SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(6), 677–699 (JUNE 1997)

Message Filters for Object-oriented Systems

RUSHIKESH K. JOSHI, N. VIVEKANANDA AND D. JANAKI RAM

Department of Computer Science and Engineering, Indian Institute of Technology, Madras-600036, India,
(email: {rushi, vivek, djram}@lotus.iitm.ernet.in)

SUMMARY

In the conventional object model, encapsulated objects interact by messages that result in method invocations on the destination object. A message is delivered directly at the destination object. As a result of the direct deliveries, the message control code performing intermediate message manipulations cannot be abstracted out separately from the message processing code in the destination object without sacrificing the transparency of the intermediate message control. We propose the filtered delivery model of message passing for object-oriented languages to provide the separation of message control from message processing in a transparent manner. An interclass relationship, called a *filter relationship*, is introduced. As a consequence, a filter object can intercept and manipulate messages sent to another object called its client via filter member functions. A filter member function in a filter object can intercept a particular member function invocation on its client object. The filtered delivery model supports both upward and downward filtering mechanisms, facilitating interception of an upward message and its return message value. Filter objects can be plugged or unplugged at runtime. Binding of filter member functions to corresponding member functions in the client is selective and dynamic. The filtered delivery model is developed for the C++ object-oriented language; its applications are described and implementation is discussed. **©1997 by John Wiley & Sons, Ltd.**

KEY WORDS: direct message delivery; filtered message delivery; filter object; filter relationship; object-oriented programming

INTRODUCTION

Objects form the basic building blocks of an object-oriented program. Objects interact by sending messages to each other. Messages are in turn mapped to invocations of member functions. When a sender object prepares the contents of a message and selects the destination object for that message, the message is said to have been generated. When a message is ready for the destination to be read, we say that the message has been delivered. In the traditional object model, *message delivery* is modeled as an activity that is directly triggered by *message generation*. We term this model of message passing as the *direct delivery* model. The direct delivery model is widely adopted in the existing object-oriented languages such as Smalltalk[1] and C++.[2]

A response to a message can be divided into two stages of message control and message processing. *Message control code* is the code that performs intermediate message manipulations before messages are delivered at the destination object. *Message processing code* is the actual code in the destination object that processes the messages to achieve the functionality desired by the caller. There can be many situations in which a finer control over messages is desired. For example, message contents might have to be checked against concerns such as validity and security. Similarly, an application may require a message preprocessing stage as in the case of an intermediate stage that implements a cache to improve the performance

CCC 0038–0644/97/060677–23 $17·50
©1997 by John Wiley & Sons, Ltd.
Received 17 July 1995
Revised 15 October 1996

of read requests sent to a server object. Separating message control from message processing can be very advantageous. The actual message processing becomes independent of message control, thereby providing the capability to modify the functionality of a message processing object in a transparent way by means of the separate message control stage. The separation makes it possible to develop message control policies in a modular way.

In the direct delivery model, the message control code is coupled with the message processing code. As a result, the message control code cannot be abstracted out without breaking its transparency. An application may require its message control policy to be changed dynamically. In such a case, the destination object needs to be modified owing to the coupling between message control and message processing. For example, for a dictionary object, a message control policy may cache items in order to lower the search time. A new policy for routing the search requests to another dictionary object may have to be added when a new dictionary server becomes available. In this case, the dictionary object needs to be modified to incorporate the new message control policy.

On the other hand, a forced abstraction of the message control code destroys its transparency. For example, a security object might be separated out from the message processing object. However, the calling semantics of direct deliveries require that the message be explicitly sent to the security object, which subsequently forwards it to the message processing object. Thus, the transparency of message control is destroyed.

We propose a new model for inter-object message communication called *filtered delivery* that achieves separation between message processing and message control in a transparent way. In this model, as opposed to direct deliveries, messages sent to a destination object can be intercepted by special objects called *filter objects*. While filter objects intercept messages, the calling semantics at the source object do not change. We incorporate the filtered delivery message passing model in the sequential object-oriented programming language C++.[2] A new interclass relationship called a *filter relationship* is introduced. Using this relationship, a filter object can be empowered to intercept messages sent to an ordinary object. Both upward calls and their return values can be intercepted.

OVERVIEW OF THE FILTERED DELIVERY MODEL

Figure 1 shows the conventional direct delivery message passing model. An object `User` sends a message `insert()` to a `ResourceQ` object. Object `User` is the source object for message `insert()`. The message is delivered directly at object `ResourceQ`, which is the destination of the message as chosen by the source object itself. As a result of message delivery, the corresponding method is invoked at the destination object.

Since messages are directly delivered, any intermediate message control has to be accommodated within the destination object. We cannot abstract out the message control part without disturbing the actual destination and without sacrificing its transparency. For example, consider the case of inserting an intermediate object in Figure 1 to route the user requests to another resource. In this case, the `User` object has to be modified to send requests to the new intermediate object. Another problem is that, when the intermediate object needs to be replaced or removed, the code in source object needs to be modified. For example, an addition of a second intermediate object requires the source object to refer to the new object instead of the old one. If the intermediate objects are removed, the message should go directly to the destination object. We propose the filtered delivery model that provides a modular way to develop objects that act as message filters for their client objects. Filter objects act transparently. Removal, addition or replacement of filter objects do not require any modification of code,

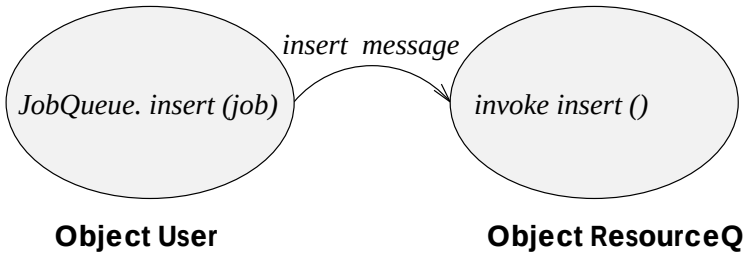


Figure 1. The direct delivery model

either at the source object or at the destination object.

Filters as message manipulators are used in various applications ranging from distributed algorithms[3] to multi-media collaborative applications.[4] However, the existing object oriented languages do not provide a formal support for designing filter objects for developing filters as modular and transparent entities. We have earlier proposed a primitive language construct for filtering called *capture specification* for ShadowObjects,[5] a model for control replication in distributed systems. Shadow objects are replicas of an object, which provide the same set of services as that of the original object. The replicas can be hidden behind the original object or exposed to the network. When the replicas are hidden, the capture specification is employed to intercept the service requests sent to the original object and schedule them on-the-fly on one of the replicas. The drawbacks of the capture specification are that it is closely coupled with a class, it is statically declared, and it cannot be changed or enhanced during runtime.

In the composition filter model,[6] input and output filters are specified within an interface of a class. Filters belong to specialized filter classes depending on their usage. *Dispatch* filters are used to conditionally accept messages. *Meta* filters can delegate messages to *abstract communication types*. In contrast to the composition filter model, we provide the separation between message processing and message control by means of dynamically pluggable filter objects that are separate entities from the destination of messages. We achieve this by a special interclass relationship called a *filter relationship*.

