A Filter Object Framework for MICO

Pranav S. Nabar[1] Amit L. Padalkar[1] Rushikesh K. Joshi[2]

[1] K. R. School of Information Technology,
IIT Bombay, Mumbai, INDIA.
[pranav, amitp]@it.iitb.ac.in,
[2] Department of Computer Science and Engineering,
IIT Bombay, Mumbai, INDIA.
rkj@cse.iitb.ac.in

Abstract. Filtered delivery model of message passing in an object-oriented distributed computing environment facilitates separation of message control from message processing in a transparent manner. In this model, special objects called *filter objects* have the ability to filter messages in transit and perform intermediate actions. We present the design and implementation of the message filtering model for transparent dynamically pluggable filter objects for MICO, an open-source CORBA implementation. For implementing the filtering framework, enhancements to the MICO implementation model are proposed. A process for development of filter objects with related tool support has also been outlined.

Keywords: CORBA, filter objects, pluggable filters, MICO, transparency.

1 Introduction

Distributed object-oriented systems are built using collaborating objects on a distributed platform. These distributed objects collaborate by passing messages. In some cases, applications may require to change their message control policies dynamically. For example, an application might need to check message contents for validity or against security concerns. In such cases, message control cannot be abstracted out without breaking its transparency. In applications like these, interceptors[8] or filtering models models such as Composition Filters[1], Filter Objects[6] and Encapsulators[9] achieve separation between message processing and message control. In the filter object model, messages sent to the destination objects can be intercepted by first class filter objects. While filter objects intercept messages, the calling semantics of the source object do not change. This behavior of filter objects can be employed to change the system behavior by transparently intercepting calls and controlling or modifying the invocation requests coming from clients. In this paper, an implementation of a filter object framework for MICO[12], an open source implementation of CORBA[12] is described.

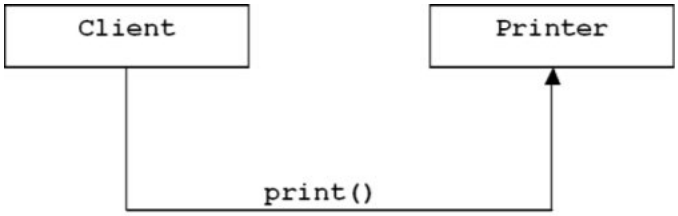

Fig. 1. Direct delivery model

1.1 Filter Objects

Figure 1 shows the conventional direct delivery message passing model. A `Client` object on a network sends a `print()` message to a `Printer` object. The message is delivered directly to the destination object, which in this case is `Printer`. As a result, the corresponding operation is invoked at the destination object. This implies that any intermediate message control cannot be decoupled from the operation without sacrificing its transparency.

For example, after a period of time, it may be found that one `Printer` is insufficient to fulfill the needs of increasing number of clients, and more printers need to be added to the system for load balancing. But it may not possible to dynamically introduce this solution without making considerable changes to the `Printer` object code.

Filter objects provide an elegant, modular solution to the above problem. Removal, addition and replacement of filter objects do not require any modification of code, either at the source object or at the destination object. Figure 2 shows how load balancing can be introduced through the filtered delivery model. Here a `LoadBalancer` object, which maintains a list of additional `Printer` objects, is plugged to original `Printer` object. It intercepts all the incoming `print()` requests to original `Printer` object, and redirects them to one of the `Printer` objects to achieve load-balancing.

CORBA implementations such as Visibroker[2] and Orbix[3] provide interceptors. Visibroker provides various kinds of client and server interception points such as bind interception, client request interception, POA create/destroy interception, server request interception etc. Orbix provides filters in per-process and per object categories. Various kinds of filtering points may be specified such as pre-marshaling and post-marshaling filter points per process and pre-invocation and per-invocation filter points per object are supported.

