Application specific) and the messaging infrastructure (Canonical) while keeping the two independent of each other? The messaging infrastructure in this design exposes Message Endpoint components to systems. The two specifically chosen in this design is the Messaging Gateway on the consumer's side and the Service Activator on the Service's side. This pattern does NOT mean that the mapping takes place from domain specific entities to the message. The mapping is from domain specific form to Canonical form. Mapping of the Canonical form to the actual message is the responsibility of the Message Endpoint. To ensure that the implementation details of the messaging infrastructure is properly encapsulated behind the interface of the Message Endpoint, the Endpoint exposes the Canonical form of the message, but in the environment of the application. This has the effect that the Messaging Mapper maps between domain specific form and Canonical form where the mapped entities reside in the same environment. In a .Net / Java environment, both the Canonical and domain forms of the information will be objects. In COBOL, both will be copybook layout structures. Therefore the application has no knowledge of the message infrastructure, semantics or format. Message mapping as it is described here is an application responsibility, while the generation of the Message Endpoints, is an infrastructure responsibility. Message Endpoints are deployed separately form applications. This pattern is used to support the following patterns: • Canonical Data Model • Service Activator • Messaging Gateway Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Envelope Wrapper	EAI	Answers the following question: • How can existing systems participate in a messaging exchange that places specific requirements on the message format, such as message header fields or encryption? The main purpose of this pattern is to enable us to solve both functional and non-functional informational requirements using the same Messaging infrastructure, without compromising loose couple between them. This design suggests that the message envelope contain a header part and a body part. The header is mainly used to transport information that is used by the infrastructure and infrastructure services. Examples of this type of information are compression data, encryption data, quality of service parameters, routing information and security information. Basically the possibilities are endless and one can start with a limited subset and migrate to include more complex non-functional requirements. The body part is used to transport a Command Message, which in turn contain the information needed to complete the functional requirements of the message exchange. The primary principle is that application infrastructure handle and encapsulates the information in the header and that the application handles the Command Message. Any information in the header is only accessible to the application through a set of API's that the infrastructure provides. It must be noted that the infrastructure will need to manipulate the body for utility processing like compression or encryption, but the emphasis is on manipulate and far less on change. For actual changes to the content of the Command Message, the infrastructure should rather provide hooks into the message-processing pipeline, where application supplied components can be invoked to make the changes. By viewing it in this manner, encapsulation and separation of concerns are not compromised. This pattern is used to support the following patterns: • Canonical Data Model • Service Activator • Messaging Gateway • Message Bus • Pipes and Filters Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Message Bus	EAI	Answers the following question: • What is an architecture that enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others? This pattern is a specialisation of Messaging Channel and it is the closest fit to the DTSP architecture. Applications bind to the Message Bus using a Message Endpoint. Other patterns are implemented between Message Endpoints and are hidden from the application. The net effect is that the application binds only to its local Message Endpoint. It is important to keep in mind that there might be more than two Endpoints involved in certain messaging patterns. For example, a message might have multiple destinations or pass through an intermediary processor like a Messaging Bridge. This pattern is used to support the following patterns: • Channel Adapter • Messaging Bridge • Messaging Gateway (Message Endpoint specialisation) • Service Activator (Message Endpoint specialisation) Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Channel Adapter	EAI	Answers the following question: • How can you connect an application to the messaging system (Messaging Channel and in this case more specifically the Message Bus) so that it can send and receive messages? This pattern shields the rest of the design from the intricacies of the lower five layers of the 7 Layer OSI model for communications between applications. It implements the Send and Receive logic for messages in a consistent way, whatever the underlying communications infrastructure. Examples of protocols that are hidden behind this pattern are: • TCP/IP Sockets • UDP/IP Sockets • HTTP (S) • SMTP • Message Queuing • Named Pipes • Shared Memory • Memory Mapped Files • Custom This pattern is used to support the following patterns: • Message Bus • Service Activator • Messaging Gateway • Messaging Bridge Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Messaging Bridge	EAI	Answers the following question: • How can multiple messaging systems (more correctly, the Messaging Channels they contain) be connected so that messages available on one are also available on the others? A message bridge is basically a pair of Channel Adapters and a Message Translator that work in union to Bridge messages between Messaging Channels. It must be viewed as an evolutionary pattern to help bridge the period while middleware is not yet mature on all platforms. This pattern is used to support the following patterns: • Message Bus (in this case multiples of them) Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.
Message Translator	EAI	Answers the following question: • How can systems (Message Bus's in the case of this document) using different data formats communicate with each other using messaging? In this design the Message Translator pattern is optional, since it is only necessary if we implement more than one Message Bus. We need the same capability on all platforms in terms of Messaging to eliminate this pattern. It must be viewed as an evolutionary pattern to help us bridge the period while middleware is not yet mature on all platforms. This pattern is used to support the following patterns: • Messaging Bridge Architecture objectives solved by this pattern: • Adhere to architecture principles.

Pattern	Reference	Rationale
Messaging Gateway	EAI	Answers the following questions: • How does an application connect to a messaging channel to send and receive messages? • How do you encapsulate access to the messaging system from the rest of the application? This pattern is a specialisation of the Message Endpoint pattern. This is the only messaging infrastructure pattern visible to Consumer type applications. It acts as the de-coupling point between Consumer applications and the messaging infrastructure. It is the main controller pattern that drives the sequencing and operations of the other patterns on the Consumer side. It works in union with the Service Activator pattern that performs a similar role on the Service side. Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.
Service Activator	EAI	Answers the following questions: • How does an application connect to a messaging channel to send and receive messages? • How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques? This pattern is a specialisation of the Message Endpoint pattern. This is the only messaging infrastructure pattern visible to Service type applications. It acts as the de-coupling point between Service applications and the Messaging infrastructure. It is the main controller pattern that drives the sequencing and operations of the other patterns on the Service side. It works in union with the Messaging Gateway pattern that performs a similar role on the Consumer side Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.

Pattern	Reference	Rationale
Pipes and Filters	EAI	Answers the following question: • How can we perform complex processing on a message while maintaining independence and flexibility? This pattern is a flexibility pattern that helps to introduce new ways of non-functional message processing, which was not originally intended or known, in a non-intrusive way. It basically introduces a pipeline that manipulates incoming and outgoing messages before it leaves a Message Endpoint (either on the Application side or the Message Bus side). It is common that pipeline processing done on the one side of a conversation, is undone or reversed by the pipeline on the other side of the conversation. Examples of this kind of behaviour are context serialisation/de-serialisation, compression, encryption and digital signing of messages. It is imperative to note that this pattern is introduced inside the application infrastructure to solve non-functional type processing. Functional processing is part of the application domain and MUST NOT be seen as part of the pipeline discussed here. This does not mean that applications cannot implement the same pattern to solve functional type processing, but that would be in a different context and outside the boundaries of this application framework. This restriction is to ensure proper encapsulation and de-coupling. This pattern is used to support the following patterns: • Messaging Gateway • Service Activator Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Consolidate application integration patterns. • Establish looser coupling. • Establish enterprise re-usable application infrastructure. • Ensure that application infrastructure is replaceable.
Factory	GoF	Answers the following question: How can one define an interface for creating an object, but use other factors to decide which class to instantiate. This design does not use an actual direct mapping to the GoF family of factory patterns. It uses the concepts of thode patterns to enable context sensitive component instantiation. Detail design may use more a direct mapping to one of the factory patterns, depending on the platform capabilities. Architecture objectives solved by this pattern: • Adhere to architecture principles. • Consolidate application design patterns. • Establish looser coupling. • Ensure that application infrastructure is replaceable.

5.2 General Design Objectives

Objectives of this technical end-to-end design are as follows:

1. Adhere to architecture principles and objectives.
2. Fully take advantage of known design patterns to limit the amount of design time and the errors that can occur when designing from scratch.
3. Provide a consolidated end-to-end high-level design required to implement a first incarnation of an application infrastructure that will start Nedbank on the journey towards a service-oriented architecture.
4. Fully separate the concerns and responsibilities between applications and application infrastructure.
5. Provide a new design pattern for applications to enable them to expose service interfaces.
6. Provide input to infrastructure designers and developers to:
   - Understand the architectural requirements and constraints
   - Assist application infrastructure delivery and refinement.
7. Provide input to application designers and developers to:
   - Understand the architectural requirements and constraints
   - Ensure compliance with the application infrastructure.
8. Reduce the amount of infrastructure detail in application design documents.
9. Reduce the amount of dependency between high-level components.
10. Provide guidance for architectural and environmental compliance across all IT disciplines.
11. Reduce the number of concerns raised in technical reviews in the latter part of the project process.
12. Improve the quality of solutions as well as reduce time-to-market.

6 KEY CONCEPTS

6.1 Overview

This design describes an enterprise level application infrastructure based on principles and patterns that enable migration to service oriented architecture. Enterprise level implies that it should be implemented on all platforms across the enterprise to achieve full effectiveness. It also specifies the integration patterns that must be used between system and platform boundaries. This section defines the significant concepts involved and is intended to provide background information by which the rest of the document can be evaluated.

6.2 Definitions

6.2.1 System

A system is best described as “A key architectural component of the overall IT landscape”. It is typically a feature rich set of functionality that solves a particular aspect of the total business. Systems typically contain multiple applications to fulfil its responsibilities. The architecture department in conjunction with business should describe which systems we use, their boundaries and how they interrelate. Examples of systems in our environment are:

- Customer (CIS is not the system)
- Card (CAMS is not the system)
- Profile
- Payments
- ABF

6.2.2 Application/Program/Component

In this document they mean the same thing just on different platforms. To solve a particular processing problem on the MVS mainframe, one will used a set of programs that work together in a particular way. The same problem on a Windows server will be solved by a set of components. In essence an Application/Program/Component is just a set of compiled instructions that can be deployed autonomously with the minimum effect on other Applications/Programs/Components. This document will refer to them interchangeably and the reader can choose the one most applicable to his environment. In essence there is very little difference.

6.2.3 Service Program

This is one deployable unit that acts as the main controller to execute the functional part of a Service’s operations. In this design, a Service typically contains one Service Program that will honour all the operations of that Service. The reason is to shield the Service infrastructure from application implementation intricacies.

6.2.4 Service Operation

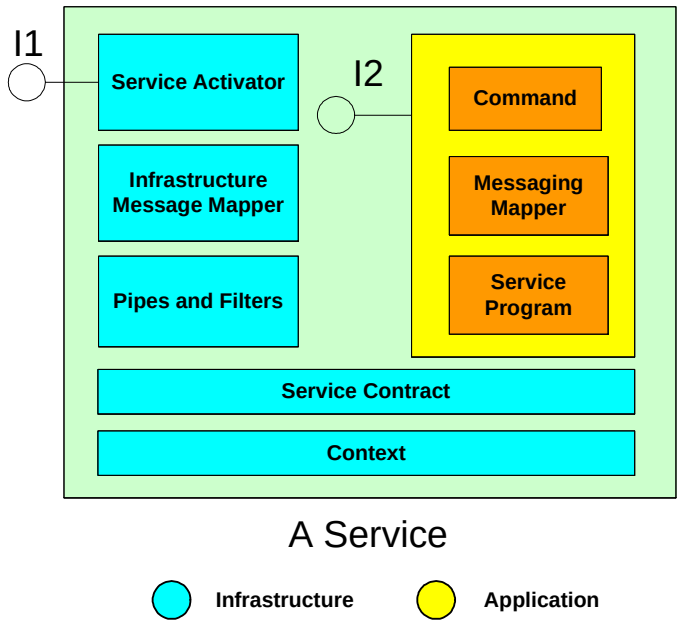
The operation is the functional part of the Service. A Service can expose multiple operations through its interface. An operation performs either a full scenario in a Use-Case or a sub part of a bigger Use-Case. Since Services are located at System boundaries, their operations are the most visible functions that those Systems expose. The physical locations for operations inside Services are within the Service Application.

6.2.5 Service

A service is best described as an aggregation of applications, which is published for use, forming a new functional block. A Service is referred to using a noun. The verb part of a Service is its operations. Services live at the boundaries of systems. All communication with a system should go through Services. Services have the following characteristics:

- A service is autonomous – A consumer never interacts with the resources under the service's control directly. It sends a message to the service and in turn the service will perform the operation on the consumer's behalf. It is very important to note that communication with the service is only achieved through messages. This allows for a very loose relationship between the consumer and the operations and information that is under the control of the service.
- A service has a published, discoverable contract – The service contract is specified in a common language, which should allow the consumer to dynamically bind to its operations and information model and to dynamically invoke the operations of the service. In the web services world this language is the "Web Service Definition Language" or WSDL. It must be noted that the contract definition language is not limited to WSDL alone. In the web services world the standard mechanism used for service contract discovery is "Universal, Description, Discovery and Integration" or UDDI. It must be noted that discovery is not limited to UDDI. A service contract can include the following:
    - Supported operations – A service typically supports more than on operation. Operations are typically invoked through messages. Operations may also result in messages going back to the consumer.
    - Data types and formats supported – Integer, String, Customer, Account and Business Unit. Data types are the building blocks of messages.
    - Protocols supported – HTTP, TCP and SOAP.
    - Conversation Semantics supported – One-Way, Events, Request-Reply and Publish-Subscribe.
    - Service Levels supported.
    - Security semantics.
- Assumes cross-platform use – A service must be invoked in a platform independent way. When a service replies with a response it must do so in a platform independent way. This attribute of a service works hand-in-hand with the previous one. It means that the service contract, message formats and messaging mechanism are available to and understandable by multiple systems and platforms (see the Canonical Data Model design pattern).
- There is no shared state between the consumer and the service – All information necessary to complete its operations must be contained in the message that enters the service. Likewise will all information that resulted from the operation performed by the service be embedded in the message leaving it.
- Communication with a service is only possible through an exchange of messages - It should be noted that passing a message to a service is NOT analogues to invoking an operation on a component. When invoking an operation on a component, the calling thread of execution is used to execute the logic in the operation. This means that the invoking function blocks until the operation returns to it. Services on the other hand also support one-way semantics. This means that sending a message to a service does not necessarily require a resulting message to be sent back. All messages must utilise the Canonical Data Model.
- Data that leaves the service must be treated as a snapshot – Once as service operation completes, the consumer may make no assumptions as to the state of the data that the service operated on. Services serve multiple consumers simultaneously. Any number of these consumers can invoke the same operation on the same data. One operation may be seen as autonomous, but no cross operation atomicity can be assumed. In conclusion, when a consumer receives the resultant information from a service operation invocation, the real state of the information could have already changed. Designers of Service Applications must therefore cater for an optimistic concurrency model.

