Extending Message-Oriented Middleware[1]

Elsa Estévez
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Argentina
ece@cs.uns.edu.ar

Tomasz Janowski
International Institute for Software Technology
United Nations University
Macao
tj@iist.unu.edu

Abstract: Different types of middleware exist to facilitate the integration of software running on heterogeneous computing platforms. Message Oriented Middleware (MOM), for instance, enables the interaction between heterogeneous applications by exchanging packets of structured data (messages) through communication channels. The core responsibility of a MOM is asynchronous delivery of messages from senders to receivers, as well as management of the corresponding message queues. However, realistic software applications need many more messaging functions, for instance functions to enable auditing, encryption, tracking and transformation of messages. Such functions should be clearly provided by the underlying MOM and not implemented and re-implemented by applications themselves. In this paper, we present an approach for extending the core functionality of a MOM. In particular, we investigate how such extensions can be configured and combined, to ensure correct delivery of messages.

Keywords: software engineering, message-oriented middleware, software integration.

1. Introduction

In the past, private and public sector organizations were focused on the custom development of software applications and paid little attention to the integration issues. Software applications were integrated within each organization and the integration required using the same Information Technology (IT) platform to run all of them.

Recently, many organizations are faced with the software integration problem. Sometimes, the problem is a consequence of the globalization process. For instance, when different organizations are merged, each with its own IT infrastructure, there is a need to consolidate data produced and maintained by the applications that cross organizational boundaries, often developed in different programming languages and running on different platforms. The integration problem may also arise when the applications are

[1] This work was partially funded by Universidad Nacional del Sur, Bahía Blanca, Argentina, PICT 2003 Nº 15043, PAV 00076 (Agencia Nacional de Promoción Científica y Tecnológica) and Macau Foundation through the e-Macao Project.

designed to run as web services, with high-level services built by integrating multiple individual services. Not only such services are running in different environments, the integration must be done dynamically at runtime. The integration problem is particularly acute in the public sector, where government agencies are developing their own IT solutions based on individual needs. Now, trying to deliver better public services and improve efficiency through the use of Information and Communication Technologies (ICT), they have to integrate heterogeneous software applications to deliver public services that seamlessly cross agency boundaries.

Different programming abstractions, called middleware, were created to facilitate the integration of software applications running on heterogeneous computing platforms [1]. These include systems based on: Remote Procedure Calls (RPC), Transaction Processing (TP) monitors, object brokers, object monitors, Message-Oriented Middleware (MOM) and Message Brokers. Currently, MOM and Message Brokers are most popular solutions for software integration. MOM implements the administration of message queues for sending and receiving asynchronous messages. Built upon an old idea, MOM promotes a totally new approach for designing distributed systems, particularly well suited to tackle the integration problem. A message broker extends a MOM with functions for filtering and transformation of messages, and others. A MOM is implemented in several products: IBM WebSphere MQ [2], Microsoft Message Queuing Services (MSMQ) [3], WebMethods Enterprise by WebMethods [4], and also the messaging services of CORBA (Common Object Request Broker Architecture) [5]. Common functions include: distribution of real-time information, secure delivery of messages, validation and transformation of messages between different formats, and routing of messages based of the pre-defined business rules.

Despite the active development of MOM, several problems exist that limit their potential. One is the absence of specific functions required by business applications running on top of a MOM. While a MOM provides only the core functionality for queuing and delivering messages, additional functions, for instance for auditing and tracking of messages, would have to be build by each application individually, unless provided as MOM extensions. Another problem is the lack of formal models to support reasoning and verification of how different extensions can be combined while preserving the core behavior and its required properties. For instance, if the middleware is required to encrypt and validate messages, how to assure that the encryption and validation extensions can seamlessly work with each other, delivering messages that are both encrypted and valid. Finally, as most MOM solutions are proprietary products, it is crucial for their vendors to openly publish all interfaces and data formats used. The opposite would cause lack of interoperability between different MOM implementations as well as vendor dependence for maintenance support and future enhancements.

