Middleware Platform Management Based on Portable Interceptors

Olaf Kath[1], Aart van Halteren[2],
Frank Stoinski[1], Maarten Wegdam[3], and Mike Fisher[4]

[1] Humboldt-Universität zu Berlin
{kath,stoinski}@informatik.hu-berlin.de

[2] KPN Research
A.T.vanHalteren@kpn.com

[3] Lucent Technologies - Bell Labs Twente
wegdam@lucent.com

[4] BT Advanced Communications Technology Centre
mike.fisher@bt.com

Abstract. Object middleware is an enabling technology for distributed applications that are required to operate in heterogeneous computing and communication environments. Although hiding distribution aspects to application designers proves beneficial, in an operational environment system managers may need detailed information on information flows and the locality of objects in order to track problems or tune the system. Therefore, hooks are required inside the processing core of the middleware to obtain inside-information and to influence the processing of information flows. We present the use of portable interceptors for the management of CORBA as well as COM/DCOM middleware. Management information is structured in a middleware technology independent way, using XML for representation. Our approach shows two aspects of "management transparency": application designers are not burdened with designing management functionality, and system managers can manage CORBA and (D)COM from a single set of management tools.

1 Introduction

This paper gives an overview of some intermediate results from the EURESCOM Project 910, Technology Assessment of Middleware for Telecommunications. The project builds on the claim that Public Network Operators (PNOs) and Service Providers can benefit from distributed object technologies and middleware platforms. The purpose of the collaboration is the assessment of middleware technology as a means of providing large scale software infrastructures, suitable for wide range of telecommunication services. The approach is to assess middleware technologies by means of hands-on experience gained through actual experiments.

A key element for a large-scale software infrastructure is the operational management of the middleware and the software components that constitute the infrastructure. Management of middleware is essential for service providers to operate large-scale software infrastructures. These infrastructures will unavoidably consist of multi-vendor software solutions. Operational management of heterogeneous multi-vendor software infrastructures requires the development of middleware management concepts, such as relevant management information and management policies.

A. Ambler, S.B. Calo, and G Kar (Eds.): DSOM 2000, LNCS 1960, pp. 107- 118, 2000.
@ Springer-Verlag Berlin Heidelberg 2000

These concepts must be represented in a concrete, but middleware platform technology independent notation. Section II explains our approach towards developing middleware management concepts and their representation. In addition, middleware management information must be obtained in a standardized, portable way to hook management tools into the core of middleware software infrastructures, e.g. hooks into ORBs and object services. Examples of such hooks are CORBAs Portable Interceptors and COM/DCOM interceptors. Due to the unavailability of a standard and products supporting portable interceptors during the project lifetime, our own specification was developed for CORBA and (D)COM platforms, prototypically implemented in a multi vendor CORBA environment and contributed to the OMG standardisation process[7]. Section III explains our interceptor design.

All presented middleware management concepts were implemented as prototypes in a multi-vendor CORBA environment; some of the implementation concepts are presented in Section IV. Conclusions and future work are presented in Section V.

2 Middleware Management Concepts

2.1 Management Transparency and Management Information

Object middleware offers the so-called distribution transparencies [5]. This means that the application developer does not have to be aware of issues like the location of objects, the user programming languages used, the used network, available transport protocols etc. He can simply focus on the business logic of the software component he is developing, reducing time-to-market and decreasing development costs. Along the same lines, we believe that the middleware should also offer management functionality, and that this management functionality should also be transparent for the application developer. This management transparency does not only have to be offered to the application developer, but also to the system manager. A system manager has to manage an application consisting of software components running distributed objects which have been developed independently using different technologies. He should be able to manage using a single management platform. Management information in a distributed system can roughly be divided into three different categories; application specific management information, information which has to do specifically with a middleware platform, and operating system and network resources specific management information. Only the middleware platform management information is within the scope of this paper. Figure 1 depicts the various information categories, and further divides the middleware management information into object related information, request related information, message level information and network level information.