Our filter model has similarities with the filter mechanism provided in the Orbix Corba product.[7] Orbix allows programmers to supply filter code for clients and servers mainly for packaging requirements such as authentication, debugging, performance statistics, auditing and encryption. The main purpose of the filter mechanism is to keep the Orbix implementation flexible and lightweight. Our filter model is provided at the language level by introducing a *filter relationship* between classes. The main features of our model include dynamic creation of filter relationship between objects, per message and selective filtering and dynamic changing of filtering policies.

Some design patterns[8] such as decorator and proxy provide some functionalities of the filtered delivery model. However, the filters introduced in this paper are at the programming language level, and the main intent is to *glue* client object and server object using a filter object in a transparent way. Design patterns are at the level of structuring object-oriented systems. It is possible to construct specific design patterns based on the the filter object model, especially for gluing objects together.

Now we take a closer look at the filtered delivery model. Figure 2 shows the conceptual view of the filtered delivery model. A message filter functions as a message manipulator for its

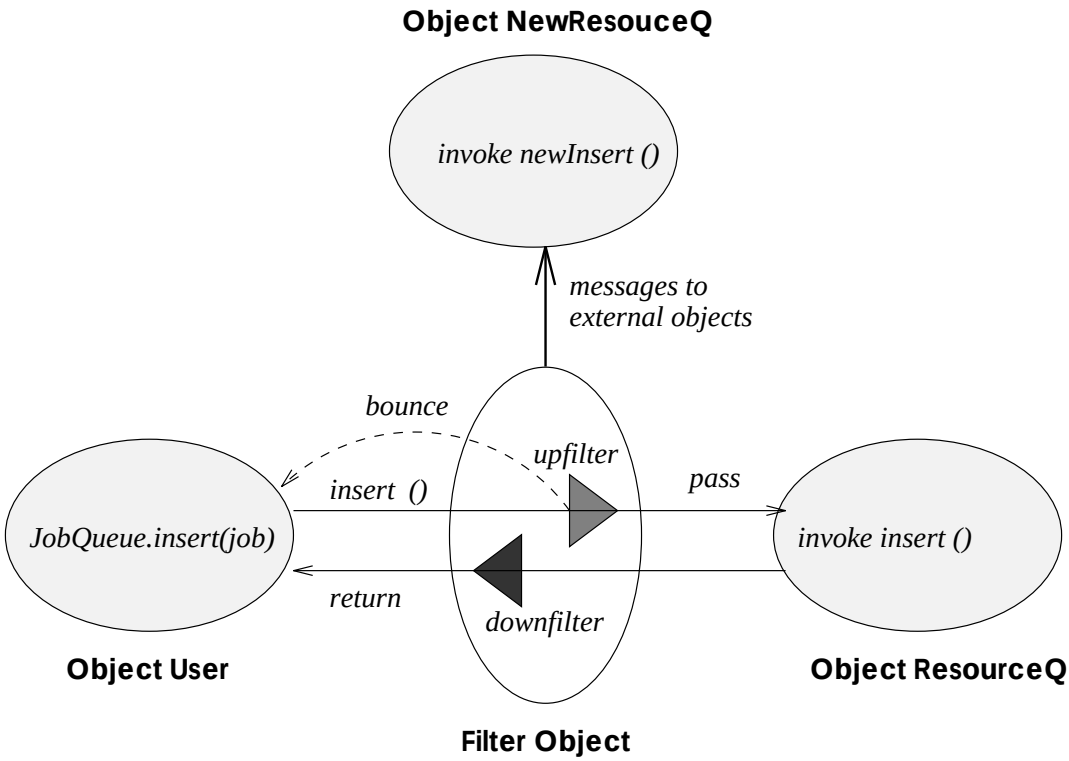


Figure 2. The filtered delivery model

client object. In the figure, `ResourceQ` is a *filter-client* object for the filter object. Messages sent to object `ResourceQ` pass through the filter object. `User` calls the member functions of `ResourceQ` directly. The filter object intercepts these messages when they are on their way.

The filter object can take the following actions upon interception of a message:

(a) *Interception of upward messages:* upward messages are messages from a source object to a destination object that has a filter. The filtering function that filters an upward message is called an *upfilter*.
(b) *Manipulation of messages:* an upfilter may change the arguments of a message and process an arbitrary code.
(c) *Bounce:* an upfilter returns a value to the source of the message on behalf of the destination.
(d) *Pass:* the upfilter passes the message on to the filter-client after a possible manipulation of the message contents.
(e) *Intermediate invocations on other objects:* a filter object may send requests to other objects as part of its control code.
(f) *Interception of downward messages:* downward messages are the return values from the destination. A downward message can also be filtered. The function that filters a downward message is called a *downfilter*.

The above actions and the various paths that a message can take in presence of a filter object are shown in the figure. Various properties of filter objects can be described from the point

of view of their functionality and usage. We divide these properties into two categories of essential and extended properties. Essential properties concentrate on the basic functionality of filter objects. They describe a bare minimum filter object model. The extended properties explore their flexibility from the point of view of their usage.

The essential properties of filter objects

We define the following properties as essential properties for filter objects:

(a) *Support for basic filtering actions:* filter objects are specified to intercept messages arriving at an ordinary object called a *filter-client*. Filters may manipulate the messages. They eventually *pass* or *bounce* them. *Bounce* is a return performed by the filter. Whereas, *pass* specifies forwarding of the message to its client. This property provides elegant language constructs to enable the design of objects that carry out tasks related to message control such as range checking, security checking, data conversions, message preprocessing and message routing.

(b) *Modularity:* specification of a filter object is separate from the specification of its client object. Neither a filter object breaks the encapsulation of its client, nor the client breaks the encapsulation of its filter.

(c) *Transparency:* sender may not know the existence of a filter object. Hence, direct delivery call semantics are preserved for the source of a message. Since the sender does not know the existence of a filter, it directly sends messages to the destination. No code changes in the source object are required when a filter object is added, removed or replaced. Filters can thus be used to act on-the-fly.

(d) *Selective filtering:* filters can intercept messages selectively. Some of the methods may remain unfiltered, whereas some may be filtered. This property enables a filter object to implement independent message control codes for multiple messages.

The extended properties of filter objects

(a) *Group filtering:* group filtering allows multiple filter-clients to be served by a single filter. This property defines an obvious extension to the power of a filter object. A filter object may intercept messages sent to a number of its client objects that are instances of the same client class.

(b) *Dynamic filtering:* filters can be changed for a client over its lifetime. This property specifies the *dynamic binding* capability of filters. Binding is done at two levels. At the first level, the filter objects may be removed and replaced. At the second level, individual member functions within a filter object that filter their corresponding counterparts in the client object may be changed at runtime.

(c) *Layered filtering:* this property specifies that filters can be nested. With this property, *multilevel filters* can be designed by specifying filters to filters. Multilevel filters can be used for designing multilevel message processing. For example, one filter may take care of security whereas another may be attached to function as a router to redirect messages to other servers.

As can be seen from the above properties, the basic filter object model provides support for *one to one* filter relationship between a filter object and a filter-client object. The the extended model provides support for *one to many* (group filtering), and *many to one* (layered filters) filter

relationships. In the following section, we describe the basic filter object model in detail. The basic model consists of various mechanisms that provide support for the essential properties discussed above. The support for the extended properties is discussed in the subsequent section. The base language used is C++. We have emphasized examples rather than syntactic specifications in our description.