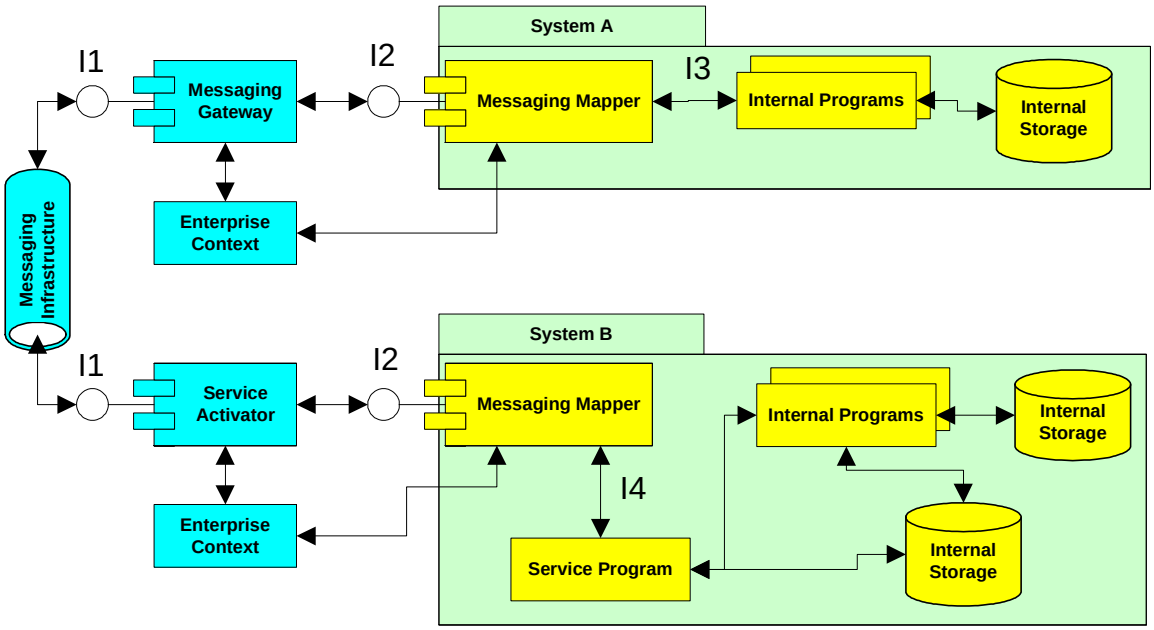
6.2.6 Service Diagram



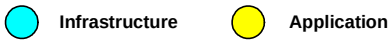
This diagram explicitly shows that a service is a collection of programs that work together to honour the service contract. Infrastructure programs typically solve the non-functional requirements of the service contract, while application programs solve the functional part.

Interface I1 is the full service interface as described in the interface part of the service contract. It is implemented in terms of an Envelope Wrapper and contains interface I2 in its body portion. Service developers typically only develop the Service Program and the Messaging Mapper, because the infrastructure components will be re-usable across services. The infrastructure programs are platform re-usable and the Messaging Mapper can be system re-usable. The command that forms part of interface I2 identifies the operation that is being invoked on the service program. The Service Activator may choose to map a particular command message to a different Service Program.

6.2.7 A new application design pattern



The Position of Services in the IT landscape



Before evaluating this diagram, be sure to read and understand the description and role of the various design patterns.

This diagram depicts the essence of positions services in the overall IT landscape. Services lives only at the boundary of systems. This design is only concerned with inter-system integration and do not yet attempt to specify intra-system integration patterns and mechanisms. This design specifies that all forms of interactive inter-system integration use the pattern indicated in this diagram.

Please note that some components have been collapsed to minimise clutter on the diagram. For instance the Channel Adapter pattern has been collapsed into the Messaging Gateway and Service Activator components.

This pattern will enable us to take advantage of more optimised, platform specific, messaging infrastructures where the need arise and the capability exist. Consider the following scenarios:

- System A resides on a different platform than System B: The infrastructure needs to provide for full messaging capabilities, since shared memory and direct program calls are not available.
- System A and System B reside in CICS: Shared storage and direct CICS calls are available. A messaging infrastructure that capitalises on that can be used. Even a channel like MQ can be utilised.

Key aspects to note about the diagram:

- The integration landscape is essentially split into three groups of functionality: Consumers, Service Providers and the Infrastructure that connect them together.
- Messages that flow across the messaging infrastructure are always in canonical form (interfaces I1 and I2) even if System A and System B lives in the same environment.
- To the internal programs in the consuming system (System A) the Service is the Messaging Mapper and Messaging Gateway pair. It must be totally unaware of what happens beyond that.

- To the internal programs of the Service Provider system (System B) the Consumer is the Messaging Mapper and Service Activator pair. It must be totally unaware of what happens beyond that.
- New infrastructure and integration mechanisms can be introduced as long as I1 and I2 are honoured.
- The enterprise context is flowed by the messaging infrastructure between the environments that System A and System B lives in.
- The enterprise context is available to the messaging infrastructure and applications.

The following migratory steps should be followed to fully implement this pattern:

- Define system boundaries.
- Define service interfaces on those boundaries.
- Implement messaging mapper components and service programs to honour those interfaces.
- Implement the required service activator programs. More that one might be needed to cater for different invocation mechanisms and optimisations. At least one that will cater for cross platform messaging should be implemented as a start.
- Implement one or more forms of Messaging Infrastructure.
- Implement the required Messaging Gateway programs for the different consuming environments.
- Refine and enhance the Messaging Infrastructure.

6.2.8 Service Contract Rules and Infrastructure

- The service contract is immutable. The dictionary definition is “Not subject or susceptible to change”. This does not mean that a service contract cannot change, it just implies a different way of applying change to a service contract. When change is needed on a particular service contract, it may only be handled in one of the following ways:
    - A new version of the contract is implemented and the old version is still supported. This method in effect exposes 2 service contracts concurrently on the same system. This method should be used when both versions are valid.
    - A new version of the contract is implemented and the old version is depreacated using the contract lifcycle management process (discussed below).
    - Remove the contract in totality. In this case the service will not be available anymore. This is the method to use when services themselves are deprecated.
- An owner role must exits for service contracts. In real terms this role will be a coordinated multi-dissiplinary team. This role encaptulates the following characteristics:
    - The ownership of the contract spans projects and project liftimes.
    - All planning around the usage or migration of contracts ar controlled or approved by the owner role.
    - The owner acts as a librarian for service contracts and related artefacts. Contract artefacts should be stored in a repository controlled by the librarian function. These artefacts include the following:
        - Documentation around service contracts.
        - Service Contract definitions.
        - Canonical entity definitions.
        - Generated re-usable Messaging Endpoints.
    - Service Contract lifecycle management and maintainance of artefacts.
    - Provide project support. Two types of projects need support:
        - Projects that create services.
        - Projects that consume services.
- Lifecycle management processes and rules needs to be established, because of the immutability attribute of services contracts.
- Usage of service contracts needs to be tracked and monitored. This will assist grately with lifecycle management. Both services and consumers are considered as users of the service contract. Once nobody uses a service contract it may be archived or removed from the repository and management infrastructures.

7 END-TO-END TECHNICAL INFRASTRUCTURE CONTEXT

7.1 Overview

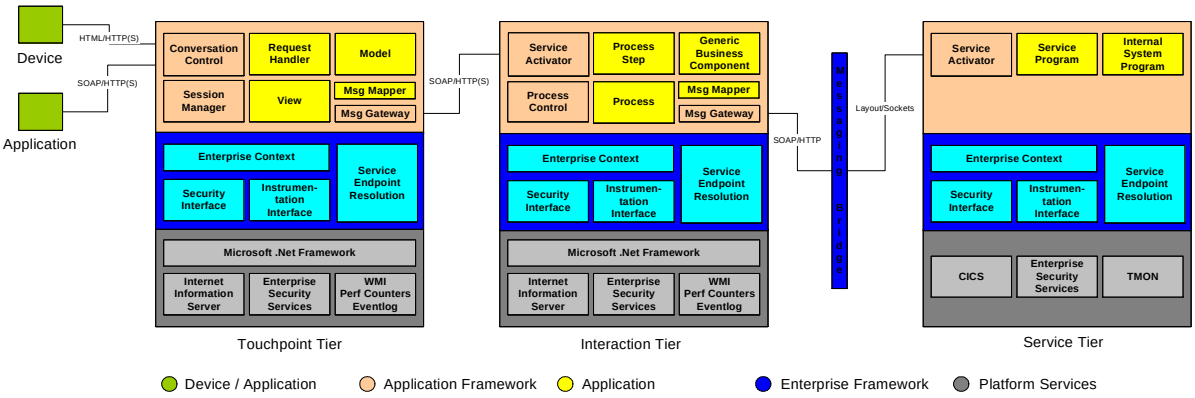
This section explains the end-to-end view of the infrastructure that will be used in the enterprise. The Channel Convergence project will implement parts of this infrastructure and in the latter part of this section a suggested mapping for that project will be done. As stated in the architecture principles, the implementation of infrastructure will be a migratory process and at the time of writing this document it was unclear of exactly how much of the infrastructure will be implemented and finalised by the first phase of the Channel Convergence project.

Choices around the extent of infrastructure implementation will be made as a collaborative effort as part of the detail design. Architecture would like to have the full design implemented, but recognises that certain practical constraints exist and therefore supports a migratory effort in terms of the implementation of the patterns suggested in this document.

Please make an effort to understand the goal of this design and let us find ways to realise it. Feel free to contact the author if there is still uncertainty on the end goal or to help with enhancements to the document.

7.2 End-To-End Diagram

The following diagram depicts the end-to-end view of the significant architectural components and how they interrelate. It specifically concentrates on the Tiers and Layers in the solution.



7.3 Application Framework Layer

In applications a certain amount of functionality usually exists to support other functionality and as a rule that functionality is re-used many times throughout the rest of the application. The framework layer bundles and standardises this supporting type functionality. This framework does not have to be the same for all tiers of a distributed application. In this design there are different and distinct responsibilities on each of the tiers and the application framework layer is in control of those responsibilities. In each tier application blocks are build in and on top of the application framework layer.

7.4 Enterprise Framework Layer

In contrast to the application framework layer, the enterprise framework layer ensures consistency across all the tiers of the distributed application. Building applications across multiple tiers does not mean building multiple applications. To enable multiple scalability models the Interaction Tier and the Service Tier frameworks and applications are design to be stateless and with no server affinity.

The enterprise context in this layer helps to communicate common information that is used throughout the execution of functionality across the tiers in a stateless fashion. This communication is achieved by exposing the context information to the messaging infrastructure and application frameworks in a consistent way across all tiers.

Another crucial part of this layer is to provide common frameworks services in a consistent and correlated way across all tiers. These services use the enterprise context to store their state. The enterprise context exposes a Factory pattern that creates platform specific instances of the enterprise services. The platform specific instances of the services expose a consistent interface to applications across all tiers. This means that an application developer using the security interface on the Touchpoint tier has the same functionality and knowledge as an application developer using the same interface on any of the other tiers.

The reason for the factory pattern in the context is to enable us to make enterprise services context sensitive should it be required.

In this diagram only two enterprise services are depicted. These are the bare minimum that we must implement to get started. Other enterprise services can be added over time as the architecture matures and new requirements become available. Possible candidates for additional enterprise services in this layer include:

- Service Endpoint resolution
- Audit Service
- Transaction Control Service
- Enterprise Configuration Service
- Caching Service

7.5 Platform Services Layer

This layer contains functionality of systems outside the current context that this infrastructure depends upon. Operating system functionality, component frameworks, application environments and external applications fall in this category.

In this diagram the IIS and CICS are both application hosting frameworks and .Net is an application environment framework. The “enterprise security services” system is an external package that is used to expose the Security Interface in the Enterprise Framework layer. Each tier has its own local implementation to take advantage of platform and tier specific optimisation. All security related issues are encapsulated in this package and is described in the channel convergence security TED document.

When new platform service become available or others are replaced, the implementation of the enterprise services interface for that specific service will be re-implemented in terms of the new platform service. The reasoning is to keep the definition of the interface in the enterprise framework layer stable while we refine or replace its implementation in the platform layer.

7.6 Device / Application Tier

This tier hosts the physical external device or external application that needs to communicate to the Touchpoint tier. Devices that do not support the exposed Touchpoint protocols will require a device handler in this tier to translate between device specific protocols and the Touchpoint protocols. For devices that do support the Touchpoint protocols, this device handler is optional. An example of a device that would require a device handler in this tier is a cell phone using SMS as its primary communication protocol. The device handler can be in the form of a SMS gateway.

7.7 Touchpoint Tier

The primary responsibility of this tier is to shield the rest of system (other tiers) from the differences in capability of devices in the device / application tier. This de-coupling also allow us to keep our core services and processes stable when introducing new devices with their own set of abilities.

Every interaction is described in terms of a request handler and view combination. The conversation controller controls interactions. State (request and response) information that spans multiple interactions is kept in model components. To complete all the information needed for a service request to the next tier, multiple interactions might be required. Once the conversation controller deems the information complete, it triggers the invocation of the service. On completion the result of the service is stored in the model and the conversation controller controls the interactions necessary to complete deliver of the full response to the device. Multiple interactions might be needed to deliver the result as well. For devices that have the capability to deliver all the request information and to receive all the response information, the conversation will still exist, but it will consist of only one request and one view.

This tier is also located at the outer perimeter of the system. It therefore has the added responsibility of authenticating the device or more specifically the actor that uses the device. It only triggers the authentication event, but the actual authentication is delegated to the Security Interface in the enterprise framework. It will keep track of the authentication status by attaching it to the Session controlled by the Session Manager. On subsequent requests from that device, authentication status must be verified against the enterprise framework Security Interface.