Filter objects[6] on the other hand are first class dynamically pluggable objects, which are provided in the current implementation as full fledged CORBA objects. Apart from their filtering capabilities, they may also provide public in-

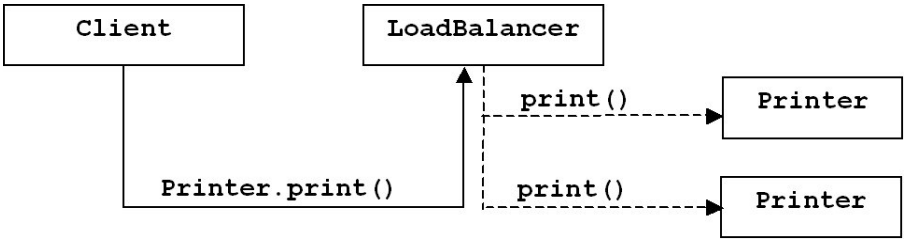

Fig. 2. Filtered delivery model

terfaces. Filter objects have been implemented for Java[7] and C++[6], and in distributed environment at user level for MICO[11] and on an AspectJ based environment[5]. The current work enhances the MICO kernel to support a kernel level implementation of first class filter objects such that the ORB becomes *filter-aware*.

1.2 Filter Object Framework for MICO

MICO[12] is a CORBA 2.3 compliant open-source implementation of the OMG-CORBA standard. We have designed and implemented a filter object framework in MICO version 2.3.4. This involved several system-level enhancements to the MICO kernel. In our framework, filter objects are full-fledged CORBA objects. Filter objects can be plugged and unplugged onto server objects transparently. The framework is based on the *message filtering model*[6] for filter objects. This paper introduces the filtering framework to a considerable detail.

We, first discuss the design and implementation of the framework in subsequent sections. The static and dynamic models are elaborated. Further, the development process, tool support and applications of filter objects are also discussed. Finally performance indices are provided for the implementation running on a Linux cluster.

2 Design of the Filter Object Framework

We begin our discussion by documenting the design requirements and subsequently discuss various design alternatives.

2.1 Design Issues and Requirements

Besides seamlessly fitting into the existing MICO framework, the filter design has to meet the following requirements.

- Support basic filtering actions like pass and bounce.
- Support extended filter object properties, like layering and grouping.

- Filter objects need to be transparent to clients as well as servers.
- Evolution of the system using filter objects should involve minimal change (ideally *no* change) in the existing system code.
- Development process of a system using filter objects should not be radically different from the existing object-oriented development processes.
- Addition of filtering framework into the system should not substantially increase the overheads on the system.
- Granularity of control over filtering actions is also a design issue.
- We assume filters are plugged and unplugged by trusted hosts. Security issues are not considered.

The design alternatives were largely based on two major considerations, the location where the mappings between the server and the filter objects would be held and the point in the actual invocation interaction where the call would be intercepted. On detailed analysis of the implementation model, it was found that the second consideration was dependent on where the mappings would be located. This meant, we only needed to evaluate the choices for locating the mappings between server objects and the filter objects plugged onto them. An analysis led us to three main design alternatives discussed below.

- **Choice 1:** *A CORBA object/service stores the mappings:* In this alternative, a service is used to maintain the mappings between server and filter objects. The CORBA object providing this service has a standard interface. This makes it similar to other standard CORBA services. Applications use `plug()` and `unplug()` interfaces on this service to plug and unplug filters onto the server objects. Whenever the server receives an invocation, it uses a reference to the mapping service to check for attached filters and take appropriate action. This design would lead to a much higher timing overloads during method invocations.
- **Choice 2:** *Mappings maintained in micod:* In this case, we store the filter-server mappings in the `micod` daemon. Local `plug` and `unplug` calls are forwarded to `micod` for plugging and unplugging filters. Whenever client makes a method invocation, it has to pass through `micod`, which checks for the filters plugged to servers and forwards the call accordingly. This leads to a substantially higher overhead during method invocation, in case of normal invocation—when no filters are plugged.
- **Choice 3:** *Mappings maintained at the server-side:* Here the mappings are maintained in an object in the server-side library. During the method invocation, the presence of plugged filters is checked at the server-side and appropriate action is taken. Overheads on *normal* method invocation is minimized.

2.2 Evaluation of Design Choices

The comparison between the design alternatives can be abstracted in a feature matrix. Choice 1, 2 and 3 refer to position of *server-filter* mappings in separate

Overheads	Design Choices		
	1	2	3
Plug/Unplug	High	Low	High
Method invocation without filters	High	High	Low
Method invocation with filters	High	High	High

Table 1. Feature matrix

CORBA object providing a mapping service, in BOA daemon (micod) and in server object respectively as described in the previous section.