In this paper we present an overview of the core Extensible Government-to-Government Messaging Gateway (XG2G) [6] and provide an approach for extending the functionality of this framework. XG2G was originally introduced to coordinate the exchange of messages between different government agencies, but is really a generic framework to enable automated messaging among any number of public or private sector organizations. XG2G has a complete formal definition and an implementation which relies entirely on open source technologies. The contribution of this paper is the definition of a set of different messaging extensions of XG2G, in terms of the features and properties provided by the core framework. The definitions use a simple graphical notation introduced for this purpose. We also provide a concrete illustration describing how the auditing extension can be built on top of the functionality provided by a MOM.

The rest of this paper is organized as follows. Section 2 describes the core messaging framework. Section 3 presents a set of possible extensions to this framework. Section 4 describes the details of the auditing extension. Finally, Section 5 presents some conclusions, related work and outlines directions for future work.

2. Core Message Gateway

MOM systems rely upon the basic concepts of messages and channels. A message is an atomic packet of data that can be transmitted through a channel. A channel is a virtual pipe that connects a sender to a receiver [7]. Many design decisions must be taken related to messages and channels when designing a messaging system, such as: how to distribute messages, how to format them, how the channels are defined, and others.

Two particular models for communicating messages through channels are publish-and-subscribe and point-to-point. The publish-and-subscribe model describes a one-to-many broadcast of messages between a single producer and several consumers of messages. The point-to-point model describes one communication between a single producer and a single consumer. In the former approach, many consumers can receive the same message. In the latter, exactly one consumer receives the message.

The core XG2G framework [6] is based on the publish-and-subscribe model and is based on the three main concepts as follows:

- *Message*: A message is a package of information. It is prepared and delivered to a particular channel by one of its subscriber (producer) and is received by all other subscribers of this channel (consumers). Messages are sent as XML documents.

- *Member*: A member is a registered user of the system, typically a software application or a human user. Three distinguished member types are: guest, administrator and owner. A guest is the member who can only communicate with the administrator, usually to register a new member. An administrator is the member who can register and un-register other members. There is only one guest and one administrator in the whole system. An owner is associated with every channel, being the member who created this channel. An owner has the right to monitor the flow of messages passing through the channel and to carry out channel management functions, including the right to destroy the channel. Different members can ask the owner to subscribe and unsubscribe the channel. Once subscribed, a member can post messages to this channel as well as receive all messages posted to the channel by other subscribers. Each channel has exactly one owner and several subscribers. The owner of a channel is also its subscriber.

- *Channel:* A channel is an abstraction created for the purpose of exchanging messages between a set of members (subscribers). A subscriber of a channel is allowed to post messages to the channel, read messages posted to the channel or both. The gateway contains three kinds of pre-defined channels: administration channel, world channel and supervisory channels, one for every member-defined channel. The administration channel connects all members with the administrator. The

world channel connects all members with each other. A supervisory channel connects all subscribers of a given user-defined channel with the owner of this channel. Other channels are member-defined.

These concepts are depicted in Figure 1 as the UML conceptual class diagram.