This tier exposes the following external interface protocols for devices or device handlers to bind to:

- HTTP(S) GET
- HTTP(S) POST
- HTTP(S) SOAP

7.8 Interaction Tier

NOTE: This tier is included in the context for completeness of the end-to-end context. It is not in scope for the project and has not been fully defined yet in terms of architecture and applicability. Once the requirements and architecture has been finalised, this document will be updated to reflect the new end-to-end context and the designs around that.

The current intent for this tier is to expose business processes as a set of coarser grained services and to control the sequencing of operation on them.

7.9 Messaging Bridge

This is a helper tier to compensate for the fact that we will not have standards based middleware in place by the time the first deliverable is required for this project. Its lifetime is not connected to the project however. This tier's primary responsibility would be to allow the consumer side of the service integration to bind to an open standard, while the service side still bind to a more restricted standard.

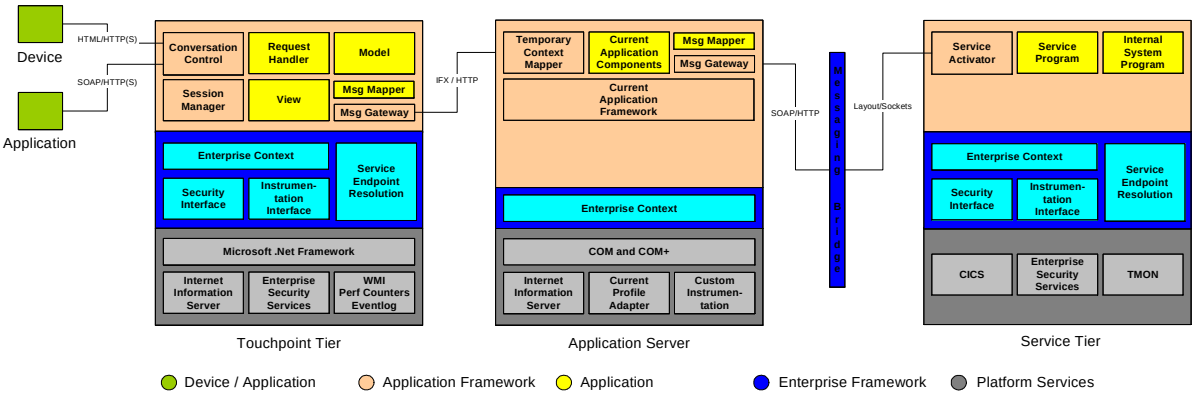
We might sit with limited infrastructure for some time to come and the intention with this tier is to smooth the migration towards the ultimate goal of enterprise middleware on all platforms. Therefor its existence and applicability must be carefully considered in detail design.

Architecture recommends that Microsoft Host Integration Server be used as a temporary Messaging Bridge.

7.10 Service Tier

This tier primarily houses the systems that expose the services that the rest of this project requires. In the diagram only the MVS mainframe is depicted in the service tier. It is the most significant one, but not the only one. Tower is another platform that fits in this tier and the application, infrastructure and integration patterns are the same for that part of the system.

7.11 Channel Convergence Context Mapping



7.12 Application Server Tier

For this phase of the channel convergence project the current Netbank application server tier will remain in place. We need to update it to flow the enterprise context at least. The enterprise context will be created in the Touchpoint tier and is needed by the Service tier. To do this we need some re-factoring in the messaging layers of the application server as well as a thin enterprise context layer in the application server framework. This is a temporary measure to help us bridge the period until we have a fully implemented interaction layer / tier.

8 LOGICAL VIEWPOINT

8.1 Overview

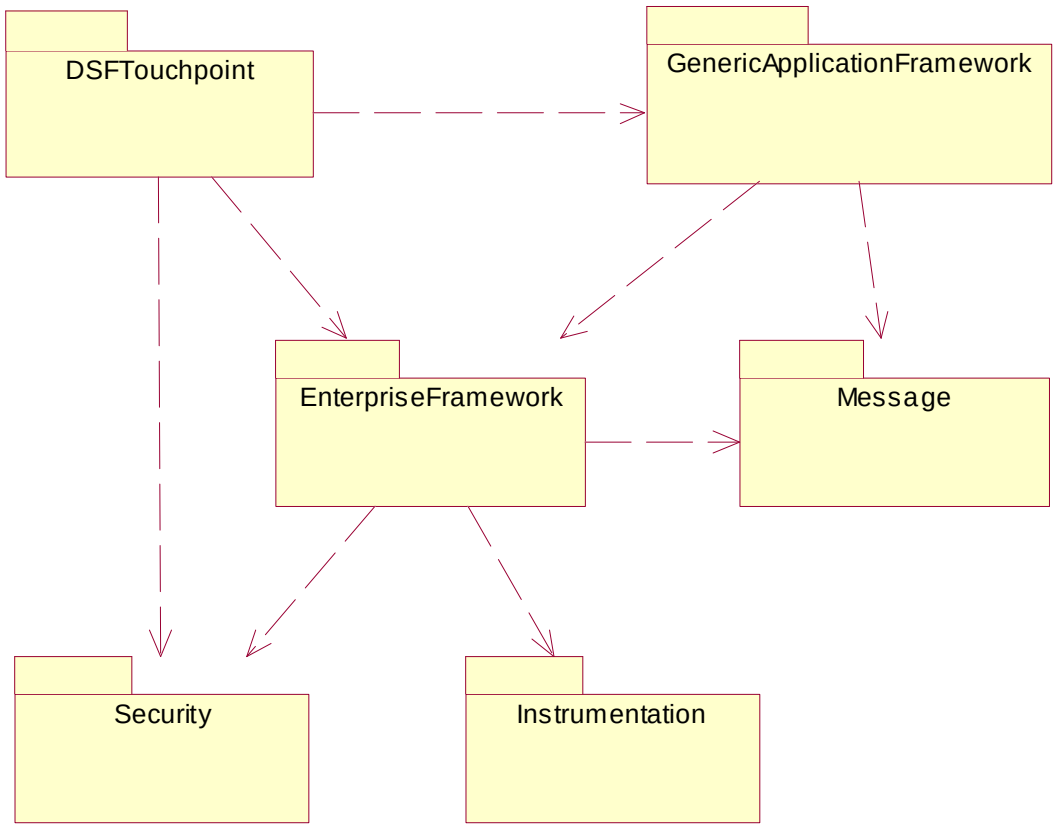
This section discusses the infrastructure at a logical level. This is the lowest level that this document will describe. Further design documents and implementation specifics will be handled by the detailed design phase of the project that will commence once this document is approved. It must be noted that only the packages that are significant for the business release of the channel convergence project are included. In subsequent phases other packages will be added. The packages shown here, with the exception of the DSFTouchpoint, will form the core around which other packages will be built. The impact of adding other packages should not significantly change the core packages.

Fist a brief description of the significant design packages will be provided. Then each design package will be described in terms of the design components it contains and the significant interactions between them. The order in which the design packages will be described in, are from the bottom of the dependency chain to the top.

At the end of this section, some detail design considerations are discussed.

By convention, this document will refer to the entities inside these packages as components. Depending on the platform they are created on, they may be entities, classes, objects or components.

8.1.1 Package model for this design



8.1.2 Security

This package is not part of this design document and is fully described in the Security TED that is part of the Channel Convergence project. It is shown in this diagram to indicate the dependency of these design elements on the elements that reside in the Security TED.

8.1.3 Instrumentation

This package is not part of this design document and is fully described in the Application Instrumentation Architecture document. It is shown in this diagram to indicate the dependency of these design elements on the elements that reside in the Application Instrumentation Architecture document.

8.1.4 Message

This package contains the design for enterprise messaging. This design describes the logical model that will be used for integration between systems.

8.1.5 Enterprise Framework

This package contains the design for the Enterprise Framework as depicted in the context section. It contains the components that applications and application frameworks will use to achieve common and re-usable tasks.

8.1.6 Generic Application Framework

This package contains most of the integration components that will be utilised by all other packages. It details the patterns for integration at a logical level that can be implemented across the platforms in the enterprise. It shows how the patterns described earlier in the document, should be used together to provide a re-usable integration environment that enables service oriented integration.

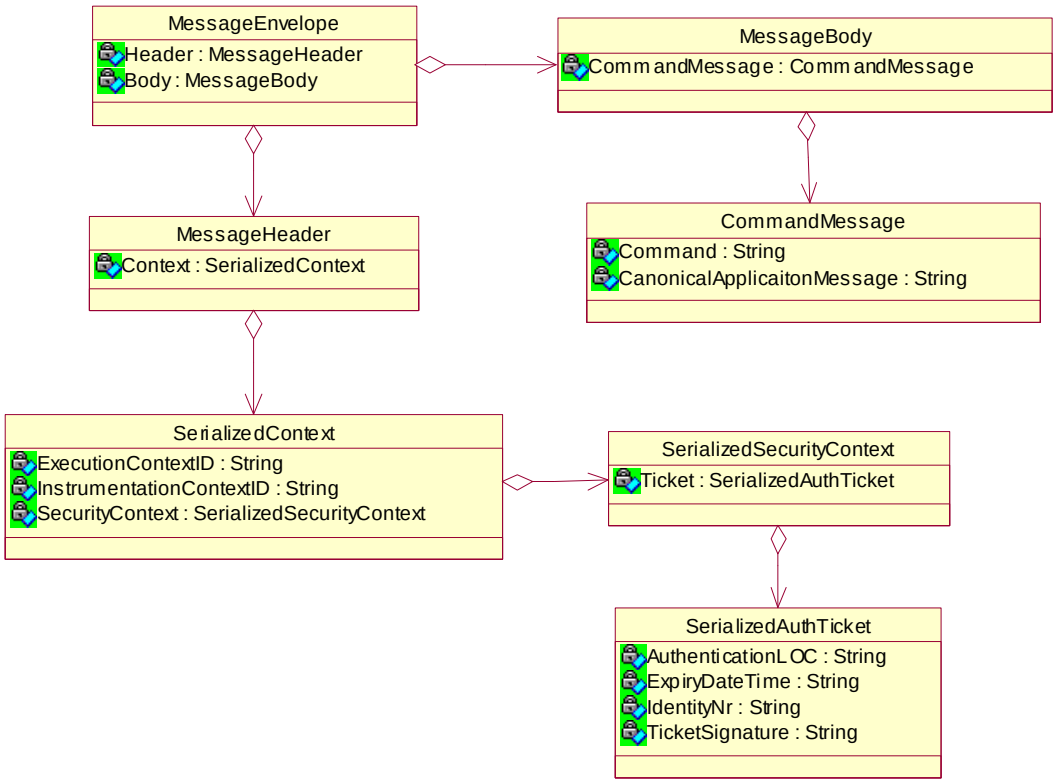
8.1.7 DSF Touchpoint

This package contains the significant infrastructure components for the application framework that will reside on the Touchpoint tier shown in the context models. It will not detail any application components. Application components are described as part of other design documents.

8.2 Message Package

The components in this package describe the logical layout of integration messages that traverses the infrastructure between services and the consumers. All the relationships is aggregation relationships and mechanisms for implementation must support this kind of relationship. Implementation mechanisms may also not place logical limitations on the multiplicity if these relationships. XML is a technology that will be ideally suited to this design, but it is not mandatory, as long as the two mentioned rules are not transgressed.

8.2.1 Components



8.2.1.1 MessageEnvelope

This is the main container for a message. In practical terms, this is the message. We need this container to enable the transportation of both application and infrastructure information in the same message. This structure is not extensible and will always contain only the elements indicated in the diagram.

8.2.1.2 MessageHeader

This structure is extensible and will evolve when new infrastructure requirements arise or new standards become available. In its current incarnation it will be used only to transport the enterprise context information.

Exception handling must still be specified. Exceptions will be used to notify the consumer of a non-functional error result and will be conveyed in the MessageHeader.

8.2.1.3 SerializedContext

This structure is the primary container for the enterprise context described in this design. It aggregates all the information that needs to flow in one element in order not to pollute the MessageHeader structure and to ease processing design on both sides of the messaging infrastructure. The fields in are discussed as part of the Enterprise Framework package that contains the enterprise context.

8.2.1.4 SerializedSecurityContext

This structure aggregates all security related information in one enterprise context element in order not to pollute the SerializedContext structure and to ease processing design on both sides of the messaging infrastructure.

8.2.1.5 SerializedAuthTicket

This structure aggregates security ticket information as described in the security TED.

8.2.1.6 MessageBody

This structure aggregates all the information to solve the functional requirements for a particular message exchange. In other words it contain information that is important to an application, rather than infrastructure.

8.2.1.7 CommandMessage

This is a helper structure to assist with function invocation across a messaging infrastructure. It aggregates a “Command” that maps to a logical service operation name. It is logical to enable run-time binding of the physical operation. This is a feature and is not mandatory (see sequence diagrams for clarity).

This structure also aggregates all the information, in Canonical form, that the service operation needs to perform its function.

For result messages this structure must still exist. The convention is to combine the name of the operation and “Result”. All resultant information is then contained in that command message. This result will contain the functional result and is always present.