We observe that all three choices are capable of supporting both essential and extended properties of filter objects. Hence the distinguishing factor between these choices is overheads incurred by each of them. We have considered two main types of overheads viz. plug/unplug overhead and method invocation overhead, with and without plugged filters. These form the basis of comparison aided by feature matrix shown in Table 1.

In choice 1, where client-filter mappings are stored in a mapping service, both the overheads are high. In this case, every plug/unplug beta message and method invocation has to consult the mapping service. This naturally results in higher overheads. In case of choice 2, mappings are maintained in the BOA daemon, micod. This choice reduces plug/unplug overhead since these beta messages are sent to the daemon instead of a mapping service. However the method invocation overhead increases even for normal method calls. Whenever a method invocation occurs, it has to pass through `micod`, which checks for the plugged filters and forwards the call accordingly. This clearly leads to higher overhead during method invocations even if the server object doesn't have any plugged filters. With plugged filters, routing invocation to them results in additional invocation overheads. With choice 3, where mappings are stored on the server-side, plug/unplug overhead is higher than that in choice 2. This overhead increases since sending these beta messages requires obtaining a server object reference and then invoking these beta operations on it. However, with this approach, there is no penalty on method invocations for objects without plugged filters. The method invocation on server objects with plugged filters results in higher overheads because of method routing to filters, which is unavoidable in all the three cases.

From the above discussion, it is clear that choice 1 has comparatively higher overheads than choices 2 and 3. In case of choice 2, though its plug/unplug overheads are low, method invocation overheads are one the higher side. Since method invocations are more frequent than plug/unplug beta messages, an implementation using choice 2 would lead to higher overall time losses than choice 3. Hence choice 3 better suits the design considerations and was selected for implementation of the filter object framework.

3 Static Model

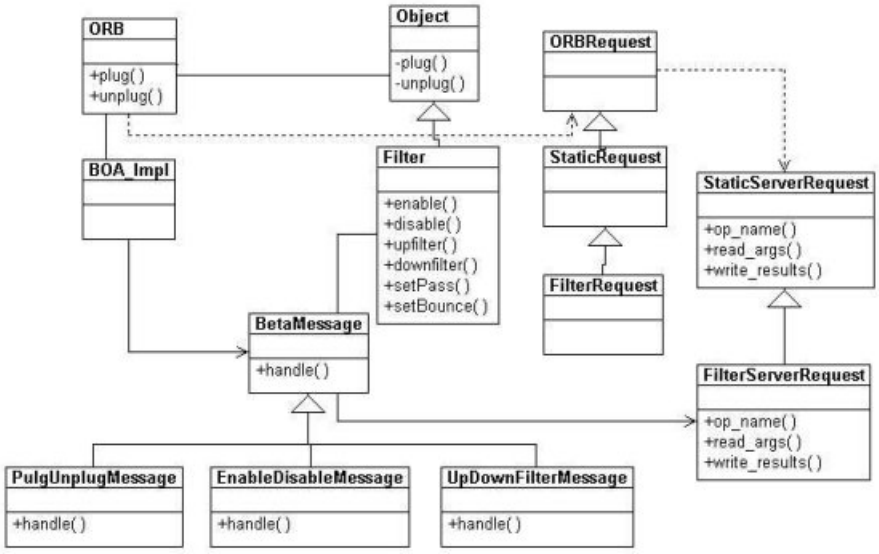

Fig. 3. Filter Framework - Static Model

The design model includes several modifications to MICO implementation to satisfy the design requirements stated in Section 2.1. In this section, we present the modifications to MICO's static model. Figure 3 shows the class diagram of filter object framework. The modification to the dynamic model are covered in the next section.

3.1 Modifications to MICO

Modifications were introduced into the MICO implementation to satisfy the filtering requirements (Section 2.1). These changes include creating new interfaces and modifying the existing interfaces by including new methods to fulfil additional set of responsibilities. We now discuss each of these.

CORBA::Filter class The `CORBA::Filter` class is the common superclass of all *filter objects*. Since all filter objects are CORBA objects, the `Filter` class inherits from the `CORBA::Object` class. `Filter` class gives concrete implementations of the following methods:

- `enable`: enables a particular filter method, passed as argument.

– `disable`: disables a particular filter method, passed as argument.
– `upfilter`: maps a filter method as an *upfilter* to a server method.
– `downfilter`: maps a filter method as a *downfilter* to a server method.
– `setPass`: to be used by the filter developer in case of a *pass* action.
– `setBounce`: to be used by the filter developer in case of a *bounce* action.

