Extending Message-Oriented Middleware using Interception

Edward Curry, Desmond Chambers, and Gerard Lyons
Department of Information Technology,
National University of Ireland, Galway, Ireland.
{edward.curry, des.chambers, gerard.lyons}@nuigalway.ie

Abstract

Varieties of Message-Oriented Middleware (MOM) platforms are available each with their own propriety functionality to solve specific messaging challenges. At present, it is not possible to mix and match these propriety features into a customized MOM solution.

A number of patterns have been identified that allow a software systems implementation to be more flexible and extensible. This work investigates the use of one such pattern, the POSA Interceptor pattern, in the construction of a MOM framework that is easily customised and extended in a structured way.

This framework, Chameleon, is designed to support the use of interceptors (message handlers) with a MOM platform to facilitate dynamic changes to the behaviour of the deployed platform. The framework also allows for interceptors to be used on both the client-side and server-side, permitting advance functionality to be deployed to the client, and for co-operation between client-side and server-side interceptors.

1. Introduction

Message-Oriented Middleware (MOM) platforms are available in wide range of implementations such as WebSphere MQ (formerly MQSeries) [1], TIBCO [2], Herald [3], Hermes [4], SIENA [5], Gryphon [6], JEDI [7] and REBECCA [8] each of these providers have been designed with specific goals and employs unique functionality to achieve these. A number of these providers are designed as centralized servers or server clusters, others as server/broker networks, some as federated services. Providers have also been designed for a specific task such as enterprise integration, mobile clients, internet-level scalability, wide-area networks, sensor networks, and ubiquitous environments. In order to achieve these design goals, each provider has its own unique services for their target audience such as message translations, filtering algorithms, mobile profiles, broker routing algorithms, distributed state persistence technologies, and so on.

At present, the vast majority of these features are not compatible with one another, nor are they transferable between MOM platforms. The goal of this paper is to illustrate a potential approach for packaging these services into reusable chunks that may be mixed and matched to create a customised messaging solution.

The remainder of this paper briefly introduces the Interceptor design pattern and its current usage, gives an overview of our work on providing message handler chains (interceptors within the messaging domain) within the Chameleon framework. The frameworks architectural design and implementation is explained along with possible uses of the framework and future development plans.

2. Interception design pattern

The ever-increasing demands placed on software systems require them to often perform beyond the scope of their original requirements. Such behaviour may not always be anticipated during their initial development phase, thus making it important to design systems that can be easily extended during their life cycle.

Systems designed for large target audiences with diverse interests will often include functionality that is only utilized by a small percentage of users. While such functionality may be vital to some users, incorporating it into the core system makes it unnecessarily bloated and increases overhead for the majority of remaining users.

The POSA Interceptor design pattern [9] is a variant of the Chain of Responsibility pattern from the Gang of Four (GoF) [10] . This pattern enhances a system by increasing flexibility and extensibility. The pattern also enables functionality to be easily added to the system in order to dynamically change its behaviour. This seamless integration of functionality can be performed without the need to stop and recompile the system, allowing its introduction at runtime.

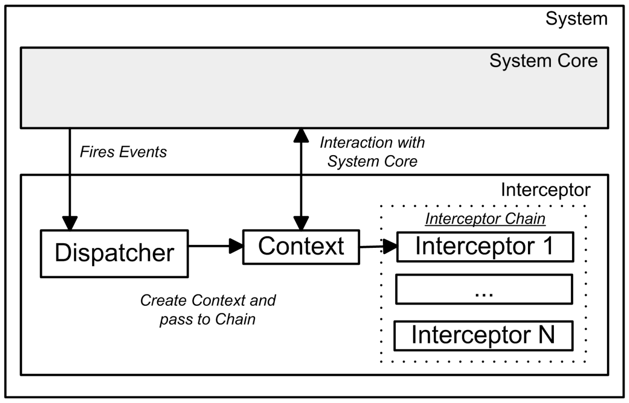

Figure 1. POSA Interceptor Pattern

The basic interceptor pattern has four main elements:

- System Core
- Dispatcher
- Context
- Interceptors

The Interceptor pattern, illustrated in Figure 1, follows a straightforward sequence of events.