Figure 1: Core Framework, Conceptual Model

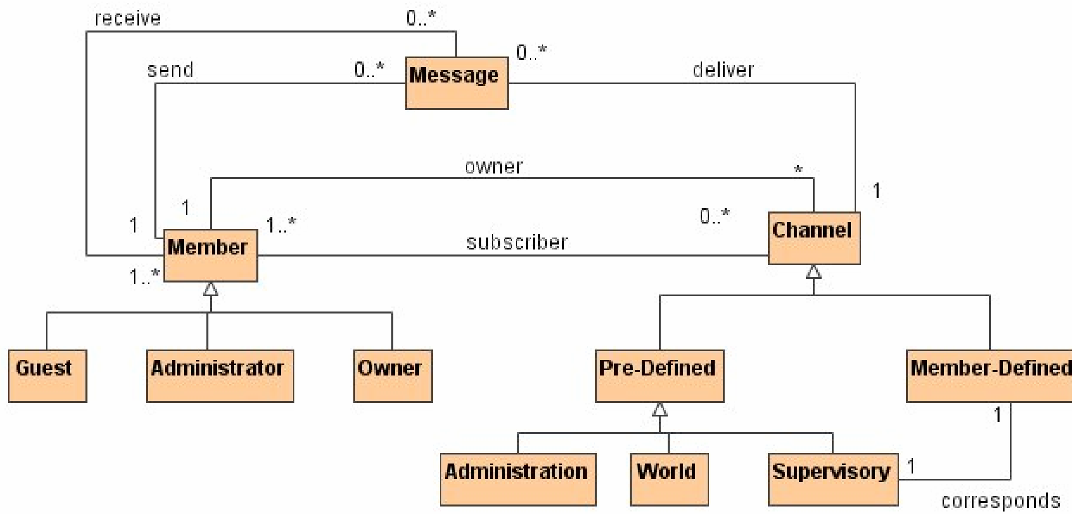
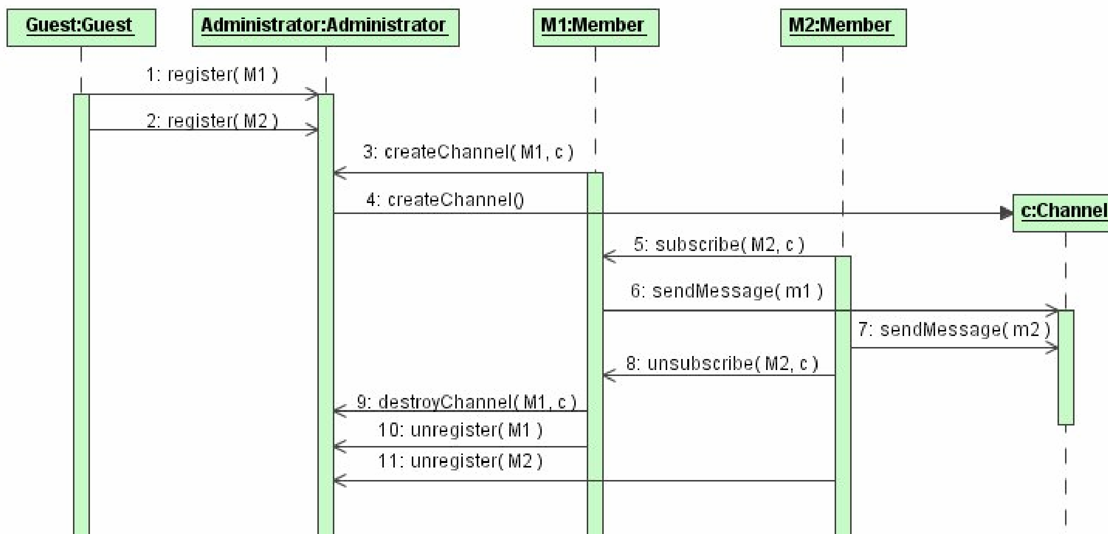


Figure 2 depicts the core functionality of the messaging framework using a UML sequence diagram. The sequence begins with *Guest* asking *Administrator* to register two new members: *M1* and *M2*. Once registered, *M1* asks *Administrator* to create a new channel *c* and *M2* requests *M1* to subscribe to this channel. Once subscribed, *M1* and *M2* post messages to *c* that the other member can read. Afterwards, *M2* requests *M1* to unsubscribe *c* and *M1* requests *Administrator* to destroy *c*. Finally, both members request *Administrator* to un-register.

Figure 2 : Core Framework, Behavioral Model



3. Message Gateway Extensions

The core messaging framework assures that the messages are delivered reliably between relevant members of the gateway. However, for realistic applications this basic property is necessary but likely insufficient. Other functions will be needed, allowing for instance to validate, transform and encrypt messages. The aim of this section is to briefly present the set of possible extensions to the core framework.

The extensions include:

1) *Auditing*: This extension enables the auditing of messages sent through a channel, so that the history of messages can be scanned later by the owner of this channel. Once the owner of a channel requests the administrator to audit the channel, all messages sent through the channel will be stored in a database. With this function enabled, the owner of the channel will be able to scan the history of messages sent as well as retrieve individual messages. A special audit member and audit channel will be created for that purpose, with the owner and audit member as the only subscribers. Possible criteria for selecting messages based on the history of the channel are: the time period during which a message was sent, a member who sent a message, the format of a message, and others.