2.2 Multi-domain Management

Our focus in this work is on middleware for large-scale distributed systems crossing organisational boundaries. It is widely recognised that business relationships are rapidly becoming much more dynamic. Increasing automation in cross-business processes means that enterprise systems need to work together in heterogeneous groupings, the details of which are unknown to the designers of any of the systems involved. A similar situation is typical in any large organisation, where there is a need for applications designed and built independently to cooperate effectively in unforeseen situations.

Management in these changing environments presents a number of new problems. There are multiple management domains with multiple uncoordinated points of control, each making independent design decisions and using a range of middleware technologies. Centralised decision making is not a viable option for large systems. Global coor-

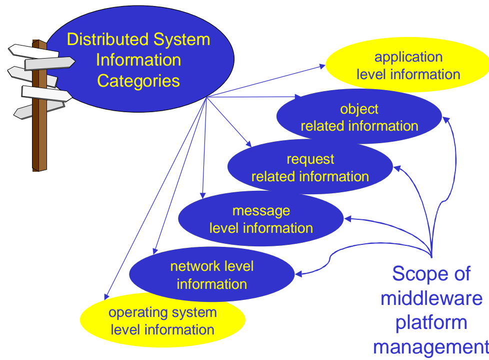

Fig. 1. Scope of Middleware Platform Management Related Information

dination is not possible and the response of the system will be the result of a collection of autonomous actions. However, end-to-end management is required so there is a need to exchange relevant management information and policies between the interacting systems.

Information exchange between domains requires common information models or, at least, common information modelling principles and syntax. Here we make the reasonable assumption that an approach based on Events (for monitoring the state of system components) and Policies (for expressing the desired behaviour of system components) will be applicable to the management of a range of different middleware technologies. Our principal focus here is on mechanisms to support management of heterogeneous middleware platforms. Management information and policies should therefore be represented in a platform technology independent way. This is essential for information that is relevant across multiple platform technologies. For specific platforms specialisation of management information is anticipated.

XML [4] is becoming the de facto standard for information representation and exchange, particularly where the information must be automatically processed. It allows users to define representations specific to their own applications with a well-defined formal syntax. XML meets our requirement for a syntax which is not tied to a specific middleware technology but which has the necessary flexibility to represent a very wide range of information. Our approach, therefore, is to define management concepts using XML DTDs (Document Type Definitions) and then to apply a mapping to platform specific formats.

Figure 2 illustrates this approach. A prerequisite is a common understanding of management concepts and processes which have relevance wider than a single product (across different CORBA implementations, for example), or middleware technology (across CORBA and (D)COM, for example). These concepts are then expressed in XML DTDs or Schema to provide reference representations of information which has a consistent meaning in multiple systems. Ideally these representations would be standardised but where this is not possible, automated transformation can be used.

The next step is to map in a well-defined way to specific technologies, such as for example CORBA and (D)COM. This approach does not impose restrictions on the message coding, protocols or management mechanisms used within a specific technology or product. This is similar to the way the CORBA specifications allow interoperability but also diversity in implementation. There is the capability of expressing information in a form that can be unambiguously understood in another

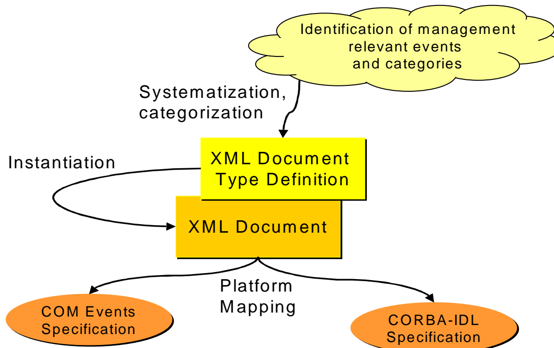

Fig. 2. Platform Technology Mapping of Management Relevant Events

technology domain when required. Although conceptually this is accomplished via the reference XML representation, there is no need actually to generate an intermediate XML document. Similarly, although it is a useful lowest common denominator, particularly in wide area networks, the XML-based approach does not imply the use of http as a transport protocol. The representation syntax and the protocols for information exchange are independent of each other.