BASIC FILTER OBJECT MODEL

The basic filter object model covers the essential properties of filter objects. Filter objects are specified separately from their client objects by a new interclass relationship called a *filter relationship*. A filter relationship is different from other relationships such as inheritance, aggregation, association, using, instantiation and metaclass relationships (see Booch[9] for a detailed description of these). As a result of a filter relationship between two classes, their instances, which are objects, also acquire the filter relationship. We define the filter relationship as:

> *the ability given by one object (filter-client object) to another object (filter object) to intercept, manipulate and forward or bounce the messages sent to it.*

A filter relationship is first established at class level. An instance of a filter class that is in a filter relationship with a filter-client class is given the ability to filter messages sent to an instance of the filter-client class. Subsequently during runtime, the instances may be plugged together to act in the filter relationship.

Specifying a filter relationship

In the following example, a filter relationship is established between a client class `ResourceQ` and a filter class `FilterQ`:

```
class ResourceQ {
      ...
};
class FilterQ : filter ResourceQ {
      ...
};
```

The relationship enables an instance of the filter class `FilterQ` to intercept messages sent to an instance of its client class `ResourceQ`. Two special operators, *plug* and *unplug* are used to specify the filter relationship between the instances of these classes.

The operators plug and unplug

Specification of a filter relationship at class level does not automatically start the relationship between their instances. The *plug* operator is used to bind two objects in filter relationship. For example, the following code binds filter object `cache` to its client `dictionary`.

```
main () {
Dictionary *dictionary = new (Dictionary);
Cache *cache = new (Cache);
```

```
      ...
      plug dictionary cache;
      ...
}
```

Similarly, the `unplug` operation is used to break the filter relationship between objects. The operation

```
unplug dictionary;
```

breaks the filter relationship between `cache` and `dictionary`. After an unplug operation is performed on a filter-client, it is on its own, and receives further messages as direct deliveries.

A filter object can selectively intercept messages sent to its client object. For example, it may intercept only one single member function among many others defined in the public interface of its client. Selective filtering is possible with the help of *beta messages*, which are described subsequently.

Organization of a filter class

A filter class defines an interface called a *filter interface* apart from the usual *private* and *public* interfaces. A filter interface defines filter member functions that are invoked automatically when their corresponding member functions in the client object are intercepted by the filter object. Thus, the members of the filter interface are invoked only by the system that executes the program. Filter member functions defined in the filter interface are not accessible as private or public members.

The filter interface

The filter interface is split into *upfilter* and *downfilter* interfaces. The member functions of the *upfilter* interface can intercept the upward messages going towards the filter-client, whereas the *downfilter* member functions can intercept the return values from the client. Both interfaces are independent, and the presence of one is not mandatory for the presence of the other. Each interface specifies a mapping from filter member functions in a filter class to member functions in a client class. As an example, the following code specifies an upward filter member function `searchCache()` and a downfilter member function `replenishCache()`, in filter class `Cache` for the member function `searchWord()` defined in client class `Dictionary`.

```
class Dictionary {
...
public:
      Meaning searchWord (Word); // returns meaning of a word
};
class Cache : filter Dictionary {
...
upfilter:
      Meaning searchCache (Word) filters searchWord;
      // returns meaning if hit
downfilter:
      Meaning replenishCache (Meaning) filters searchWord;
```

```
      // updates cache if miss
};
```

An assignment of a filter member function to a member function in its client class is specified using the keyword *filters*. An assignment gives the capability to a particular filter member in the filter class to intercept a particular member function in its client class. An assignment does not automatically mark the commencement of the filtering action, but the filter member functions have to be explicitly enabled to actually commence filtering. In the above example, if a message `searchWord()` for an instance of class `Dictionary` is generated, it can be made to go through `searchCache()` in an instance of class `Cache`. The upfilter member returns the meaning of the word on itself, if it is found in the cache. The message is not passed on to dictionary object in that case.

The downfilter members can also be specified in a similar way. The only difference in this case is that a downfilter member function takes one argument and returns one argument. A downfilter member function receives the return value from the client, and also has to return a value of the same type to the source of the message. Hence, the type of the input and the return arguments for a downfilter is the same as the return type of the corresponding client member function. In the above example, the filter `replenishCache()` can intercept the return values of calls to `searchWord()` in its client. When a meaning is on its way, the cache may be replenished by the downfilter member function.

The beta messages enable and disable

We introduce beta messages that function as messages to filter member functions. Ordinary messages such as class messages or instance messages are modeled as invocations of member functions. On the other hand, beta messages are modeled as directives to member functions themselves. Beta messages can be sent to a filter member only from within the corresponding filter object. Two standard beta messages, *enable* and *disable*, are provided to mark the commencement and termination of a filtering action. These beta messages can be applied selectively to individual upfilter and downfilter member functions. For example, if a beta message `searchCache.enable` is sent to a filter member `searchCache()`, it starts intercepting its peer member function `searchWord()` in its client class. The restriction of sending beta messages from within a filter object protects the encapsulated behavior of the filter object. Any member function (*public* or *private*) can send a beta message to a filter member function. Two additional beta messages are provided to know the current state of a filter member function. They are explained in the section on the dynamic binding of filters.

The following code demonstrates the use of beta messages. A public member invocation `start_filtering()` enables the upfilter `searchCache()` and the downfilter `replenishCache()`, while `end_filtering()` disables them. After disabling, the filter relationship between the objects terminates and messages do not go through the cache filter.

```
class Dictionary {
…
public:
      Meaning searchWord (Word);
};
class Cache : filter Dictionary {
…
```

```
upfilter:
      Meaning searchCache (Word) filters searchWord;
downfilter:
      Meaning replenishCache (Meaning) filters searchWord;
public:
      start_filtering () { searchCache.enable; replenishCache.enable; };
      end_filtering () { searchCache.disable; replenishCache.disable; };
};
main () {
Dictionary *dictionary = new (Dictionary);
Cache *cache = new (Cache);
      …
      plug dictionary cache;
      cache->start_filtering() ;
      ….
      cache->end_filtering() ;
      ….
      unplug dictionary;
}
```

Actions of an upfilter member function

The upfilter member function are specified like any other member function of a class except that they can perform two additional actions, *pass* and *bounce*, which are explained below. The prototype of an upfilter member function should match with its corresponding member function in the client class. For example, if an upfilter member in a filter class *F* has to be specified for a client member function `myReturnType C::func2(myType)` in a client class *C*, its specification can be given as `myReturnType F::filter2(myType)`. This requirement reflects the ability of a filter member to receive the message sent to its client *as it is*, and also return a value on behalf of its client. Moreover, in the presence of an overloaded client member function, this type information within the declaration of a filter member gives its corresponding overloaded meaning.

Once an upfilter is enabled, the corresponding upward messages are intercepted and the upfilter member function is invoked. All arguments in the message to the client object arrive at the upfilter as corresponding arguments. The contents of the message can be manipulated inside the filter member function. The filter member can also invoke member functions on the same or other objects just as any other ordinary member function can. It takes one of the following two actions upon completion of its message manipulation activity:

(a) *Pass the message:* the message can be passed on to the client object after the desired filter action is performed. The pass action is specified by a `pass` statement.
(b) *Bounce the message:* the filter can return a value on behalf of its client by a special return statement `bounce()`. The type of the bounced value is the same as that of the return value specified by its client. The receiver object is kept unaware of the interception unless this information is explicitly encoded in the return value.