2) *Validation*: This extension enables validating the format of messages sent through a channel. Initially, the owner of a channel asks the administrator to validate the messages posted to this channel, including the format definition document with the request. As messages are written with XML syntax, one of several languages for defining XML instance languages may be used, such as DTD, XML Schema, Relax NG or others. All messages that do not confirm to a given format would be returned to the sender with an error status or audited for later inspection.

3) *Transformation*: This extension enables the transformation of messages sent through a channel from one format to another. Initially, the owner of the channel requests the administrator to transform messages passed through this channel. The request specifies the transformation to be carried out. As all messages use XML syntax, the transformation is typically expressed as an XSLT template, but other DOM- or SAX-compliant transformations could be used as well. Different transformations may be applied depending on the member posting a message. While passing messages through the channel, the transformation is carried out before the message is delivered to the channel subscribers.

4) *Composition*: This extension enables the composition of existing channels into new, more complex channels. Some operations to carry out composition include:

    a. *Linking*: The exit-point of one channel (source) is joined with the entry-point of another channel (target). As a result, all messages received from the source channel are automatically forwarded to the target channel.
    b. *Splitting*: The exit-point of the source channel is joined with the entry-points of all target channels. This is similar to the linking operation but there are several target channels. As a

result, all messages received from the source channel are automatically forwarded to all target channels.

c. *Joining*: The exit-points of all source channels are joined with the entry-point of a target channel, symmetrically to the splitting operation. All messages passing through any of the source channels are forwarded to the target channel.
d. *Filtering*: This is a linked channel structure, with a filter inserted between a source and target channels to decide which messages received from the former channel will be forwarded to the latter channel. Other messages are discarded.
e. *Routing*: This is a split channel structure with a router inserted between the source and target channels. For each messages received from the source, the router selects the target channel to be used to forward this message on the basis of a state maintained by the router or perhaps the message itself.

5) *Tracking*: This extension enables to determine the position of a message in the system while in transit from the sender to receivers. Initially, the owner of a channel requests the administrator to add the tracking feature to the channel. Once enabled, any member posting a message to the channel will be able to track its own messages, while the owner will be able to track all messages posted to the channel. Tacking will take place by a member issuing requests to the special tracking member through the designated tracking channel.

6) *Encryption*: This extension is responsible for transforming messages in transit through a channel to a secret, not legible code. Encryption is requested by the channel owner, specifying the encryption algorithm to be used. Messages passing through an encrypted channel cannot be interpreted without knowing the decryption method. Messages are decrypted or transformed to a legible code before being delivered to the subscribers.

7) *Authentication*: This extension helps to assure the identity of the subscribers of a channel using digital signatures, assuring the integrity, confidentiality and non-repudiation of the messages passing through the channel. Once an owner asks the administrator to authenticate the channel, all subscribers will be required to provide their digital signatures issued by a Certification Authority. Thereafter, every message posted to a channel will contain the digital signature of the sender.

8) *Mailboxes*: By default, each member is provided with a single mailbox to store all incoming messages and a single mailbox to store all outgoing messages from all subscribed channels. In order to facilitate the processing of messages from heavy-traffic channels, a member can ask the owner of a channel to provide a dedicated pair of mailboxes for that particular channel. The request would be forwarded to the administrator. It successful, the member would be assigned a dedicated inbox to contain the messages received from the heavy-traffic channel and a dedicated outbox to store the messages delivered to this channel.

9) *Alliances*: This extension enables the creation and management of member alliances. A member alliance is a group of members that decide to act together to carry out messaging on particular channels. A member may carry out messaging individually and at the same time belong to one or more member alliances. In addition to its regular members, the alliance includes a member who acts as its supervisor. The supervisor ensures that the whole member alliance appears to other members in exactly the same way as a single regular member. The supervisor acts on behalf of the

whole alliance to add or remove regular members, to create or destroy channels, to subscribe or unsubscribe channels, and to send and receive messages. The alliance exists as long as its supervisor does.

10) *Directories*: This extension enables members to find existing channels and registered members though a directory service. The service includes white and yellow pages. White pages allow finding the unique identifier of a member or a channel through their names, provided during respectively member registration or channel creation. Yellow pages allow finding all members or channels that specialize in a particular topic, supported through the topics extension and specified during member registration of channel creation.