Interceptors are registered with the system dispatcher; the system core may perform registration, the interceptors may self-register, or they may be registered from an external source (parent server/master server). Once the interceptors are registered, the system core notifies the dispatcher of any events that have occurred. Upon receiving an event, the dispatcher examines the event to determine which interceptors need to be notified. The dispatcher then packages the event and any relevant information into a context, the context may be provided by the system core. The dispatcher then notifies the relevant interceptors or interceptor chains (a ordered/unordered collection of interceptors) by passing them the context containing the event. When triggered, the interceptor examines the context and executes its related functionality.

An optional addition to the pattern is to allow interceptors access to the internals of the core system state and to provide a mechanism to control the system by altering its state.

2.1. Uses of the interception pattern

Interceptors are utilised in a broad range of domains to increase flexibility and extensibility; such systems include CORBA ORBs (TAO, Orbix) for infrastructure and support services, web browsers (Microsoft Internet Explorer) for plug-in integration, and web servers (Apache 2.0) to allow modules to register handlers (interceptors) with the core server. The JBoss J2EE [11] application server also uses the interceptor design pattern to provide customized functionality in the areas such as transactions, security, remoting and life cycle support.

Currently, no message service has exploited interception as a mechanism for extending its core-messaging functionality.

2.2. Evaluation

The Interceptor pattern has a number of advantages and disadvantages; benefits of the pattern include the decoupling of communications between a sender and receiver of an interceptor request, this permits any interceptor to fulfil the request and allows interceptors to change system functionality, even at run-time.

The pattern also has a number of drawbacks that if left unresolved may lead to a number of issues in the system design. One of the main drawbacks is increased complexity in design, the more interceptors can hook into the system the more bloated its interface. The inherent openness of the pattern also introduces potential vulnerabilities into systems. With such an open design, malicious interceptors or simply erroneous ones may be introduced resulting in system corruption or errors.

Another important issue to consider is the possibility of incompatibilities between interceptors and potential infinite interceptor loops whereby an event produced by an interceptor triggers another interceptor that in turn generates an event that triggers the original interceptor. Such errors will only occur at runtime and may be difficult to locate.

When used within the messaging domain the abstract generic interceptor pattern is implemented using a customised context or *'Message Context'* and '*Message Handlers'* as interceptors. The remainder of this paper uses such terminology.

3. Chameleon

The goal of this research is to implement the Chameleon message framework to allow message handler chains (Interceptors) to augment functionality onto a base MOM platform. In order to achieve this objective the Java Message Service (JMS) Application Protocol Interface (API) [12] is used as an interface to the underlying core messaging platform, the Chameleon framework sits on top of this Java Message Service System Core (JMSSC) platform. MOM services/features can be packaged as handlers and deployed to add functionality on top of the base service, enhancing its functionality.

The remainder of this section examines the architecture of Chameleon and discusses its ability to allow handlers to be deployed on both the client-side and server-side of a MOM platform.