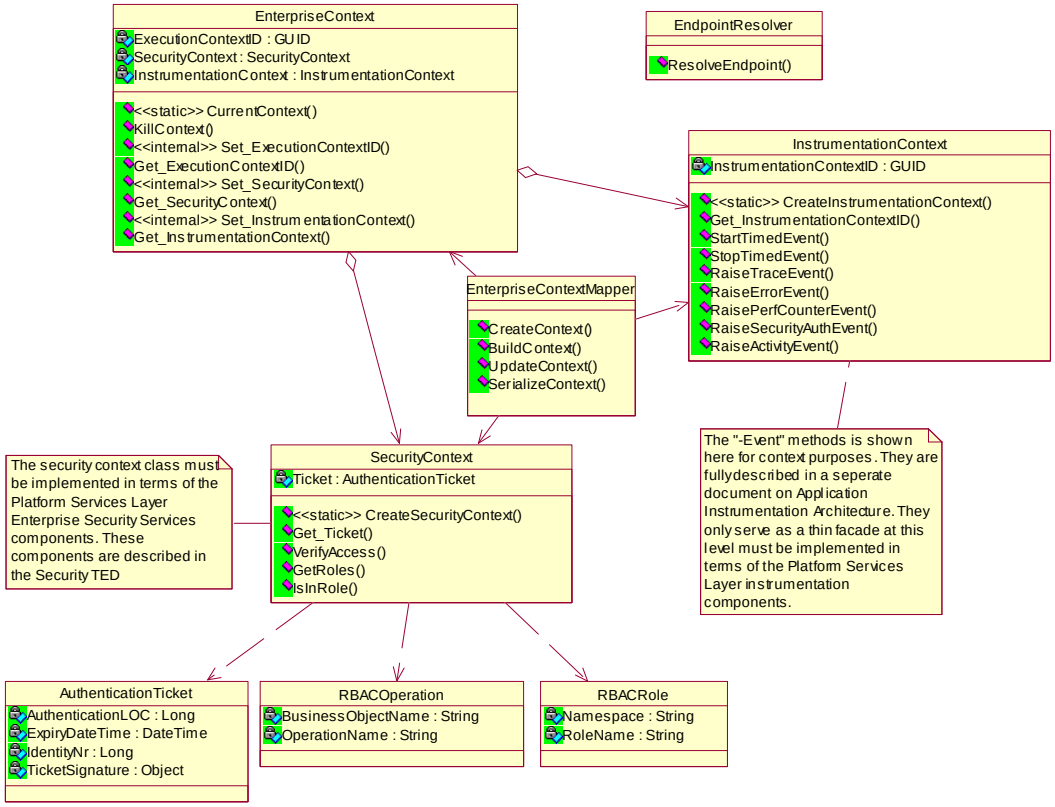
8.2.2 Interactions

This is a data package and it has no interactions.

8.3 Enterprise Framework Package

The components in this package are mainly re-used by other packages. They provide enterprise consistent information and services that can be used by applications or application frameworks. All the enterprise context sensitive service components are controlled via the EnterpriseContext component.

8.3.1 Components



8.3.1.1 EndpointResolver

This component encapsulates and abstracts the resolution of logical service endpoints to physical service endpoints, or more specifically, ServiceActivator components. It can be implemented using a local infrastructure configuration, an enterprise infrastructure configuration, UDDI or even call a service to do the resolution. We should start with a local configuration based resolution model and migrate to more flexible models. The endpoint descriptors must be valid URI definitions.

Since service endpoint resolution happens as part of the infrastructure, changing its mechanisms will have no impact on applications and can therefore be done as part of maintenance releases of the infrastructure. This means that we can make small tactical changes to one part of the infrastructure, stabilise it and then deploy it to the rest.

8.3.1.2 EnterpriseContext

This component is the main container for context information at runtime. It also acts as a resource dispenser for context-based enterprise service components and uses a factory pattern to provide these to the caller. Using the factory pattern enables this component to supply different service

components based on context information, should the need arise. For discussion on its methods, refer to the interaction diagram discussions.

8.3.1.3 EnterpriseContextMapper

This component is used for populating other components in this package. It contains methods with the following functions:

- CreateContext – This method is used to create the EnterpriseContext for the very first time, when there is no serialised context. At this level it is used to create the context from parameters, rather than the serialised context. It achieves the same goal as the BuildContext method. It is show separate at this level to help with clarity. In detail design the two may be combined to form one method where practical.
- BuildContext – This method rebuilds the in memory context from the serialised context coming from the integration infrastructure. This method is called when a request message enters a new node / tier. It de-serialises the enterprise context and populates the various components with the information contained therein.
- UpdateContext – During processing on a particular node / tier, request messages can be sent to other nodes / tiers. When the reply messages come back, they also contain the enterprise context. The context could have changed on the other node and therefore needs to be updated using this method. It de-serialises the enterprise context and populates the various components with the information contained therein.
- SerializeContext – When a node / tier wants to send a request message to another node / tier. It needs to send the enterprise context with the message. This method is called to serialise the information in this package for insertion in the message.

8.3.1.4 InstrumentationContext

This component keeps track of the key information used by the Instrumentation package. Its methods act as a façade to the functions contained in the Instrumentation package. See the Instrumentation package description for more information.

8.3.1.5 AuthorisationTicket

This is a helper component that contains the key information used by the Security package.

8.3.1.6 RBACOperation

This is a helper component that is used by the infrastructure to communicate the intent of a message to the Security package. The “Business Object Name” is the service name and the “Operation Name” is the operation on that service. This structure is used during access control.

8.3.1.7 RBACRole

This is a helper component that is used to describe a user role. This structure is used during authorisation and custom application processing.

8.3.1.8 SecurityContext

This component keeps track of the key information used by the Security package. Its methods act as a façade to the functions contained in the Security package. Its methods have the following roles:

- CreateSecurityContext – This is a factory method used to create instances of this component.
- Get_Ticket – This is a property access method to supply the calling application with the contained Authentication Ticket. The caller can then access the ticket directly to obtain specific information.
- VerifyAccess – Used mainly by infrastructure functions to enforce access control. If this method fails, the infrastructure must raise an exception.
- IsInRole – Used by applications or application frameworks to determine whether the executing identity is in a particular role. The use of this method and the information it returns is entirely up to the application designer.

For further clarity on the use of these methods, please refer to the interaction diagrams throughout this document.

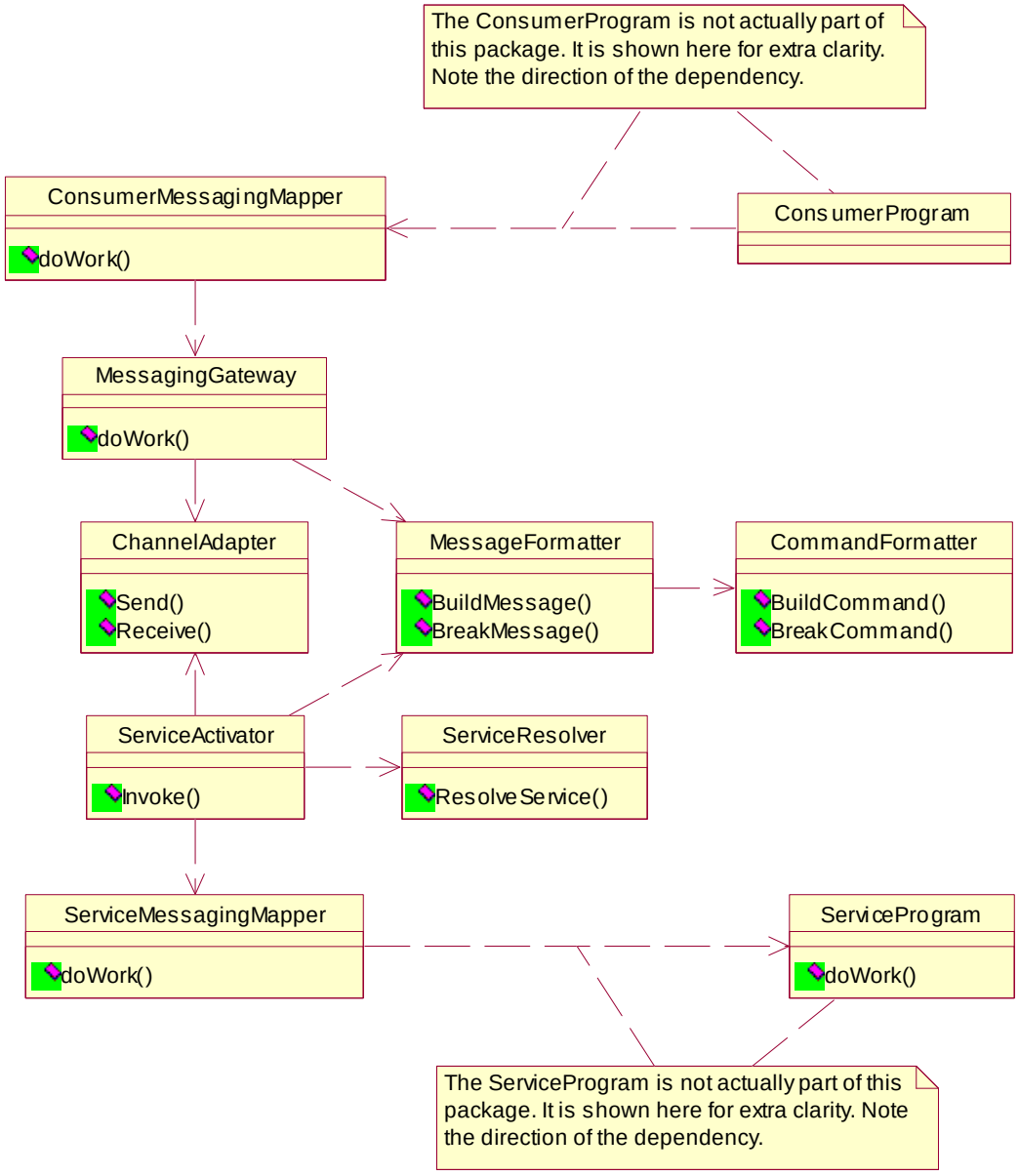
8.3.2 Interactions

This package has no significant interactions at this time. Most of its components will be utilised in the interactions of other packages. Please refer to those interactions for clarity.

8.4 Generic Application Framework Package

The components in this package are mainly concerned with integration. They provide enterprise consistent design to implement the concepts described in this document.

8.4.1 Components



8.4.1.1 ConsumerMessagingMapper

The consumer application designer creates these components. It forms part of the application and is deployed as such. It is strongly recommended that these components be created in a way, which makes them re-usable from various parts of the application. It can implement part of a service's operations or the whole set of them. This decision is up to the detail designer.

This component basically acts as an application specific proxy to a service. The methods exposed by this component matches those exposed by the MessagingGateway for the specific service. The difference here is that methods on this component take domain specific entities from the containing application as parameters. This component then binds to the Canonical entities exposed by the MessagingGateway and does a mapping between the two sets. It is imperative that all entities, domain specific and Canonical, are in the development environment of the application.

Also note that this component should be used for consumer side filtering and enrichment of the information passing between the consuming domain and the messaging infrastructure. This ensures clear separation of infrastructure and domain responsibilities.

8.4.1.2 MessagingGateway

An infrastructure team creates these components. It exposes the operations of one full service contract as a public application interface. This interface is in the development environment of the application and in Canonical form.

This means that a MessagingGateway component will exist per service contract, per application environment. For clarity, assume a customer service contract. One MessagingGateway component for the customer service will exist for COM, one will exist for .Net, one will exist for Java and one will exist for COBOL. The environments mentioned here are for explanation purposes and it does not mean that these are the only ones catered for. Remember that only the application interface needs to be environment specific. Detail infrastructure design may therefore re-use components internally to create multiple interfaces.

These components are re-usable across applications and therefore they cannot be supplied and managed by any particular application team. They must be separately deployable and must support run-time binding. They must be generated, either in source form or binary form. An infrastructure team must supply and manage them separately from the applications that use them.

Refer to the interaction diagrams in this document for more clarity.

8.4.1.3 MessageFormatter

This component is part of the infrastructure and controls the serialisation and de-serialisation of the Canonical entities exposed to applications to their message counterparts. Multiple of these components can exist to cater for different messaging schemes.

It is responsible to control both the infrastructure and application portions of the message serialisation. Messages must conform to the schema described in the Message package.

8.4.1.4 CommandFormatter

This component is part of the infrastructure and acts as a helper to the MessageFormatter component. It understands the mapping between Canonical entities and messages. Its primary responsibility is to enable service operation invocation semantics across the messaging infrastructure.

On the consumer side it will format a Command message containing the Canonical application data. The command must match the operation that needs to be invoked the service. When a resultant message is received, it will inspect the Command message to determine whether it is an application reply or an infrastructure fault. Application reply Command message has “Result” as a postfix to the Command. Infrastructure faults have “Fault” as the Command. The infrastructure must handle all infrastructure faults and raise only general error conditions to the application when they occur. These can be exceptions on platforms that support them. It does not necessarily indicate success when an application reply is received. The processing of the application reply, and therefore application failures, resides with the application domain and will not be discussed here.

On the service side this component does the inverse of the consumer side. For incoming messages it extracts the Command and the application data in Canonical form. For resultant messages, it packages the resultant application data in an application result Command message.

It must be assumed that any part of the infrastructure can generate a fault. When a fault does occur, this component is used to format a fault Command message. To do this the component will implement a special case of the BuildCommand method.

It is important that no infrastructure semantics are leaked across the MessagingGateway and ServiceActivator application interfaces.

8.4.1.5 ChannelAdapter

These components are part of the infrastructure. They handle the intricacies associated with sending and / or receiving messages over a channel. Various ChannelAddapter components will be implemented to cater for various channel protocols.

These components must know how to send and receive byte streams across the channel it is implemented for. They may not have any knowledge of the Envelope contained in the byte stream being transmitted.

It is strongly recommended that they be separately deployable to enable the maximum amount of re-use.

8.4.1.6 ServiceActivator

This component is part of the infrastructure and is generated by an infrastructure team. It is managed and maintained separately from any application.

These components are the main controllers on the service side of the invocation. A ServiceActivator component can invoke the functionality on one ore more ServiceMessageMapper components. A logical service endpoint is associated with a physical ServiceActivator through the EndpointResolver component in the enterprise framework package. It is up to the detail design to decide the most effective coupling between ServiceActivator and ServiceMessagingMapper components for a particular platform. Some environments support reflection, which greatly assist with the mapping of these components.

As with the MessagingGateway components, the ServiceActivator components support an environment specific interface on the application side. In this case the application side is the ServiceMessagingMapper.

They must be separately deployable and must support run-time binding. They must be generated, either in source form or binary form. An infrastructure team must supply and manage them separately from the applications that use them.

Refer to the interaction diagrams in this document for more clarity.

8.4.1.7 ServiceResolver

This component is part of the infrastructure. It must not be confused with the EndpointResolver component in the enterprise framework package. The EndpointResolver is used to associate the next node on route to a ServiceActivator component. Once a message enters a ServiceActivator component, this component may be used to resolve which ServiceMessagingMapper component should process it. The decision must be based on the Command in the CommandMessage.