3 Plug-In of Management Services into the Middleware Core

Management of a multi-vendor middleware infrastructure requires a standard way for obtaining management information and controlling the infrastructure (i.e. application of policies). This requirement in turn implies the need for a technology which allows management components to interact with the core components of a middleware platform in a portable, product independent manner. By the core of a middleware platform we mean runtime components that provide all the necessary basic capabilities for the interaction between distributed objects. Examples of such core components are object request brokers (ORBs) running on nodes belonging to a CORBA based platform, or containers that host Enterprise Java Beans (EJB Containers).

Current approaches for providing such standardized interaction hooks are CORBA Portable Interceptors [9] and Interceptors in COM+. During the project, both approaches were evaluated for their applicability for platform management purposes.

3.1 CORBA Interceptors

Interceptors were introduced into CORBA to provide a way for security mechanisms to interact with the ORB without the ORB having to be explicitly aware of the mechanisms. Interceptors clearly have potential application in a number of other areas, in particular management of CORBA systems, and several ORB vendors have provided interception mechanisms to increase the flexibility of their products. It turns out, that Portable Interceptors can be used to collect dynamic management information of a CORBA runtime system and to verify if the system complies with the management policies. However, interceptors in CORBA 2.2 are underspecified and not portable. The OMG issued a Portable Interceptors RFP [9] for 'a portable definition of interceptors so that system services and users may “plug into” ORB processing at particular points'. Due to the unavailability of an adopted portable interceptors specification, we have developed our own architectural model and a detailed specification, and contributed that to the standardization process within the OMG [7].

Based on an abstract ORB processing model, intercepted actions or interactions can be categorised into conceptual different abstraction levels, mainly object level interceptors, request related interceptors (request level and message level interceptors) and network level interceptors. For reasons of brevity, we present here only the conceptual model and a number of design decisions we have taken, and not the whole specification.

Object Lifetime Related Interception

We have identified the need to monitor and control the lifecycle of objects, proxies and object adapters. These lifecycle states can be intercepted just before and/or just after the lifecycle state changed, using an Object Adapter (OA) interceptor.

We focus here on the Portable Object Adapter (POA), since we do not consider it worth the effort to also include the deprecated Basic Object Adapter. Information that can be monitored or changed with an OA-interceptor includes the IOR template, the name, the parent POA and the policies that will be used. In a POA context there are four state changes for an object: creation, activation, deactivation and destruction. Potentially each of these four can be intercepted just before and just after the state change.

At the client side we identified a need to intercept the creation of a proxy. The term proxy refers to the client side ORB internal representation of an object referenced by an interoperable object reference (IOR). With a request interceptor it is possible to discover a connection between a client and a server as soon as it is actually used (at the first request). A proxy interceptor however allows the discovery of a logical connection between a client and a server before it is actually used, or even if it is never used. Available information is the IOR that was used to create the proxy, and the resulting object (i.e. proxy).

Request Related Interception

Request related interceptors are invoked during the ORBs processing of operation invocations. During information modelling, we identified several interception points and the information available at each point.

There are four interception points which directly correspond to activities of an ORB or POA during the processing of a particular operation invocation:

- The client side ORB receives the operation invocation from a client co-located to it. For this activity, we identified the corresponding interception point *ClientPreInvoke*. The client side ORB communicates the operation invocation in some form, corresponding to a particular interoperability protocol, to the target object, using either a network connection or some local invocation paradigm, if the target is in the same capsule.
- The target ORB receives the operation invocation and identifies a servant which implements that operation. Before the servant is invoked, we identified the corresponding interception point *ServerPreInvoke*.
- When the servant completes the processing of the requested operation, the ORB at the target side gains the control of the invocation processing. At that point, we identified the corresponding interception point *ServerPostInvoke*.
- The target side ORB communicates the reply for the requested operation back to the client side ORB, which in turn receives that reply and hands it back to

the client. Before the client receives the reply, we identified the corresponding interception point *ClientPostInvoke*.
- During the processing of a particular request, at both client and target side ORBs, system exceptions may be thrown. Because this can happen at any activity an ORB processes, we identified the corresponding interception point *SystemException*.

Several other interception points may be of great interest at request level, for example the ability to intercept the ORB's identification of a servant on the target side. To define the semantics of this activity and to include this in the specification, is an item for future work.