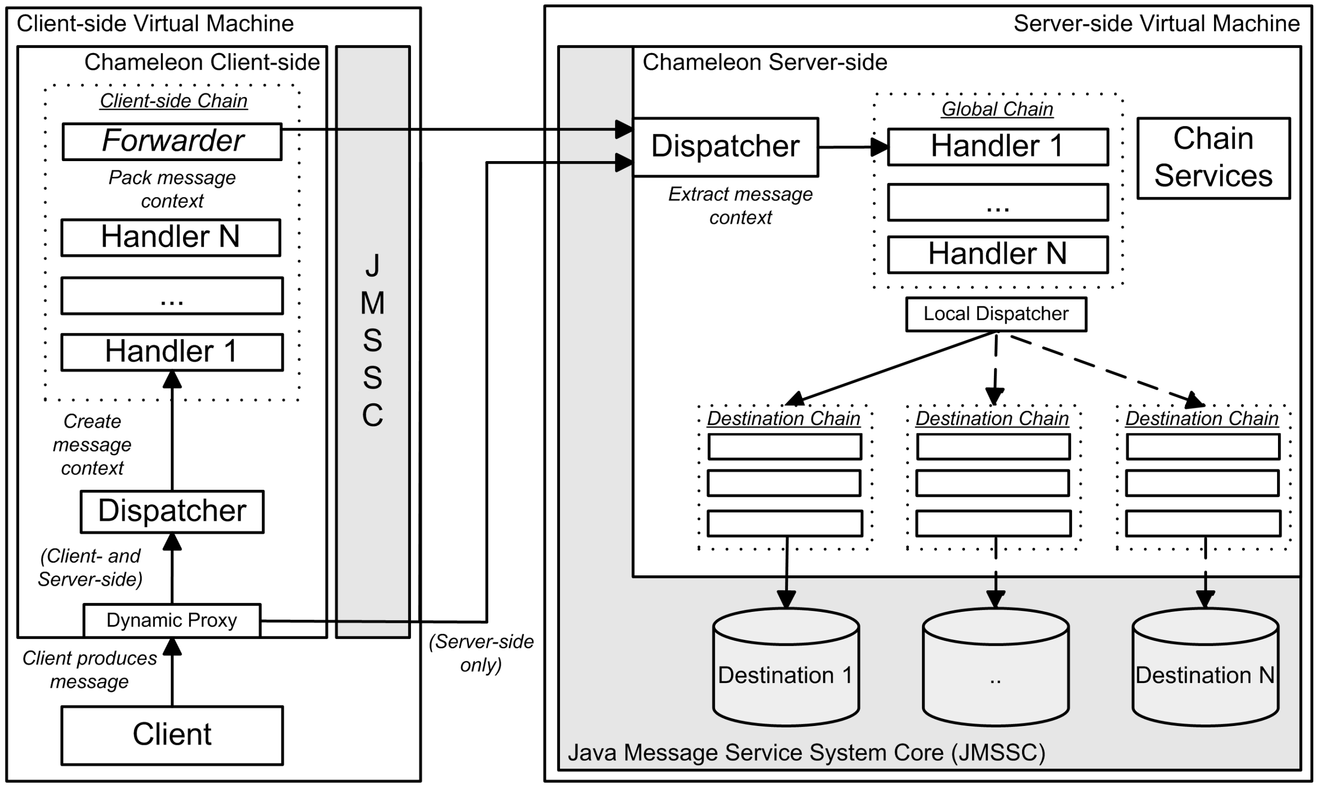

Figure 2. Chameleon Framework Overview

3.1. Server-side

The core of the Chameleon framework exists in its server-side deployment. When the framework initialises, it first registers any handler chains present in its start-up configuration. Server-side chains are constructed from local configuration files or under the guidance of a master or peer server within a broker network. This configuration may also be updated at runtime to adapt the chains for operating conditions.

When a message is passed from the JMSSC, chameleon first checks for the existence of a client-side message context, if one exists it uses it as the basis to create a server-side context. Alternatively, it creates a new context for the inbound message. Once the context is ready, it is passed to any relevant global server-side chains. After the global dispatcher has completed evaluating the message it is passed on to the local-chain dispatcher. This local-dispatcher is responsible for triggering any local-chains associated with specific destinations; chain scopes are discussed in more detail in section 3.5.

Server-side handlers have the ability to interact with a client-side handler (if one exists); this topic is covered in more details in the section 3.3 of this paper.