This component is optional, but it is strongly recommended that it be implemented from day one.

8.4.1.8 ServiceMessagingMapper

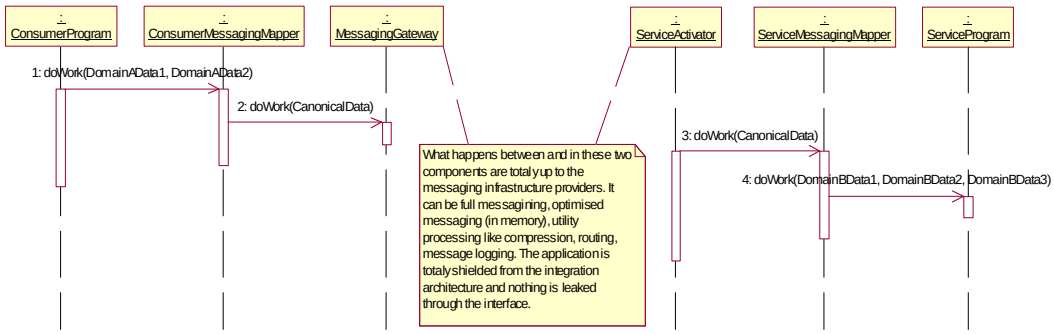
The service application designer creates these components. It has the same semantics than the ConsumerMessagingMapper, just on the service system side. See ConsumerMessagingMapper for related discussion.

Once it has mapped from the Canonical entities to the domain entities, it calls the ServiceProgram to do the specific processing required from the operation.

Also note that this component should be used for service side filtering and enrichment of the information passing between the messaging infrastructure and the service domain. This ensures clear separation of infrastructure and domain responsibilities.

8.4.2 Interactions

8.4.2.1 Application View of the integration infrastructure



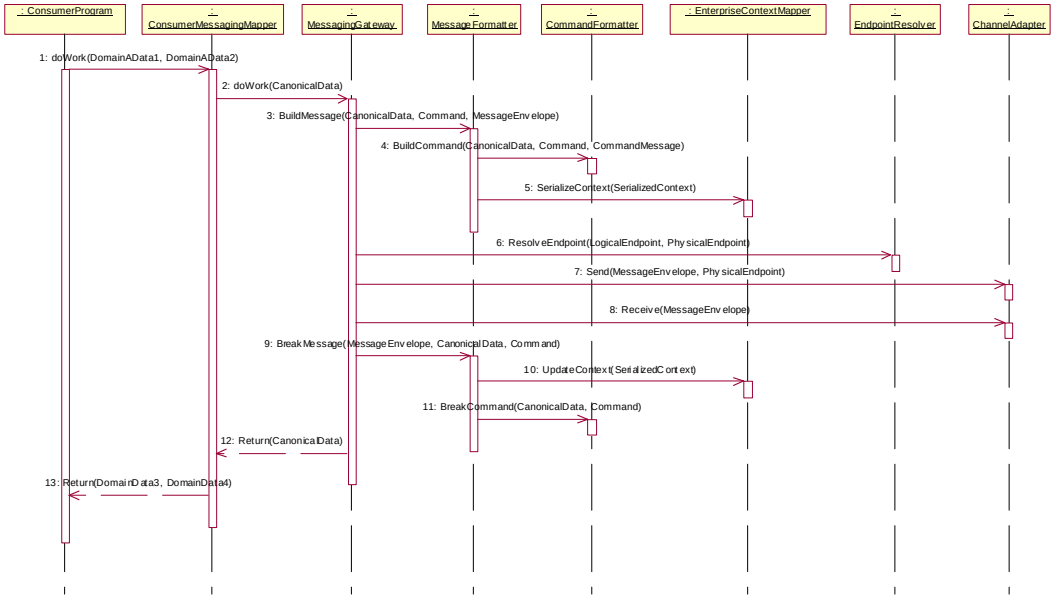
This diagram and subsequent discussion focuses on the application view of integration. The steps discussed here are the responsibility of the application designer. There is an opportunity for parallel development here. Once the Canonical interface has been defined, both sides of the integration infrastructure can be developed in parallel. The infrastructure components, MessagingGateway and ServiceActivator, can also be defined in a separate project stream.

Here follows a description of the interactions:

1. The ConsumerProgram calls the doWork() method on the ConsumerMessagingMapper as if it was the actual service operation. Parameters are passed as consumer domain entities. The doWork() method then instantiates and binds to a MessagingGateway component using run-time binding semantics. It then instantiates the Canonical entities exposed by the MessagingGateway component. It now proceeds to map between the domain specific parameter entities and the instantiated Canonical entities. As mentioned in the component discussion, any consumer side information filtering or enrichment may take place as part of this method. Finally it proceeds to the next step.
2. The ConsumerMessagingMapper calls the doWork() method on the MessagingGateway as if it was the actual service operation. Parameters are now passed as Canonical entities.

   The MessagingGateway now proceeds with the “consumer side processing for service activation” interaction, described in the next sub-section.
3. Once the messaging infrastructure performed its duty and the ServiceActivator has instances of the Canonical entities on the service side, it proceeds to instantiate an instance of the ServiceMessagingMapper, again using run-time binding semantics, and call the doWork() method on it as if it was the actual service operation. Parameters are passed as Canonical entities. It now proceeds to instantiate service domain specific entities and map between the two types of entities. As mentioned in the component discussion, any service side information filtering or enrichment may take place as part of this method. Finally it proceeds to the next step.
4. The ServiceMessagingMapper instantiates an instance of the ServiceProgram and calls the doWork() method on it. This call is the actual service operation. Parameters are passed as service domain specific entities.

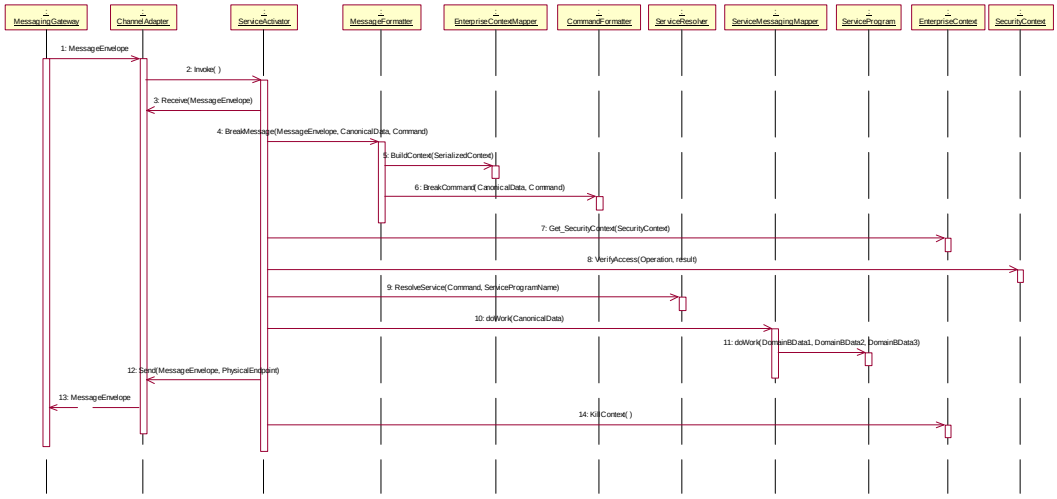
8.4.2.2 Consumer side processing for service activation



This diagram and subsequent discussion focuses on the consumer side infrastructure view of integration. The steps discussed here are the responsibility of the infrastructure designer. All information beyond the MessagingGateway is in Canonical form.
Here follows a description of the interactions:

1. As discussed in “Application View of the integration infrastructure”.
2. As discussed in “Application View of the integration infrastructure”.
3. The MessagingGateway calls on the consumer side MessageFormatter to produce a MessageEnvelope.
4. The MessageFormatter calls on the CommandFormatter to produce a CommandMessage. The command can be derived from the method call or be passed in as a parameter. It is up to the infrastructure detail designer to specify the most elegant way for his specific platform.
5. Serialisation of the enterprise context happens next. There is a separate interaction diagram for this method.
6. The MessagingGateway must now resolve the physical service endpoint. To do this it must provide a logical service endpoint. This can be derived from the binding on a specific MessagingGateway or passed in as a parameter. It is up to the infrastructure detail designer to specify the most elegant way for his specific platform.
7. Once the physical service endpoint has been determined, the MessagingGateway can now select and configure an appropriate ChannelAdapter and call the Send() method on it.
8. The call to the Receive() method is made to get the reply message. In this scenario a “Request-Reply” pattern is followed. This does not mean that it is the only pattern supported. In detail design other patterns can be implemented as well as long as the same semantics apply.
9. The MessagingGateway proceeds with de-serialisation of the reply message.
10. The enterprise context is updated. It could have changed as part of the service processing. The case in point is the instrumentation context – see the instrumentation discussion in the component subsection.
11. The reply is evaluated. If no error occurred on an infrastructure level, the Canonical data is returned. If an infrastructure error occurred, it is bubbled back up the call stack. It is up to the infrastructure detail designer to specify the most elegant way for his specific platform.
12. As discussed in “Application View of the integration infrastructure”.
13. As discussed in “Application View of the integration infrastructure”.

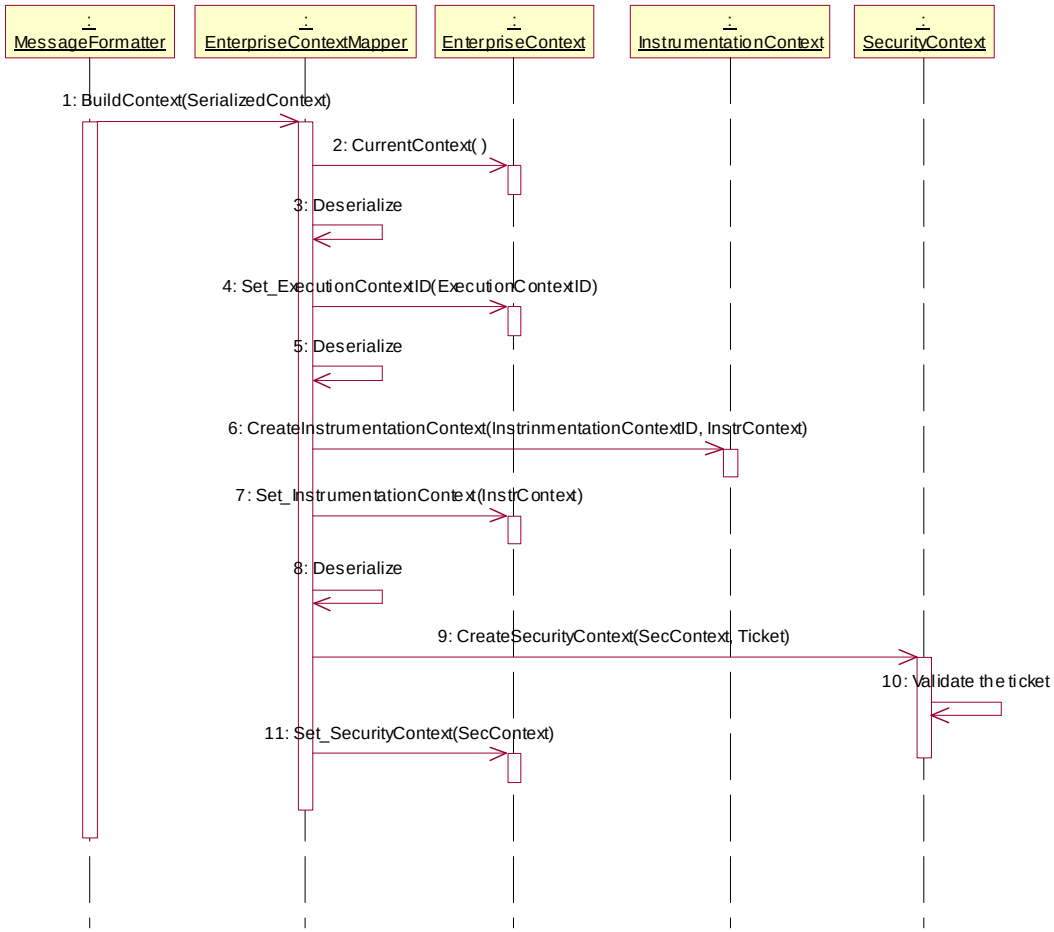
8.4.2.3 Service side processing for service activation



This diagram and subsequent discussion focuses on the service side infrastructure view of integration. The steps discussed here are the responsibility of the infrastructure designer.
Here follows a description of the interactions:

1. The message arrives at the ChannelAdapter that is located at the physical service endpoint.
2. The ChannelAdapter logically calls the Invoke() method. This is a logical call since a MessagingEndpoint typically registers with a ChannelAdapter to be notified of channel events. At this level, this call just indicates such an event.
3. The ServiceActivator receives the MessageEnvelope via the ChannelAdapter Receive() method.
4. It then proceeds with message de-serialisation.
5. The context must be built first. This will enable instrumentation for the rest of the executions. It will also ensure that all appropriate enterprise services and information is available.
6. De-serialise the CommandMessage and return the command and the Canonical entities.
7. Now that the service and operation are available, it can proceed to build an RBACOperation strucuture and get an instance of the current SecurityContext and;
8. Call the VerifyAccess() method to make sure that the AuthenticationTicket has access to invoke the operation on the service. Should this method fail, an infrastructure fault must be generated and sent back to the consumer. Under no circumstances may any processing continue. The KillContext() method must however be called after returning the fault.
9. Resolve the physical service program name using the request command. This is actually the name of the ServiceMessagingMapper to use.
10. As discussed in “Application View of the integration infrastructure”.
11. As discussed in “Application View of the integration infrastructure”.
12. The ServiceActivator now returns the reply via the ChannelAdapter Send() method.
13. The ChannelAdapter ensures delivery to the requesting MessagingGateway.
14. Destroy the context by calling the KillContext() method on the EnterpriseContext. This must be the last step as indicated here. The reason is that when something goes wrong, we still need to instrument it using the correct EnterpriseContext.