CORBA::Object class The `CORBA::Object` class is the common superclass of all CORBA objects. This class gets the additional responsibility of maintaining filter-server mappings. The system can access these mappings through private methods -

– `plug`: Plugs the filter object onto the server object.
– `unplug`: Unplugs the filter object from the server object.

CORBA::ORB class The `ORB` class implements the ORB interface defined by OMG-CORBA standard. For plugging and unplugging filter objects onto the server objects, additional public interfaces are required. These are provided by the `ORB` class through the `plug` and `unplug` methods. The process managing filter objects called the *catalyst* uses this interface for plug/unplug actions on the server object.

Carrying Filter Requests In the MICO implementation, static invocations are encapsulated into objects of `StaticRequest` class on the client side, and into objects of `StaticServerRequest` class on the server side. We specialize these two classes to carry *filter requests*—invocations routed to filter objects. Hence classes `FilterRequest` and `FilterServerRequest` inherit `StaticRequest` and `StaticServerRequest` classes respectively. Methods `op_name`, `read_args` and `write_results` of the `StaticServerRequest` class are overridden in the `Filter–ServerRequest` class for specialized implementation.

Carrying Beta Messages All the privileged filter messages like plug–unplug, enable–disable and set method mappings (upfilter–downfilter) are handled internally by objects of a specialized handler class called `BetaMessage`. This class defines a virtual `handle` function. The concrete handler implementation appears in the three subclasses of the `BetaMessage` class. These are -

– The `PlugUnplugMessage` class: Handles plug and unplug messages.
– The `EnableDisableMessage` class: Handles enable and disable messages.
– The `UpDownFilterMessage` class: Handle upfilter and downfilter messages to set upfilter and downfilter mappings at the filter objects.

4 Dynamic Model

In this section, we present behavioral view of our design. We start with discussion of the beta message handling. In the following sections, we discuss modifications to normal method invocation sequence by the filter object framework and persistence of information supporting the framework.

4.1 Beta message handling

As discussed in Section 3.1, *beta messages* are special control messages, which must be handled by the filter object framework. Although these are special messages, they follow normal method invocation sequence followed in MICO. All beta messages are handled by BOA in its `invoke` method. We will discuss the handling of the three different categories of beta messages, mention earlier, one by one.

Upfilter/Downfilter beta messages Each filter object interface consists of *upfilter* and *downfilter* methods. *upfilter* methods filter corresponding client methods whereas *downfilter* methods filter return results. Multiple upfilter and downfilter methods can be associated with single client method. To provide this facility to every filter object, `Filter`, base class of each filter object, maintains mappings between client method and corresponding upfilter and downfilter methods. These mappings can be manipulated by sending *upfilter* or *downfilter* beta messages. Each beta message takes client method name and corresponding upfilter/downfilter method name as arguments. The filter object framework includes *filterconf* utility that facilitates sending these messages to filter objects. We now present the algorithm for sending and handling these messages which is implemented in the `handle` method of the corresponding classes.

Scenario: Sending Up/Down filter message

1. Create `StaticAny` objects representing client and filter method names.
2. Create `StaticRequest` object representing appropriate beta message.
3. Add the arguments and invoke the request.

Scenario: Handling Up/Down filter message

1. Create server side request and read the client and filter method names using it.
2. Check whether filter object already contains mapping corresponding to client method.
3. If yes, get the corresponding Up/Down filter methods and add the filter method name if it didn't exist already.
4. If no, create appropriate mapping in the filter object between received client and filter method.
5. Add the filter method name with *disable* status if it didn't exist already.

Enable/Disable beta messages These beta messages control status of individual filter methods. `Filter`, base class of every filter object, maintains filter method names and their status. The filter method status can be changed by sending *enable* or *disable* beta messages. Each beta message takes filter method name as argument. The filter object framework includes *filterconf* utility that facilitates sending these messages to filter objects. We now present the algorithm

for sending and handling these messages.

Scenario: Sending Enable/Disable message

1. Create `StaticAny` object representing filter method name.
2. Create `StaticRequest` object representing appropriate beta message.
3. Add the argument and invoke the request.