11) *Topics*: This extension maintains a hierarchy of topics that can be used to describe the interest of the members and the content of the channels. During registration, a member may choose any number of topics to describe itself. Likewise, during channel creation a member may choose any number of topics to describe the content of the channel. Each topic is selected from the tree-like hierarchy of topics related using ancestor and descendant relationships. Members can use the topics defined in the hierarchy to describe themselves and to search for other members through yellows pages. They can also modify the hierarchy, by adding and removing topics, and adding and removing relationships between topics.

12) *Persistence*: The core framework realizes the push model for message delivery - all messages posted to a channel are automatically pushed into the inboxes of all channel subscribers. This extension enables an alternative pop model - messages posted to a channel are stored in memory until explicitly requested by each subscriber. In this way, a member can retrieve both new and historical messages on demand. While push channels are created by default, this extension allows choosing the delivery mode (push or pop) during channel creation. Not only different channels may realize different delivery modes, the owner of a channel may request the administrator to change the mode at run-time.

All extensions should be implemented using the features and components implemented in the core framework. Moreover, it is crucial that all extensions conservatively enrich the whole messaging framework, so that the core behavior and properties are maintained regardless of the presence of the extra functionality.

4. The Auditing Extension

Following a brief overview of different messaging extensions, this section investigates the audit extension in some detail.

The required functionality includes:

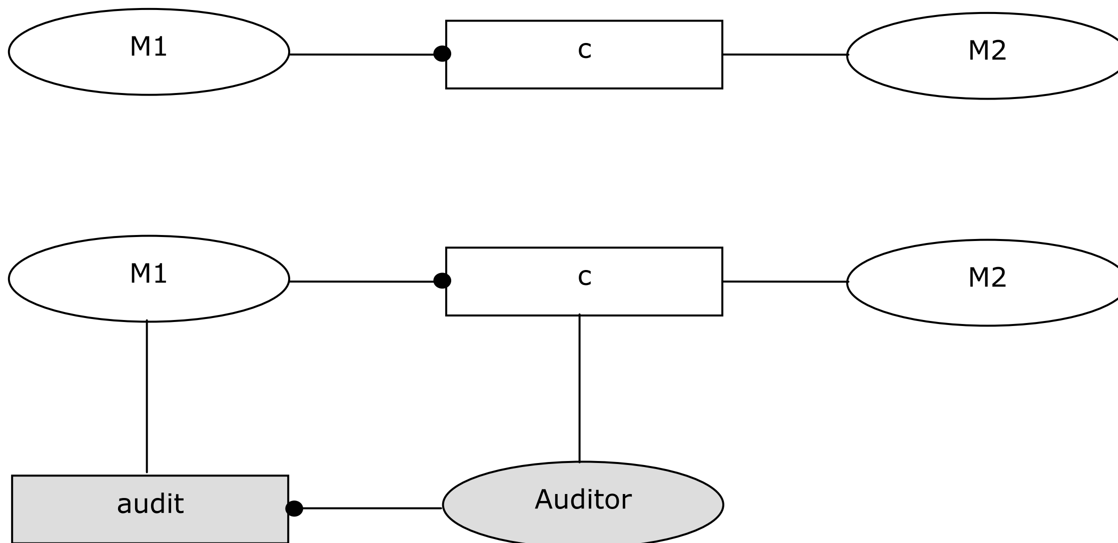
1) the owner of a channel can request the administrator to audit the channel,
2) when audited, all messages sent through the channel will be stored in a database,

3) the owner of an audited channel can retrieve historical messages according to some selection criteria such as the period during which a message was sent, the sender of a message, the format of a message, message content, or others.

Figure 3 depicts two configurations of a simple message gateway, one without and one with the audit extension. We use a simple graphical notation where members are represented as ellipses and channels as rectangles. Drawing a line between a member and a channel objects means that the member subscribes to the channel. If the member owns the channel, the line is drawn with a bullet on the side of the channel. For instance, in the upper part of Figure 4, a channel *c* is owned by the member *M1* and is subscribed by the member *M2*.

