



Misys Message Manager V4 Technology

White Paper

Contents

Overview	3
Business Issues	3
Standards Proliferation.....	3
The SWIFTNet 'Single Window'.....	3
Back-office System Integration	4
Straight Through Processing.....	4
Compliance	4
Cost Reduction.....	4
Misys Message Manager Functionality	5
Overview	5
Message Manager and the SWIFTNet Integration Model.....	5
Back-office Systems Integration.....	6
Orchestration.....	6
Standards.....	7
Directory Services	7
Hub-Ready	8
Value Add.....	8
Manual Intervention.....	9
SWIFTNet Connection	9
Message Manager Architecture	10
Overview	10
Data Flow	12
Security	13
Message Broker	14
Adapters, Connectors and Connector Types	21
Message Manager Application.....	26

Overview

Misys Message Manager V4 is a J2EE application for processing financial messages throughout their lifecycle: from extracting data from host systems to creating messages, through manual and automated message management to message delivery, tracking, audit and archive. The purpose of this whitepaper is to describe the problems that Message Manager sets out to solve, the functionality it implements and the technology on which that functionality is based.

Business Issues

Financial institutions face increasing challenges when dealing with financial messages:

Standards Proliferation

Until recently, banking financial messaging was primarily concerned with just one message standard, SWIFT FIN. However, although FIN remains of prime importance, a number of other standards are appearing and most financial institutions will need to support at least some of them:

- UNIFI or ISO 20022 XML
- FIX and FIXML
- TWIST
- RosettaNet
- FpML
- XBRL
- MDDL
- OMGEO

The SWIFTNet 'Single Window'

By the end of 2004 SWIFT have migrated fully from the x.25 network to the new SWIFTNet IP network (SIPN). One of the key benefits of this change is that SWIFTNet can be used for more than just FIN traffic; for connections to market infrastructures such as CLS, for other

SWIFT services such as FileAct and InterAct and for connections to non-financial institutions, such as the members of a MA-CUG. Although this is a powerful capability, the challenge is to manage effectively the various different types of traffic that converge on the SWIFTNet connection.

Back-office System Integration

Conventionally, the generation and consumption of financial messages has been seen as the domain of back-office systems. This is a natural development because it is in these systems that the data required for the messages usually resides. However, as message standards proliferate and evolve, the overhead of maintaining compatibility in back-office systems that may be old and expensive to change becomes prohibitive.

Straight Through Processing

To increase Straight Through Processing rates is a common goal of all back-offices, but there are difficulties:

- Legacy systems may produce invalid or incomplete messages
- Legacy systems may support only crude batch style interfaces to financial networks – or they may not support an interface at all, necessitating costly and error-prone re-keying of messages
- Message control systems may be insufficiently flexible - forcing users to manually review messages that should be sent straight through in order to catch those that should not
- Incoming messages may lack necessary data (such as account numbers), or that data may be present but incorrect, in either case requiring costly manual repair
- Standard Settlement Instructions (SSIs) may be maintained in a number of different systems, requiring co-ordinated manual effort to renew and maintain

Compliance

As increasingly strict regulatory requirements emerge, watch-list checking has become an important fact of life for the back-office, and a source of cost and processing bottlenecks. Also, compliance with new regulations such as SEPA in Europe requires multiple changes to systems over lengthy periods.

Cost Reduction

Increasing STP and minimising processing delays can lead to significant cost savings, but there are other potential sources of savings that can be addressed by a modern centralised messaging infrastructure:

- Message processing hubs can reduce the need for costly infrastructure in branches
- Message processing hubs can switch messages between users of the hub without requiring messages to be sent via the SWIFT network, resulting in significant savings on messaging fees

Misys Message Manager Functionality

Overview

Message Manager addresses the issues set out in the previous section from a number of angles:

- Supports a variety of message standards in a single central architecture
- Provides a single point of control for financial message traffic
- Interfaces with legacy systems using a variety of technical connection types
- Provides comprehensive message processing functionality that can be shared by all the systems that connect to it, including interfaces to reconciliation, matching and compliance systems
- Supports regional or global hubs, high availability, multi-entity, multi-lingual, multi-host

The sections that follow describe these functions in more detail, linking them to SWIFT's published recommendations for SWIFTNet message processing systems.

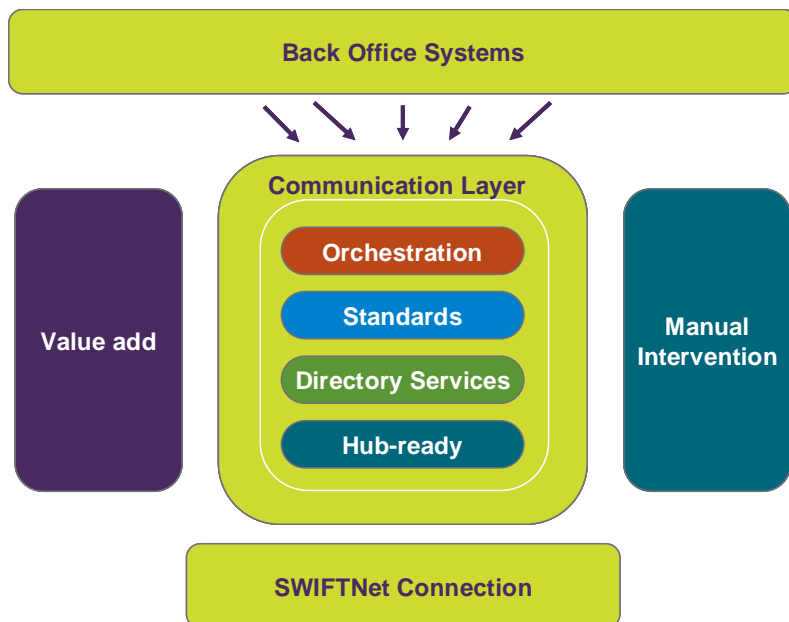
Message Manager and the SWIFTNet Integration Model

SWIFT has published a whitepaper that describes an abstract 'reference architecture' for banks connecting to SWIFTNet¹. This paper proposes a 'value-added EAI² hub' that should be positioned between back-office systems and the SWIFTNet network. The hub acts as a concentrator and consolidation point for all message traffic and is rich in message processing functionality. It also offers end users a consistent view of messages from a variety sources with a single access mechanism, and offers the IT department powerful tools to facilitate the implementation of new standards and business processes without affecting legacy systems.