Scenario: Handling Enable/Disable message

1. Create server side request and read the filter method name using it.
2. If this is enable message, enable corresponding filter method.
3. Search Up/Down filter methods corresponding to every client method to check if the filter method is part of either Upfilter or Downfilter methods. If yes, change the status of other methods to disabled. This ensures that at most one Up/Down method is enabled for any client method.
4. If this is disable message, simply disable corresponding filter method.

Plug/Unplug beta messages These beta messages allow plugging and unplugging of filters from their clients. `ORB` provides a public interface–*plug* and *unplug*–for this purpose. This interface in turn makes use of private interface of `Object`, which stores a list of plugged filters. The filter object framework includes *filterconf* utility that facilitates sending these messages to client objects. We now present the algorithm for sending and handling these messages.

Scenario: Sending Plug/Unplug message

1. Convert a filter object reference to string.
2. Create `StaticAny` object representing stringified filter object reference.
3. Create `StaticRequest` object representing appropriate beta message.
4. Add the argument and invoke the request.

Scenario: Handling Plug/Unplug message

1. Create server side request and read the stringified reference using it.
2. Convert the stringified reference to object reference and narrow it to the filter reference.
3. If this is plug message, add the filter reference to filters list if it didn't exist already.
4. If this is unplug message, simply remove the filter reference from filters list if it exists.

4.2 Filtered method invocation

We discuss modifications made to normal MICO method invocation sequence in order to incorporate the filtering functionality. First important modification to invocation sequence is creation of a specialized `FilterServerRequest` instead of `StaticServerRequest` in `make_request()` method. The responsibilities of server-side filter request object include method name translation and performing *upfiltering* and *downfiltering*. To satisfy these responsibilities, it overrides three methods viz. `op_name()`, `read_args()` and `write_results()`.

Method name translation The translation process is performed by `op_name()` method. This method returns current operation (method) name. In the filter object framework, it performs the translation from intercepted filter client method name to appropriate filter method name. Here, we present algorithm used by the translation process.

Scenario: Method name translation

1. If the current object is not a *filter*, then no translation is needed. Set invocation status to normal and return the actual method name.
2. If this is an intercepted method call from filter client:
   (a) If the current filter object doesn't have mappings corresponding to intercepted client method, then set the invocation status as bad filter client invocation and return.
   (b) If intercepted method call is *Up* call, search the Up filter methods corresponding to intercepted client method.
   (c) If intercepted method call is *Down* call, search the Down filter methods corresponding to intercepted client method.
   (d) If one of the Up/Down filter methods is enabled, set the invocation status as normal filter client invocation and return enabled method name.
   (e) Otherwise, set the invocation status as bad filter client invocation and return.
3. Else set the invocation status as normal filter method invocation and return the actual method name.

Up filtering The `read_args()` method performs the work of reading the argument data, which was sent as part of method invocation from client side. This data is then passed on to actual method at server side provided `read_args()` is successful in reading the data. In the filter object framework, `read_args()` method handles additional responsibility of *Up* filtering. Here, we present algorithm used to carry out Up filtering.

Scenario: Up filtering

1. Read the argument data. If unsuccessful, set read status as false and return.
2. Perform Up filtering setting method *pass* status.
   (a) If the invocation status is *bad filter client invocation*, set the pass status as false.
   (b) If there are no filters plugged to current object, set the pass status as true.
   (c) Else iterate through plugged filters in LIFO order till all filters are traversed or one of the filters cause method to bounce. For every filter, create a filter request and copy the current invocation request's arguments and return value to it. Make the copied arguments *inout*. Finally, invoke the intercepted filter client method call using newly created filter request.

(d) If none of the plugged filters bounce the method, set pass status as true else set it as false.
3. If pass status is true, set read status to true and return.
4. Else set read status to false and call `write_results()`.

Down filtering The `write_results()` method performs the work of communicating return results and *out* argument values to the client side. In the filter object framework, `write_results()` method handles additional responsibility of *Down* filtering. Here, we present algorithm used to carry out Down filtering.

Scenario: Down filtering

1. Perform Down filtering if method *pass* status is true.
   (a) If current method has *void* return type, do nothing.
   (b) If the invocation status is *bad filter client invocation*, do nothing.
   (c) If there are no filters plugged to current object, do nothing.
   (d) Else iterate through plugged filters in FIFO order till all filters are traversed. For every filter, create a filter request and copy the current invocation request's return value to it. Add the current invocation request's return value to the request as *in* argument. Finally, invoke the intercepted filter client method call using newly created filter request.
2. Communicate results of current invocation back to client side.
3. If current call is intercepted call at filter object, communicate method pass status to filter client.