Network Related Interception

During the implicit binding for a non capsule-local interaction between a client and the target object, an ORB actively establishes connections or accepts incoming connection requests. Such network connection related activities include the choice of a transport endpoint to listen on at the server side, the choice of a transport endpoint for connection establishment and the actual connect procedure in the client ORB, the acceptance of an incoming connection and connection closure, performed by either the client or server side ORB.

Network level interceptors may be involved during such activities at five interception points related to connection management activities:
- The *PreAccept* interception point relates to the choice of a transport endpoint to listen for incoming connection requests at the server side,
- The *PreConnect* interception point relates to the choice of a transport endpoint for connection establishment, just before the actual connection establishment, at the client side - the object reference of the target object is known, which contains a number of transport protocol specific profiles;
- The *PostAccept* interception point relates to the acceptance of an incoming connection request at the server side - the selected and activated transport endpoint for a particular network connection is known.
- The *PostConnect* interception point corresponds to the activities done by a client side ORB just after successful or unsuccessful connection establishment - the selected and activated transport endpoint for a particular network connection is known.
- The *Close* interception point relates to activities to close a network connection, performed at either a client or server side ORB.

The network connection interception concepts can be applied to connection oriented networks. Although the use of TCP/IP in the CORBA world is common, more work is needed to develop concepts for interception of network related activities for the usage of connectionless network. One use of a connectionless network is already proposed by the GIOP over SCCP mapping in [1][8], another one for GIOP over IP-M and UDP has been presented in [3].

Management of Interceptors

Interceptors can access and modify information of object interactions, passed within the ORB core. In this way, they work together with the ORB in a very tight manner. Nevertheless they are entities, which should be distinguished clearly from the ORB core.

The lifetime of an interceptor seen from an information viewpoint is only correlated with the ORB i.e. an interceptor cannot exist without the ORB it belongs to. During the

lifetime of the ORB, new interceptors can be created and existing interceptors can be deleted. Depending on the lifetime of an interceptor, the visibility of an interceptor to the ORB could be defined such that a certain interceptor may or may not take part in an invocation.

Interceptors are not assumed to inform the ORB of any modification they make to the information they receive before or after an invocation. The ORB has to trust an interceptor not to harm to the whole system in terms of security, functionality and error recovery. Furthermore no standardization of interceptor functionality can be done, so that ordering constraints on the invocation of different interceptors of the same interceptor type have to be applied from another entity.

This leads to the idea of an interceptor registry, which is responsible for registering and de-registering interceptors and for applying invocation constraints on the registered interceptors. The interceptor registry is the only entity that controls which interceptors to call during each invocation. In this role it has no knowledge about the functionality of each interceptor in principle, but classifies each interceptor according to the different interceptor types. Using this information, the interceptor registry can instruct the ORB to call the appropriate interceptors at each stage of the invocation.

The classification of an interceptor is done upon registration by providing the appropriate information about the nature of the interceptor. A registered interceptor is then able to intercept subsequent invocations. The question of whether a registered interceptor should be used in invocations or not is controlled by the interceptor registry on behalf of the application. No decisions are made by the interceptor registry itself, since the only appropriate information about the interceptor it has, is the interceptor type.

For the same reason that the interceptor registry cannot decide for itself whether to enable or disable a certain interceptor during invocations, it cannot make decisions about the ordering of interceptors, i.e. which interceptor to call first and which to call last during an invocation. Again this decision is up to the application, which must have appropriate knowledge about the registered interceptors. The interceptor registry assists the application through providing appropriate information about the current invocation order of the registered interceptors and providing a mechanism for reordering, but has no responsibility for defining the correct ordering of the interceptors.

To correlate an incoming reply with a previous outgoing request, interceptors often need to store some information about certain invocations for later evaluation. This requires some kind of cookie, which can be filled with information and can be accessed by the interceptor. These cookies, held within the interceptor registry for each registered interceptor, are created during the invocation phase of the interceptor and are made available to the interceptor again during the response phase.