¹ See http://www.swift.com/index.cfm?item_id=41985

² EAI – Enterprise Application Integration

Message Manager is Misys' interpretation and implementation of this recommendation. The diagram below shows the principal functional components:



Back-office Systems Integration

Message Manager can connect to back-office systems using a wide variety of technical interfaces in the communication layer, including:

- WebSphere MQ
- JMS
- Database (JDBC)
- Flat file
- TCP/IP sockets

Data can be exchanged in a variety of formats, including:

- XML
- SWIFT FIN
- Fixed length
- Delimited (e.g. CSV)

Message Manager's extensive data mapping capabilities can be used to create SWIFT or other standard financial messages from non-standard data extracts.

Orchestration

Message Manager orchestration directs messages to the people, functions or systems that need them. Orchestration rules can interrogate any field of the message to decide on its destination. All transitions through an orchestration are audited, and message

images are recorded if ever the content of the message changes. The user interface includes an audit viewer and query function that allows a message's path through the system to be tracked and the content of the message at any point in its progress to be viewed.

Standards

Message Manager includes detailed meta-data definitions of the financial message types it supports. Built in parsers and formatters use these definitions to convert messages from their native formats into hierarchical message objects for easy manipulation, and from message objects back to the native formats for network transmission. Message Manager's mapping and orchestration functions operate on messages in object form. The message objects combine the data content of the message with the meta-data from the definition, so it is possible in these functions to test for the value of a field and also, for example, whether the field is mandatory or optional – information which comes from the meta-data definition.

Message Manager extends the basic financial message definition to include common details that are maintained for all messages. These include:

- The host (back-office) system which produced the message (or the extract from which the message was created)
- The business entity that owns the message
- The line of Business to which the message corresponds
- The sender and receiver
- The value date, maturity date, currency, amount and beneficiary (where applicable) of the message

Some of these values are populated in the host system interface; others can be derived from the message directly (for example, Message Manager's definition for the SWIFT MT103 payment message includes meta-data that instructs the parser to extract value date, currency and amount from field 32A).

Directory Services

Message Manager incorporates an online copy of the SWIFT BIC+ directory. This directory is used to validate messages and enhance the presentation of messages for display and reports. It can also be used to automatically enrich messages, adding clearing system references, for example.

Other types of directory may also be integrated with Message Manager. For example a directory of IBAN issuers cross-referenced against BIC codes can be used to validate IBANs that appear in messages. Or an SSI directory can be used to enrich messages, obviating the need to hold SSIs in the back-office system.

Hub-Ready

Message Manager is designed for running regional or global hubs. User access is pure-thin client, delivered through a browser for zero impact deployment. The server is designed for full 24x7 operations. It maintains multiple simultaneous processing dates, and working day and time zone settings for different regions. The data model identifies messages according to both the originating host system and region. Time zone and locale information is used to offer users a locale-centric view of their messages no matter where the central server is located.

Message Manager's security model allows user access to messages to be configured in business terms, which allows a wide range of users to share the same server; a user can be authorised to work with messages from a given host, region or business entity, or a combination, depending on the needs of the business.

Use of a hubbed system offers significant advantages in terms of cost and control:

- Messaging infrastructures are costly to maintain. A Message Manager hub allows many branches to share a single connection to SWIFTNet
- A hubbed system allows a global view of message traffic, allowing many back-office functions (watch-list checking, matching, reconciliation, exception processing) to be centralised
- Single point of control. A hubbed system allows managers to monitor message traffic for all parts of the enterprise
- Often, a significant proportion of a bank's message traffic is between its own branches. Using Message Manager's internal routing capability, messages sent from one hubbed location to another can be intercepted and routed internally in Message Manager, resulting in instantaneous delivery with no SWIFT message charges
- The ability to exchange messages between branches instantly and with zero incremental cost can allow banks to offer new or improved services to customers, such as retail cross-border payments

Value Add

Message Manager includes a range of financial messaging-specific processes that can be linked into the workflow:

- Automated cancellation processing
- Automatic creation of multiple messages from single messages
- Automatic payment release on confirmation match
- Automated watch-list checking
- SWIFT FIN validation
- CLS confirmation tracking and net payment generation
- ISO-15022 Market Practice guideline checking
- Internal message routing

Manual Intervention

Message Manager includes a comprehensive thin client GUI for end-users to work with messages. The GUI is designed to support global message hubs; it is fully internationalised, recognising differences in locales, end-user languages, time zones, date formats and amount formats.

The GUI functions available include:

- User-defined queries for messages and audit events
- Manual message repair
- Manual message origination
- Manual routing
- Manual deletion

SWIFTNet Connection

Message Manager includes packaged interfaces to *SWIFT Alliance Access*³, using MQSA, for the exchange of FIN messages, including full tracking of Alliance Positive and Negative Acknowledgements (PANs and NANs), and network acknowledgements (ACKs and NAKs). Message Manager can communicate with other SWIFT CBTs such as SWIFT Alliance Entry, MINT, BESS or Merva using WebSphere MQ or flat-files.

Message Manager also connects to *SWIFT Alliance Gateway*⁴ using MQHA, for the exchange of files over FileAct or interactive messages over the InterAct service.

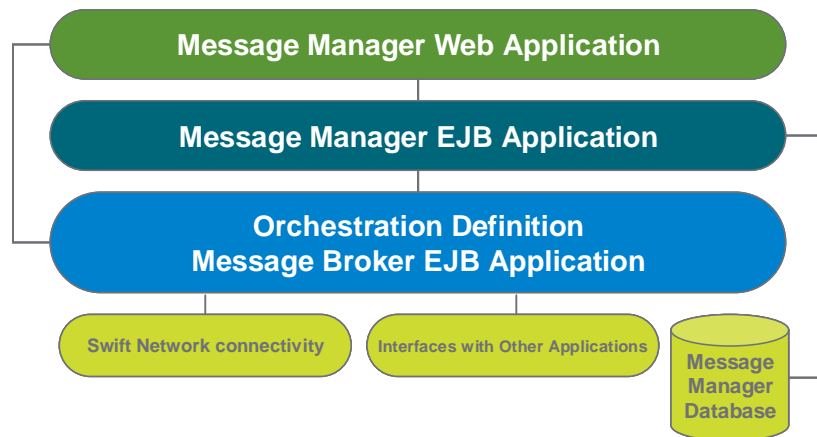
³ See http://www.swift.com/index.cfm?item_id=2341

⁴ See http://www.swift.com/index.cfm?item_id=2321

Message Manager Architecture

Overview

Message Manager V4 is a J2EE application composed of several layers, as shown below:



At the lowest level, messages are exchanged with SWIFT network devices and with other applications. A message broker component handles parsing, formatting, transformation and orchestration of these messages according to rules given in the Message Manager orchestration definition. Messages are stored in Message Manager's database.