The following code is an example of a filter member function that bounces a return value of -1 if the argument `x` is negative. The message is passed on as it is for all non-negative values of `x` less than 100. For higher values of `x`, it is held at the threshold of 100 and then passed on.

```
int aFilter :: filter1 (int x) {
            if (x < 0) bounce (-1);
            if (x > 100) x = 100;
            pass;
};
```

Actions of a downfilter member function

A downfilter member function intercepts the return messages from a client object. To intercept a return message, it is not required that the upward message be intercepted. A downward filter takes exactly one argument, which is the return message to be intercepted. Within the filter, this return value can be manipulated. A *bounce* from the downward filter returns a value to the calling object. In the following example, the downfilter code unconditionally adds a value of 2 to every return message and then returns it to the caller.

```
aFilter : filter  C {
...
downfilter:
                  int down1 (int) filters func1;
};
int aFilter :: down1 (int x) {
                  bounce (x+2);
};
```

EXTENDED FILTER OBJECT MODEL

The extended filter object model supports the extended properties of filter objects. The *group filtering* construct facilitates multiple filter-clients to be served by one filter object. *Dynamic bindings* of filters are possible at two levels. A filter object may be changed and individual filter member function may also be changed for a filter-client member function over its lifetime. Filters to filters provide support for designing *multilevel filters*.

Support for group filtering

It is possible to plug multiple client objects within a single filter object. Group filters thus define a one to many relationship. In such a case, a beta message on a filter member acts as a group beta message. Figure 3 shows a group configuration for upward messages. Similar configuration is possible for downward messages also.

In the figure, objects `file1`, `file2` and `file3` are instances of class `File` and are clients of the filter object `fileFilter`. A beta message `enable` enables the filtering action by a particular filter member function such as `readFilter()` or `writeFilter()` for the entire group of clients at a time. Similarly the `disable` beta message disables filtering for all clients. The `plug` and `unplug` operations can be performed on individual client objects. For example, object `file1` can be selectively unplugged leaving the other two objects plugged. Any number of clients can be plugged or unplugged to a filter at any time, but all of them must belong to a class that is in filter relationship with the class that defines the object `fileFilter`. Group filters can also be used for shared administration of messages sent to a group of objects. It is

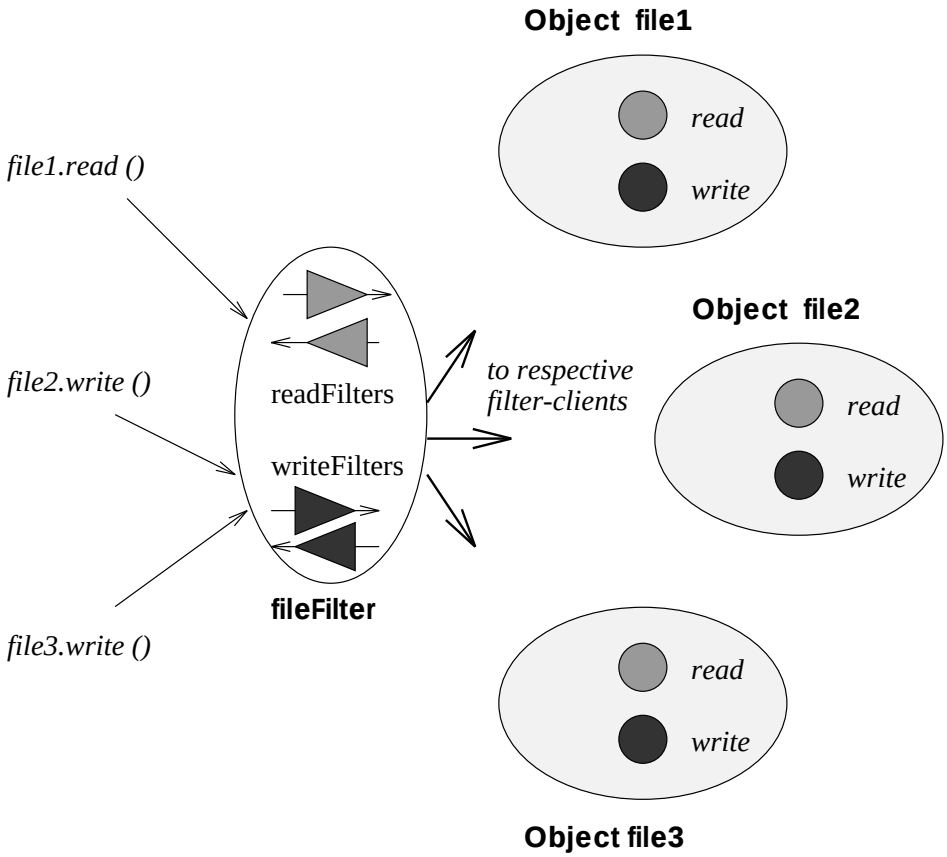


Figure 3. Group filtering

possible to design group multi-cast filters to address the specific problems of multi-casting[10] such as late-comers joining a multi-cast group.

Dynamic binding of filters

At filter object level

A filter-client may change its filter object over its life time. A filter-client may be in filter relationship with many filter classes, but at the level of instantiation, a client object can be in filter relationship with only one filter object at a time. The following example shows the dynamic binding of filter objects for a client object, which is a design object. A design object such as a machine part undergoes different levels of design process, and at each level, the design process has to satisfy different constraints. Readers are referred to the section on applications of filter objects for a detailed treatment on this application.

```
class DesignObject { ...};
class Constraint-A : filter DesignObject { ... };
class Constraint-B : filter DesignObject { ... };
main ()
{
DesignObject *designObject;
Constraint-A *constraint-A; // a filter for designObject
Constraint-B *constraint-B; // another filter for designObject
      …
      plug designObject constraint-A;
      … // design level 1
      unplug designObject;
      plug designObject constraint-B;
      … // design level 2
      unplug designObject;
}
```

At filter member function level

It is possible to specify multiple upfilter or downfilter member functions for one client member function at compile time, and bind at runtime only one of them as a filter for the corresponding client member function. In the following example, a filter object implements two caching policies `cachePolicy1()` and `cachePolicy2()`. Either of the policies can be used as an acting upfilter for member `read()` in the client object. Two additional beta messages, `is_enable` and `is_disable`, are provided for testing purposes. They return a boolean result as per the status of the corresponding filter member. A public member function `enable1()` can be invoked to initialize the upfilter to `cachePolicy1()`. Subsequently, a public member function `switch_filter()` can be invoked to switch the upfilter member function from `cachePolicy1()` to `cachePolicy2()`, or vice versa.

```
class Cache : filter Dictionary {
...
upfilter:
      cachePolicy1(..) filters read {...};
      cachePolicy2(..) filters read {...};
public:
      enable1 { cachePolicy1.enable; }
      switch_filter ();
};
Cache :: switch_filter () {
      if ( cachePolicy1. is_enable ) { cachePolicy1.disable;
      cachePolicy2.enable}
         else
             { cachePolicy2.disable; cachePolicy1.enable; }
}
```