3.2. Message context

The message context is a handler's main point of interaction with the framework and JMSSC. It is central to the invocation of handlers and is used to communicate the message/event that has triggered it; the handler then processes the message. Message contexts can be used as a medium to store data, communicate with other handlers, or interact with the JMSSC and framework. Handlers are able to store and retrieve information within the context using the setProperty() and getProperty() methods. These methods can save any serializable object and the basic Java primitive types within the context.

Contexts can also be used to communicate information between client-side and server-side handlers, this allows for the behaviour on the server-side or client-side to be dynamically altered based on events and information from either side.

The context is also a mechanism to enable a handler to alter the behaviour of the JMSSC. The context achieves this by exposing an API to control core system operations. Handlers can be granted permissions to access parts of this API. Such an approach provides a safe method to manipulate the system core in a secure and controlled manner.

3.3. Client-side

Client-side handlers offer the ability to extended and enhance the functionality of the JMSSC on its client-side. Client-side chains allow the message service to dynamically alter the behaviour of its client at run-time.

Client-side handlers can be constructed in two manners, the first approach is to build the chain from local handlers which reside on the clients machine, this approach requires the machine to have the relevant handlers installed or to use a distributed classloader. The second approach involves the construction of chains on the server-side and transmission to the client; this removes the need to have application-specific stubs that would need to be pre-installed on the client machines. Once the chain is constructed it now needs to be configured, similar to server-side chains two approaches can be used, configuration can be obtained from a local file or from the server or peer node that the client is connect too. Client-side chain configuration may be updated at runtime to adapt the chains for operating conditions.

Client-side handlers introduce a number of advantages to MOM/DEBS by permitting computational tasks and behaviour to be easily distributed to client machines. With this support framework in place, advanced features may be developed with co-operation and co-ordination between both the client and server-side of the platform. Such capabilities could be used to increase the scalability of centralized servers by distributing tasks to the clients, such as message transformation, filtering, etc.

The dynamic retrieval and configuration of client-side handlers has a number benefits; services can now be deployed to the client without any special arrangements on the client-side. This streamlined distribution of services reduces the amount of administration needed to alter a deployed system, making frequent changes to its behaviour and configuration more feasible. A service can adapt itself into a more optimal state based on its current operating conditions. Clients may now connect to multiple servers and retrieve their specific client-side chain without the need for extra configuration or intervention by a system administration.

The prospect of deploying functionality to the client side is an interesting proposition, however due diligence must be taken when considering the use of this “mobile code”. If a client-side chain can be configured from a remote location or is downloaded, it presents a number of security issues and potential vulnerabilities to the client system; such issues are covered in more details in [13].

Once client-side chain initialization has complete, the outbound/inbound message is placed into a message context and passed to the client-side dispatcher. As soon as the message has passed through the chain, the message context is packaged into the JMS messages and sent to the server-side. Upon receipt of a new message, the server checks for the existence of a client-side context and uses this in the construction of the server-side context. This process allows client-side handlers to communicate with the server-side, allowing data exchange between them, such as the results of a distributed task or the results of any computations or pre-processing carried out on the message by the client-side.

The inverse of this process is also possible, whereby server-side handlers wish to pass message specific information details to the client-side; achieved by updating the message context of an outbound message.

3.4. Chain services

Message handlers within the Chameleon framework can provide a wide variety of functionality to the underlying JMSSC. To successfully carry out many of these tasks, handlers will require access to a number of infrastructure services. The Chameleon framework offers a numbers of support services for handlers; such services include logging facilities, persistence frameworks (hibernate), usage statistics, and core system states. Other potential services also include transactions, security, hardware usage statistics (CPU, memory, hard disk) and fault-tolerant system/event recovery logs.

Interactions with the core message service can use two methods; exploitation of the message context to control the core service has already been covered. The second approach involves using a chain service to access the system core. Such a service can expose models of the core message services internal structures and state. Chameleon provides one such service that exposes the current destinations that exist within the message service; the model contains destination configuration, usage statistics, and subscriber details. This service also provides destination administration capabilities (add, remove, move, etc). Such services are designed to allow handlers access to the internal state of the system in order to increase the functionality that they provide and enhance their ability to examine and adapt the core system at runtime.