The lower part of Figure 3 depicts the auditing extension added to the upper part. A new *Auditor* member has been introduced to carry out the auditing function. *Auditor* subscribes to the original channel *c* and therefore receives copies of all messages posted to it by *M1* and *M2*; *Auditor* does not itself post any messages to *c*. The role of *Auditor* is to maintain a complete record of all messages received from *c*, including the sender of a message, the time of sending, message content and others.

Figure 3: Simple Versus Audited Channel



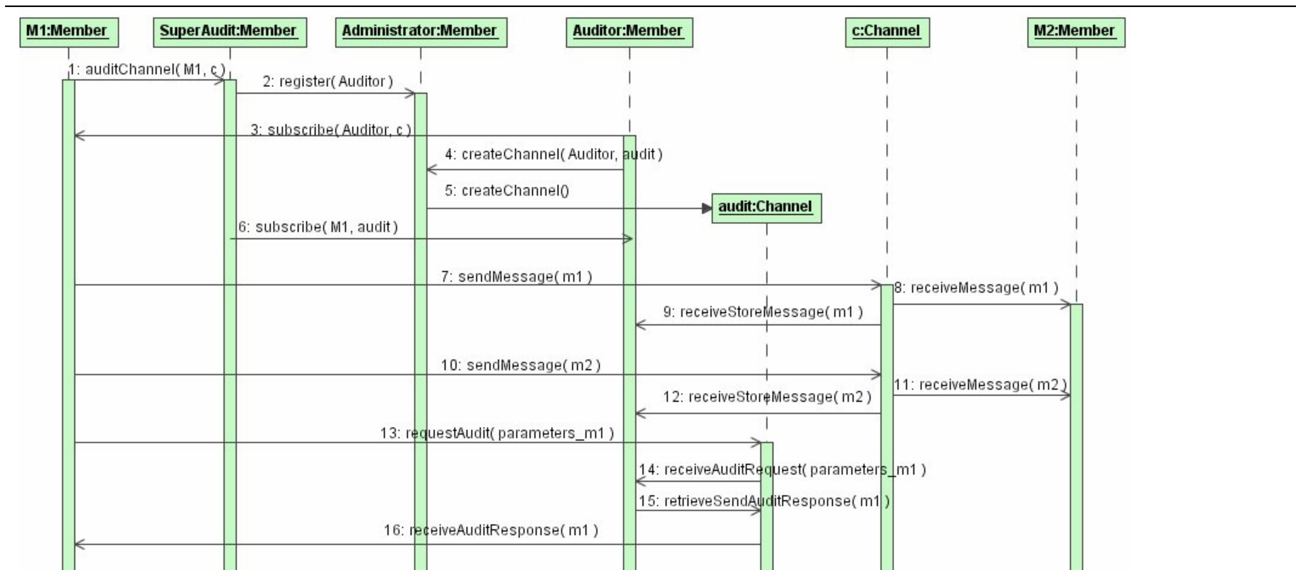
The auditing extension also includes the new *audit* channel, designed as a private channel between *M1* and *Auditor*; *M1* is the owner of the *c* channel and *Auditor* is the owner of the *audit* channel. Using the *audit* channel, *M1* can post requests to *Auditor* to retrieve the record of messages received from *c* according to certain selection criteria, and *Auditor* sends the record of all relevant messages back to *M1*.

In the simplest case, the auditing extension includes one *Auditor* member and one *audit* channel for every audited channel. The extension also allows a single *Auditor* member to audit a collection of channels. With many configurations possible, auditing individual channels can also start, end, suspend and resume dynamically at run-time, in response to requests from channel owners. A single member responsible for managing all audit requests is the audit supervisor - *SuperAudit*. When processing such requests, *SuperAudit* communicates directly with the system administrator to register and un-register *Audit* members, and to create and destroy *audit* channels.