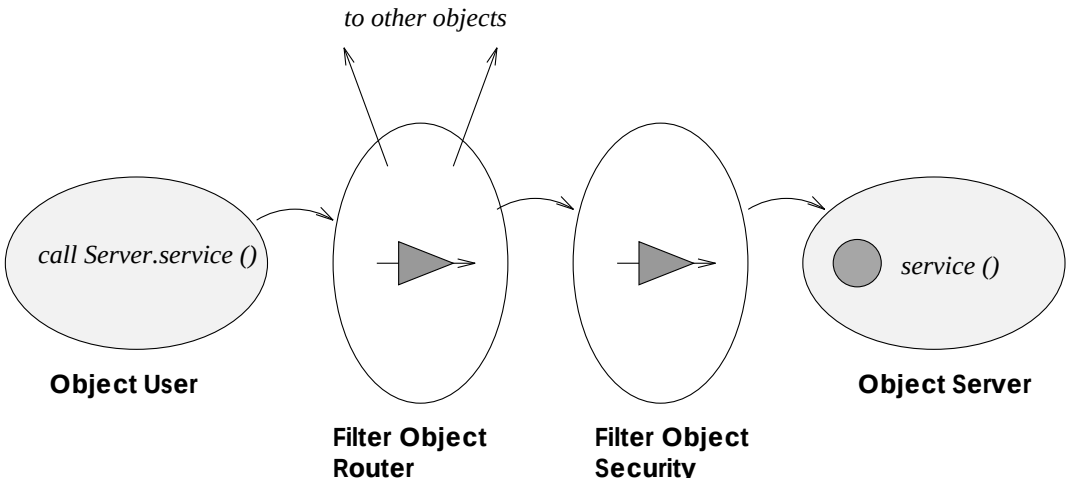


Figure 4. The Multilevel filters

Filters to filters

It is possible to design nested multilevel filters by specifying them as filters to filters. Multilevel filters can be useful in protocol software development in networking. In the following example corresponding to Figure 4, a `Security` filter filters an object `Server`. A `Router` filter is specified to intercept the `Security` filter. Thus, `Security` is a filter-client for `Router`. Filter `Router` may intercept the public and filter members of its client `Security`. Filtering the filter member functions leads to multilevel filtering.

```
class Server {...};
class Security: filter Server {...};
class Router: filter Security {...};
```

Figure 4 pictorially captures the multilevel filter relationship. The `service` requests sent to the `Server` object may be captured by the `Router` and routed to other objects instead. If a request is passed on, it goes through the `Security` filter before it reaches the object `Server`.

APPLICATIONS OF FILTER OBJECTS

In this section, we discuss some applications of filter objects in detail, which were used as examples in developing the filter object model in the earlier sections. The first application uses filter objects for implementing the constraint meta-object model for collaborative applications. The second application implements access control mechanism for replicated objects using filters. Requests to an object can be rerouted to one of its replicas by capturing the calls on-the-fly. Finally, a database application implements an *on-the-fly* cache on a host using a filter object. The filter object can change the cache policy by changing the filter members. These applications give a flavor of the applicability of filter objects.

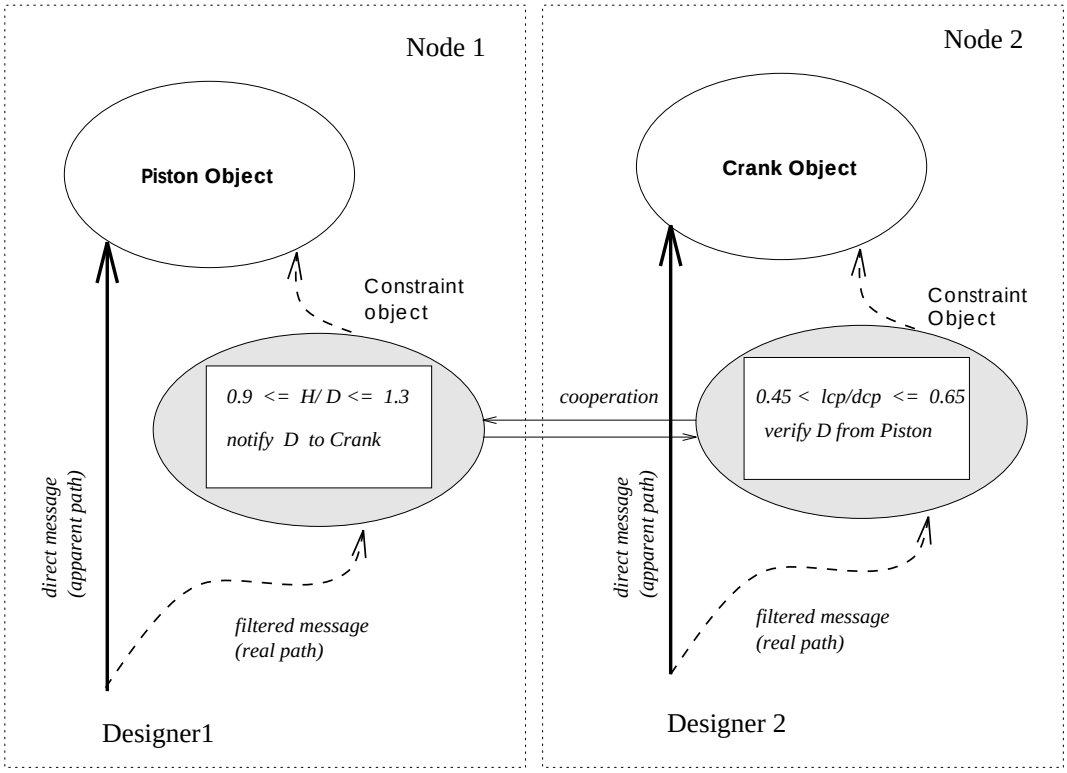


Figure 5. Filter objects in the constraint meta-object model

Filters in the constraint meta-object model, a model for distributed collaborative designing

This application demonstrates the use of filter objects for maintaining the consistency of design objects. It also uses the dynamic binding feature of filter objects.

The Constraint Meta-Object model[11] has been proposed to develop collaborative design applications. The constraint meta-object model for a mechanical design application is shown in Figure 5. In this model, designers collaboratively design through a design space consisting of design objects. The design space captures collaboration via interdependencies between design objects. Each design object has several constraints that have to be satisfied when changes are made to the design object. Due to the interdependent nature of design objects, other dependent design objects need to be notified when a change is made to a design object. Constraint meta-objects capture the constraints on the design parameters of design object. They transparently intercept the design operations on design objects, validate these operations against the constraints, and also perform the necessary notify operations. Constraint meta-objects are implemented as filter objects since the filter mechanism allows transparent interception of messages.