The Message Manager EJB application handles business logic and database access for the Message Manager web application, which provides a user interface in HTML, allowing users to query and work with messages via a web browser.

Message Broker EJB Application

The message broker component provides generic message parsing, formatting, routing, transformation and orchestration capabilities to Message Manager. It uses message definitions and a message workflow that are provided in a repository, which is shipped with Message Manager. Clients who have purchased the Message Manager Meridian Toolkit can create custom repositories. A *repository editor* creates an EAR archive based on each repository and deploys it into the application server.

Messages in an orchestration pass through a series of queues, which can be realised as physical destinations (e.g. a SWIFT network device) or simply as states (e.g. verified or confirmed); each message's current queue is recorded in the database. Message routing and transformation rules written in Java are attached to queues in the repository. Queues can be configured to route messages automatically at regular intervals, or when prompted by a user via the user interface.

Each queue has an *adapter*, which allows it to communicate with a storage or transport mechanism. Standard adapters are included for the most commonly used mechanisms, such as JMS, relational databases and file systems. An API is provided for custom adapters. Message Manager uses its own adapter to persist messages in its database and to maintain audit logs that track changes to messages.

Queues can be configured to send messages through application-specific *filters* during routing; Message Manager includes filters that process message confirmations and cancellations as messages move through the workflow.

The J2EE application server provides transaction coordination, using the two-phase commit process, for transactions involving multiple adapters.

The message broker provides an EJB façade in the form of a stateless session bean; other applications can make RMI calls to the façade to inject messages into the workflow, pick up messages from queues, parse and format messages, check the validity of messages using SWIFT validation rules and access other services provided by the message broker in a SOA.

In order to provide long-running processes (which are not a feature of J2EE), the message broker includes the *Control Module* – a client application that prompts the activity of components that access messages. The control module also implements a JMX⁵ management capability, which can be used by the message broker's own management console, and by third party JMX-enabled system management tools. The management interface provides methods for stopping and starting processes, and provides feedback on the state and health of the system.

Message Manager EJB Application

In addition to messages, the Message Manager database stores application-specific data such as security information (users, roles and capabilities) and user-defined message queries. In keeping with J2EE design conventions; this data is managed by an application-specific EJB layer with its own session bean façade.

Message Manager Web Application

The Message Manager user interface is implemented using JavaServer Pages (JSPs) and Servlet technology. The Jakarta Struts⁶ framework is used to provide a robust implementation of the Model-View-Controller design pattern, in which the Controller is a Servlet (Sun Model 2).

⁵ See <http://java.sun.com/products/JavaManagement/>

⁶ See <http://struts.apache.org/>

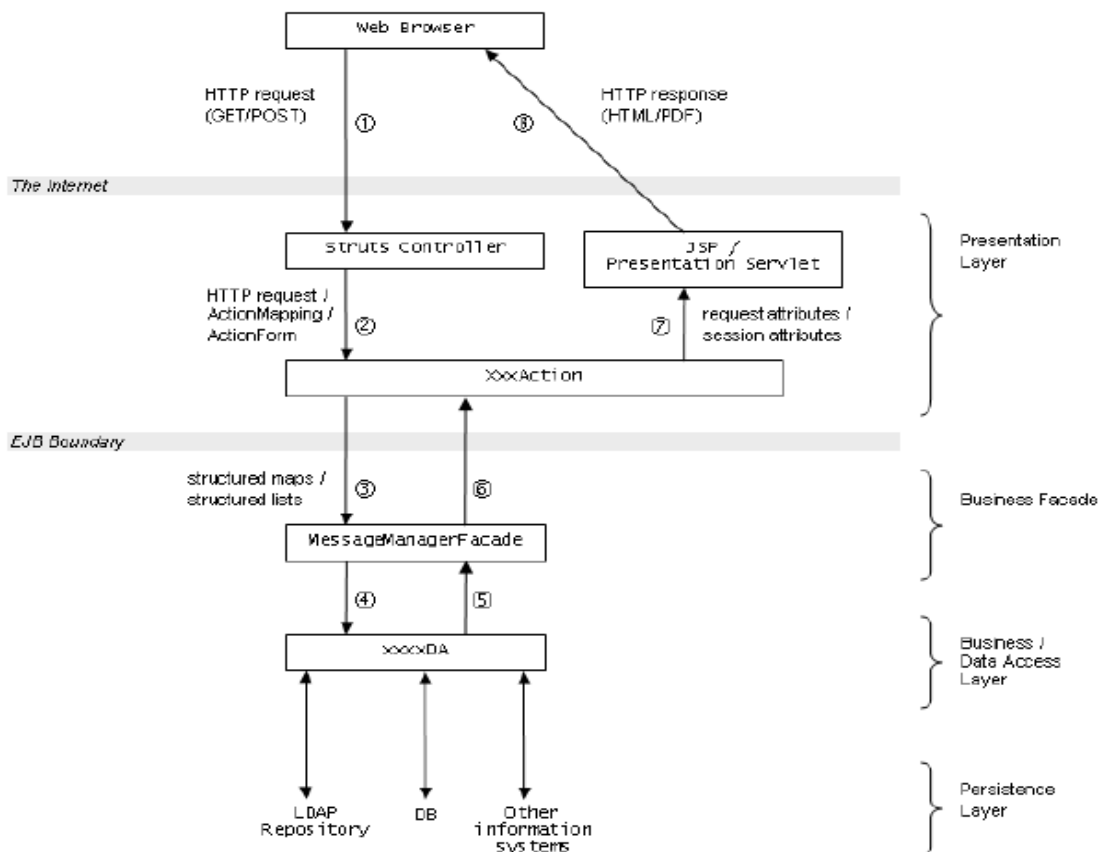
All mark-up is XHTML. Dynamic HTML techniques are used to deliver a user experience that is close to a thick client in terms of ease of use for example: status information is updated automatically; menu options are presented in pull-down menus. All JavaScript operates on the W3C Document Object Model (DOM) API.

There is no mixing of scriptlets and mark-up in JSPs: all code is maintained in reusable tag libraries. Where appropriate, functionality from the JavaServer Pages Standard Tag Library (JSTL) is used. Where there is no standard function custom tags are added to Message Manager's own tag library.