Computational Model

After completing the phase of defining interception points and their related information, we developed a computational model for the interactions between the ORB runtime system and interceptor objects as well as computational concepts for the management of interceptors. The purpose of the computational model for interceptors is to define the interfaces and operations which are provided by both the ORB runtime system and the interceptors. Beyond that, the architectural model for inter-

ceptor invocations, the related interceptor management concepts and instantiation semantics are specified. A main goal was to ensure the portability of interceptor implementations across different ORB products.

More than one interceptor may be registered with an ORB for a particular interception point at the same time. These registered interceptors are invoked during the ORB's processing of an invocation.

We found three basic architectures for how an ORB may invoke interceptors:

- Daisy-chained invocation - The ORB calls the first interceptor, which in turn calls the second directly or indirectly, and so on.
- Serial invocation - The ORB calls each interceptor at each interception point, and the invoked interceptor returns control to the ORB before the ORB invokes the next interceptor.
- Conditionally serialized - The ORB calls each interceptor serially, but may omit an interceptor based on some constraint or condition.

A main advantage of daisy chained interceptor invocations is stacking of interceptors. In daisy-chained invocations of interceptors, there is no need for the ORB to explicitly remember, which interceptor was invoked during the request processing and should therefore be invoked again during reply processing. On the other hand, daisy-chained interceptors require different request related operations to handle synchronous, deferred synchronous and asynchronous requests. Moreover, an application of daisy-chained interceptors together with time independent invocations in CORBA Messaging [11] seems to require another set of operations to handle requests. Finally, a serial native ORB architecture would have to jump through hoops to emulate a daisy-chained architecture.

Serialized interceptor invocations require explicit mechanisms within the ORB to register which interceptors were called while request processing in order to invoke those interceptors again during reply processing in reverse order. On the other hand, this more complex ORB behaviour will be compensated by a number of advantages. First, a serial architecture for portable interceptors can be implemented over a wider variety of existing ORB implementations. A serial architecture is less sensitive to the various types of invocations that may be made in a CORBA client, including those invocation types supported by CORBA Messaging. Due to the mentioned advantages of serialized interceptor invocations and with respect to ease of implementation using existing ORB sources and the proprietary filter technologies of some ORB products, we decided to base all interceptor specifications on the serialized approach.

Another design consideration regards the number of interceptor local objects involved in the processing of a request or reply at an interception point. Each interceptor local object fulfils a particular functionality, e.g. an interceptor local object that implements a transaction service, or another that supports a certain management task. The question now is, whether such an interceptor local object should be instantiated each time that interception point is reached. Interceptor instantiation can be managed in two ways. An interceptor instance is created for each interception processing, e.g. an instance creation for each request/reply or an interceptor instance is created once and will be activated for each interception processing, e.g. for each request/reply. The first approach is more flexible, but requires extensive management capabilities as well as standardized factory interfaces for interceptors. The second approach is not as flexible but doesn't require such extensive management capabilities as the dynamic one.

Another criterion for interceptor processing is whether an interceptor may affect the ORB's behaviour with respect to a certain invocation processing or not. If an interceptor

is not allowed to change the way an ORB controls the request processing, this interceptor can only be applied to monitoring tasks. If an interceptor affects the ORB's interaction processing, this may be done in several ways. A request related interceptor may change parameters of a request or reply message, but not the control flow with respect to the handled interaction. Changed request or reply parameters may or may not include changes to the service context of an interaction. A request related interceptor may change the control flow of an ORB regarding a particular interaction. This means for example, that a request related interceptor registered at the client side with respect to a certain object may produce a reply for a given request and send this reply back to the client directly. A network level interceptor may or may not send a given byte stream by itself, i.e. the interceptor instead of the ORB core sends the byte stream through a network API.

To identify to what extent an interceptor influences the ORB's behaviour with respect to a certain interaction processing, we defined capability sets for interceptors support within a particular CORBA runtime environment. These capability sets include

- Monitoring capability - an interceptor is only able to monitor the ORB's processing of a certain interaction,
- Parameter changing capability - an interceptor is able to change parameters of a certain interaction, like operation request parameters or service contexts,
- Control flow changing capability - an interceptor is able to change the ORBs normal control flow while processing a certain interaction.

3.2 COM/DCOM Interception Mechanisms

The architectural layering of COM is different from CORBA, and the interlocking of the COM runtime system with the Windows system is very tight. For this reason, it is very difficult to extract information from a (D)COM method invocation in an uniform way.