3.5. Chain scope

Handler chains provide a powerful method of augmenting a message service with dynamic functionality; this ability can be extend by using multiple chains and attaching them at multiple interception points within the service core. An interception point is a place of execution within a system in which handler chains can be triggered to inject functionality. In order to achieve this, the systems behaviour must be modelled to allow chains to be attached to specific points within the model. A good quality system model will help to identify appropriate locations for interception points, this can also help to determine what handlers should be grouped together. Within the Chameleon framework, three chain interception points or interception scopes are available:

- Global (system-wide – all destinations)
- Local (per destination)
- Hierarchy (per branch)

Global-scoped chains are designed to operate on all incoming messages into the service and are triggered before any other scoped chains; global chains are designed for implementing system-wide services such as auditing, logging, or usage monitoring.

Local-scoped interception points work on a per-destination basis, allowing for chains to be attached to a single or group of destinations, once a message arrives for a given destination the chains attached to that destination (if any) will be triggered before the message is delivered to the destination. These interception points allow functionality to be added/extended at the destination level; this scope of chain allows highly specific behaviour to be attached to a single destination.

Hierarchy interception points allow chains to be associated with a channel within a hierarchy. Similar to local-destination interception points, hierarchy interception points work on the principle that each leaf (channel) of the hierarchy can have a local chain associated with it. The absolute chain for a branch consists of this local chain and the absolute chain of its parent, this recursive approach to interception continues up the tree until it reaches the root channel. This results in a very powerful mechanism for processing messages submitted to a channel hierarchy structure. This mechanism is similar to inheritance within object-oriented programming, where an object (channel) inherits the functionality of its ancestors (parents chains) and can augment this functionality with its local implementation (local chain).

4. Framework benefits

When compared to alternative techniques for system extension, such as method-call interception, reflection, or aspect-oriented programming, Chameleon provides a non-invasive message-centric method for augmenting any JMS compatible message service without needing access to its source code for mass-refactoring and recompilation.

Besides offering a flexible method of integrating MOM technologies, the Chameleon framework has a number of benefits for the development and deployments of MOM services. It provides a unique ability to easily deploy functionality to the client-side of a MOM without the need for any application-specific stubs to be present. MOM systems can be easily extended and their behaviour can be altered at run-time, this allows for a number of unique MOM features to be developed.

Chameleon can facilitate the development of reflective and adaptive MOM services, reflection has been advocated for advanced adaptive behaviour, the reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports development of flexible and adaptive systems and applications [14]. Reflective flexibility diminishes the importance of many initial design decisions by offering late and runtime-binding options to accommodate actual operating environments at the time of deployment, instead of anticipating the operating environments at design time [15].

Chameleon also facilitates the development of services that can co-ordinate their behaviour with the client-side of the message producer or consumer. This allows the development of a service that can easily distribute tasks to client machines; such capabilities open the possibility of developing new dynamic services for MOM. Such services could request clients to transforms their message payloads into the desired format for a destination, this would substantially reduce the workload of message brokers, for example each destination within the service may provide a XML schema and relevant XSLT stylesheets to transform incoming XML messages with on the client-side.

Chameleon also allows behaviour to be easily packaged as message handlers and dynamically added to the core service. Once packaged, behaviours can be mixed and matched to create customised messaging solutions. This process reduces the effort required to dynamically add and remove behaviour such as auditing, accounting, security, transactions, filters, message transformations (XSLT), and load balancing from a MOM service.

5. Future plans

The next stage of our research is to complete the development of the framework and to develop a number of services that exploit the virtues of the framework. The first of these services involves the utilisation of adaptive and reflective techniques with channel hierarchies [16].