An upfilter member is generated for a constraint meta-object to make a temporary copy of its filter-client, the design object. The temporary copy is used in recovering the design object back when a completed design operation does not satisfy the constraints. A downfilter

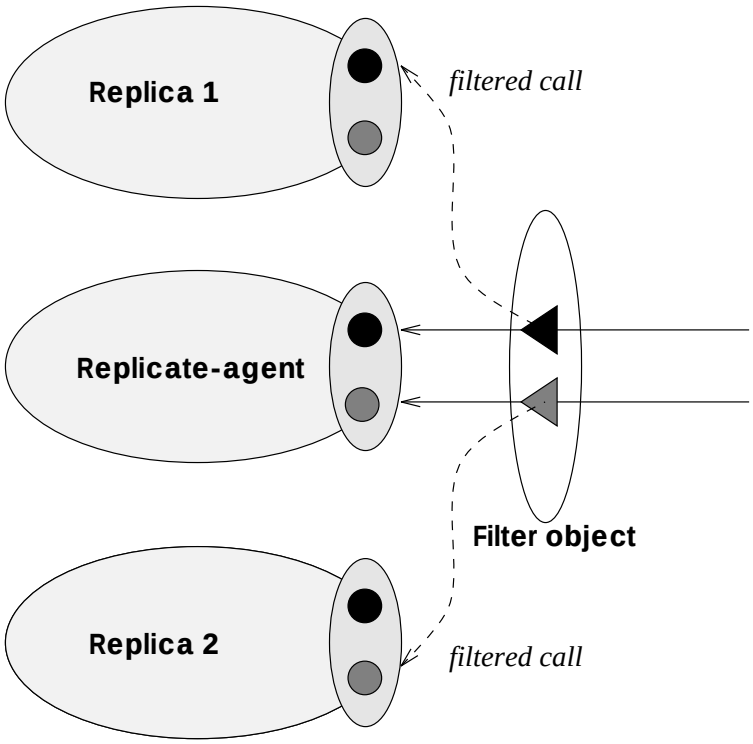


Figure 6. The ShadowObjects model

performs the validation of the object against the constraints and restores the original state of the object if necessary.

Dynamic plugging and unplugging of the filter to an object helps in capturing a dynamically changing constraint environment such as during the migration of a design object from designer's environment to manufacturer's environment.

Filters in the ShadowObjects model

The ShadowObjects model[5] demonstrates the applicability of filter objects in routing captured messages to other desired objects. The model is developed for control replication in distributed systems. Figure 6 depicts the use of filter objects for the ShadowObjects model. In this model, an object can replicate itself into a number of replicas at any point of time. The replicas are hidden behind the original object called the *replicate-agent*. Requests to a replicate-agent can be routed to one of the replicas. Routing is performed by capturing the calls to the replicate-agent by a router modeled as a filter object.

The calls from outside are still made to the original object providing the replicate-agent the ability to encapsulate its replicas. The ShadowObjects model also provides a mechanism to selectively *expose* the hidden replicas, in which case, the outside objects can contact the exposed replicas directly. A filter may be disabled when a replica is exposed and enabled when it is hidden.

```
class DB { // database front end
...
public:
      void connect_to_back_end ();
      record *read (int rid);
      void write (record *rec, int rid);
};
class Cache : filter DB { // filters read and write requests
private: ...
      cache_object CO;
      float hit_ratio;
upfilter: // access through cache
      record *directCache (int rid) filters read;
      void invalidate_entry(record *, int rid) filters write;
downfilter: // implements replacement policies
      record *replenishLIFO (record *) filters read;
      record *replenishFIFO (record *) filters read;
public: ...
      Cache () {hit_ratio = 0; };
      switch_policy ();
};
Cache :: switch_policy () {
      if ( replenishLIFO.is_enable ) {
         replenishLIFO.disable;
         replenishFIFO.enable;
      } else if (replenishFIFO.is_enable ) {
               replenishFIFO.disable;
               replenishLIFO.enable;
      } else replenishFIFO.enable; // default action
};
```

Figure 7. Filters for on-the-fly caching

Filters for on-the-fly caching

This application demonstrates the capability of filter objects to add functionalities to existing messages on-the-fly. It also demonstrate the applicability of switching between various filter member functions. A filter object functions as a cache with multiple cache policies for a database front-end server. Caching is performed on-the-fly. The filter object captures read requests and caches their answers transparently when required. If a read request can be replied by detecting a cache hit, the filter answers the request by itself.

Figure 7 shows a filter object `Cache`, which implements different cache replacement policies. The filter object can change the active policy on receipt of a message `switch_policy`. Class DB is the front-end sever class to the actual data base. The filter object filters `read` and `write` queries. If a write query arrives, it invalidates the cache entry if one exists for that record. Read queries are intercepted by an upfilter member, and if an entry exists in the cache, it is served by the filter itself without having to go through the data base server. In the case of

a cache miss, the read request is forwarded to the data base front and when the result is on its way home, it is intercepted by a downfilter member. The downfilter member replenishes the cache by invoking the cache replacement policy.

The filter object mechanism models the on-line cache conveniently without having to modify the front end object. Cache is an added functionality to the front end. A filter object conveniently captures an added functionality. Experimentation with varying cache policies would require changes only to the filter object keeping the front end intact. Another advantage of modeling the cache as a filter object is that the cache object can easily accommodate multiple front ends by means of the group filtering mechanism.

The above examples show how filters can be used to control messages. Filter objects are developed independently of their clients in a modular way. The message processing code remains with filter-clients whereas the message control is abstracted by the filter objects. Various functionalities can be programmed as a part of the message control code. A filter can dynamically tune its characteristics.

Experience with filter objects

We have been able to extensively use the filter object model in developing different applications such as those discussed above. We generally found that the filter object model provides ease of programming especially in the context of distributed program development. Our experience also showed that the most attractive features of the model are its ability to dynamically plug filter objects, group filtering and layered filtering. Though we have not been able to apply layered filtering to practical problems, we foresee their use in the context of protocol software engineering. We have also observed that the filter overhead is practically negligible in these applications.

IMPLEMENTATION OF FILTER OBJECTS

We describe a user level implementation scheme for filter objects in C++. Due to the flexibility of C++, it was possible for us to convert a C++ code that uses filters to a plain C++ code. We use various features of C++ such as function pointers, inheritance, polymorphism, parameter passing by reference, and the ability to specify one to one association using buried pointers.[12] The code conversion is achieved with the following steps:

1. Identify all classes that are in filter relationship with a client class. Consider the following relationships as shown in Figure 8.

   ```
   class JobMix1 : filter JobServer ...;
   class JobMix2 : filter JobServer ...;
   ```

   In this example, `JobMix1` and `JobMix2` are the filter classes for client class `JobServer`.
2. Form a hierarchy of these filter classes rooted at a new special filter class. For the above example, a new class `JobServerFilter` is defined, and `JobMix1` and `JobMix2` are made subclasses of class `JobServerFilter`. Thus, we translate the above relationships in the following hierarchy, which is pictorially shown in Figure 8 by dotted lines.