Internationalisation of screens uses JSTL fmt: tags and Java resource bundles. All style information (colours, types, borders, etc.) is contained in Cascading Style Sheets (CSS), and can be changed easily without affecting functionality.

The web application makes method calls into both the message broker EJB façade and the Message Manager EJB façade, depending on the type of data it is working with. For example, it uses the Message Manager façade to get the headers of messages that satisfy the criteria in a user-defined query; when the user clicks on a message header, it uses the message broker façade to retrieve the full contents of the message.

Data Flow



Message Manager is a web-enabled multi-tiered J2EE application. The tiers consist of presentation, business and data layers. The Presentation layer is composed of servlets/JSPs/struts actions, the business and data layers execute within EJBs, and the data access itself is performed through Spring and Websphere-supplied authentication mechanisms.

The diagram shows how information flows through the tiers during a typical HTTP request.

1. The end-user clicks on a link or performs some other task that causes their browser to send a HTTP request to the Message Manager web application. This request is received by the application server, which determines the web application for the request, and (presuming this is a request that actually performs work) invokes the strut Controller Servlet to manage the request
2. The struts controller matches the request against the Action code that will process that request. The controller performs coarse-grained security checks, then passes control to the action, supplying the original request, the mapping, and any struts-supplied Action Form objects
3. The action receives the request, validates all user-supplied input, and either returns directly if it can service the request, or hands it to a back-end EJB if information must be sent or retrieved from the database or LDAP. The request is sent to the Message Manager façade, which is a stateless session EJB containing entry points to all Message Manager related business logic functions
4. The EJB performs fine-grained security checks, performs the work requested, usually through passing to a static method in a class defined in the Data Access package
5. After the Data Access class performs its data access tasks, it returns the result to the EJB
6. The EJB performs standard exception handling and benchmarking, and returns the result to the web application
7. The web application may invoke further EJBs to perform processing (going back to step 3 in the diagram), or determine which page is responsible for rendering the result. The struts framework takes this logical name (e.g. 'success') and maps it to a JSP or servlet (e.g. '/query/queryUpdated.jsp'). The action code is responsible for setting any request attributes used by the JSP to display dynamic data
8. The JSP generates XHTML, JavaScript, or some other browser-visible resource, which is sent back to the user

Security

Authentication

Message Manager does not include authentication mechanisms of its own; instead, it relies on the application server's authentication mechanisms. A system administrator

can configure the application server to use pluggable user registries containing usernames and passwords, such as the LDAP directories used by many enterprises.

Authentication is enforced in the web application using J2EE form-based authentication: when a user attempts to access a JSP page, the application server checks whether the user has logged in, and redirects the user's browser to a login page if necessary. When the user submits the login form, the application server checks the user's username and password in the user registry configured by the system administrator.

The EJB façades also require an authenticated user. When the web application calls an EJB method, the application server transparently passes the user's credentials to the EJB container. A standalone Java client program that wants to call EJB façade methods can use JAAS programmatic login to authenticate itself with the application server.

Authorisation

The Message Manager database includes data about role membership and the capabilities assigned to users and roles. A system administrator can edit this information using Message Manager's user interface. Capabilities are used in the following ways:

- In the EJB layer, to restrict data sets returned to the user. For example, users can define their own queries to search for messages of interest to them, but they are only shown the messages that they have permission to see
- In the web application, to enable and disable user interface elements such as buttons and links
- In the EJB layer, to verify that, when a user requests an operation to be performed (e.g. by clicking on a link or submitting a form), they have permission to perform the requested operation

A capability is defined as an action (such as *view* or *edit*) and a resource type (such as *message*) and optionally a set of criteria (e.g. *value date* is today, and *value* is less than *10,000 pounds sterling*). These criteria can then be evaluated in memory (e.g. to enable or disable aspects of the user interface) and can also be used to generate WHERE clauses to restrict SQL queries.

Message Broker

Overview

Host system connectivity, mapping, routing and orchestration services are provided by Message Manager's message broker, which sits between back-office and network

interface systems, accepting messages from message producers, transforming them into the form required by message consumers and delivering them. Some systems may be both producers and consumers of messages, and there may be many connected systems, with complex rules governing their interactions; the message broker coordinates the message flows between the systems, ensuring that messages arrive at the right place and in the right format. The message broker is also used to realise Message Manager's workflow orchestrations. In this case, data is not physically transported across system boundaries; rather, messages are moved between logical queues that represent the state of the message – 'where it is', what has happened to it, what can happen to it next.

The logic that determines the transformation and distribution of messages is provided by the user in the form of mapping and routing *scripts*⁷. The message broker provides an execution environment for the scripts, which includes components for interfacing with data transport and persistence mechanisms, parsing and formatting services and mechanisms for ensuring transactional integrity.

The message broker's behaviour is defined by a *repository*; a single file that contains all the data required to describe the communicating systems, their connections, the messages they produce or consume and the logic required for mapping and routing messages. The sections that follow describe the major components of the repository.

Systems

Systems define the computer systems that connect to Message Manager. Systems are principally characterised by the messages that they produce or consume, and the interfaces that they support to other systems. *System Interfaces* are defined in terms of the mappings required to transform messages defined for one system into those defined for another.

Message Manager is supplied with a *SWIFT* system, which includes definitions for SWIFT FIN and UNIFI / ISO 20022-XML messages.

Message Definitions

Message definitions describe the messages processed by the system. A message definition has a unique name within a system, including an optional version qualifier. Messages are composed of *fields*, *field lists*, *repeat blocks* and *sub-messages*:

⁷ 'Scripts' in this context are not scripts in the commonly used sense of interpreted source code. Message broker scripts are written in Java and compiled before deployment.

Fields

A field is a single data item in a message. Fields may be of several types:

Type	Comment
String	Any number of characters (<i>a Java String</i>)
Number	A floating point number (<i>a Java double</i>)
Date	A date/time (<i>a Java java.util.Date</i>)
Day count	A date expressed as a number of days from a given base-date

Field Lists

A field list is a list of fields that can be defined once then used in a variety of messages. It is commonly used for headers or other sequences of fields that are used in many places. At runtime, fields derived from a field list are indistinguishable from fields that are defined directly in the message definition.

Sub-messages

A reference to one message definition can be inserted into the definition of another message, where it becomes a *sub-message*. In practice, a sub-message is very similar to a field list, except that a field list cannot be used as a message in its own right, and at runtime a reference to a sub-message must be obtained before its fields can be accessed.