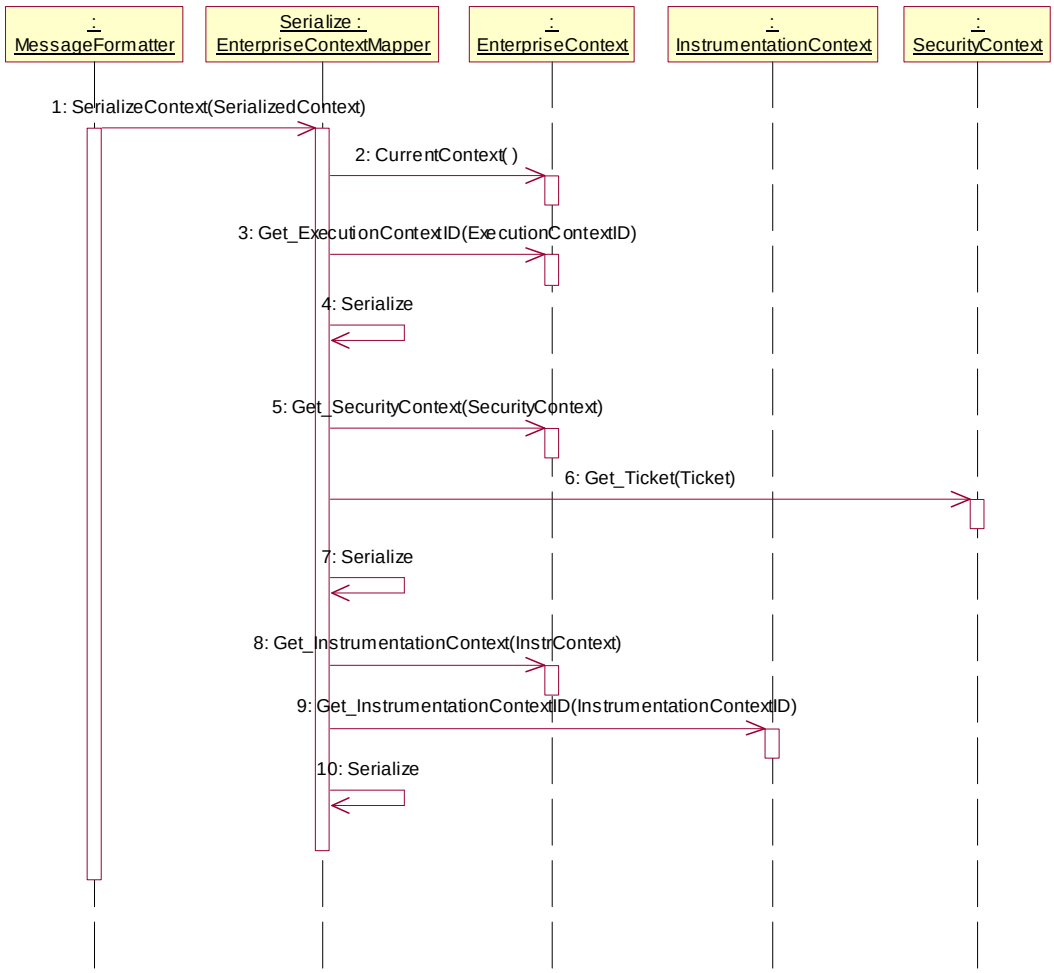
8.4.2.4 Building the Context components at system entry



After this sequence is complete the EnterpriseContext is ready for use.
Here follows a description of the interactions:

1. This diagram.
2. Call the CurrentContext() factory method. At this time the enterprise context does not exist, so this method will create an empty one and return it to the EnterpriseContextMapper.
3. Parse the message to de-serialise the ExecutionContextID.
4. Set the ExecutionContextID on the EnterpriseContext component.
5. Parse the message to de-serialise the InstrumentationContextID.
6. Create an instance of the InstrumentationContext using its own factory method.
7. Attach the InstrumentationContext to the EnterpriseContext component.
8. Parse the message to de-serialise the AuthenticationTicket.
9. Create an instance of the SecurityContext using its own factory method.
10. The CreateSecurityContext() method validates the ticket via the Security package. This procedure is described in the security TED. If the validation fails, an infrastructure fault must be generated and sent to the client. After this is done the context must be destroyed via its KillContext() method. Remember that the full InstrumentationContext already exist at this point in the sequence and the failure can be reported.
11. If nr. 10 succeeds, attach the SecurityContext to the EnterpriseContext component.

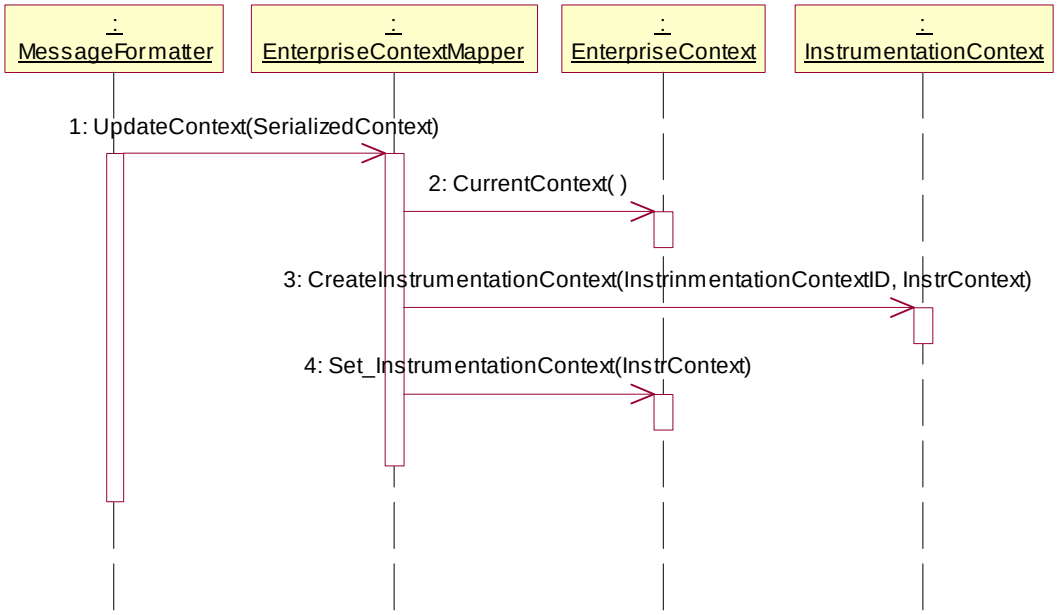
8.4.2.5 Serializing the Context components at system exit



Here follows a description of the interactions:

1. This diagram.
2. Call the CurrentContext() factory method. At this time the enterprise context does exist, so this method will just return it to the EnterpriseContextMapper.
3. Get the ExecutionContextID from the EnterpriseContext.
4. Add it to the SerializedContext.
5. Get the SecurityContext from the EnterpriseContext.
6. Ask it for the AuthenticationTicket.
7. Create a SerializedSecurityContext, add the ticket to it and add the whole lot to the SerializedContext.
8. Get the InstrumentationContext from the EnterpriseContext.
9. Ask it for the InstrumentationContextID.
10. Add it to the SerializedContext. It is important to note that instrumentation information MUST be serialised last to allow instrumentation to still take place during the serialisation process.

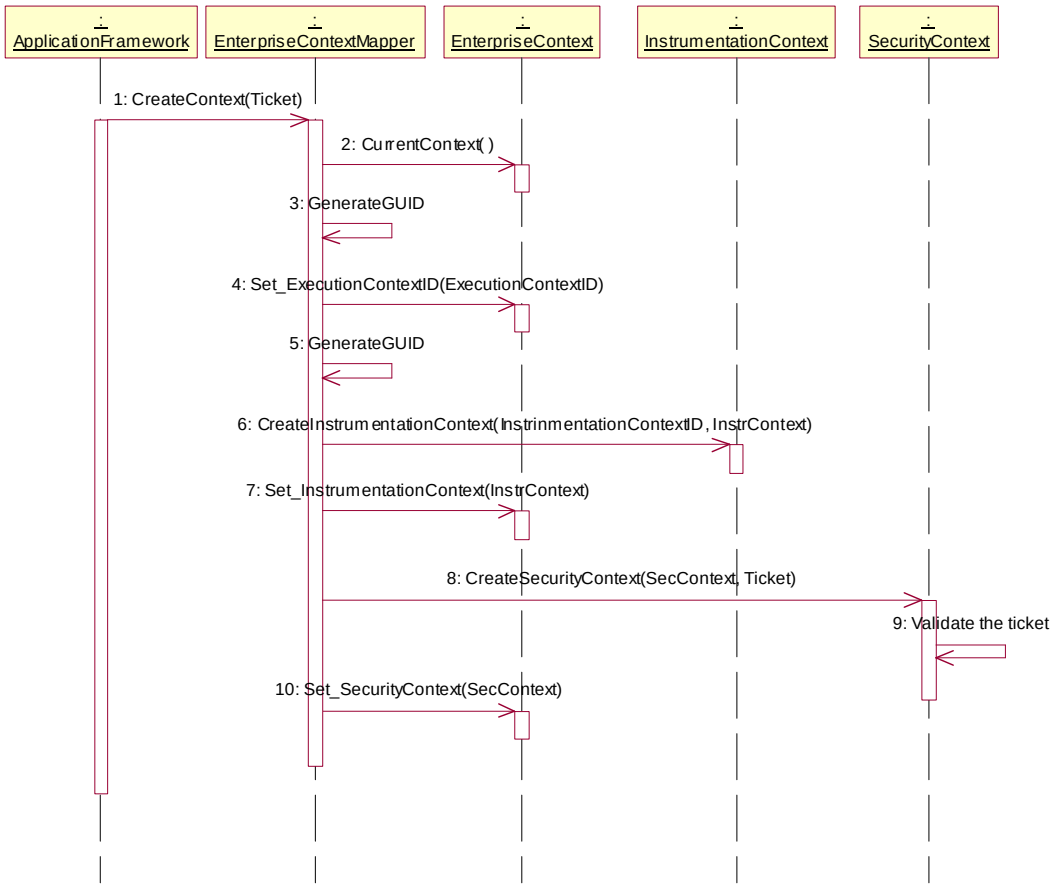
8.4.2.6 Updating the Context objects at system re-entry



Close inspection of the context serialisation diagrams will reveal that this one does less than the others. This is intentional. A service is only allowed to return changed information for the InstrumentationContext. Under no circumstances may it return changed information in the other areas of the enterprise context.
Here follows a description of the interactions:

1. This diagram.
2. Call the CurrentContext() factory method. At this time the enterprise context does exist, so this method will just return it to the EnterpriseContextMapper.
3. Create a new InstrumentationContext.
4. Attach the InstrumentationContext to the EnterpriseContext component.

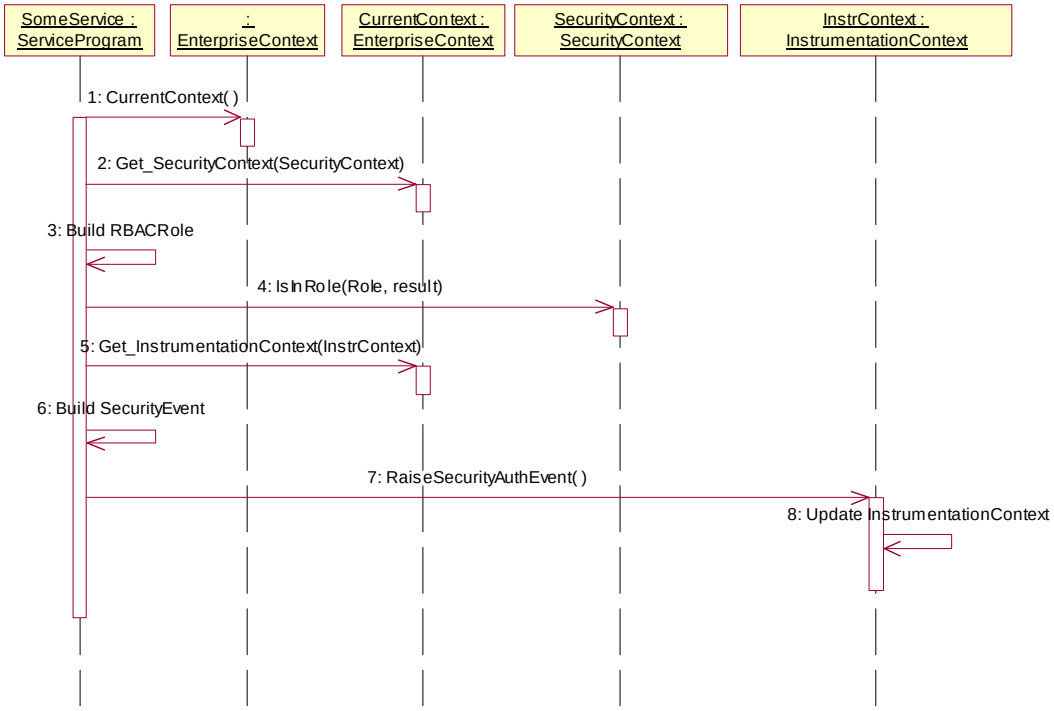
8.4.2.7 Creating the Context components for the first time



After this sequence is complete the EnterpriseContext is ready for use.
Here follows a description of the interactions:

1. This diagram.
2. Call the CurrentContext() factory method. At this time the enterprise context does not exist, so this method will create an empty one and return it to the EnterpriseContextMapper.
3. The ExecutionContextID is a GUID do create one from scratch.
4. Set the ExecutionContextID on the EnterpriseContext component.
5. The InstrumentationContextID is a GUID do create one from scratch..
6. Create an instance of the InstrumentationContext using its own factory method.
7. Attach the InstrumentationContext to the EnterpriseContext component.
8. Create an instance of the SecurityContext using its own factory method and the AuthenticationTicket that was passed into the CreateContext() method.
9. The CreateSecurityContext() method validates the ticket via the Security package. This procedure is described in the security TED. If the validation fails, a platform exception must be generated. After this is done the context must be destroyed via its KillContext() method. Remember that the full InstrumentationContext already exist at this point in the sequence and the failure can be reported.
10. If nr. 10 succeeds, attach the SecurityContext to the EnterpriseContext component.