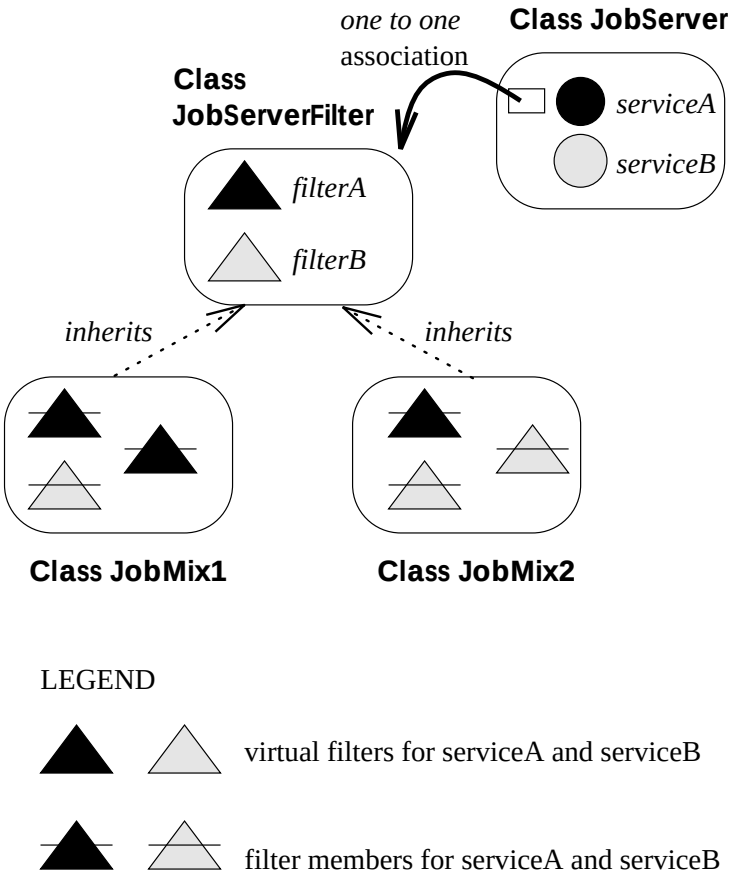


Figure 8. Hierarchy of filters

The new hierarchy

```
class JobServerFilter {...};
class JobMix1 : public JobServerFilter {...};
class JobMix2 : public JobServerFilter {...};
```

We know that, at any time, either an instance of `JobMix1` or an instance of `JobMix2` can be plugged to the client object. `JobMix1` and `JobMix2` may implement different codes for the actual filtering actions, but both of them filter only the member functions defined in the client. Hence, this behavior is earned by treating an instance of `JobMix1` or `JobMix2` as an instance of their super class `JobServerFilter` by means of `inclusion polymorphism`.[13]

3. All the filter members are defined as virtual functions in class `JobServerFilter`. The number of virtual functions in class `JobServerFilter` is equal to the total number of member functions in `JobServer` that are filtered by `JobMix1` or `JobMix2`. The subclasses `JobMix1` and `JobMix2` define the actual filter member functions. In the

figure, class `JobMix1` provides two member functions for `serviceA`. Class `JobMix2` has two filter member functions for `serviceB`. Whenever there are multiple filter members for one client member, one of them is bound at runtime. The binding is achieved by making the appropriate function call through a function pointer from a function that overrides the virtual function defined in the superclass `JobServerFilter`. The `enable` and `disable` beta messages assign and remove these function pointers.

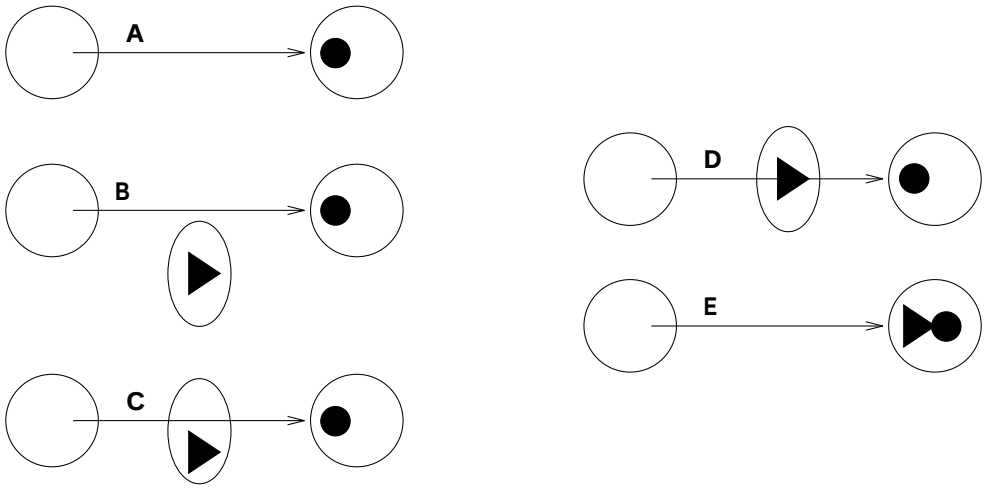
4. The client class `JobServer` declares a pointer to an instance of class `JobServerFilter`. This pointer specifies a `one to one association` relationship with an instance of class `JobServerFilter` and an instance of class `JobServer`. The relationship is shown by a thick line in the figure. Whenever an instance of class `JobServer` is plugged to a filter object that is an instance of `JobMix1` or `JobMix2`, this pointer is made to point to the corresponding instance.
5. Now, every member function in `JobServer` that is upward filtered, is made to go through the corresponding upfilter member function defined as virtual in `JobServerFilter`. All parameters except pointers are passed by reference to the filter member. This scheme allows the filter member function to manipulate the message arguments. A *pass* statement in the filter member brings back the control to the member function of `JobServer`. On the other hand, a *bounce* statement returns the control to the caller object.
   Similarly, in the case of an enabled downfilter, the return statement in the client member function relinquishes control to the filter by sending the return value as a parameter to it. The downfilter subsequently returns a value to the caller on behalf of the client object.
6. The same steps are followed for filters to filters. In such cases, a filter that intercepts method invocations of another filter considers it as its filter-client.
7. Provisions are made for checking possible runtime errors regarding the usage of filter objects. Some of the possible runtime error messages are the following:

   Cannot enable two filter members at a time for one client member
   Cannot enable a filter member of an unplugged filter object
   Cannot plug two filter objects for one client object

Performance test

The performance test was carried out on a Sun Sparc machine to observe various overheads involved in the filter object model. The test setup was configured as shown in Figure 9. Five different kinds of messages are shown in the figure. Message A is a direct message from a source object to a destination object that has no filter specified even at the class definition level. Message B is a message to an object that has a filter class but does not have the filter object instance plugged. Message C goes through a filter object that is only plugged but not yet enabled. Message D is the filtered message that goes through a filter member function. In the configuration for message E, the action that a filter member takes is embedded as a function call into the destination object itself. Table I shows the absolute timings for these five configurations. The number of arguments is varied from none to 4, where each argument is an integer taking 4 bytes.

Various overheads are computed from these absolute figures and are presented in Table II. The comparison between the timings for messages A and B gives the overhead of declaring a filter class. In Table II, this overhead is shown as *filter declaration overhead*. All the figures in Table II are quoted in terms of null body, null argument direct member function call, measured to 0.527 micro seconds as shown in Table I. For example, the filter declaration overhead is



A: Direct call to an object which has no filter
B: Call to an object which has an unplugged filter
C: Call to an object which has a filter plugged, but not enabled
D: Call to an object which has a filter enabled
E: Call to an object which has no filter, but the filtering action is embedded

Figure 9. Performance test setup

computed as $(B - A)/0.527$. The filter declaration overhead, a constant, involves checking of a condition whether the destination object has a filter object specified. As can be observed from the table, it does not depend on the number of arguments, and on average, remains within 0.2 times the direct null member function call.