Repeat Blocks

A repeat block represents an iterated sequence in a message. A sequence of fields defined within a repeat block may appear more than once: either a fixed number of times, or an arbitrary number – zero or more. Repeat blocks may contain fields, field lists, sub-messages or other repeat blocks. Iteration may be nested to any number of levels.

Message Meta-data

All message components are further described by meta-data, which is required by the system to parse and format messages. The meta-data required depends on the technical formats in which the messages are exchanged. An extended discussion of the meta-data required by the various built-in formatters and parsers can be found in the *Formats* section of this document.

Namespace Definitions and Namespace Lists

Namespaces⁸ are used to qualify XML tags in XML documents to guarantee their uniqueness. A namespace prefix is concatenated with an XML tag to ensure that tag's uniqueness in the document (e.g. `<abc:customer> ...</abc:customer>` where 'abc' is the namespace prefix). The prefix is associated with a URI in a namespace declaration that appears in the header of the XML document, which ensures the global uniqueness of the namespace.

⁸ See <http://www.w3.org/TR/REC-xml-names/>

A *Namespace Definition* can be created at the system level. The name of the definition is the URI. Its only attribute is the prefix to be used when formatting a message (when parsing a message the prefix used to identify an element is the one given in the document's namespace declaration).

A *Namespace List* is a collection of Namespace Definitions that can be associated with a message. Typically a message will have a principal namespace that will be used to qualify most of the fields (the *Active Namespace*). But some fields or repeat-blocks may require a different namespace. In this case the active namespace can be overridden at the field or repeat-block level by one of the namespace definitions found in the message's namespace list.

Function Definitions

Function definitions allow static Java functions to be defined that can be accessed from other Java scripts. Functions are a convenient place to define logic that may be re-used by several mapping or routing scripts. Functions can be defined at the system level, if they are relevant to only one system, or at the global level, where they are accessible to scripts defined for any system.

Database Definitions

Message Manager can access database tables to read or write messages. *Database definitions* are used to associate message definitions with database tables or queries. A database definition consists of a series of *Message Table definitions*, which comprise the following settings:

Setting	Description
Selection	A table name, or SQL SELECT statement that selects records to be read by the message broker (or written to if the database is the target of a message)
Key Field List	The database field (or fields – expressed as a comma-separated list), which can be used to identify records uniquely.
Read Complete	By default (if this setting is left blank) the message broker will delete the records it reads from a table once it has processed them as messages (the assumption is that the table in question is defined for interfacing purposes, not an application data store). This behaviour can be overridden by defining an UPDATE statement in this setting, which should include a 'processed' flag field and value – for example: UPDATE MyTable SET Processed = 'Y' The message broker will use this statement (combined with a WHERE clause constructed from the Key Field List) to mark processed records as complete. The Selection setting should exclude processed records: SELECT * FROM MyTable WHERE Processed <> 'Y'

Interfaces

Overview

Interfaces represent the links between one system and another. An interface is defined from a source system to a target system. If message traffic between System A and System B is bidirectional two interfaces are required: A -> B (defined under System A) and B -> A (defined under System B). Interfaces are composed of several components:

Mapping Scripts

A mapping script defines the logic required to transform a message defined for the source system into a message defined for the target system. Mappings therefore are identified by their source and target messages: Message A -> Message B. Mapping logic is written in Java using the *Mapping Editor*. This component of the Repository Editor presents a view of the source and target messages and provides simple tools to help the user build the necessary mapping logic – in the case of a simple assignment, by dragging a field defined in the source message and dropping onto a field defined in the target message.

Mapping Selection Scripts

In some cases there may be more than one possible target message in an interface for a given source message. By default, the message broker will map the source message to whatever target messages it has a mapping for, but this may not always be the desired behaviour. In these cases *mapping selection scripts* may be used to examine the content of the source message and decide which (if any) of the available mappings to execute. Mapping selection scripts are Java methods that are passed an object representation of the source message, which they can interrogate, and which can call the mapping scripts defined for the source message in the interface. For example, one system might send a payment request that could be mapped to either a domestic payment or an international payment in the target system. In this case a mapping selection may be used to access the currency of the source message and if it is the domestic currency map the message to a domestic payment, otherwise to an international payment.

Lookup Definitions

A common requirement when transforming messages is to be able to look up data in a database table, to convert from the source system's customer code to that of the target system, for example. Lookup definitions allow the necessary database settings to be captured. Scripts can use lookups to access databases without needing JDBC or other database access logic. The runtime ensures that lookups are managed efficiently: connections are pooled, statements are closed and so on.

Connectors and Connector Types

Connectors define the message broker's connections to external data transport and persistence mechanisms: databases, message queues, email servers and so on.

There are various *Connector Types* supplied with the system. Connector types specify the information the user needs to provide to configure a connector. For example, a database connector type may specify that to configure a connector of that type the user must specify the hostname of the database server, the database name, and a valid user and password. Message Manager ships with a range of connector types for common resources including Oracle, DB/2 and SQL Server databases, WebSphere MQ, flat files, POP and SMTP mail. Advanced users can provide their own Connector Types for resources that are not supported as standard.

Formats and Format Types

Computer systems can exchange messages in a variety of technical formats. When Message Manager receives a message for processing it uses its message definitions and information about the data format expected to parse the message text to create a rich message object. It is message objects that Message Manager uses to represent messages internally in memory, and it is message objects that are passed to message broker scripts. When the message broker delivers a message typically the reverse needs to happen: the message object is converted (serialised) into a format suitable for exchange with another system.

Message Manager is supplied with components that can be used to parse and serialise a variety of formats. Each is defined in the repository as a *Format Type*. Format types define parameters that can be provided by the user to specify *Formats*. For example, a 'comma separated' format is based on the 'Delimited' format type, specifying a comma as the delimiter.

Format type	Comment	Parameters	Parameter details
SWIFT	Financial messaging standard	–	–
Delimited	Fields are separated by a delimiter character.	Field Separator	The character used to delimit fields
		Quote character	The character used to quote field values
		Message Terminator	The character used to terminate the message
Fixed	Fields appear at fixed positions in a record layout.	Pad Character	Character to pad data with to obtain desired length. Default is a space character.
XML	Industry data mark-up standard.	Use Fixed Tags	If true, all elements are given a hard-coded XML tag that identifies the message element type: Message, Field, etc. The element name appears as an XML attribute. If false, the XML element name is the same as the element name in the definition.