Channel Hierarchies are structures that allow channels to be defined in a hierarchical fashion, so that they may be nested under other channels. Each sub-channel offers a more granular selection of the messages contained in its parent. Clients of the hierarchy *subscribe* to the most appropriate level of channel in order to receive the most relevant messages. In large-scale systems, the grouping of messages into related types (i.e. into channels) helps to manage large volumes of different messages [17].

Current MOM platforms do not define the structure of channel hierarchies. Application developers must therefore manually define the structure of the hierarchy at design-time. This process can be tedious and error-prone. To solve this problem the Chameleon messaging architecture implements reflective channel hierarchies [18] with the ability to autonomously self-adapt to their deployment environment. The Chameleon architecture exposes a causally-connected meta-model to express the configuration and structure of the channel hierarchy, this enables the run-time inspection and adoption of the hierarchy.

In order for handlers to be interoperable between services, a number of standards will need to be defined to regulate interaction with the core message service. Models are needed to represent the internal state of the service, its destinations, message producers and consumers, subscriptions, filters, usages statistics, and so on. If MOM behaviour is to be truly portable between implementations, such standards will need to be defined in co-operation with the entire MOM community.

Plans are also in place to port propriety functionality from a third-party message service to act as a proof of concept for the approach. We also plan to investigate further services, tools, and techniques to offer streamlined integration of disparate MOM services.

6. Conclusions

This work forms a necessary step in integrating Message-Oriented Middleware (MOM) technology from disperse implementations. While not claiming to be a sliver bullet, it provides a useful non-invasive mechanism to add/extend the functionality of an underlying MOM implementation. The process of augmenting functionality upon this core message service is achieved though the use of the POSA Interceptor pattern.

The Chameleon framework is designed to support the use of message handlers (interceptors) with a JMS compatible MOM platform. Chameleon allows MOM behaviours to be easily packaged as message handlers and dynamically added to the core service. Once packaged, behaviours can be mixed and matched to create customised messaging solutions.

The framework permits handlers to be used on both the client and server-sides, allowing for advanced functionality to be deployed to client systems, and for co-operation between client-side and server-side handlers. Handlers can also be added and removed at run-time; this enables the application of reflective and adaptive techniques within a MOM service.

7. Acknowledgement

The support of the Informatics Research Initiative of Enterprise Ireland is gratefully acknowledged.

8. References

[1] L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*. New York: John Wiley, 1996.
[2] D. Skeen, "An Information Bus Architecture for Large-Scale, Decision-Support Environments," presented at USENIX Winter Conference, 1992.
[3] L. F. Cabrera, M. B. Jones, and M. Theimer, "Herald: Achieving a Global Event Notification Service," presented at 8th Workshop on Hot Topics in OS, 2001.
[4] P. R. Pietzuch, "Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems?," 2002.
[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service," presented at Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR, 2000.
[6] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, "Gryphon: An Information Flow Based Approach to Message Brokering," presented at International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998.
[7] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, pp. 827--850, 2001.
[8] L. Fiege and G. Mühl, "Rebeca", http://gkpc14.rbg.informatik.tu-darmstadt.de/rebeca/
[9] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, vol. 2: Wiley & Sons, 2000.
[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.
[11] M. Fleury and F. Reverbel, "The JBoss Extensible Server," presented at Middleware 2003, Rio de Janeiro, Brazil, 2003.
[12] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java Message Service Specification v1.1," Sun Microsystems, Inc., 2002.
[13] D. M. Chess, "Security issues in mobile code systems," in *Mobile Agents and Security*, vol. 1419, *Lecture Notes in Computer Science*: Springer-Verlag, 1998.
[14] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, 2002.
[15] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering*: Wiley & Sons, 2001.
[16] E. Curry, D. Chambers, and G. Lyons, "Could Message Hierarchies Contemplate?," presented at 17th European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, 2003.
[17] P. R. Pietzuch and J. M. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," 2002.
[18] E. Curry, D. Chambers, and G. Lyons, "Reflective Channel Hierarchies," presented at 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003, Rio de Janeiro, Brazil, 2003.