Message C, when compared with message A, gives the *plug overhead*. This overhead checks whether the corresponding filter member is enabled. The plug overhead is shown in the table. It does not depend on the number of arguments, and is also a constant. On an average, it remains less than one direct null member function call.

When message D is compared with message E, we get the total overheads of a filter object that performs its intermediate action and passes the arguments to the the destination. The overhead is given as *filter overhead* in the table. As can be observed from the table, these overheads sum up to around three times a direct null member function call. It is observed that with an increase in the number of arguments, there is a slight increase in this overhead. However, this overhead does not depend on the the body of the destination member function or the body of its filter member since both get canceled while comparing D with E.

EVALUATION AND FUTURE WORK

The filter object model provides a flexible mechanism for developing dynamically pluggable filters for object oriented systems. Filter objects can be developed independently from their clients. A filter class does not break the encapsulation of its client, nor does the client need any modification to its code to agree with an added filter class. Only one restriction on the type

Table I. Timings for the five configurations

No. of Arguments	A	B	C	D	E
	μ sec.	μ sec.	μ sec.	μ sec.	μ sec.
0	0.527	0.628	1.033	1.963	0.727
1	0.607	0.710	1.054	2.258	0.870
2	0.866	0.958	1.301	2.724	1.260
4	1.274	1.370	1.730	3.630	1.910

declarations has to be maintained. The declarations of the filter-client member functions and the corresponding filter member declarations should match. However, between the corresponding arguments, it is possible to have ordinary subtyping relationships. The matching declarations are required, since a filter member takes and returns arguments of the same type as that of its client member function. The type information for filter members is also required to give them the overloaded meanings for their corresponding overloaded client members.

Filter objects can be maintained separately from their clients since the encapsulation between them is ensured. The code in a filter class can be reused by other classes at the level of class inheritance, since a filter class is treated at par with any other class. Methods defined in a filter class may be reused to design a new filter for a new or a derived class. However, filter relationships are not reusable. Reuse of filter relationships to obtain polymorphic filtering is a topic for further research.

The implementation of filters is at the user level, which involves overheads of a preprocessing stage during compilation and also involves run time overheads. A filter class has to be recompiled with its filter-client when it undergoes changes.

In this work, we have restricted ourselves to the design and development of filter objects for a sequential object-oriented paradigm. We are carrying out research work in order to achieve smooth integration of the filtered delivery model with concurrent distributed systems. The filtered delivery model finds several applications in distributed systems. A server object can be equipped with a filter to restrict requests depending on the current load on the server. A filter may reroute the requests to a new server making this decision transparent to the clients. We have used filters in our earlier work on control replication in distributed systems.[5] Filters act as schedulers to the replicas of a server.

The sequential and the distributed object models differ over the function call semantics. In a

Table II. Performance Overheads in Terms of Null Member Call

No. of arguments	Filter declaration overhead (B - A)/0.527	Plug Overhead (C - A)/0.527	Filter Overhead (D - E)/0.527
0	0.192	0.96	2.34
1	0.195	0.85	2.63
2	0.175	0.83	2.78
4	0.182	0.87	3.26

sequential object oriented language like C++, sending of a message is replaced by a blocking function call. The return value is treated as an implicit message that arrives at the originator and not as an explicitly programmed message. Whereas, in concurrent object oriented languages such as Charm++[14] and Mentat,[15] a sender object may be located in a different address space on a different processor than that of the destination object. The concurrency available is exploited by various non-blocking message passing mechanisms. The sender continues processing immediately after dispatching a message and does not have to block until a return message arrives. In such a case, if the return address is known, any object may return a value to the caller object on behalf of the destination object. A filter object can exploit the non-blocking call semantics by supporting forwarding of member function calls. For example, a filter may forward a Remote Procedure Call (RPC) request to a lightly loaded machine and permanently forget about it. Filters can serve various purposes in distributed systems such as fault tolerance, forwarding, routing, and load balancing. For example, a filter object may implement a *redundant invocation* mechanism or a *reinvoke upon timeout* mechanism to provide fault tolerance to message invocations. Filters to filters have applications in multilevel security architectures and in the development of layered network protocols.

CONCLUSIONS

We have discussed the filtered delivery model for message passing between objects. In this model, the messages can be intercepted and manipulated by special objects called filter objects. An interclass relationship called a filter relationship is introduced. A filter relationship empowers a filter class to provide filter member functions that can intercept messages sent to another class in a transparent manner. Filter objects can be dynamically plugged and unplugged to their clients. The model supports selective filtering of member function invocations. Beta messages, which act as directives to member functions are introduced to achieve dynamic and selective filtering. The filter object model was developed for the C++ object-oriented programming language, and a scheme was discussed to implement filter objects in C++ itself. The filtered delivery model separates the message control code from the message processing code in a transparent manner. Three practical applications for the filter object model were discussed. The model supports modular design of transparent filter objects. It is possible to design application specific filters and plug them to their client objects on demand.

ACKNOWLEDGEMENTS

We thank T. N. Shrikanta who wrote a part of the translator. We thank the anonymous referees whose comments and suggestions have improved the quality of the work.

REFERENCES

1. A. Goldberg and D. Robson, *Smalltalk-80*, Addison-Wesley, 1989.
2. B. Stroustrup, *The C++ Programming Language*, 2nd Ed., Addison-Wesley, 1991.
3. G.R. Andrews, 'Paradigms for process interaction in distributed programs,' *ACM Computing Surveys*, **23**(1) 49–90, (March 1991).
4. J. Reidl, V. Mashayekhi, J. Schnepf, M. Claypool and D. Frankowski, 'SuiteSound: a system for distributed collaborative multimedia, *IEEE Transactions on Knowledge and Data Engineering*, **5**(4) 600–610 (August 1993).

5. R.K. Joshi and D. Janaki Ram, 'ShadowObjects, a programming model for control replication in distributed systems', *Technical Report IITM-CSE-DOS-95-003*, 1995.
6. M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, 'Abstracting object interactions using composition filters', *Proceedings of ECOOP-1993*, LNCS-791, Springer-Verlag, 1993, pp. 152–184.
7. IONA Technologies Ltd., Dublin, Ireland, *The Orbix Architecture*, January 1995.
8. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.
9. G. Booch, *Object-oriented Analysis and Design, 2nd ed.*, Benjamin/Cummings, 1994
10. D. Janaki Ram and L. Rajesh Kumar, 'Multimedia multicast routing algorithm', *Proceedings of Networks-96*, Bombay, India, 1996, pp. 1-10.
11. D. Janaki Ram, Vivekananda, Ch. S. Rao and Krishna Mohan, 'Constraint Meta-object: a new object model for distributed collaborative designing', *IEEE Transactions on Systems, Man and Cybernetics (PART A)*, **27**(2), 208–221 (March 1997).
12. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
13. L. Cardelli and P. Wegner, 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys*, *17*(4), 471–522, (December 1985).
14. L. Kale and S. Krishnan, 'Charm++: a portable concurrent object oriented system based on C++, *8th Annual Conf. on Object-oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, *28*(10), 91–108 (October 1993).
15. A.S. Grimshaw, 'Easy-to-use object-oriented parallel processing with Mentat', *IEEE Computer* 39–51 (May 1993).