In addition, all formats include a 'Can Be Header' attribute. When this is set to *true*, the format becomes available for use in defining message *headers* – standard data sequences that appear before the payload of a message, which can be used by the message broker to identify the type of the message.

Advanced users may define their own message formats by providing sub-classes of Message Manager's Formatter and Parser classes. The classnames of the implementations are defined in a format type, along with any parameters that are required. User defined format types can then be used to create formats.

Some format types require meta-data for message elements. For example, in order to process a fixed format message, the message broker needs to know the maximum length of each field. The meta-data required for each of the standard formats is given in the table below:

Format type	Element	Meta-data	Comment
Fixed	Field	Length	The maximum length of the field (shorter values will be padded to the maximum length using the pad character defined for the format).
XML	Message	Namespace List	List of namespace definitions that can be used to qualify elements in this message
	Message, Repeat-block, Field	Active namespace	The name space to be used to qualify elements in the scope of this element, unless overridden.
SWIFT	Field	Field tag	For SWIFT Block 4 fields
		Fixed text	For ISO15022 block delimiter fields 16R and 16S

Queues and Routing

Overview

Queues represent symbolically the source, destination or state of messages for a given system. By default, a queue can send and receive any of the messages defined for the system to which it belongs. Queues can be connected together in a many-to-many scheme to create orchestrations. Message flow between the queues is determined by user-defined routing logic associated with the queue that is the source of the message.

If source and target queues belong to different systems the message broker will search for and execute a mapping that will convert the message sent from the source queue

into a message of a type defined in the target queue's system. If there are multiple potential target message types the user can allow them all to be created, or select a subset. The subset can be defined in a number of ways: 1. a fixed subset can be specified at the system interface level; 2 a mapping selection script can be specified at system interface level, which can interrogate the fields of the source message in order to decide which mappings to perform; 3 the routing logic can specify the mapping required.

There are two ways of specifying routing logic. In the simplest case the user can select, using a series of checkboxes, the downstream queues that a message should be routed to. Alternatively the user can specify a *routing script*. Routing scripts are Java methods that receive the source message object as an argument, and can interrogate it to decide how it should be routed. A routing decision is specified in the script by executing a *routeTo()* method, passing the name of the target queue as an argument. A second form of the *routeTo()* method allows the target queue and the target message format to be specified. Routing scripts can also be used to modify messages as they pass through the system. The repository editor provides a GUI script builder that simplifies the process of creating routing scripts.

Routing logic can be specified at two levels: message type specific and global. Message type specific routing logic is only executed when a message of the relevant type arrives. Global logic is executed for all messages. Typically one or the other is used at any given queue, but not both. If the routing logic is the same (or similar) for all message types, specifying all routing at the global level is easier, and easier to maintain. Conversely, if routing behaviour differs markedly from one message type to another, it is easier to specify the logic at the message type specific level, because global routing logic would need to include a lot of conditional logic to test for message type. It is possible to use a combination of the two, but this can lead to confusion if the responsibilities of each are not clearly defined. One scheme might be to use message type specific logic to enrich or modify messages, and global logic to perform the routing.

Adapters, Connectors and Connector Types

Queues are attached to the computer systems they represent by *adapters*. Adapters are components that connect queues to data persistence or transmission mechanisms such as databases, message queues, flat files or sockets.

Message Manager includes a special *Message Manager Data Access* (MMDA) adapter, which implements message state. Each message in the Message Manager database includes a queue field that represents its state in the orchestration (for example, Awaiting Verification, Transmitted, Logically Acknowledged). The MMDA adaptor updates the message in the database, and changes its state to that denoted by the name of the queue to which it is attached.

Adapters can be used to either read or write messages; most can perform either function depending on whether the queue to which they are attached is the source or the target of messages. The details required to define an adapter depend on the adapter type, but all adapter definitions need a reference to a *connector*. A connector is a collection of parameters that the message broker uses to establish a connection to a resource (a database, or a queue manager, for example) at runtime. The parameters required to define a connector are determined by the connector's *connector type*.

Types of Queue

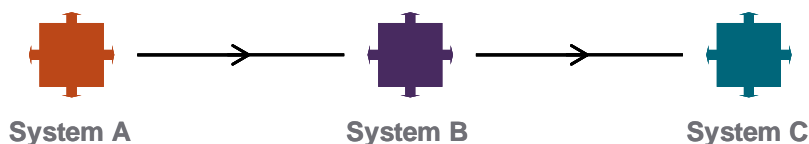
Generator

Generator queues are the sources of data in a workflow. At runtime the message broker polls the adapter attached to each generator queue. If a message is found it is parsed and routed to the queues downstream of the generator queue according to the routing logic defined by the user.

Pass-through

Pass-through queues are queues without adapters. They can be used as placeholders for routing logic to implement a decision point in a workflow. They can also be used to perform a 2-step mapping, where the intermediate result is not required. For example, if two mappings exists:

SystemA-MessageA->SystemB-MessageB, and SystemB-MessageB->SystemC-MessageC, a System B pass-through queue can be used to map from MessageA to MessageC without persisting MessageB:



Standard

Non-generator queues with adapters are normally the targets for messages sent by generator queues. They can also be the source of messages when the message broker is programmed using its API.

Message Manager orchestrations are modelled using queues attached to a Message Manager –specific adaptor that denotes the message's state, and records any changes in the content of the message as it passes from one queue to the next.

System

The system defines three queues that are mandatory in all repositories:

Queue	Purpose
Alert	To capture user-defined alert messages. Alerts can be sent to the Alert queue from user scripts.
Log	To capture user-defined log messages. Log messages can be sent to the log queue from user scripts
Error	To capture errors. If a message cannot be delivered to its intended destination because of an error, it will be delivered to the error queue, wrapped in a system error message.

Message Manager constantly monitors the content of the Error queue, and alerts users whenever a new error message appears.

Filters

Filters are classes that may be associated with a queue on either the input or the output side to perform processing on messages as they pass through an orchestration. Filters can be arranged in a list, to implement the *Chain of Responsibility* pattern; each filter can either handle the message itself, or if it cannot, pass it on to be handled by the next filter in the chain.