The behavior of the auditing extension is depicted in the sequence diagram shown in Figure 5. A typically action sequence includes the following steps:

1) *M1* requests the audit supervisor, *SuperAudit* to audit the channel *c,*
2) *SuperAudit* requests the *Administrator* to register the new *Auditor* member,
3) *Auditor* requests *M1* to subscribe to the channel *c,*
4) *Auditor* requests *Administrator* to create the *audit* channel,
5) *Administrator* creates the *audit* channel,
6) *SuperAudit* asks *Auditor* to subscribe *M1* to the *audit* channel,
7) *M1* sends a message *m1* to *c*,
8) M2 receives the message *m1*,
9) *Auditor* receives and stores the message *m1*,
10) *M1* sends a message *m2* to *c*,
11) *M2* receives the message *m2,*
12) *Auditor* receives and stores the message *m2,*
13) *M1* sends a request to *audit* to retrieve the first message sent by itself,
14) *Auditor* receives the request from *M1,*
15) *Auditor* retrieves the message *m1*, and sends it to the *audit* channel*,*
16) *M1* receives the message *m1*.

Figure 4 : Channel Auditing, Behavioral Model



5. Conclusions

The aim of this paper is to present an approach for extending the functionality provided by message-oriented middleware. Based on the core functionality for delivering messages between members through dynamic, member-defined channels, we propose a set of well-defined extensions to carry out a variety of useful messaging services. We outlined 12 concrete services: message auditing, validation, transformation, encryption and tracking, channel composition, member authentication, member alliances, channel-dedicated mailboxes, directory services, hierarchies of topics, and persistent channels. All extensional can be implemented on top of the core middleware. As an illustration, we described the auditing extension in some detail.

Several message-oriented middleware solutions exist, most as commercial products. They assure reliable delivery of messages and provide a range of complementary functions such as: user account management, transaction support, mail query monitoring, persistency and encryption of messages [9][4]. In addition, many support the standard Java-based API – Java Message Service (JMS) [10], and usually realize both publish-and-subscribe and point-to-point delivery models. In our approach, these complementary functions are explicitly regarded as functional extensions of the core messaging framework, all expressed and realized in terms of the core messaging concepts. In addition, the core framework has received a fully formal definition, and has been developed using open-source technologies [6]. The contribution of this work is to extend this core messaging framework with a range of useful messaging services. We aim to develop formal specifications for each of the proposed extensions, analyze the mechanisms for their composition, and provide corresponding implementations.

The first step in our future work is to study in detail and model each of the extensions proposed in this paper, providing formal specifications for each of them. The second step is to develop a mechanism for seamless composition of extensions on top of the core framework, also providing the formal specification. Based on the formal models of individual extensions and their composition, the third step is to formally analyze the preservation of properties and behaviors realized by the core messaging framework when adding new extensions. This step should ideally result in a well-founded theory to underpin the development of provably correct messaging services. The fourth step is the application of the resulting message gateway in different domains. Our particular interest is electronic government and how to develop distributed software applications to facilitate communication between public and private sector organizations in order to produce seamless services to citizens and businesses.

Acknowledgements

We wish to thank Adegboyega Ojo, Gabriel Oteniya and Benedikt Mas y Parareda for collaboration, comments and useful discussion about this work.

References

[1] Web Services, Concepts, Architectures and Applications. Gustavo Alonso, Favio Casati, Harumi Kuno, Vijay Machiraju. Springer, 2004.

[2] IBM WebSphere MQ, http://www-306.ibm.com/software/integration/wmq/v60/
[3] Microsoft Message Queuing Services, MSMQ, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnfoxgen/html/ msmqwvfp6.asp
[4] WebMethods Enterprise, WebMethods, http://www.webmethods.com/meta/default/folder/0000005877
[5] Messaging service Common Object Request Broker Architecture (CORBA) http://www.omg.org/technology/corba/corba3releaseinfo.htm
[6] xG2G, Extensible G2G Message Gateway, Tomasz Janowski and Benedikt Mas y Parareda, e-Macao Project, UNU-IIST, Macao SAR, China.
[7] Enterprise Integration Patterns, Designing Building and Deploying Messaging Solutions, Gregor Hohpe and Bobby Woolf, Addison Wesley 2004.
[8] Enterprise Service Bus, David A. Chappell, O'Reilly Media Inc, 2004.
[9] IBM WebSphere MQ, http://www-306.ibm.com/software/integration/wmq/features/
[10] Java Message Service Tutorial, http://java.sun.com/products/jms/tutorial/