The extraction of method call information can be done on the component side using a message filter. A message filter is a component that exposes the interface `IMessageFilter`. Through this interface, the COM runtime system tells about incoming calls to components. The method that will be called in this case, is:

```
DWORD HandleInComingCall (
[in] DWORD dwCallType,
[in] HTASK htaskCaller,
[in] DWORD dwTickCount,
[in] LPINTERFACEINFO lpInterfaceInfo
);
```

A message filter component can extract the interface identifier from the iid parameter, and the method number. The method number is actually the offset of the called method in the virtual function table of the object inside the component implementing the called interface. On this point at least the interface and the invoked method can be extracted. In contrast to CORBA there is no easy way to extract the method name or method parameters for management purposes.

This could be overcome with COM+, the successor to (D)COM. COM+ (version 1.0) introduces interceptors for attaching standard services (e.g. MTS) to components. Microsoft promises for future version of COM+ the introduction of user-de-

fined interceptors. It is supposed that these user-defined interceptors can play a similar crucial role in management, like interceptors in CORBA.

4 Management Service for CORBA Based Middleware Platforms

The Portable Interceptors framework, as the base for management information collection and management policy verification, was implemented during the project in order to verify its applicability for management purposes. In general, the interceptors specification was used for interactions between the specific management interceptor implementations and the ORB core, while the management interceptors interact with the environment using specific management channels (see Figure 3).

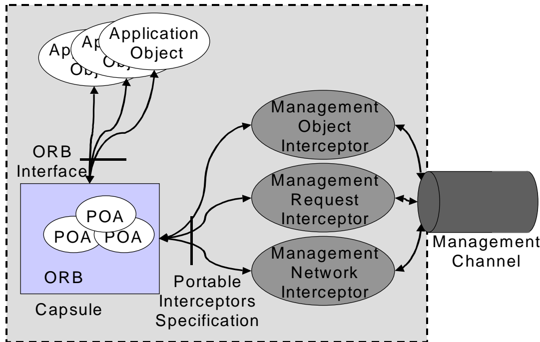

Fig. 3. Management Interceptors Connected to the ORB and Management Channels

In more detail, a management specific interceptor object interacts with the ORB in a portable way to obtain object interaction information as well as to set and verify certain management policies. On the other hand, such interceptors interact with management channels to notify management-relevant events which occur during an ORB's object interaction processing. Furthermore, management interceptors receive management policies from such management channels. In that sense, management interceptors are context specific software components that interact with the platform core components in a portable, product independent manner.

The implementation of the portable interceptors framework focused on commercially available ORB products, we chose to use Inprise' VisiBroker, IONA's OrbixWeb and OOC's ORBacus. For these products, different implementation strategies were applied. While for VisiBroker and OrbixWeb, the product specific proprietary filtering mechanisms were wrapped in order to comply with the specification, for the ORBacus, the available source code was extended in order to support the interception of object interaction events presented above. Our management interceptors can be plugged into the interceptor framework implementation for each product in a portable way.

Management interceptors interact on one side with platform core components, like Object Request Brokers. On the other side, they communicate management-related notifications to management applications and receive management policies from those. For that purpose, the document type definition for management-related events and policies is mapped to CORBA specific constructs for transmission purposes. To achieve this, we defined a generic mapping between XML DTD constructs and constructs of the Interface Definition Language (IDL) of CORBA. This mapping in general uses IDL `valuetypes` as target constructs for XML elements. Attributes of elements are mapped to members of their representing valuetypes. This language mapping was ap-

plied to both the management-relevant events specification and the management policies definition, resulting in IDL definitions that represent the XML specification. These IDL definitions are used as communication elements between a management interceptor and management applications via management channels. Management channels are realized using the Notification service adopted by OMG [12]. Each conceptual management channel maps to a notification channel within a CORBA platform. The complete picture of how a management interceptor collects management information is depicted in Figure 4.