Each filter class must implement the Filter interface, which consists of one method:

```
public interface Filter
{
    public FilterResult execute(Message msg, FilterContext
context)

    throws TransactionalException;
```

...where *FilterResult* contains the modified message and one of the following constants:

Constant	Description
FILTER_NEXT	This filter has not completely handled the message, execute the next filter in the list (if there is one)
FILTER_STOP	This filter has handled the message; no further processing required
FILTER_DISCARD	This filter has decided that the message should be discarded

The *FilterContext* passed to the filter is an object that implements convenience methods to allow the filter to access other message broker objects. Toolkit users can

implement their own filters. Message Manager includes standard filters to perform application-specific processing:

Filter	Description
CharacterSet	Ensures SWIFT FIN messages contain only valid characters (the SWIFT character set is very restrictive). Invalid characters are converted to a valid alternative according to settings in a table of character conversions that is maintained by the system administrator.
Cancellation	Performs cancellation processing for messages resulting from an amended or cancelled transaction. Supports 3 cancellation models: <ul style="list-style-type: none"> • Codeword (for category 3 messages) • ISO 15022 (for category 5 ISO15022 messages) • MTn92 (for other message types)
Validation	Performs field- and message- level validation for SWIFT FIN messages.

Long-running Processes: Services

J2EE does not cater well for long-running processes (daemons, services). However, message broker *generator queues* are required to access sources of data autonomously, and hence require a long-running process capability. The message broker therefore implements its own scheme for prompting data collection, which has been generalised to allow other long-running processes, referred to in the repository as services, to be configured and deployed.

In the repository, a *service profile* defines the Java class that implements the service and any configuration parameters it requires. A *service* is an instance of the service profile for which values for the configuration parameters are supplied.

At runtime, service activity is prompted by the message broker's control module; the message broker sends a prompt request by calling the *prompt* method on the service, which must implement the *UserPromptable* interface:

```
/**
 * Prompts this Promptable.
 *
 * @param threadID a string containing a globally unique
 * ID for the calling client thread.
 * @return PROMPT (prompt again as soon as possible),
 * PROMPT_AFTER_INTERVAL (prompt again after a preconfigured
 * interval) or RECOVER (call recover()).
 */
public int prompt(String threadID)
    throws RemoteException;
```

...the service returns an enumerated integer which determines whether it should be prompted again immediately, after a timeout period, or requires recovery after an error.

Services are used to implement a number of Message Manager's standard message processing features. Toolkit users can implement their own services.

Packaging and Deployment

A complete message broker repository is transformed into an enterprise application by the *packaging* process. The packaging process performs a number of steps:

1. Compiles user-supplied Java 'scripts' and creates a JAR file that encapsulates the business logic described by the repository:
 - Java classes for business logic
 - EJBs for queues
 - Façade EJBs
 - Kernel message broker code
2. Creates all the application server configuration settings required by the message broker runtime:
 - JDBC data sources
 - JMS queue connection factories
 - JCA resource adaptors
 - JAAS security aliases
3. Creates scripts to set up the JMS queues required by the message broker's transaction co-ordination mechanism, and all necessary JNDI bindings
4. Packages all deployed items into a single EAR file, ready for deployment

The *deployment* process deploys the packaged application to the application server:

1. Uninstalls any previous version of the application
2. Creates uninstall information for the new version of the application
3. Creates all required JMS queues and JNDI bindings
4. Applies server configuration settings
5. Installs the new application on the application server

Packaging and Deployment may be initiated directly from the repository editor. When the repository editor is running on the same machine as the application server (typically during development and testing), Packaging and Deployment can be performed in one step. Otherwise packaging may be performed on the developer workstation, and the resulting package deployed (using a command line tool) on the server.

Message Manager Application

Overview

The Message Manager Application provides Message Manager's end-user functionality. It is a web application designed for use by back-office and wire-room staff. The functionality of the Message Manager application is fully described in the Functional Specification. The purpose of this section is to highlight some of the significant technical aspects of the design and explain how some of the functionality is realised.

Data design

Hosts and Host Groups

Messages may be keyed in by a Message Manager user, but are more likely to be generated automatically by a host back-office system, or from data extracted from a host back-office system. To allow tracking of messages, Message Manager marks all messages with the details of the host system to which they belong. Many work-groups within a financial institution may use multiple host systems to book and account for transactions. Message Manager's data design allows host systems to be aggregated as Host Groups.

Messages

All financial messages in Message Manager are maintained in the same table, and can be queried together, irrespective of their type. The strength of this approach is that a user query for messages relating to a particular transaction may return SWIFT FIN messages, ISO 20022-XML messages, or other formats in the same summary view. This is achieved by extracting message data and message management data common to most (if not all) messages into a series of columns in the messages table that can be queried, and storing the message object itself (serialised in XML) in a Character Large Object (CLOB) in the same table.

The common columns that can be used in user queries include:

Column
Message ID
External Message Type
Internal Message Type
Host Type
Host ID
Host Reference
Transaction Reference
Deal Reference
Direction
Action

Cover
Priority
Queue
Message Status
Cancel
Network Exchange Time
Value Date
Deal Maturity Date
Network Exchange Time
Network Device
Currency Code
Amount
Network
Sender Address
Destination Address
Business Entity
Possible Duplicate
Error Status
Error Details
Line Of Business
Payment Beneficiary
Confirmation Codeword

Market Centres

Message Manager may be deployed as a regional or global hub. Users in different regions performing time- or date- based queries need the ability to express those queries in terms that are adjusted for their locations. For example, a user in London requesting messages with value date equal to date of *next working day* would expect to see different messages from a user in New York issuing a similar request if the following day were a public holiday in New York but not in London. Similarly a user defining an alert to warn of JPY payments that might miss a clearing cut-off in Tokyo would want to specify the cut-off in Tokyo time, which may not be the time zone of the server.

Market Centres allow time zone, working day and holiday information to be captured, which is then used to adjust dates and times according to the location of the user. Typically a user is allocated a default Market Centre – assuming that the user works in or on behalf of one location – but this can be overridden (a London user can specify the JPY cut-off in Tokyo time, for example).