8.4.2.8 Application Example: A service program doing role based authorisation and other Enterprise Services



This diagram depicts an example of a service program doing a role base authorisation check and then event the outcome through the enterprise instrumentation context.
Here follows a description of the interactions:

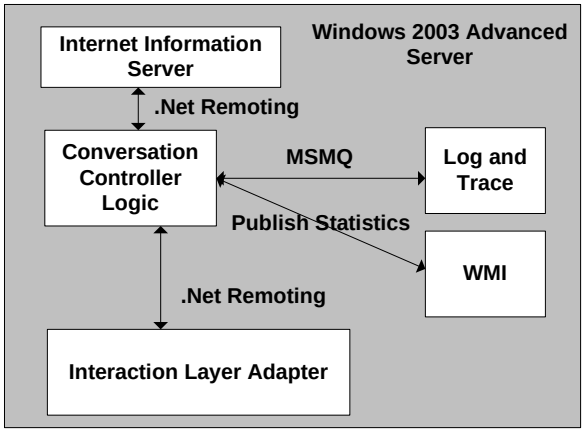
1. Call the CurrentContext() factory method. At this time the enterprise context does exist, so this method will just return it to the service program.
2. Get the SecurityContext.
3. Build a role descriptor.
4. Check for role membership. A result of true means that the current SecurityContext belongs to the specified role.
5. Get the InstrumentationContext.
6. Build a security event containing the outcome of the role membership check.
7. Raise the event.
8. Raising the event will cause the InstrumentationContext to change. This is described in the Application Instrumentation Architecture document.

8.5 DSF Touchpoint Package

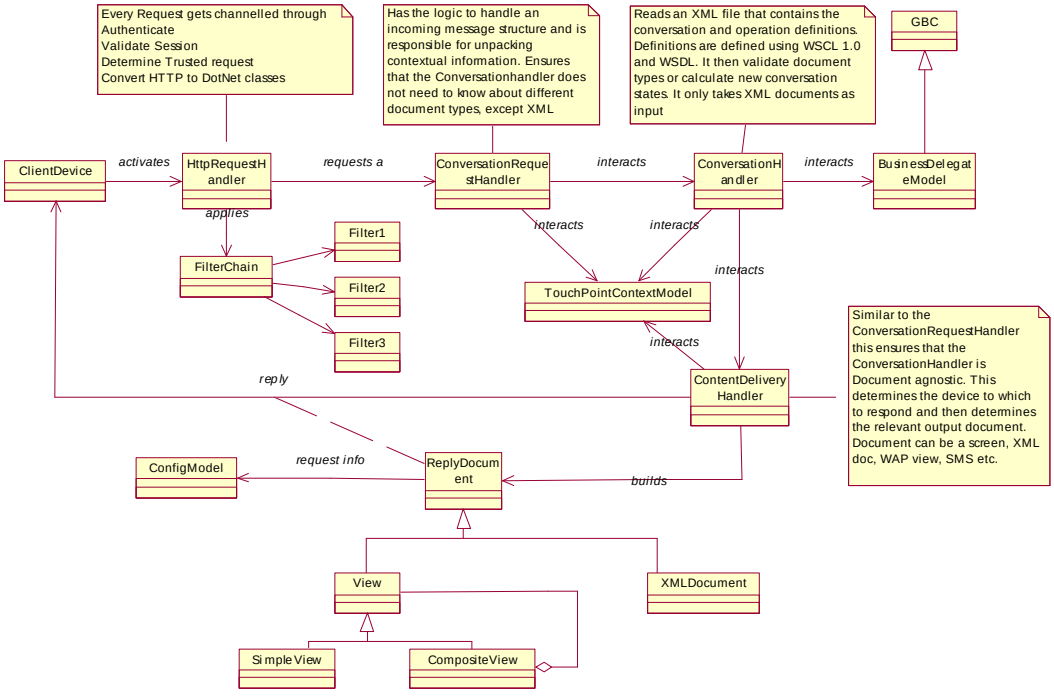
8.5.1 Components

Within the DSF Future view the following description was given to the Touchpoint:
"The Touchpoint Service Layer is the only layer that is aware of the channel specifics, and formats the content into a data stream that will be accurately interpreted and rendered by the browser on the channel (e.g. PC or Mobile phone). This layer will provide a mechanism for the easy incorporation of new distribution channels, dependent on the "style" that the new channel requires."

To accomplish this the following high level design was done for the Touch Point:



During a more detailed design a decision was made to base the design for the touch point on the Model-View-Controller pattern as well as the Strutts framework. HP has also done work on the concept of a conversation controller, although to handle conversations between web services. The result design for the touch point is therefore as follows:



The description of a conversation will be done as per the Web Services Conversation Language (1.0) as defined by HP and presented to the W3C. This was accepted by the W3C. The conversation is in the form of a finite state machine.

Note: Due to certain limitations to the standard to achieve the goal of the conversation controller, the following extensions was made to the standard:

- An action was added to a conversation interaction
- An event was added to determine which document to be used for output
- Nested conversations was added

Due to the power of XML, that allows extensions to be added without breaking a system that requires a pre-determined format, the extensions should be added without breaking the current WSCL 1.0 structure, ie it should still be possible for an existing WSCL 1.0 application to read the Nedcor conversation file without breaking.

8.5.1.1 Client device

The client device can be any device from which the user would like to interact with an application that resides on the Touch point. Technology currently exists that will convert most interactions into an HTTP request. For this reason the Client device is seen as coming in through a HTTP request and responces will be send back via an HTTP response. The request however will have to carry the type of device.

8.5.1.2 HTTP request handler

The HTTP request handler is based on the front controller pattern. The HTTP request handler manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing to the conversation controller and handling session validation errors.
Centralizing control in the request handler promotes code reuse across requests. It is a preferable approach to the alternative-embedding code in multiple views-because that approach may lead to a more error-prone, reuse-by-copy- and-paste environment.
While the Front Controller pattern suggests centralizing the handling of all requests, it does not limit the number of handlers in the system, as does a Singleton. An application may use multiple controllers in a system, each mapping to a set of distinct services.
An HTTP request handler is currently provided by MS IIS, this can be used to implement the HTTP request handler. The HTTP request will also convert the request into DotNet classes that can be handled by the rest of the framework and application.

8.5.1.3 Filter Chain and filters

All requests received are channelled through a filter chain that will call a set of filters. By creating pluggable filters to process common services in a standard manner, eliminates the need for changes to core request processing code. The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing. We are able to add and remove these filters unobtrusively, without requiring changes to our existing code.

We are able, in effect, to decorate our main processing with a variety of common services, such as security, logging, debugging, and so forth. These filters are components that are independent of the main application code, and they may be added or removed declaratively. For example, a deployment configuration file may be modified to set up a chain of filters. The same configuration file might include a mapping of specific URLs to this filter chain. When a client requests a resource that matches this configured URL mapping, the filters in the chain are each processed in order before the requested target resource is invoked.

8.5.1.4 Conversation Request Handler

This handler has the logic to handle an incoming message structure and is responsible for unpacking contextual information. Ensures that the Conversationhandler does not need to know about different document types, except XML.

8.5.1.5 Conversation Handler

The Conversation Handler, reads an external XML file that contains the conversation and operation definitions. Definitions are defined using WSCL 1.0 and WSDL. It then validate document types or calculate new conversation states. It only takes XML documents as input.

8.5.1.6 Business Delegate Model

The Business Delegate Model (BDM) The Business Delegate acts as a client-side business abstraction; it provides an abstraction for, and thus hides, the implementation of the business services. Using a Business Delegate reduces the coupling between presentation-tier clients and the system's business services. Depending on the implementation strategy, the Business Delegate may shield clients from possible volatility in the implementation of the business service API. Potentially, this reduces the number of changes that must be made to the presentation-tier client code when the business service API or its underlying implementation changes.
However, interface methods in the Business Delegate may still require modification if the underlying business service API changes. Admittedly, though, it is more likely that changes will be made to the business service rather than to the Business Delegate.
Another benefit is that the delegate may cache results and references to remote business services. Caching can significantly improve performance, because it limits unnecessary and potentially costly round trips over the network.

8.5.1.7 TouchPoint context model

This is the context that will keep information that is required by the conversation controller as well as by the Touch Point to perform it functions. This context will be valid fo rthe duration of the client's session.

8.5.1.8 Content Delivery handler

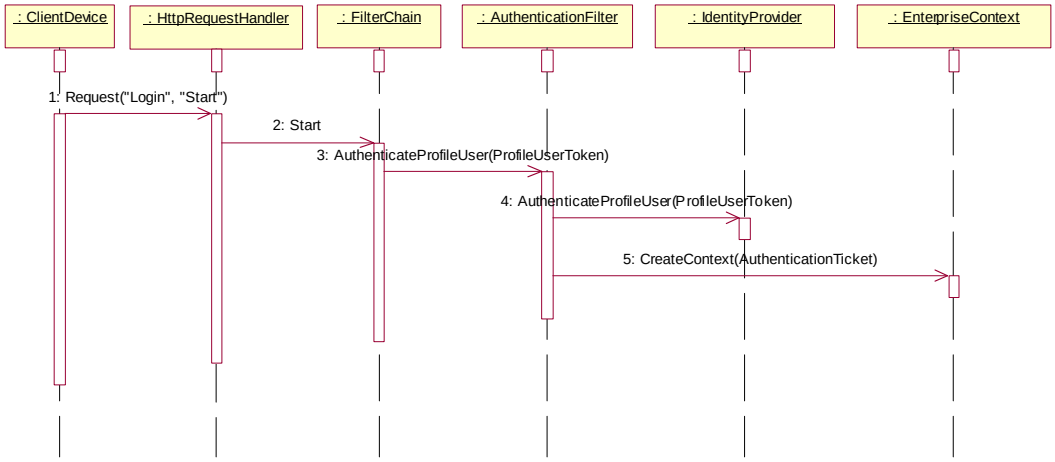
Similar to the ConversationRequestHandler this ensures that the ConversationHandler is Document agnostic. This determines the device to which to respond and then determines the relevant output document. Document can be a screen, XML doc, WAP view, SMS etc.

8.5.1.9 Reply Document and config model

This is the Document that is produced to sent back to the requestor as a response. Depending on the device, the relevant document will be build using data from the touchpoint context model. The config model for each type of document will provide detail about the document such as horizontal or vertical screen size, lines of data per document, etc. These exact config details will need to be agreed upon during detailed design.

8.5.2 Interactions

8.5.2.1 Logon / Authentication and Enterprise Context Creation

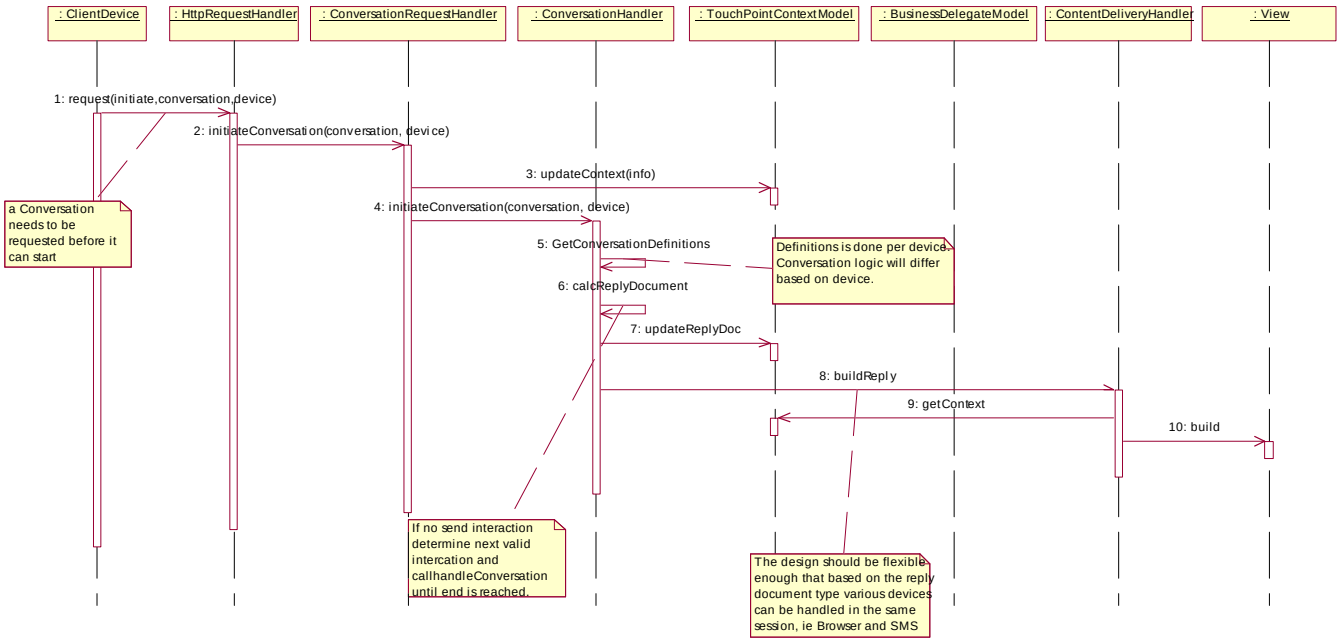


This diagram and subsequent discussion focuses on the infrastructure and security view of integration. The steps discussed here are the responsibility of the infrastructure and security designer.

Here follows a description of the interactions:

1. The Client device requests a login which will contain the Profile, PIN and Password information if it is a single profile user. For a multi-user profile the Login request will contain the Profile, UserID and Password. The login data is known as a ProfileUserToken. See security TED for more detail.
2. The HTTPRequestHandler starts the FilterChain. The filterchain determines that the first filter is to authenticate the user.
3. The FilterChain initiates the AuthenticationFilter with a ProfileUserToken. The Filter checks whether this user has a valid AuthenticationTicket.
4. If no valid AuthenticationTicket exists, then the IdentityProvider is called with the ProfileUserToken. This token is validated, see Security TED for more detail, and an AuthenticationTicket is send back.
5. Once the authentication ticket has been received the EnterpriseContext is called with the AuthenticationTicket.