4.3 Persistence of framework related information

As described in Section 4.1, every filter client maintains a list of plugged filters and every filter maintains mappings from client method to upfilter/downfilter methods and status of each of its methods. Any CORBA object providing a service may *deactivate* itself to save resources if it anticipates period of inactivity. Deactivation involves stopping the service and freeing all resources. The object will be reactivated whenever the service it offers is requested. Since filter clients and filters are CORBA objects, they can also choose to be deactivated. If the framework related information mentioned above is not saved while deactivation, it will be lost making the framework ineffective. In this section, we discuss the strategy adopted by us to make the framework related information persistent.

A CORBA object requests its deactivation by calling `shutdown()` on BOA. The shutdown procedure in BOA involves deactivating object implementations. Before deactivation actually takes place an object is given a chance to save its data by `save_object()` method in BOA. This method calls `_save_object()` on every object, which does actual work of storing object data, usually in a file. On reactivation, object initializes itself using the stored data.

In our approach for making the framework information persistent, we have modified the `save_object()` method of BOA. Before giving an object a chance to save its data, framework specific information in filter client and filter object is stored in files as appropriate. The information is stored in two files, one contains client specific data whereas other contains filter specific data. Note that creation of these files is not mutually exclusive since a filter client can itself be a filter e.g. in case of multi-level filtering.

On reactivation of the object, saved framework information is not immediately restored since it is required only when a method is to be invoked on that object. The method invocation request is represented at server-side by `FilterServerRequest`. Hence, the framework information is restored only while creation of this server-side request. File names, where the framework information is stored, are made unique for every object by using `_ident()` method of `Object`.

5 Key Features

MICO filter object framework has certain key advantages over existing filter implementations. These are listed below:

1. The existing software can be evolved with minimal change in code and in most cases no change what-so-ever.
2. All the filtering properties, essential as well as extended, have been implemented.
3. Multiple methods can filter a single server method, though, at a time, at most only one is enabled.
4. Filter methods need not comply to any naming rules, thus giving more flexibility to the filter developer. The methods though need to have a compatible signature.

There are certain limitations of the current implementation. The current implementation is designed for intercepting only static invocations on server objects using the Basic Object Adapter (BOA). It does not intercept dynamic invocations made using the dynamic invocation interface (DII). Since the server-filter mappings are maintained at the server-side, every invocation has go through the server first, before the appropriate filter method is called. Logically, though, the filtering semantics are retained using this strategy, its leads to some finite overheads. There might be cases where all the filter methods corresponding to a server method are disabled. Even in such case, an invocation of that method is redirected to the filter, leading to unnecessary overheads.

6 Filter Object Development Process

The following sections depict the steps involved in building filter objects using this framework to evolve the existing system.

6.1 Building Filter IDL from Server IDL

Filter IDL can be obtained from the server IDL in two ways:

- By manually writing the code for the IDL complying a set of predefined relationship rules mentioned below.
- By using the fidlgen utility.

Relationship between server and filter interfaces:

- For every method in the server interface, there exists at least one *up-filter* method in the filter interface.
- For each *up-filter* method in the filter interface, all the arguments corresponding to those in the server method are inout.
- For every method in the server interface returning non-void value, there exists at least one *down-filter* method.

The fidlgen utility can be used to directly generate the filter IDLs.

6.2 Compiling the Filter IDL

Once filter IDL is obtained, it can be compiled using MICO IDL compiler (idl) to obtain the files defining the filter, the filter stub, and the filter skeleton classes. Note that the current filter framework only intercepts static invocations using the Basic Object Adapter (BOA) on server objects with shared activation policy. Hence idl –no-poa –boa options should be used during compilation.