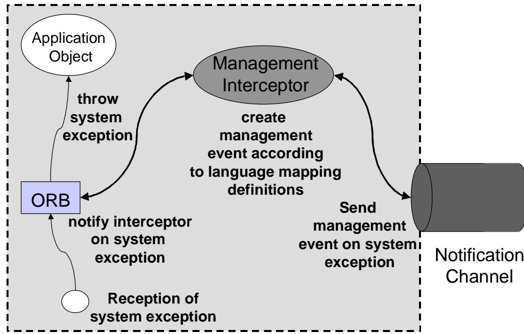

Fig. 4. Collection of Management Information

If for instance an ORB receives a system exception as result of a operation invocation by an application object, it first verifies, whether an interceptor is registered for the SystemException interception point. If so, then the registered interceptor is notified on the happened system exception. The management interceptor in turn creates a IDL construct, that represent the according management event and sends that event through a notification channel to a management application. The application object, that originally invoked the operation that resulted in the system exception, receives that exception as usual.

If on the other hand a management policy is received by the management interceptor, it uses the standard ORB interface to override policies for the current thread of invocation. The mapping between management policies arriving through the management channel and those policies that can be set at the ORB is currently implemented as known by the interceptor object.

5 Conclusions and Further Work

Management services for middleware platforms must be provided with hooks within the core platform components so that basic object interaction processing can be monitored and controlled. "Management transparency" is an extremely desirable feature of heterogeneous systems - it should be possible to achieve end-to-end management in a distributed system which is built using a range of different middleware products and technologies. The use of Interceptors in CORBA and COM+, as described in this paper, offers the prospect of being able to "break out" of a specific middleware technology when required, which is an essential capability if portable management services are to be realized. In addition, some degree of standardization in the mechanisms for extracting relevant state information (i.e. Events) and inserting control information (i.e. Policies) is required. The correspondence or relationships between the idi-

oms used in particular technology domains must be established where it is necessary to exchange management information between domains.

It is not envisaged that all middleware management will be restricted to some common subset of concepts applicable to all platforms. In other words, not all management services will be portable. Particular products are likely to offer management functionality that is product-specific. Equally, families of products (e.g. CORBA ORBs) are likely to have some management functionality in common. Beyond this, there will be a core of management information which is applicable across technologies. In addition a neutral reference representation and syntax must be chosen. XML seems to be a suitable choice for a standard representation. It is flexible enough to represent a very wide range of structured information, has a strict enough syntax to allow automated processing and has broad industry acceptance. Standard document types will be required to define the structure of management information but the text-based nature of XML documents and their straightforward structure makes automated transformation realistic.

There are several open issues, not addressed in this paper. These include techniques and concepts to support the correlation of multiple management events in heterogeneous distributed systems. In addition, while the concepts seem to provide many attractive features for introducing management functionality into heterogeneous middleware systems, the performance implications of specific implementations, based on CORBA Portable Interceptors or on COM+ interceptors need to be assessed.

References

[1] Fischbeck, Kath: *CORBA Interworking over SS7*, in Proc. of ISN'99

[2] Fischbeck, Holz, Kath, Vogel: *Flexible Support of ORB Interoperability*, in Proc. of Interworking ´98

[3] Halteren, Noutash, Nieuwenhuis, Wegdam: *Extending CORBA with specialised Protocols for QoS Provisioning*, in Proc. of DOA'99

[4] W3C Rec.: *Extensible Markup Language V. 1.0*

[5] ITU Rec. X.901-X.904: *Reference Model for Open Distributed Processing*, ITU-T '95

[6] OMG: *The Common Object Request Broker Architecture: Architecture and Specification*, Revision 2.3, OMG docs. formal/99-07-01 to formal/99-07-28

[7] Expersoft et.al.: *Portable Interceptors Joint Initial Submission*; OMG doc. orbos/99-04-10

[8] AT&T et.al.: *Interworking Between CORBA and TC Systems*, OMG doc telecom/98-10-03

[9] OMG: P*ortable Interceptors Request for Proposals*, OMG doc. orbos/98-09-11

[10] BEA Systems et al: *Portable Interceptors*, OMG doc. orbos/99-12-02

[11] OMG: *CORBA Messaging*, OMG doc. ptc/00-02-05

[12] OMG: *Notification Service Specification*, OMG doc. telecom/98-11-01