8.5.2.2 Initiate Conversation

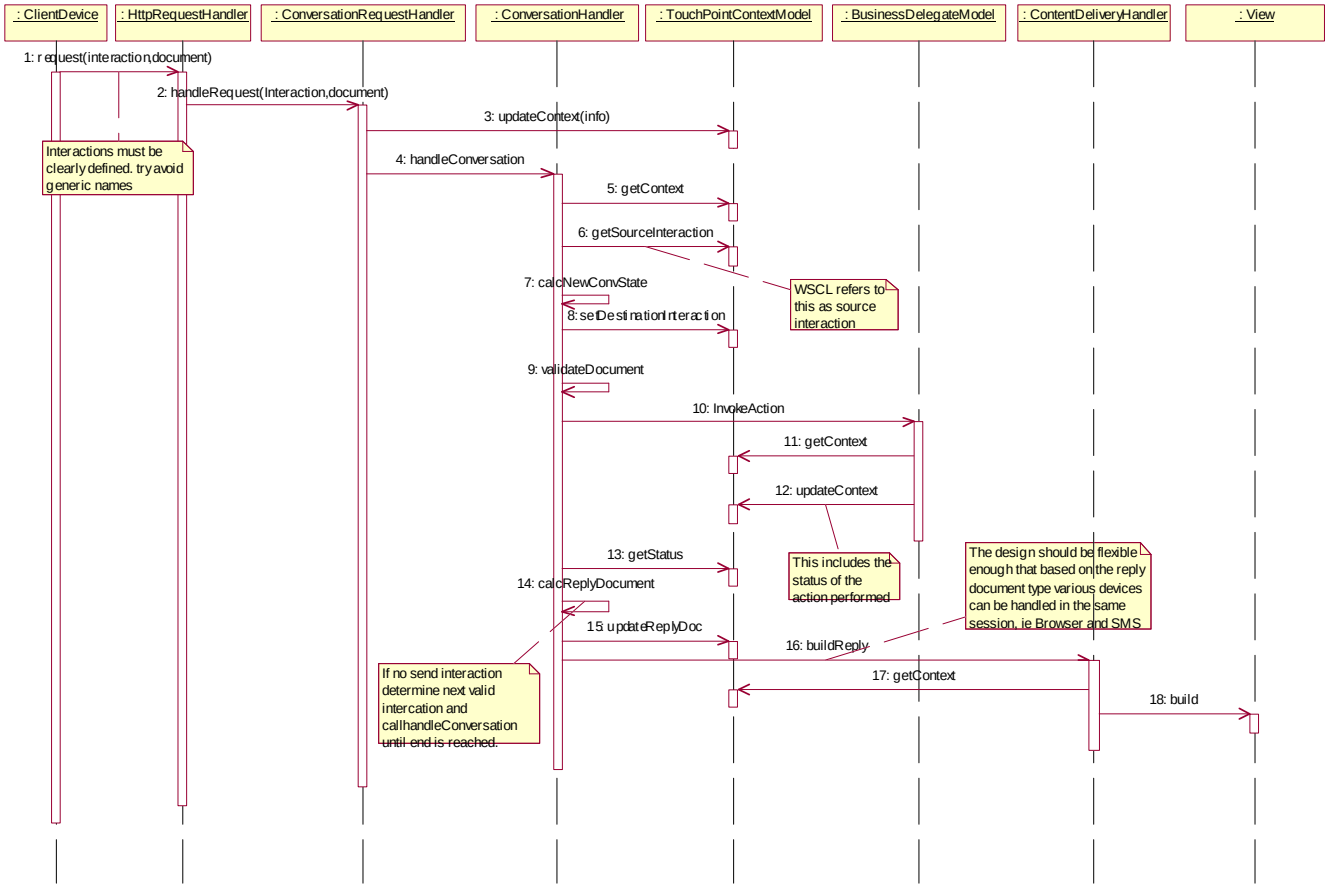


This diagram and subsequent discussion focuses on the infrastructure. The steps discussed here are the responsibility of the infrastructure designer.

Here follows a description of the interactions:

1. The Client device requests the Touch Point to start with a conversation on a specific device.
2. The HTTPRequestHandler calls the ConversationRequestHandler to start the requested conversation.
3. The ConversationRequestHandler informs the TouchPointContext of the new conversation that is to be started as well as the status of the conversation.
4. The ConversationRequestHandler requests the ConversationHandler to initiate the Conversation.
5. The ConversationHandler loads the relevant conversation from a configuration file and starts the conversation.
6. The ConversationHandler then determines the Reply document.
7. The TouchPoint Context is then updated with the Reply document
8. The ContentDeliveryHandler is then requested to build the Reply document.
9. The ContentDeliveryHandler gets all the relevant information from the TouchPoint Context and determines the type of document that needs to be build.
10. The relevant object is then requested to build itself and this is returned to the ClientDevice.

8.5.2.3 Request Conversation Interaction



This diagram and subsequent discussion focuses on the infrastructure. The steps discussed here are the responsibility of the infrastructure designer.

Here follows a description of the interactions:

1. The Client device requests a conversation interaction with a document. The specific device is send with the request.
2. The HTTPRequestHandler calls the ConversationRequestHandler to handle the interaction and progress the conversation.
3. The ConversationRequestHandler informs the TouchPointContext of the interaction and the document with its information that has been received.
4. The ConversationRequestHandler requests the ConversationHandler to handle the interaction.
5. The ConversationHandler gets the Context information from the Context
6. as well as the document that was sent as the interaction
7. Given the Conversation state and the interaction document received, the ConversationHandler determines the conversation state based on the Conversation that was loaded when the conversation was initiated.
8. The ConversationHandler now informs the context of the response interaction.
9. The data in the Interaction document is validated and
10. the relevant action is triggered on the relevant Business Delegate Model.
11. The BDM will get all the data it needs from the context to fulfil the required interaction.

12. The BDM will also update the Context after completing it's action with the result as well as the status of the action, ie successful completion, error codes, etc.
13. The ConversationHandler will get the status of the action from the Context and
14. based on the result of the action performed on the BDM, it will determine the next Interaction and reply document based on the conversation that was loaded during the initialisation of the conversation.
15. The TouchPoint Context is then updated with the Reply document
16. The ContentDeliveryHandler is then requested to build the Reply document.
17. The ContentDeliveryHandler gets all the relevant information from the TouchPoint Context and determines the type of document that needs to be build.
18. The relevant object is then requested to build itself and this is returned to the ClientDevice.

8.6 Detail Design Considerations

8.6.1 Open Standards

Open standards are those that a recognised and used by the broader IT industry. Only mechanisms and specifications that conform to open standards should be considered for use when doing the detail infrastructure design.

A compromise in this department should be considered very temporary and must be deprecated and replaced by standards based alternatives through maintenance releases of the infrastructure.

Where open standards causes design and implementation issues on a particular platform, the vendor of that platform must be involved to help specify alternatives and plan migration to the open standards.

While there are absolutely no alternative but to use a proprietary mechanism or specification, we must at least cater for a standards base solution in parallel to it.

8.6.2 Software that can create its own MessagingGateway and ChannelAdapter

If we ignore the consideration on open standards, we run the risk of not being able to integrate to COTS type applications that do adhere to the standards.

8.6.3 Existing components

On platforms where there are existing frameworks available, these should be exploited as far as possible. The detail design must first map in existing platform supplied components, before developing components are considered.

Platforms like Microsoft .Net and J2EE already supply a lot of the framework components to enable this infrastructure design. Even the current versions of CICS, MVS and DB2 have some support toward this infrastructure.

Should platform capability change, according to the vendor supplied migration plan, the infrastructure should be re-factored to use the new platform capabilities and deprecate self-developed code. This should be done through maintenance releases.

8.6.4 Context handling with no context in message

This condition can occur when integrating to applications that does not run the enterprise framework described in this document. When a message, containing no enterprise context, arrives at a node / tier, two options must be catered for:

- If the sending consumer cannot be authenticated, processing must terminate with a resultant fault message being sent back.
- If the sending consumer can however be authenticated, processing MAY continue. Should the node configuration allow processing to continue for particular consumers, that node will be the creator of the enterprise context, using the security information of the consumer. Any request made from this node will process the enterprise context as per this design.

8.6.5 Handling errors that occurs when the context does not exist yet

Errors like all events should be reported using the InstrumentationContext. There are however tiny areas of processing where the InstrumentationContext is not fully instantiated yet. Should errors occur during this processing, they must be reported using the standard error reporting mechanism for that particular platform.

It must be noted that errors reported in this fashion will not be correlated with other events that are reported via the InstrumentationContext. The occurrence of this condition will in all likelihood be very rare and then only under exceptional circumstances.

It cannot however be ignored.

8.6.6 Handling instrumentation between SerialiseContext() and UpdateContext()

Calling the instrumentation methods on the InstrumentationContext component will change the InstrumentationContextID. The InstrumentationContextID is used to correlate instrumentation events that happen across the enterprise. This means that once it is serialised care should be taken not to break this correlation. Careful thought must be given to this condition in the detail design of the infrastructure.

8.6.7 Handling duplicate messages

Messaging infrastructures can suffer from a particular condition, where a particular message arrives at a processing endpoint more than once. This can be as a result of various factors, including programming errors. This can have adverse effects like inconsistent business state or denial of service.

The infrastructure must treat each request or result from an application as a unique message. Even if that message contains exactly the same application data as a previous message, there is no consistent way for the infrastructure to understand the application's intent and therefore must honour the request or reply.

Architecture suggest that this condition may be catered for as part of the BuildContext() method of the EnterpriseContextMapper component. Detail design must give careful thought to how to implement this on each platform, since the state management can become complex very quickly.

The bare minimum implementation must ensure that a particular message from an application will be delivered only once per endpoint. More exotic implementations that implement full idempotent behaviour may be explored, but is not currently an absolute requirement.

Since messages are delivered between endpoints, it would be admissible to add a unique MessageID (GUID) to the header of the MessageEnvelope (not to the SerializedContext – it does not span the enterprise).

8.6.8 Service Identification in a generic ServiceActivator

On the .Net platform the standard Web Service ServiceActivator is implemented to be generic. That means that there exist only one ServiceActivator that can invoke any operation on any service. Service components inherit from this generic ServiceActivator. It uses some channel information to determine which service is being invoked. The channel it relies on is the SOAP / HTTP channel that is part of the .Net framework and the information used is the URL of the request.

When using Microsoft Host Integration Server, the HIS runtime can send a transaction code with its message to the host. A generic form of the ServiceActivator on the host can then use the transaction code to identify the specific service being invoked.

Service endpoints should be defined as URI descriptors. MessagingGateway components should bind to a logical URI descriptor, while the physical descriptor must be resolved at message routing time.

The point is that a service is identified by its endpoint and during detail design careful consideration should be given to defining a service identification scheme that will be common across the enterprise.

Please note that this problem does not exist in the case where a particular ServiceActivator only ever invoke the functionality on one service program.

8.6.9 Use of the Pipes and Filters pattern for the infrastructure processing

Where possible detail design should design infrastructure processing using the Pipes and Filters design pattern. The pipeline should support registration of new processing components as well as sequencing. This will enhance future extensibility.

8.6.10 Use of low level Authentication

See “Context handling with no context in message” in this section.

8.6.11 De-Serialisation overflow

Another problem with messaging infrastructure is that there is no consistent way to know how much data is actually on its way and still on the wire. This is one of the biggest concerns for denial of service attacks. Care should be taken in detail design not to make any assumptions on the size of network messages read of a particular channel. Platform restrictions should be strictly enforced and parameterised. Offending channel connections must be dropped and proper instrumentation and logging must be done for any attempts to breach the platform restrictions.

9 PHYSICAL VIEWPOINT

9.1 Technology Usage

Technology	Use Location	Notes / Comments
Customer Browser	Customer Workstation	This is one of the devices that a customer can use to access the DSF Touchpoint. The browser type will affect the applications and views that run in the Touchpoint. From an infrastructure viewpoint it only needs to support HTTP (S) and the Nedcor security framework.
Customer Application	Customer Workstation OR Customer Server OR Customer Backend	This is one of the devices that a customer can use to access the DSF Touchpoint. The infrastructure requires the same functionality as the browser plus the capability to consume standard WSDL based Web Services.
Microsoft Internet Information Server (IIS)	Touchpoint Server, Application Server, Interaction Server	Hosting environment for Wintel servers.
Microsoft COM+	Touchpoint Server, Application Server, Interaction Server	Hosting environment for Wintel Servers.
Microsoft SQL Server	Front-End data stores.	Primary Microsoft database solution.
Microsoft Host Integration Server (HIS)	Messaging Bridge	This software can help us to integrate easier between off-host consumers and on-host services. By applying the Messaging Bridge pattern, HIS can assist us in bridging between a SOAP / HTTP channel and various host supported channels.
Microsoft .Net Framework	Touchpoint Server, Application Server, Interaction Server	The current de facto standard for developing solutions on a Windows platform.
DLI	ChannelAdapter and MessagingGateway	The DLI is currently prevalent in the integration space in Nedcor. It should be evaluated against the constraints put down in this design to determine its applicability in the infrastructure space.
MQ	Channel and Protocol	MQ is another way of delivering messages in a queued and asynchronous way. As with the DLI, it should be considered in detail design for applicability.
DB2	Host	Primary Host database solution.

Technology	Use Location	Notes / Comments
PFC (Process Flow Controler)	CICS	Currently the PFC is used to to pipeline processing control and transformation management for programms that run in CICS. This component can still be used as part of the application processing internal to service programs. It is only available on the host and is therefore not seen as an enterprise component, but rather an application support component for applications that run in the CICS environment. Detail design can asses its applicability to do infrastrucuture processing, but architecture propses that we do not use it for that purpose, because of the architectural objective listed in this document..
IBM CICS	Host	Hosting environment for services on the Host.
Tivoli	Everywhere	Primary platform for consuming instrumentation