Users and Roles

Message Manager delegates user authentication to the application server. However once a user is authenticated, their data rights and capabilities are derived from

Message Manager's application security model. The following concepts are used to describe the security model:

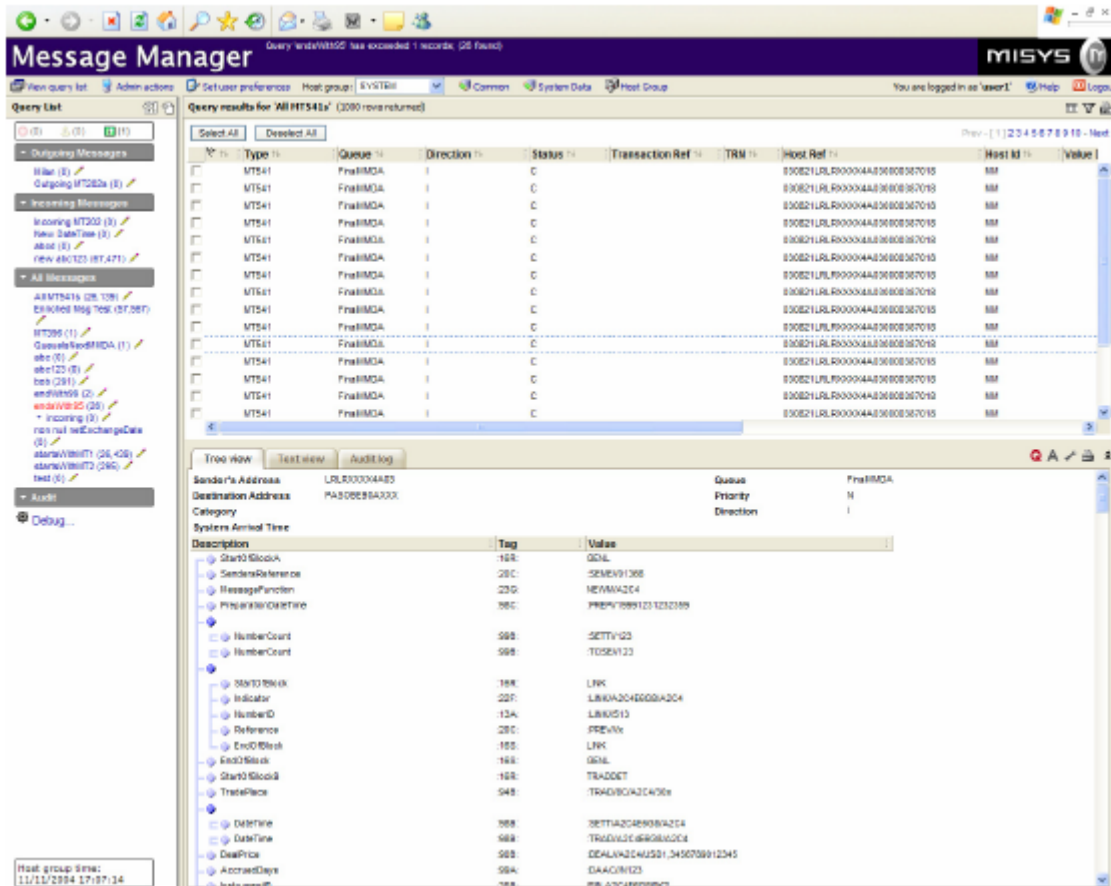
- *User* - an end-user of the site, either via a webapp, SOAP service, or other 3rd-party API. A user is identified by the username + customerID, and authenticated by their password.
- *Role* - a grouping mechanism for users. Security restrictions can apply to roles, as well as users, to make it easier to apply the same restrictions to sets of users. Typical roles may include: 'Administrator', 'Verifier', 'Editor' .
- *Activity* - A type of action (verb) that can take place on a resource. Typical activities may be 'create', 'delete', 'print', 'view'. Not all activities are relevant for all resource types.
- *Resource* - A type of thing (noun) that we wish to place security controls over.
e.g.:message
userConfig (for user-specific settings)
appConfig (for system-wide application settings)
- *Conditional Resource* - A resource that provides an expression (in an SQL-like syntax) that can be used to identify a subset of the item referred to typically messages). E.g.:
Messages where Host = 'Midas' AND BusinessEntity = 'Treasury'
- *Permission* - a combination of an activity and a resource; e.g. view message permission, update message permission.
- *Application* - a distinct set of roles, activities, and resources that are used to secure a system. Note that users are shared between applications.

Message Manager's security administration function is used to configure Users and Roles, to assign users to roles, to create the conditional expressions that define conditional resources (e.g. All messages for Host 'Midas' and Business Entity 'New York Branch', or 'All confirmations'), and to assign permissions to roles and users.

The use of Conditional Resources allows very fine-grained security decisions to be made about which users can perform which activities on which messages.

Message Screen

The Message Screen is the principal screen of Message Manager (note that the screenshot is from a pre-release of the product and may differ from the final version):



The column on the left-hand side shows query definitions that can be used to subset the messages into manageable views (e.g. *All Payments Value Date Today*). Users can define queries using the Query Builder described below. Administrators can also impose queries on users. The number in brackets to the right of the query description shows the number of messages the query would select. These numbers are reworked in the background and updated on the screen at a frequency defined for the system - typically every minute.

Clicking a query definition executes it. The results appear in the summary view in the top-right of the screen. The columns shown in this view, their order and widths can be customised by the user. Clicking a message in the summary causes the message detail to be displayed in the pane below. The message can be viewed in an expanded form, with its hierarchy of repeated sequences delineated and terse data such as currency codes and BICs expanded. Or it can be viewed in pure text format. The information shown in the message header in the expanded view can be customised by the user.

From this screen the user can also view audit details for the message, including side-by-side images of the message where the content has changed on its progress through the system.

Query Builder

The query builder allows users and administrators to build powerful queries without knowing SQL. Queries are expressed as a series of statements that are built up by the query builder, such as:

The type of the message starts with MT3

The value-date of the message is between <today> and <date of next working day>

The user can specify whether messages should be selected only if all statements apply or if any statements apply. The results are automatically filtered still further by the user's *view message permission* security setting.

Actions

Message Manager's message broker is responsible for moving messages between queues. Queues represent the state of messages – *Transmitted*, *LogicallyAcknowledged*, *AwaitingVerification*, etc. Messages on some queues are processed by automated processes on the server (such as the SWIFT Transmission process). Other queues hold messages awaiting manual processing (e.g. Verification, Repair). The user *actions* available at each queue are configured when the system is set up. When working with a message the valid actions (subject to security settings) are presented to the user. When the user selects an action the message is forward via the orchestration. The selected action is associated with the message and is available to the message broker's routing script, which decides which queue to send the message to next.

Errors and Alerts

Should an error occur, the Message Broker relays details of the error to the Error Queue. The Message Manager application monitors the error queue, and alerts logged-in users within 60 seconds of the error occurring. Suitably authorised users acknowledge error messages, providing details of the work-around, if any, used to circumvent the problem. The acknowledging user, date and time are logged for future reference.