6.3 Post-processing the Filter Header File

Post-processing of the filter header file can be done in two ways:

- By manually inheriting the filter class (declared in the filter header file) from `CORBA::Filter` instead of `CORBA::Object`.
- The above task can also be accomplished by using filtergen utility. Along with the filter header file, we also need to supply the filter class name to the utility. The syntax is as given below:

```
filtergen <filter header file> <filter class>
```

6.4 Implementing Filter Objects

Since filter objects are full-fledged CORBA objects, implementing a filter object is similar to implementing any other CORBA object. Each of the filter method is implemented keeping in mind the server method it is going to up-filter or down-filter.

```
/* Filter section with unique key filter */
[ filter ]
{
     [ FilterRepo ] /* contains filter IMR repoid */
     {
          /* Filter id and Tag */
          IDL:AccountFilter:1.0   foobar
     }
     [ Enable ]  /* contains filter method to be enabled. */
     {
          balanceUp
          balanceDown
     }
     [ Mappings ]  /* contains up & down method mappings. */
     {
          Up balance balanceUp
          Down balance balanceDown
     }
}
[ client ]  /* Client section with unique key 'client' */
{
     /* contains client and filter ids to be plugged. */
     [ Plug ]
     {
          [ ClientRepo ]  /* contains client IMR repoids */
          {
               IDL:Account:1.0 foobar
          }
          [ FilterRepo ]  /* contains filter IMR repoids */
          {
               IDL:AccountFilter:1.0   foobar
          }
     }
}
```

Fig. 4. Sample Configuration File

6.5 The filterconf Utility and BetaFiles

BetaFiles are configuration files for using filters. Figure 4 is a sample BetaFile. This file can contain multiple sections. Each top level section is identified by unique key. Top level sections may in turn contain subsections. There are two basic kind of top level sections. First type of section contains information needed by a filter object and second type of section contains information related filter-client (server) object. These two type of top level sections with keys *filter* & *client* sections respectively are described in this file.

Filter section This section can contain three subsections viz. *Enable*, *Disable* & *Mappings*. All three subsections need not be present in this section but it must at least contain one of the three subsections. Each subsection requires an *id* of the destination filter. This *id* can be either naming service identifier or implementation repository identifier (*repoid*). The filter id is written in another subsection either *FilterNS* OR *FilterRepo* based on type of id. Note that allowed

subsections are *Enable*, *Disable*, *Mappings*, *FilterNS* and *FilterRepo* with no restrictions on their order.

Client Section This section can contain two subsections viz. *Plug* & *Unplug*. Both subsections need not be present in this section but it must at least contain one of them. Each subsection contains list of filter-client (server) and filter ids. These ids can be naming service or implementation repository repoids. These ids must be specified in appropriate sub-subsections, i.e., *ClientNS*, *ClientRepo*, *FilterNS* and *FilterRepo*. For given client and filter ids, if they are equal in number, corresponding ids are used for plug or unplug calls. If client ids are greater, filter corresponding to last id will be plugged or unplugged from multiple servers. If filter ids are greater, multiple filters will be plugged or unplugged from last server.

6.6 Working with Catalysts

Catalysts are processes which manage the filter objects in the system. This involves plugging/unplugging filter objects, mapping filter methods to server methods, and enabling and disabling filter methods selectively. This is achieved using several interfaces provided by the framework.

Toggling with filter objects Filter objects can be plugged and unplugged from a server object dynamically. This is achieved using the `plug` and `unplug` methods of the `CORBA::ORB` class. Hence before we can plug or unplug filter objects from the system, we need to obtain a local ORB reference. The filter and server object references are passed as parameters.

Mapping method names Though there are strict rules for filter method signatures with respect to corresponding server method, no such rules exist regarding naming of filter methods. Hence the names filter methods need to be mapped to the corresponding server methods, they are supposed to filter. The `upfilter` and the `downfilter` methods of the `CORBA::Filter` class map the upfilter and downfilter method names to that of the server class. The mappings are established after a filter object is created. Unless the methods are appropriately mapped, filtering action cannot occur even if the filter has been plugged to the server. In such cases the call will continue normally as if no filter exists.

Toggling with filter methods A filter object can have more than one methods mapped to a single server object method. But at the same time, at the most, only one method can be enabled and acts as a filter method for the corresponding server method. The `enable` and `disable` methods of the `CORBA::Filter` class are provided for this purpose. All the filter methods corresponding to a server method being disabled is semantically equivalent to normal invocation of that server method.

7 Applications using Filters

Distributed object-oriented systems built using MICO can be evolved using transparent filter objects based on inter-class filter relationship. Applications of transparent filter objects include on-line pluggable caches [6] and filter configurations [4] such as loggers, replacers, balancers, routers, monitors etc. Various configurations resulting from the filter relationship along with other meta patterns can be applied to carry out system evolution. It is possible to inject filter objects or a network of filter objects into the system at runtime and satisfy certain kinds of evolutionary requirements without having to bring down an existing system.

8 Performance Evaluation

The performance implications of using the filtered delivery model are discussed in this section. The test setup included Pentium IV machines connected through a 100 Mbps LAN, running MICO version 2.3.4 with integrated filter object framework on Linux.

Table 2 indicates time required to make direct calls to local and remote servers in absence of plugged filters. These timings are used to calculate the overheads of the filter object framework.

Table 3 shows timings for beta messages (control messages) to local and remote servers and filters. Timings for *filtered* method invocation with one filter plugged are shown in Table 4. The table presents the time required to make a filtered call with *Up/Down* filtering enabled and disabled. Time measurements are shown for the following four configurations of clients and filters: Local Client/Local Filter, Local Client/Remote Filter, Remote Client/Local Filter and Remote Client/Remote Filter.

Local server (μs)	660
Remote server (μs)	850

Table 2. Direct call

Plug/Unplug	Local server (μs)	Remote server (μs)
	2150	2450
Mappings/Enable	Local filter (μs)	Remote filter (μs)
/Disable	450	530

Table 3. Client and filter beta messages

Up/Down		LC/LF	LC/RF	RC/LF	RC/RF
Enabled	Bounce	1090	1300	1160	1190
	Pass	1850	2050	1950	1880
Disabled		1450	1600	1550	1500

Table 4. Filtered call

By comparing direct calls (Table 2) and *passed* filtered calls with Up/Down filtering enabled (Table 4), it can be observed that later incurs approximately 2.5 times overheads over direct call. Similarly overheads of *bounced* filtered calls with Up/Down filtering enabled over direct calls are approximately 1.6 times. With Up/Down filtering disabled, overheads are approximately twice that of a direct call. Even with disabled Up/Down filtering, these high overheads can be attributed to filtered method call always consulting plugged filter for its method status, which is controllable at runtime.

Conclusions

A filtering framework was designed and implemented for MICO, an open-source CORBA implementation. The model supports all essential as well as extended filter properties. Filter objects are first class full fledged CORBA objects, which are dynamically pluggable. Filtering framework is supported through modifications to an implementation of MICO version 2.3.4 which is compliant with OMG-CORBA 2.2 standard. Tools have also been developed to support the development process.

References

1. M. Aksit, J. Bosch, L. Bergmans: Abstracting Object Interactions using Composition Filters. Proceedings of the ECOOP'93 Workshop on Object-based Distributed Programming, 1994.
2. Inprise Corp., Visibroker for JAVA Programmer's Guide, 1999.
3. IONA Technologies PLC., Orbix Programmer's Guide C++ Edition, 2000.
4. Rushikesh K. Joshi: Filter Configurations for Transparent Interactions in Distributed Object Systems. Journal of Object Oriented Programming. June 2001, pp. 12-17.
5. Rushikesh K. Joshi, Neeraj Agrawal, AspectJ Based Implementation of Dynamically Pluggable Filter Objects in a Distributed Environment, In Proceedings of the 2nd German Workshop on Aspect Oriented Software Development, University of Bonn, Feb. 2002.
6. Rushikesh K. Joshi, N. Vivekananda, D. Janaki Ram: Message Filters for Object-oriented Systems. Software Practice and Experience. June 1997, pp. 677-699.
7. Maureen R. Mascarenhas, Rushikesh K. Joshi: Filter Objects for JAVA, Technical Report, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, Jan. 2001.

8. P. Narasimhan, L. Moser, P. Melliar-Smith, Using Interceptors to Enhance CORBA, IEEE Computer, July 1999.
9. James Noble, Encapsulators in Self, in ECOOP 1996 Workshop on Prototype Base Object Oriented Programming.
10. Object Management Group (OMG): The Common Object Request Broker: Architecture and Specification 2.4.1, Nov 2000.
11. G. Srirami Reddy and Rushikesh K. Joshi: Filter Objects for Distributed Object Systems. Journal of Object Oriented Programming. Jan 2001, pp. 12-17.
12. Kay Romer, A. Puder and F. Pilhofer, MICO: An Open Source CORBA Implementation. http://www.mico.org.